On the Impact and Defeat of Regular Expression Denial of Service

James C. Davis

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Applications

April 30, 2020
Blacksburg, Virginia

Keywords: Regular expressions, denial of service, ReDoS, empirical software engineering, software security

On the Impact and Defeat of Regular Expression Denial of Service

James C. Davis

(ABSTRACT)

Regular expressions (regexes) are a widely-used yet little-studied software component. Engineers use regexes to match domain-specific languages of strings. Unfortunately, many regex engine implementations perform these matches with worst-case polynomial or exponential time complexity in the length of the string. Because they are commonly used in user-facing contexts, super-linear regexes are a potential denial of service vector known as Regular expression Denial of Service (ReDoS). Part I gives the necessary background to understand this problem.

In Part II of this dissertation, I present the first large-scale empirical studies of super-linear regex use. Guided by case studies of ReDoS issues in practice (Chapter 3), I report that the risk of ReDoS affects up to 10% of the regexes used in practice (Chapter 4), and that these findings generalize to software written in eight popular programming languages (Chapter 5). ReDoS appears to be a widespread vulnerability, motivating the consideration of defenses.

In Part III I present the first systematic comparison of ReDoS defenses. Based on the necessary conditions for ReDoS, a ReDoS defense can be erected at the application level, the regex engine level, or the framework/runtime level. In my experiments I report that application-level defenses are difficult and error prone to implement (Chapter 6), that finding a compatible higher-performing regex engine is unlikely (Chapter 7), that optimizing an existing regex engine using memoization incurs (perhaps acceptable) space overheads (Chapter 8), and that incorporating resource caps into the framework or runtime is feasible but faces barriers to adoption (Chapter 9).

In Part IV of this dissertation, we reflect on our findings. By leveraging empirical software engineering techniques, we have exposed the scope of potential ReDoS vulnerabilities, and given strong motivation for a solution. To assist practitioners, we have conducted a systematic evaluation of the solution space. We hope that our findings assist in the elimination of ReDoS, and more generally that we have provided a case study in the value of data-driven software engineering.

On the Impact and Defeat of Regular Expression Denial of Service

James C. Davis

(GENERAL AUDIENCE ABSTRACT)

Software commonly performs pattern-matching tasks on strings. For example, when validating input in a Web form, software commonly tests whether an input fits the pattern of a credit card number or an email address. Software engineers often implement such string-based pattern matching using a tool called *regular expressions* (regexes). Regexes permit software engineers to succinctly describe the sequences of characters that make up common "languages" like the set of valid Visa credit card numbers (16 digits, starting with a 4) or the set of valid emails (some characters, an '@', and more characters including at least one '.'). Using regexes on untrusted user input in this manner may be a dangerous decision because some regexes take a long time to evaluate. These slow regexes can be exploited by attackers in order to carry out a denial of service attack known as *Regular expression Denial of Service* (ReDoS). To date, ReDoS has led to outages affecting hundreds of websites and tens of thousands of users.

While the risk of ReDoS is well known in theory, in this dissertation I present the first large-scale empirical studies measuring the extent to which slow regular expressions are used in practice. I found that about 10% of real regular expressions extracted from hundreds of thousands of software projects can exhibit longer-than-expected worst-case behavior in popular programming languages including JavaScript, Python, and Ruby. Motivated by these findings, I then consider a range of *ReDoS solution approaches*: application refactoring, regex engine replacement, regex engine optimization, and resource caps. I report that application refactoring is error-prone, and that regex engine replacement seems unlikely due to incompatibilities between regex engines. Some resource caps are more successful than others, but all resource cap approaches struggle with adoption. My novel regex engine optimizations seem the most promising approach for protecting existing regex engines, offering significant time reductions with acceptable space overheads.

Dedication

Praise God from whom all blessings flow.

Acknowledgments

So many people have contributed to this labor! Many thanks to all of you, and particularly...

To Kirsten: Thank you for suggesting we go to graduate school, for sharing in its triumphs and tribulations, and for listening with a smile to years of incomprehensible jargon.

To my multitudinous family: Thank you for your love, your questions, and for fruitful family colloquia. To quote my sister, Dr. Nan Pond: "*I joyfully acknowledge the absurd cross-disciplinary thinktank that is my family.*" I am also grateful to B. Danielak — you were the first of my friends to go to graduate school, and your passion has been an inspiration to me.

To my friends (you know who you are): Thank you, variously, for many rounds of racquetball, many madcap conversations, many tea parties, and many Tuesday evenings in the Word.

To L.C. Gayne, S. Duersch, and the GPFS team: Thank you for introducing me to the practice of computing, and for supporting me in the transition into research.

To P. De Arras and the Idego Coffee team: Thank you for helping me through the days and (a few) nights.

To my collaborators: Thank you for challenging me and for expanding my perspectives. I am particularly grateful to C. Coghlan, J. Donohue, Sk A. Hassan, A. Kazerouni, L. Michael IV, D. Moyer, and E. Williamson, for their expert and enthusiastic contributions to the research presented in this dissertation.

To D. Craig, S. Fulton, T. Pennings, and T. Nishikawa: Thank you for holding my hand in my first fumbling attempts at research.

To my doctoral committee: Thank you for your candid criticisms, and for your guidance and assistance on this path.

To my advisor, Dongyoon Lee: Thank you for inviting me into your research lab, infecting me with your curiosity and delight in discovery, and asking more of me than I thought possible. I would not be here without your unceasing support.

List of Figures

List of Tables

Part I

Introduction and Background

Chapter 1

Introduction

"Some people, when confronted with a problem, think 'I know, I'll use regular expressions.' Now they have two problems."

–Jamie Zawinski

1.1 Context and problem statement

Recognizing patterns is one of the most important challenges of life. Many tasks rely on the ability to determine whether a new object belongs to a class of objects of interest, or whether it resembles one you've seen before. We have long sought to train computers to solve pattern recognition problems on our behalf [170]. In the context of computing, many pattern recognition problems take the form of *string matching problems*. For example, business processes often require testing whether a user's input resembles an email address, or searching unstructured text for phone numbers, tax identifiers, or phrases of interest. By expressing these pattern recognition tasks as string matching problems, software engineers can use computers to automate time-consuming manual activity.

One of the most widely used techniques for solving string matching problems is known as *Regular Expressions* (regexes). A regex is a notion and a notation for concisely describing a set of strings that share a property. Rather than enumerating all such strings, a regex provides the machinery to generalize the content of these strings into a pattern. An algorithm can then be used to determine whether a candidate string matches this pattern. For example, the set of all valid email addresses might be described using the pattern "A non-empty sequence of characters, an '@' symbol, and then another non-empty sequence of characters containing at least one period." This pattern for email addresses can be encoded in traditional regex notation as `/.+@.+\..+/`.

Many practical problems can be described in the form of string matching using regexes [163]. The general nature of string matching has made regexes a popular tool, and software engineers report that they frequently use regexes [238]. By one estimate, 40% of software projects use regexes to solve string matching problems [115]. For example, string matching can be used to validate input [338], search for relevant code [300], and as part of linting or compiling software programs [65].

Although regexes are sometimes used in low-risk and ephemeral ways, e.g., during program comprehension [300], regexes are also used in business-critical contexts. Of particular interest in this dissertation, regexes are widely used in web servers to validate untrusted input, e.g., to confirm that an entry on a web form is truly an email and not an SQL injection attack. When used in this manner, regexes act as a defensive filter and prevent illegitimate data from poisoning a program, avoiding crashes and security vulnerabilities. But are the regexes themselves a potential vector for abuse? *Quis custodiet ipsos custodes?*

This dissertation focuses on the denial-of-service implications of using regexes as a defensive filter. Not all regex matches can be solved quickly in current computer systems, with potentially dire consequences for software stability. For some regexes, there are input strings that will cause the standard matching process to take super-linear time in the length of the pattern and input strings [284, 313, 336, 341]. To determine a match using the standard matching process, the worst-case time complexity is polynomial in the length of the input for some regexes, and exponential in the input length for others.[1] This high algorithmic complexity exposes software to a potential denial of service vector [246, 247] known as *Regular expression Denial of Service* (ReDoS) [135, 136]. A malicious actor can conduct a ReDoS attack by first identifying (or guessing [307]) a super-linear regex that is used by a web server to process untrusted input, and then submitting as input one of the strings that triggers that regex's worst-case behavior. Such an input will cause the web server to dedicate an inordinate amount of computational resources to the matching process, diverting resources away from legitimate users. If the attacker can divert sufficient resources, the effect will be to deny service to other users.

The technology of regexes has a great deal of inertia, and will only be changed for a compelling reason. To date, there have been only a few publicized examples of super-linear regex evaluations that disrupted a web service. Although some of these examples had substantial impact — e.g., rippling across thousands of websites (Chapter 3) — on the whole I believe that regex engine maintainers feel no urgency to address this issue, viewing ReDoS as a parlor trick rather than a pressing concern. For example, Google's V8 JavaScript engine has had defects describing its super-linear behavior since 2009 [274], and many Perl monks believe such regexes are unrealistic [277]. The motto of the maintainers of critical software like regex engines is quite reasonable: *"If it ain't broke, don't fix it. (And even if it is broke, tread carefully)."*

In light of this inertia, this dissertation investigates two questions:

Part II : Is ReDoS a significant threat to real-world software?
Part III : What is the solution space for ReDoS, and how effective are existing solutions?

[1] In Chapter 8, we extend prior analyses to show that some regexes require super-exponential time to match using the standard matching process.

1.2 Thesis

Because 10% of regexes exhibit super-linear worst-case behavior in typical regex engine implementations, ReDoS is a significant security threat to real-world software. Existing solutions are ineffective or impractical, while our new approaches appear promising.

1.3 Scientific contributions and applications

This dissertation has contributed to computer science in two ways:

Part II: I determined that a significant fraction — up to 10% — of real regexes exhibit super-linear behavior. Since regexes are commonly applied to untrusted input, this finding suggests that ReDoS may be a significant problem in practice. Some of these results were shared with the scientific community at ESEC/FSE [139, 141] and ASE [142].

Part III: I analyzed the ReDoS solution space and evaluated the effectiveness of various solutions. Some of the solutions evaluated in this work are novel. Others have been proposed by researchers or implemented in production systems, and I have provided the first empirical evaluation of their effectiveness. My findings have been shared with the scientific community at ESEC/FSE [139, 141] and USENIX Security [140]. Other findings are under preparation.

Beyond contributions to the state of knowledge, my findings will impact two engineering communities: software engineers and regex engine developers.

In the short term, software engineers should take more care with the use of regexes in their software. My empirical research has shown that most programming languages have exponential worst-case behavior, despite the adoption of safeguards in several programming languages. Unless software engineers incorporate regex vulnerability scans into their code review process, they should be suspicious of the use of regexes on time-critical paths. They may wish to respond *"Regexes considered harmful"* to code changes that introduce new regexes into their software.

In the long term, regex engine developers should modify their regex engines as a result of my findings. My empirical research has shown that software engineers frequently rely on super-linear regexes in their code, and so regex engine developers should address the widespread potential for ReDoS. With the adoption of regex engines comes the responsibility to offer users a secure foundation. But this task can be difficult to implement in the absence of usage data and a systematic empirical evaluation of the solution space. My work has addressed these gaps, guiding regex engine developers toward an effective solution appropriate to their context.

1.4 Organization

This dissertation is divided into four parts.

- Part I presents background and related work. Additional related work will be introduced as needed in the subsequent chapters.
- Part II describes my research showing that ReDoS is a problem in practice.
- Part III describes my research evaluating approaches to address ReDoS.
- Part IV concludes with a summary of the findings and opportunities for future work.

1.5 Statement of authorship, attribution, and copyright

As required by the Virginia Tech Dissertation Handbook,[2] *this section clarifies the intellectual ownership of the material in this document.*

This dissertation is in the "manuscript" style, with several chapters derived from published work that had multiple co-authors. Each chapter begins by indicating the publication(s) from which the material is drawn. Where appropriate, the content has been modified and extended to form a coherent dissertation.

I am the author of all of the material included in this dissertation. I identified the problems and potential solutions, designed and performed the experiments, and documented my findings. Where my co-authors had intellectual ownership of portions of a publication, I did not include that material in this dissertation. For example, Chapter 7 is derived from [141]; the qualitative portions of [141] were conducted by L. Michael and appear in his Master's thesis [237] rather than this dissertation.

All figures used in this document are of my own creation.

[2]See `https://guides.lib.vt.edu/c.php?g=547528&p=3756998`.

Chapter 2

Background and related work

"'Begin at the beginning', the King said, very gravely, 'and go on till you come to the end: then stop.'"

–Lewis Carroll

2.1 Outline

This dissertation is motivated by the importance of regexes within software engineering practice. In this chapter I will discuss the theoretical underpinnings of regexes, as well as their manifestation in computer software as experienced by a software engineer. This chapter covers the following material and related work:

- The history and theoretical foundations of regexes (§2.2);
- The various algorithms for testing whether a string is in the language of a regex (§2.3);
- The use of regular expressions (regexes) in software engineering practice (§2.4);
- The nature of Regular Expression Denial of Service (ReDoS) (§2.5); and
- Other research related to the work described in this dissertation (§2.6).

It concludes by summarizing what can be learned from the literature and identifying open questions that are resolved in this dissertation (§2.7).

The material on ReDoS (§2.5) and the summary of gaps in the literature (§2.7) is the most critical for understanding the contributions of this dissertation. I have provided the other material to give readers a more complete understanding of the context of my work, and to assist those unfamiliar with the material. Subsequent chapters will refer back to this material as needed, so readers may skip some or all of it for now.

2.2 The theory of regular languages

2.2.1 Mathematical origins

The theory of regular languages emerged from concurrent studies of computability and linguistics. Fundamental work in the 1930s by Gödel [173], Church [123], and Turing [324] sketched the general limits of computability using a machine with a finite set of states and an unlimited tape comprising its working memory. Subsequent work by McCulloch and Pitts posited that the behavior of neural systems could be modeled using "neural nets" with finitely many states and no memory at all [230]. Various extensions of these neural nets are now known more generally as *finite state automata*. Kleene described an algebra for the "regular events" that these automata could identify, and showed the correspondence between this algebra and the behavior of the automata [210].[1] Mealy soon observed that similar machines could be used to model the behavior of an electrical circuit [235]. Moore [253] and others [296] demonstrated and delineated the expressive power of various formulations of finite state automata. Eventually Rabin and Miller provided the canonical formalization of finite automata and the classes of problems that they could compute [281], while the linguist Chomsky described the limits of their expressiveness in the context of human language [121].

By the end of the 1950s, computational and linguistic researchers had introduced the parallel ideas of finite state automata (as a model of computation) and of regular languages (as an algebra and a linguistic model). It was known that these concepts had equivalent expressive power, but the precise correspondence between them had not yet been clearly shown. This task was accomplished independently by McNaughton and Yamada [234] and by Glushkov [171], both of whom provided algorithms for converting between a finite state machine and Kleene's algebraic language of regular expressions. After this, many researchers explored the expressive power of various classes of finite state machines, and the computational complexity they face in solving different problems. Many survey papers [192] and textbooks [93, 194, 301] summarize their efforts. I discuss work on the specific problem of *membership testing* in §2.3.

2.2.2 Regular expressions and finite automata

2.2.2.1 Kleene regular expressions

Kleene introduced the class of *regular events* that a McCulloch-Pitts neural net (finite state automaton) might identify [210]. A regular event is any sequence of elementary events from an alphabet Σ that can be constructed using a finite number of disjunctions, concatenations, and repetitions. Kleene introduced algebraic notation to describe this class of events inductively. A grammar for the *Kleene regular expressions* (K-regexes) is given in Grammar 2.1.

A Kleene regular expression consists of a sequence of terminal characters σ taken from an alphabet Σ, as well as operations on groups of these characters. Kleene defined three operations on the sub-patterns of an expression:

[1] It was Kleene who proposed the term "regular languages". In the same breath he made an appeal for a "more descriptive term" that never emerged.

$$
\begin{aligned}
\langle R \rangle \;\; ::= \;\; & P \\
| \;\; & (P) \\
| \;\; & P* \\
| \;\; & P \cdot P \\
| \;\; & P \mid P \\
\\
\langle P \rangle \;\; ::= \;\; & \sigma \in \Sigma \cup \{\varepsilon\} \\
| \;\; & \mathrm{R}
\end{aligned}
$$

Grammar 2.1: ***Kleene's grammar for regular expressions.*** *A Regular expression consists of one or more Patterns. Parentheses provide precedence. Patterns can be repeated (" $*$ ") and combined using concatenation (" $\cdot$ ") and disjunction (" $|$ "). Disjunction is also denoted " $\cup$ " and " $+$ " in the literature. My introduction of a second non-terminal P is intended to emphasize the recursive nature of sub-patterns, but is not strictly necessary.*

- **Disjunction**, permitting a nerve net to recognize the union of a set of regular events $(R_1|R_2)$;
- **Concatenation**, permitting a nerve net to recognize two regular events in sequence $(R_1 \cdot R_2)$;[2] and
- **Unbounded repetition**, permitting a nerve net to recognize a regular event zero or more times $(R*)$.

The *language* of a Kleene regular expression R, i.e., the set of strings that it describes, is denoted $L(R)$. This language can be described inductively in terms of the basic operations of the grammar:

$$
\begin{aligned}
&L(\varepsilon) = \{\varepsilon\} \\
&L(\sigma) = \{\sigma\} \\
&L(P_1 \cdot P_2) = L(P_1) \cdot L(P_2) \\
&L(P_1*) = L(P_1)* \\
&L(P_1|P_2) = L(P_1) \cup L(P_2)
\end{aligned}
$$

For example, with the alphabet $\Sigma = 0, 1$, the Kleene regular expression

$$R = (\ 0 \cdot 0 * \cdot 1\ ) \mid (\ 1 \cdot 0\ )$$

describes the sequence of events consisting of either (1) one or more 0's, followed by a 1; or (2) a 1 followed by a 0.

[2]In Glushkov's algebra, this operation is referred to as multiplication and can be thought of as a Cartesian product.

Table 2.1: ***Components of a finite automaton.*** *Components of a Rabin-Miller finite state automaton $A = \langle Q, q_0 \in Q, F \subseteq Q, \Sigma, \delta \rangle$. Whether the automaton is deterministic or non-deterministic depends on its transition function δ. The components of finite automaton A can be denoted, e.g., A_Q or A_{q_0}.*

Component	**Meaning**
Q	The (finite) set of states of the automaton: $Q = \{q_1, q_2, \ldots, q_m\}$, with $\lvert Q \rvert = m$
$q_0 \in Q$	The initial state of the automaton
$F \subseteq Q$	The accepting (Final) states of the automaton
Σ	The input alphabet for strings: $w \in \Sigma*$
$\delta : Q \times \Sigma \cup \{\varepsilon\} \to \mathbb{P}(Q)$	The transition function of the automaton, i.e., its "edges" in a graph representation

Kleene showed that every regular expression corresponded to the behavior of some finite state automaton, and vice versa. In other words, every finite automaton describes a class of regular events, and every regular expression describes some finite automaton. McNaughton and Yamada, as well as Glushkov, provided algorithms to show "which ones", i.e., to convert from a Kleene regular expression to an automaton and from an automaton to a Kleene regular expression [171, 234]. Expressions in Kleene's algebra can thus be viewed both as a description of a set of regular events, as well as a notation to describe the behavior of a finite state machine. I will describe their constructions using the Rabin-Miller model of a finite state machine, or *finite automaton* [281].

2.2.2.2 Finite automata

Rabin and Miller provided the canonical model of a finite state machine. Unlike machines based on Turing's model, which were automata with internal states and unbounded memory, Rabin and Miller studied finite automata with a fixed number of states to represent memory and computation. These automata can be described using the five-tuple $A = \langle Q, q_0 \in Q, F \subseteq Q, \Sigma, \delta \rangle$ with the meanings given in Table 2.1.

In Kleene's terminology, such a finite state automaton encodes a class of regular events of interest. Suppose that we encode a regular event in terms of a string over the input alphabet: $w \in \Sigma*$. We denote the empty string $w = \varepsilon, \lvert \varepsilon \rvert = 0$. Given a finite automaton A, starting from its start state A_{q_0}, we can repeatedly apply the transition function A_δ a total of $\lvert w \rvert$ times, once for each character in w. If the automaton ends in an *accept state* $f \in A_F \subseteq A_Q$, we say that the regular event described by the string w is in the language of the automaton: $w \in L(A)$. If the automaton does not end in an accept state, we say that the regular event described by the string w is not in the automaton's language: $w \notin L(A)$. The set of strings that drive a finite automaton to an accept state is known as the *language* of that automaton.

The behavior of a finite automaton can be represented by means of a directed *graph*: the states in Q comprise the vertices, the δ mapping is expressed in the form of *transitions* or

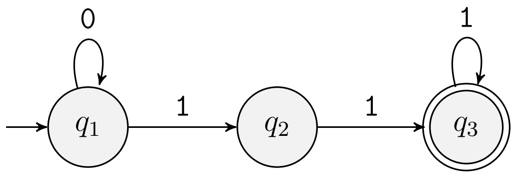


Figure 2.1: ***Finite automaton example 1.*** *Example of a finite automaton whose language can be described using the regular expression* $0 * \cdot 1 \cdot 1 \cdot 1*$. *There are implicit edges from* q_2 *and* q_3 *to the implicit reject state on the input* 0.

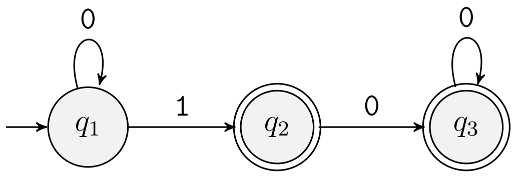


Figure 2.2: ***Finite automaton example 2.*** *Example of a finite automaton whose language can be described using the regular expression* $0 * \cdot 1 \cdot 0*$. *Note that this automaton has two accept states:* q_2 *and* q_3.

edges between the states, each edge is labeled with the corresponding character Σ, the vertex for the start state q_0 is denoted by an unlabeled edge entering from outside the automaton, and the vertices for the accept states F are indicated with a double circle. To avoid cluttering such a graph, there is usually an implied *reject state* to which implicit sink edges are directed from any vertices missing an outgoing state transition.

The correspondence between a finite automata and a regular expression can be seen intuitively from examples like the graphs shown in Figure 2.1 and Figure 2.2. These automata both have the start state q_1. The operation of these automata is to track their current state, consume a character from the input string w, and "move" (i.e., update their current state) by following the corresponding edge. While they consume 0's from the input string, they will remain in state q_1. Once they observe a 1, they will advance to state q_2. For a string to be in the language of the automaton depicted in Figure 2.1, it will then have to conclude with at least one 1, up to an unlimited number thereof. For a string to be in the language of the automaton depicted in Figure 2.2, it will then have to conclude with between zero and an unlimited number of 0 symbols. Such strings would cause the corresponding automata to terminate while in one of their accept state(s). Any other strings would cause these automata to transition to an implicit reject state.

The finite automata we consider come in two kinds: *deterministic* and *non-deterministic.* The determinism of an automaton is based on its δ component.

- If a finite automaton is deterministic (DFA), then for any input string there is exactly one path through a deterministic finite automaton (DFA). This property is ensured by means of an additional constraint on the automaton's transition function δ: a finite automaton

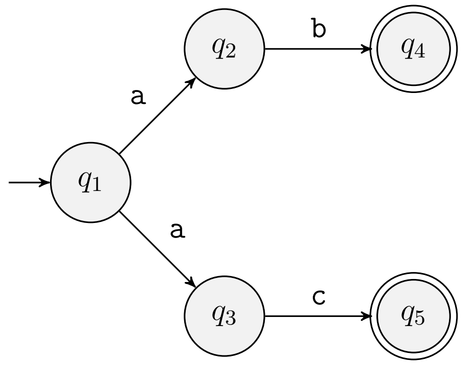


Figure 2.3: ***Finite automaton example 3.*** *Example of a non-deterministic finite automaton whose language can be described using the regular expression* $(a|a)(b|c)$*. This automaton is non-deterministic due to the multiple transitions from* q_1 *on the input* a*.*

is deterministic if every $(q, \sigma) \in \delta$ maps to a single state, i.e., $|\delta(q, \sigma)| = 1 \; \forall q \in Q, \sigma \in \Sigma$.

- If a finite automaton is non-deterministic (NFA), then for some input string, there is a point at which the simulation could advance to different states on the same character. More formally, a finite automaton is non-deterministic when δ is permitted to map some state to multiple states on some input character. In an NFA, there exists a $q \in Q$ and a $\sigma \in \Sigma \cup \{\varepsilon\}$ such that $|\delta(q, \sigma)| > 1$. In a non-deterministic automaton, there may be multiple paths through the automaton for a given input string, thus complicating its simulation.

The automata of Figures 2.1 and 2.2 are deterministic. The automaton of Figure 2.3 is non-deterministic.

Let us now consider the cost to store a finite automaton. A finite automaton with $|Q|$ vertices can have at most $\mathcal{O}(|Q|^2)$ edges if it is completely connected. To be fully described, the edges of the automaton must describe $\delta(q, \sigma)$ for every node in Q and the entire alphabet Σ. An automaton can be represented by a *transition table*, a two-dimensional matrix of automaton states and alphabet characters [105]. Such a table contains $|Q|$ columns and Σ rows, and each cell contains one or more states from Q. It may thus have $\mathcal{O}(|Q| * \Sigma * |Q|) = \mathcal{O}(|Q|^2 * \Sigma)$ storage cost. As Σ can be the Unicode character set, reducing this cost is critical for practical applications. This cost can be substantially lessened by means of an efficient encoding scheme for ranges of Σ that map to the same vertex [133]. Therefore, in the analyses of §2.3, we will assume that each edge has a constant storage cost, for an automaton storage complexity of $\mathcal{O}(|Q|^2)$.

2.2.2.3 Constructing an NFA corresponding to a regular expression

Researchers have proposed myriad constructions of an automaton corresponding to a given regular expression. These constructions may yield distinct automata with equivalent languages [92, 234]. Some constructions aim to avoid non-determinism, others to minimize the number of states, and others to minimize the number of transitions. The original construction of McNaughton and Yamada produces a DFA, unsuitable for exposition because the mapping from the expression to the automaton can be difficult to see. I will instead follow the NFA construction algorithm given by Thompson [321]. Thompson's construction resembles that of Glushkov [171],[3] but is more accessible thanks to Thompson's use of what are known as ε-*edges*. These edges can be taken by the automaton without consuming an input character.

Thompson's construction begins by parsing a regular expression into an abstract syntax tree, following Grammar 2.1 and applying precedence rules. The automaton is then built from a bottom-up traversal of the tree, beginning at the leaf nodes (terminals; $\sigma \in \Sigma$) and working its way to the root of the tree (the entire expression). At the leaf nodes, simple automata are constructed that can match each symbol. At successive layers, these automata are composed according to the meaning of the unary or binary operator being visited, connected using ε-edges. The final automaton thus corresponds to the set of all events.

The automaton for a leaf node, a symbol $\sigma \in \Sigma$, is depicted in Figure 2.4. From an initial state, there is a single edge on σ to the accept state.

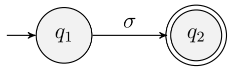


Figure 2.4: ***Thompson sub-automaton for $\sigma \in \Sigma$.*** *This figure shows the Thompson sub-automaton for a terminal node corresponding to the sub-pattern $\sigma \in \Sigma$. This automaton will accept $w = \sigma$ and reject all other inputs. In other words, its language is $L(P) = L(\sigma) = \{ \sigma \}$.*

The automaton for the unary operation $P*$ is depicted in Figure 2.5. The automaton will accept the empty string, as well as one or more strings from $L(P)$. In this and the following figures, the notation $L(P)$ indicates the sub-automaton for the pattern P. This sub-automaton has previously been produced using Thompson's construction, which proceeds from the leaves of the parse tree to its root and processes child nodes before parent nodes.

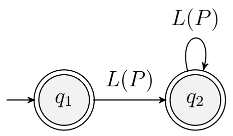


Figure 2.5: ***Thompson sub-automaton for repetition.*** *This figure shows the Thompson sub-automaton for repetition: an internal node corresponding to $P*$, for some sub-pattern P. This automaton will accept the family of strings $\{x^k \mid x \in L(P), k \geq 0\}$ (where x^k denotes x repeated k times).*

[3] See Glushkov's "minimal automaton" addendum to his Theorem 16.

The automaton for an inner AST node for the disjunction operation $P_1 \mid P_2$ is depicted in Figure 2.6. This automaton will accept a string from $L(P_1)$ or a string from $L(P_2)$.

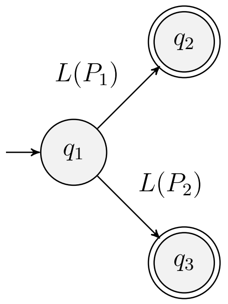


Figure 2.6: ***Thompson sub-automaton for disjunction.*** *This figure shows the Thompson sub-automaton for disjunction: an internal node corresponding to $P_1 \mid P_2$, for sub-patterns P_1 and P_2. This automaton will accept the family of strings $\{x \mid x \in L(P_1) \text{ or } x \in L(P_2)\}$.*

The automaton for the concatenation operation $P_1 \cdot P_2$ is depicted in Figure 2.7. The "accept" states of P_1 serve as the entry points into P_2.

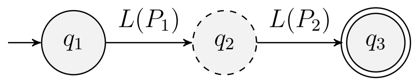


Figure 2.7: ***Thompson sub-automaton for concatenation.*** *This figure shows the Thompson sub-automaton for concatenation: an internal node corresponding to $P_1 \cdot P_2$, for sub-patterns P_1 and P_2. This automaton will accept the family of strings $\{xy \mid x \in L(P_1) \text{ and } y \in L(P_2)\}$. The dashed node q_2 represents the former accept state(s) of the sub-automaton for P_1, which may no longer be the accept state(s) of the combined sub-automaton for $P_1 \cdot P_2$.*

Figure 2.8 illustrates an automaton for the regular expression $(a \mid b) * c$. Note that it is non-deterministic — for example, from node q_2 the automaton can take either ε-edge to reach q_3 or q_5.

As presented here, Thompson's construction introduces up to three states at each node in the parse tree. The resulting automaton thus has $\mathcal{O}(|R|)$ states for a regular expression R of length $|R|$. Thompson often relies on ε-edges to join his sub-automata. In contrast, Glushkov introduces new states sparingly, with one state for each symbol $\sigma \in \Sigma$ from the regular expression. Under Glushkov's approach, algebra on Kleene's grammar (Grammar 2.1) is used to determine the viable destinations from each state. Glushkov does not rely on ε-edges; his approach is analogous to applying an ε-edge removal algorithm by computing the transitive closure of the ε-edges [301].

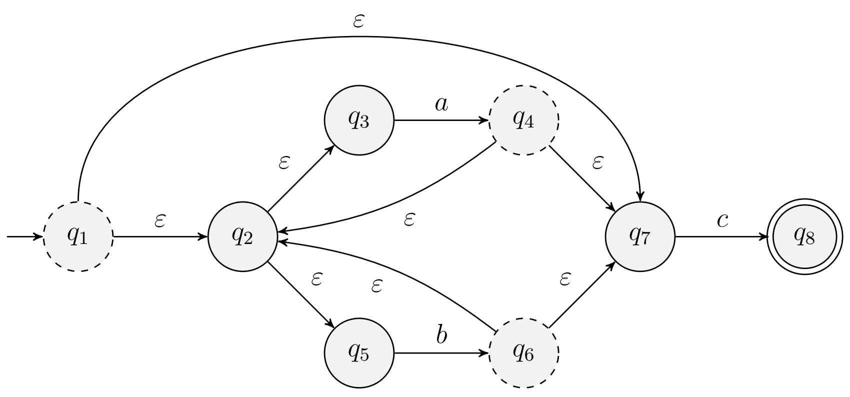


Figure 2.8: ***Example automaton following the Thompson construction.*** *This figure shows the automaton produced by Thompson's construction for the regular expression* $(a \mid b) * c$. *Vertices with a dashed circle indicate the accept states of some sub-automaton prior to being composed into the full automaton. In this case:* q_4 *and* q_6 *were the accept states of the automata for* a, b, *and* $a \mid b$; q_1, q_4, *and* q_6 *were thus the accept states of the automaton for* $(a \mid b)*$; *and the accept state for the automaton for* c *remains the overall accept state after completing the concatenation. Thompson automata may make liberal use of* ε*-edges.*

Table 2.2: ***Summary of regular expression membership testing algorithms.*** *Summary of the worst-case complexities of the regular expression membership testing (i.e., recognition) algorithms presented in this section, given a K-regex* R *and candidate string* w*. For the finite automaton algorithms, the complexities stated here use addition to distinguish between (time) the cost of constructing and simulating the automaton, and (space) the cost of storing the automaton (dominated by the transition table), the candidate string, and the simulation state. For the space complexity of DFA simulation, observe that $|Q|$ in this case may be exponentially larger than for an equivalent NFA. Spencer's and Thompson's NFA simulation algorithms use, respectively, depth-first search (DFS) and breadth-first search (BFS) to resolve non-determinism. The time and space complexity of derivative-based approaches vary, but some approaches can run in linear in $|w|$ (akin to the Thompson NFA simulation).*

Matching algorithm	**Summary**	**FA size**	**Time cxty.**	**Space cxty.**
DFA simulation (§2.3.1.1)	Apply δ	$\lvert Q\rvert = \mathcal{O}(2^{\lvert R\rvert})$	$\mathcal{O}(\lvert Q\rvert^2 + \lvert w\rvert)$	$\mathcal{O}(\lvert Q\rvert^2 + \lvert w\rvert + 1)$
Spencer NFA simulation (§2.3.1.2)	DFS	$\lvert Q\rvert = \mathcal{O}(\lvert R\rvert)$	$\mathcal{O}(\lvert Q\rvert^2 + \lvert Q\rvert^{\lvert w\rvert})$	$\mathcal{O}(\lvert Q\rvert^2 + \lvert w\rvert + \lvert Q\rvert * \lvert w\rvert)$
Thompson NFA simulation (§2.3.1.3)	BFS	$\lvert Q\rvert = \mathcal{O}(\lvert R\rvert)$	$\mathcal{O}(\lvert Q\rvert^2 + \lvert Q\rvert^2 * \lvert w\rvert)$	$\mathcal{O}(\lvert Q\rvert^2 + \lvert w\rvert + \lvert Q\rvert)$
Brzozowski derivatives (§2.3.2)	Algebraic ops.	–	*Varying*	*Varying*

2.3 Algorithms for regex membership testing

At the heart of Kleene's theory of regular events [210] is the notion that some sequences of events will be recognized by a given nerve net (finite automaton), and other sequences will not. As described in §2.2.2.1, Kleene introduced a notation called a regular expression that describes families of events, and that corresponds to an equivalent finite automaton. Given a regular expression and a sequence of events — or, equivalently, given a finite automaton and a string (§2.2.2.3) — we now wish to determine whether the sequence is in the *language* of the regular expression (i.e., accepted by the finite automaton).

This section describes the four common algorithms to solve the *regular expression language membership testing problem*, also known as recognition. The input to this problem is a pair (R, w): a regular expression R and a string w describing the sequence of events of interest. This is a decision problem, so the output is a Boolean, either "yes" or "no". The first three approaches solve this problem by leveraging the automaton interpretation of a regular expression, while the fourth approach relies on algebraic manipulation. These algorithms have varying time and space complexities summarized in Table 2.2. If the regex pattern R is fixed, terms like $|Q| = f(R)$ can be viewed as a constant. It is sometimes assumed that a candidate string w is much larger than the regular expression R, i.e., $|w| \ll |R|$, in which case terms involving $|w|$ may dominate other terms [221]. In practice, however, $|R|$ may be large, under circumstances we discuss in §2.4.3 (e.g., expansion of limited repetition).

Listing 1 General algorithm for solving the match (recognition) problem by simulating a finite automaton.

```
def isInLanguage(regexPattern, candidateString):
  # Build the automaton.
  FA = buildFA(regexPattern)
  # Simulate the automaton.
  finalState = FA.simulate(candidateString)
  # Check if we ended in one of the FA's accept states.
  return FA.acceptStates.contains(finalState)
```

2.3.1 Membership testing via automaton simulation

The equivalence between regular expressions and finite automata has led to a two-phase membership testing approach via automata simulation. In the first phase, the regular expression R is converted to an equivalent finite automaton. In the second phase, this automaton is simulated on the input string w.

The general algorithm is given in Listing 1. The three algorithms in sections 2.3.1.1 to 2.3.1.3 differ based on the automaton they build and the simulation they perform. I assume that all of these automata have a special "reject state" (i.e., "sink state"), q_{reject}, such that $\delta(q_{\text{reject},\sigma}) = q_{\text{reject}}\ \forall\ \sigma \in \Sigma$.

In explaining the three approaches, I will make reference to the NFA depicted in Figure 2.9. This NFA accepts the language of the regular expression `(a|a)b` and was produced using Thompson's construction. I will also assume that a finite automaton's δ function has the following properties:

- $\delta(q, \sigma)$ can be calculated in $\mathcal{O}(1)$ time (e.g., by storing outgoing edges at q).
- $\delta(q, \sigma)$ will return $\mathcal{O}(|Q|)$ vertices: 1 vertex for a DFA and between 1 and $|Q|$ vertices for an NFA.
- $\delta(q, \sigma)$ returns a unique set of vertices, no duplicates.

2.3.1.1 The McNaughton-Yamada-Glushkov DFA-based algorithm

Algorithms to convert a regular expression into a corresponding deterministic finite automaton (DFA) were proposed independently by McNaughton and Yamada [234], and by Glushkov [171, 172]. To understand these DFAs, it is helpful to contrast the NFA in Figure 2.9 with an equivalent deterministic automaton. The automaton given in Figure 2.9 is non-deterministic because from state q_1 on the input a the automaton can move to state q_2 or state q_3. The non-determinism can be removed using the Rabin-Miller *power set* construction algorithm [281], which creates a DFA with one "multi-state" for each of the subsets

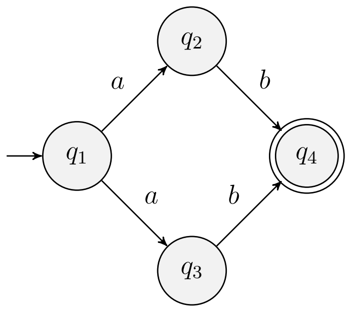


Figure 2.9: ***Automaton used to illustrate membership testing.*** *This figure shows the automaton produced by Thompson's construction for the regular expression $(a|a)b$. It will be used to illustrate the behavior of the automaton simulation algorithms in sections 2.3.1.1 to 2.3.1.3.*

of the states of an NFA. These multi-states are connected with edges such that there is an edge labeled σ between multi-state $q = \{q_a, q_b, ...\} \subseteq Q$ and multi-state $r = \{q_x, q_y, ...\} \subseteq Q$ provided that the original NFA had an edge labeled σ between some state in q and some state in r.

Given a regular expression of length $|R|$, these algorithms produce a DFA with $|Q| = 1 + 2^{|R|}$ states in the worst case, combining a reject state with the $2^{|R|}$ subsets in the power set of the $|R|$ original NFA states. If the DFA state-graph is complete, it may have a transition table of size $\mathcal{O}(|Q|^2) = \mathcal{O}((2^{|R|})^2) = \mathcal{O}(2^{2|R|})$. This DFA must be created and stored prior to simulation.

Once a DFA is created, simulating it is straightforward because of its deterministic nature. On each input character, there is exactly one possible next state. An algorithm for simulating the DFA is given in Listing 2. This algorithm takes constant space in addition to the cost of constructing the DFA. The simulation phase performs $\mathcal{O}(|w|)$ iterations of the loop, with each iteration making constant-time accesses to the transition function DFA_δ.

2.3.1.2 Spencer's NFA-based backtracking algorithm

"It can be pretty inefficient."

–Henry Spencer [305]

The trouble with constructing and then simulating a DFA is that the DFA may be exponentially large in the worst case. Thus, the algorithm discussed in §2.3.1.1 may be intractably expensive in practice. As an alternative to building a DFA, an equivalent non-deterministic finite automaton (NFA) may be created and simulated instead. This NFA may exist explicitly as states and edges [305], or implicitly, e.g., embedded in a grammar [157, 295]). Either way, NFA simulation is an accurate model for the behavior of this class of algorithms [335]. Thompson [321], McNaughton and Yamada [234], and Glushkov [171] all proposed NFA con-

Listing 2 Algorithm for solving the match problem by simulating a DFA.

```
def isInLanguage(regexPattern, candidateString):
  # Build the automaton
  DFA = buildDFA(regexPattern)

  # Simulate the automaton
  currState = DFA.q0
  for char in candidateString:
    nextState = DFA.deltaTransitionFunction(currState, char)
    currState = nextState

  # Check if we ended in one of the FA's accept states.
  finalState = currState
  return DFA.acceptStates.contains(finalState)
```

struction algorithms.[4] Although their construction algorithms yield automata with somewhat different numbers of states ($|Q|$) and transitions ($|\delta|$), they all yield NFAs with $\mathcal{O}(|R|)$ states and $\mathcal{O}(|R|^2)$ transitions.[5]

The *Spencer backtracking algorithm* (this section) and the *Thompson lockstep algorithm* (§2.3.1.3) both operate on an NFA by *simulating non-determinism.*

Spencer's approach is to simulate non-deterministic choices using a technique called backtracking [90, 248].[6] Any time Spencer's algorithm encounters ambiguity, it tries one path first, and if that path fails it will try the other(s) later. To implement a backtracking algorithm, a *backtracking stack* is used to store paths to try again later. If any path ends in an accept state, it is unnecessary to try the remaining options in the stack. If a path fails, however, the stack is consulted to identify the ambiguous points where another non-deterministic choice might have led to success. A version of this algorithm is given in Listing 3.

For example, consider the behavior of Spencer's algorithm as it simulates the NFA from Figure 2.9 on the candidate string $w = \mathrm{ac}$. The automaton simulation begins in state $\langle q_1, 0\rangle$ (where $w[0] = a$). Applying the transition function, $\delta(q_1, a)$ yields $\{q_2, q_3\}$. Thompson's

[4]The automata produced by McNaughton and Yamada and by Glushkov are known as *position automata* and do not require ε-edges. The automata produced by Thompson are rife with ε-edges. If an NFA is ε-free, it can be simulated directly using the algorithms in this section and the next. If not, it can either be converted to an ε-free NFA [301], or the ε closure can be calculated when determining each next set of states.

[5]Although the worst-case $\mathcal{O}(|R|^2)$ does not always occur; Nicaud [262] has shown that McNaughton-Yamada-Glushkov position automata have on average a linear number of transitions in $|R|$.

[6]I do not know who first proposed the algorithm that I here call "Spencer's algorithm". Spencer used this algorithm in the Ur-regex engine on which many popular programming languages have based their implementations, and so I credit him here to acknowledge his influence. But the general technique of backtracking had apparently been applied to NFA simulation as early as 1968, when Thompson refers to it [321].

simulation will arbitrarily order these states and try one first — say, q_2. It will push the alternative simulation state $\langle q_3, 1\rangle$ onto the stack. Proceeding with its choice, the simulation will discover that $\delta(q_2, c)$ yields the reject state, and will have exhausted the candidate string. Not having ended in an accept state, Thompson's algorithm will pop from its backtracking stack to examine another possibility: $\langle q_3, 1\rangle$. Proceeding with this alternative choice, the simulation will find that $\delta(q_3, c)$ yields the reject state. Now the backtracking stack is empty, so the simulation concludes by reporting that the candidate string is not in the language of the automaton (nor the regular expression from which the NFA was derived).

Beyond the $\mathcal{O}(|R|^2) = \mathcal{O}(|Q|^2)$ space complexity of storing an NFA, the space complexity of simulating it using Spencer's algorithm is a fairly intuitive $\mathcal{O}(|Q| * |w|)$ in the case of a completely-connected NFA, i.e., one for which all state-vertices are connected on every character $\sigma \in \Sigma$. The stack will have no more than $|w|$ distinct indices amongst its backtracking points, and at each index it can add at most $|Q|$ states to the backtracking stack — so there will be at most $\mathcal{O}(|Q| * |w|)$ stack entries to which it may backtrack at any particular time.

In terms of time complexity, it costs $\mathcal{O}(|R|^2) = \mathcal{O}(|Q|^2)$ to construct an NFA. The worst-case time complexity of simulating this NFA using Spencer's algorithm can be bounded as $\mathcal{O}(|Q|^{|w|})$ by counting the number of *simulation states* processed during the backtracking simulation. To see this, let's again examine the case of the same pathological completely-connected NFA as before, and this time suppose that the NFA has no accepting states. Consider the following two properties of the simulation of this NFA on an input w:

- Each time we consume a character at index j, we handle one simulation state directly and add $|Q|-1$ simulation states to the backtracking stack. This is because in the pathological NFA, the δ transition function returns all of the states Q from any character.
- Because the NFA has no accepting states, none of the potential solutions (paths through the NFA) will end in an accepting state. Thus we will eventually process each of those $|Q|$ backtracking points (one immediately and the other $|Q| - 1$ via backtracking). When processed, each of *those* states will themselves introduce $|Q|$ backtracking points.

Combining these two facts, we see that the number of simulation states we process compounds for each additional input character: $|Q|$ states to consider for $|w| = 1$, $|Q|^2$ states for $|w| = 2$, and so on — $\mathcal{O}(|Q|^{|w|})$ states overall. The cost of handling the states returned by the transition function δ at each point will itself cost $\mathcal{O}(|Q|)$ (for there are $|Q|$ states returned), giving us a total estimated cost of $\mathcal{O}(|Q| * |Q|^{|w|}) = \mathcal{O}(|Q|^{|w|+1})$.

It is worth noting that the number of simulation states processed by this algorithm will grow exponentially under a much weaker condition. The NFA need not be *completely* connected, so long as there is *some* vertex in the NFA such that there are two different paths that originate and terminate at this vertex under the same string ρ. If there are two such paths, each time the simulation sees an instance of ρ it will double the number of paths to explore, leading to a cost of $\mathcal{O}(2^{|w|})$ instead of $\mathcal{O}(|Q|^{|w|})$. This is of course a *more* desirable worst-case behavior, but still exponential in the length of the input string w. This behavior lies at the heart of ambiguity leading to Regular Expression Denial of Service (ReDoS), discussed further in §2.5.

Finally, it is also worth noting that Spencer's algorithm can perform in best-case linear time in the length of the candidate string: $\mathcal{O}(|Q| * |w|)$. This time can be achieved if the candidate string *never* requires the simulation to make a non-deterministic choice. Then the simulation will follow a deterministic path through the NFA, its backtracking stack is never used, and the simulation will cease after $|w|$ iterations of the inner loop and only one iteration of the outer backtracking loop.

Listing 3 Algorithm for solving the match problem using Spencer's backtracking NFA simulation. The backtracking stack contains NFA simulation states in the form of tuples $\langle q, j \rangle$, where $q \in Q$ and j is an index into the candidate string w.

```
class BacktrackingPoint:
  def __init__(self, state, stringIx):
    self.state, self.i = state, stringIx

def isInLanguage(regexPattern, candidateString):
  # Build the automaton
  NFA = buildNFA(regexPattern)

  # Initialize the backtracking stack
  stack = Stack() # Contains BacktrackingPoints
  initialState = BacktrackingPoint(NFA.q0, 0)
  stack.push(initialState)

  # Backtracking loop to evaluate choices we haven't yet considered
  while not stack.empty():
    # Simulate the next backtracking point to completion,
    # appending to the stack at each non-deterministic choice.
    bPt = stack.pop()
    currState = bPt.state
    i = candidateString[bPt.i]

    for j, nextChar in enumerate(candidateString[i:]):
        # Where might nextChar lead?
        possibleStates = NFA.deltaTransitionFunction(currState, nextChar)

        # Pick one for now
        currState, others = possibleStates[0], possibleStates[1:]

        # Unlike Frost, we can take the path less traveled later
        backtrackingPoints = [BacktrackingPoint(q, j+1) for q in others]
        stack.push( backtrackingPoints )

    # End of candidateString.
    # Check if we ended in one of the FA's accept states.
    finalState = currState
    if NFA.acceptStates.contains(finalState):
      return MATCH

  # None of the paths reached an accept state.
  return MISMATCH
```

2.3.1.3 Thompson's NFA-based lockstep algorithm

Thompson's approach [321] is to simulate non-deterministic choices using what has been called the lockstep algorithm [284]. Where Spencer's approach saved unexplored paths for later in a backtracking stack, Thompson's approach instead tracks all of these paths together by managing a *frontier* of possible states. Thompson saves the current set $\Phi_{curr} \subseteq Q$ of potential states, and upon consuming each character he updates this set to $\Phi_{next} \subseteq Q$ by applying the δ transition function to each state in Φ_{curr} in turn. Thus the frontier of possible positions in the NFA is updated at each character in lockstep. A version of this algorithm is given in Listing 4.

For example, consider the behavior of Thompson's algorithm as it simulates the NFA depicted in Figure 2.9 on the candidate string $w = \text{ac}$.

1. The automaton simulation begins at the initial state: $\Phi_{curr} = \{q_1\}$.
2. Applying the transition function to the states in Φ_{curr} on the input character a, the simulation obtains $\Phi_{next} = \{q_2, q_3\}$ and then copies Φ_{next} into Φ_{curr}.
3. Repeating this, on the next input character c, the simulation obtains the next $\Phi_{next} = \{\text{REJECT}\}$ — both $\delta(q_2, c)$ and $\delta(q_3, c)$ yield the same state, and Φ_{next} removes the redundancy.
4. Having exhausted the candidate string, Thompson's simulation now examines Φ_{curr} to see whether $\Phi_{curr} \cap F \neq \emptyset$.
5. As $\Phi_{curr} = \{\text{REJECT}\}$, the simulation concludes by reporting that the candidate string is not in the language of the automaton (nor the regular expression from which the NFA was derived).

Beyond the $\mathcal{O}(|R|^2) = \mathcal{O}(|Q|^2)$ space complexity of storing an NFA, the space complexity of simulating it using Thompson's algorithm is $\mathcal{O}(|Q|)$. The algorithm maintains only the state-sets $\Phi_{curr} \subseteq Q$ and $\Phi_{next} \subseteq Q$.

In terms of time complexity, it costs $\mathcal{O}(|R|^2) = \mathcal{O}(|Q|^2)$ to construct an NFA. The worst-case time complexity of simulating this NFA using Thompson's algorithm can be bounded as $\mathcal{O}(|w| * (\textit{the cost to query } \delta \textit{ and update nextFrontier for each } q \in \Phi_{curr}))$ — in each of the $|w|$ iterations, we query δ for each of the $\mathcal{O}(|Q|)$ states in Φ_{curr}, and must merge the $\mathcal{O}(|Q|)$ states returned by each query. Based on the properties of δ assumed in §2.3.1, the time complexity is expected to be $\mathcal{O}(|Q|^2 * |w|)$.[7]

A common interpretation of Thompson's algorithm is that it is a "dynamic DFA construction" [132, 332]. To see this, consider the Rabin-Miller power-set construction for a DFA, as described in §2.3.1.1. This construction produces one multi-state for every subset of Q in the original NFA, with edges connecting these multi-states corresponding to potential routes in the NFA. The state of the Thompson algorithm — Φ_{curr} and Φ_{next} — are precisely the multi-states that would be taken during a traversal of the full power-set DFA under the

[7]Google's RE2 implementation caches these computations to lower the time cost [133].

Listing 4 Algorithm for solving the match problem using Thompson's lockstep NFA simulation. The simulation simulates non-determinism by tracking all possible NFA states together, i.e., an NFA "state frontier".

```
def isInLanguage(regexPattern, candidateString):
  # Build the automaton
  NFA = buildNFA(regexPattern)

  # Choice frontier begins at q0
  currFrontier = set( [ q0 ] )

  for nextChar in candidateString:
    # Compute the next choice frontier: the union of all destinations
    # from the current frontier on this character.
    nextFrontier = set()
    for state in currFrontier:
      possibleStatesFromHere = NFA.deltaTransitionFunction(state, nextChar)
      nextFrontier.add(possibleStates)

    # Update currFrontier.
    currFrontier = nextFrontier

  # End of candidateString.
  # Check if any path ended in one of FA's accept states
  if currFrontier.intersect( NFA.acceptStates ):
    return MATCH

  # None of the paths reached an accept state.
  return MISMATCH
```

candidate string of interest. So Thompson's algorithm dynamically performs the power-set construction, but only for the path through the DFA that would be taken on one particular input string.

2.3.2 Membership testing via algebraic manipulation: Brzozowski derivatives

The previous section described membership tests that leverage the equivalence of regular expressions and finite automata. Those approaches required two steps: converting the regex into an automaton, and then simulating the automaton. In this section I describe an alternative approach, that of algebraic manipulation of the regular expression in the context of the candidate string as pioneered by Brzozowski [103]. To the best of my knowledge, this approach is not used in any production regex engines, so I will not treat it in detail. Researchers from Microsoft have, however, demonstrated its potential in a research regex engine [293], and may plan to incorporate it into the .NET framework.

Brzozowski's approach relies on *regular expression derivatives*. A derivative describes the effect of a symbol $\sigma \in \Sigma$ on a regular language L, i.e., the strings that are prefixed with this symbol. For example, consider the regular expression $R = foo|bar$, whose language is $L(R) = \{foo, bar\}$. The derivative of this language with respect to $f \in \Sigma$ is $D_f(R) = \{oo, ar\}$ — these are the strings that remain after subtracting off the prefix "f". A regular expression derivative is a language of strings, and Brzozowski showed that it is also regular. In addition to defining the derivative explicitly in terms of a set of strings, we can also define the derivative implicitly in terms of the regular expression that denotes this language.

With this in mind, here are the derivatives on the regular expression terminals from Grammar 2.1:

$$D_\sigma(\emptyset) = \emptyset$$
$$D_\sigma(\varepsilon) = \emptyset$$
$$D_\sigma(\sigma) = \varepsilon$$
$$D_\sigma(\sigma') = \emptyset \text{ if } \sigma' \neq \sigma$$

and derivatives for regular expressions that use operators:

$$D_\sigma(R*) = D_\sigma(R) \cdot R*$$
$$D_\sigma(R_1 \mid R_2) = D_\sigma(R_1) \mid D_\sigma(R_2)$$
$$D_\sigma(R_1 \cdot R_2) = D_\sigma(R_1) \cdot R_2 \text{ if } \varepsilon \notin L(R_1)$$
$$D_\sigma(R_1 \cdot R_2) = (D_\sigma(R_1) \cdot R_2) \mid (D_\sigma(R_2)) \text{ if } \varepsilon \in L(R_1)$$

Having done so, we can define the recognition problem in terms of *repeated derivation*. To learn whether a string $w = \sigma_1\sigma_2 \ldots \sigma_n$ is in the language of a regular expression R, compute $D_1 = D_{\sigma_1}(R)$, then $D_2 = D_{\sigma_2}(D_{\sigma_1}(R))) = D_{\sigma_2}(D_1)$, and so on until you reach D_n. At D_n, all characters from w have been consumed; now, if $\varepsilon \in D_n$, then $w \in L(R)$.

There are both dynamic and static approaches to solving the recognition problem using Brzozowski derivatives. Dynamically, Brzozowski derivatives can be computed on a per-candidate-string basis to answer recognition queries, as we have just discussed. Statically, as Thompson observed [321], his algorithm can be understood as the automaton analogue of the repeated application of derivatives. The Thompson NFA simulation considers all possible NFA states at once based on the characters consumed so far, just as the Brzozowski derivative tracks all possible regular suffixes based on the prefix for which the derivative has been calculated. A static alternative to calculating dynamic derivatives is thus to pre-compute all possible derivatives: $D_\sigma(R)$ for each $\sigma \in \Sigma$, then derivatives of those D_σ, and so on. Brzozowski showed that there are finitely many such derivatives, and Owens et al. show that there are $\mathcal{O}(2^{|R|})$ equivalence classes for a given starting expression R [271]. A DFA can then be constructed from these equivalence classes, with one DFA state per class and edges to indicate transitions upon derivatives. The time and space complexity of using Brzozowski derivatives will therefore range from that of Thompson's NFA simulation (dynamic derivatives relate to w) to that of a DFA simulation (static pre-computation of all derivatives).

Brzozowski derivatives have been extended beyond the recognition problem. Sulzmann and Lu showed that they can solve the parsing problem (sub-matches) [316], and Saarakivi et al. have recently demonstrated a more full-featured regex engine via Brzozowski derivatives [293].

2.4 Regular expressions in software engineering practice

2.4.1 A brief history of the adoption of regexes by computing practitioners

As discussed in §2.2, the theory of regular expressions was initially developed as a model for biological systems [210, 230] Once that theory was applied to string matching problems [121, 321], computing practitioners began to adopt it for general engineering purposes. Using regexes for string matching was popularized through UNIX tools like ed (1973 [233]), sed, awk, grep, and egrep.[8] Soon C libraries for regexes emerged, e.g., the `regexp` module in the GNU C library (1980s) [309]. Led by Perl [329], programming languages began to incorporate regexes as a language primitive or within their standard libraries. Now virtually all mainstream programming languages have built-in support for regexes, including JavaScript, Java, PHP, Python, Ruby, Go, Perl, Rust, and C# (.NET) [178]. These programming languages maintain *regex engines* whose purpose is to accept the pattern representation of a regex and answer variations on the regex match problem.

The regexes used in software engineering practice have diverged from the Kleene regular expressions (K-regexes) defined and studied by theorists. Whether at the behest of software

[8]The program name grep is derived from the command `g/re/p` in the ed editor, which would Globally apply a Regular Expression and Print the result [285].

engineers hungry for conciseness or expressiveness, or motivated by inter-tool competition, or merely due to the vagaries of the implementers, the regexes used by software engineers bear little resemblance to the regexes of the automata theorists (§2.2). I will use the term *Extended Regular Expressions* (E-regexes) to distinguish the regexes supported by programming languages from the well-studied K-regexes of yore.

From the perspective of a software engineer, a regular expression is a tool and a domain-specific language to match strings — a regular expression is a string pattern language. Software engineers can apply regular expressions in many contexts where strings are processed. Software engineers can use regexes in all mainstream programming languages [178]. They can also turn to regexes on the command line; UNIX command-line tools have long supported regexes for use in shell scripting, from primitive regexes like shell globs to the increasingly-expressive regexes supported by grep, egrep, sed, and other command-line interfaces. Regexes are often offered in a search API for text-centric applications, such as web browsers [251, 292], word processors [242, 269], and IDEs [152, 243]. Regexes are also supported in broader contexts where content can be interpreted as strings, e.g., databases [13, 200] and network monitoring tools [77, 120, 127, 155].

In this section I will describe the syntax and semantics of regexes used in modern programming languages (§2.4.2), discuss the relationship between these Extended Regular Expressions (E-regexes) and the Kleene Regular Expressions (K-regexes) of the theorists (§2.4.3), consider implementation concerns (§2.4.4), and sketch alternative semantics proposed in the literature (§2.4.5).

2.4.2 Regex syntax and semantics in modern programming languages

Regexes reached maturity in the 1980s, with the release of Henry Spencer's regex package in 1986 [305], Larry Wall's Perl in 1987 [163], and the GNU regex lib in 1988 [309]. Programming languages have followed the example of Perl, and regexes are now supported in all mainstream general-purpose programming languages. "Scripting" languages [268] often include regexes in the language specification, e.g., Perl [330], PHP [181], Ruby [99], and JavaScript [147]. Other programming languages and frameworks offer regexes through a core module, as is done in Python [216], Java [130], Rust [146], Go [176], and the .NET framework [240].

In the context of programming languages, regexes are a string pattern language. I introduce this language using its syntax (§2.4.2.1) and semantics (§2.4.2.2).

2.4.2.1 Perl-Compatible Regular Expressions (PCRE) syntax

There have been several specification efforts for E-regexes [163]. Since 1992, the POSIX standard has described two types of regexes: POSIX Basic Regular Expressions and POSIX Extended Regular Expressions [201]. These variations share core semantics but differ in their expressive power (i.e., conciseness and feature support). Despite the efforts of the POSIX committee, their standard has not been accepted by the regex engine developer community. Instead, Wall's Perl regexes have dominated the discussion.[9] In an effort to establish a Perl-oriented specification that could be used by other tools and programming languages, in 1997 Hazel published the *Perl-Compatible Regular Expression* specification (PCRE) [187]. The POSIX standard uses the same syntax as PCRE for the features that it supports.

The regex sub-language of mainstream programming languages is derived from the PCRE syntax and semantics. All programming languages support a core set of regex features, and some support various regex extensions. The regex language in this dissertation will therefore follow the PCRE standard, which is a superset of the regex dialects in most programming languages. The PCRE syntax is given in Table 2.3.

In this dissertation, "Extended Regular Expressions" (E-regexes) refers to the syntax and semantics of PCRE regexes. I have omitted some of the more exotic PCRE features, most notably inline options,[10] conditionals,[11] recursive patterns,[12] and callouts.[13]

[9]Perhaps this is because Perl regexes have greater expressiveness than the two forms of regexes proposed by the POSIX specification.

[10]Inline options permit changing the match configuration in parts of the evaluation, e.g., enabling or disabling case-sensitivity while evaluating a sub-pattern. For example, in the pattern $R_1(?i)R_2$, the sub-pattern R_2 is matched in a case-insensitive manner.

[11]Conditionals permit testing pattern R_1 for a match, and applying R_2 or R_3 depending on whether R_1 matched.

[12]Recursive patterns permit encoding a "stack" to match, e.g., matching left- and right-parentheses.

[13]Callouts permit executing arbitrary code during a regex evaluation.

Table 2.3: ***Features and notation of E-regexes (PCRE).*** *Summary of the major features and notation of regexes in mainstream programming languages. This table uses the notation from the PCRE specification [188]. Abbreviations are consistent with Chapman and Stolee where possible [115]. A P denotes a sub-pattern consisting of any combination of the regex features. In most cases regex features can be arbitrarily combined, though some regex engines restrict interactions between certain features.*

Abbreviation	Feature	Notation
	Regular features (K-regexes)	
CAT	X followed by Y	P_1P_2
KLE	Zero-or-more repetition	$P*$
OR	Logical OR (Disjunction)	$P_1\|P_2$
CG	Capture group	(P)
NCG	Non-capturing group	(?:P)
PNG	Named capture group	(?<name>P)
	Additional quantifiers	
ADD	One-or-more repetition	$P+$
QST	Zero-or-one repetition	$P?$
DBB	m-to-n repetition	$P\{m,n\}$
LWB	At-least-m repetition	$P\{m,\}$
SNG	Exactly-n repetition	$P\{n\}$
	Shorthand for a subset of Σ	
CCC	Custom character class	$[aeiou]$
RNG	CCC with range	$[a\text{-}z]$
NCCC	Negated CCC	$[\hat{}aeiou]$
ANY	Built-in class: non-newline character	.
WSP	Built-in class: whitespace character	$\backslash s$
DEC	Built-in class: numeric character	$\backslash d$
WRD	Built-in class: word character	$\backslash w$
NWSP	Built-in class: non-whitespace character	$\backslash S$
NDEC	Built-in class: non-numeric character	$\backslash D$
NWRD	Built-in class: non-word character	$\backslash W$
VWSP	Built-in class: vertical whitespace	$\backslash v$
	Zero-width assertions	
STR	Start-of-{string,line}	$\hat{}P$, \A$P$
END	End-of-{string,line}	P\$, $P\backslash$ Z
WNW	Word/non-word boundary	\b
NWNW	Negated WNW	\B
PLA	Positive lookahead	(?=P)
NLA	Negative lookahead	(?!P)
PLB	Positive lookbehind	(?<=P)
NLB	Negative lookbehind	(?<!P)
	Backreferences	
BKR	Backreference (numeric)	$(R)\ldots\backslash 1$
BKRN	Backreference (named)	(?<name>P)...\k<name>
	Prioritization and backtracking controls	
LZY	Non-greedy repetition	$P*?$, $P+?$, $P\{m,n\}?$
ATM	Atomic group (cut)	(?>P)
POS	Possessive quantifier (cut)	$P++$, $P*+$

Table 2.4: ***Testing whether a string is in the language of a regex.*** *This table provides examples of language membership testing for simple regexes. The syntax follows Table 2.3.*

Pattern	Candidate string	Is in language?	Explanation
/(ab)|(cde)/	*ab*	Yes	Matches the left side of the disjunction
/a+b?c{3}[xy]/	*aaccc*	No	The candidate string is missing a trailing x or y
/(a+)(?=b)\1/	*aabaa*	Yes	One or more a's, the lookahead is satisfied, and the captured a's are repeated

2.4.2.2 PCRE semantics

Having defined the syntax we will use for regexes, we now turn our attention to their match semantics. As with Kleene regular expressions, with E-regexes one tests whether or not a candidate string is part of the language of a regex by applying the sub-patterns of the regex (left to right) against the characters in the string (left to right). For example, suppose we have the regex `/^ab+c$/`. The strings *abc* and *abbbc* are in its language. The string *ac* is not in the language, because the pattern requires at least one *b*. Likewise, the string *bac* is not, because the order of characters is incorrect. Further examples are given in Table 2.4.

The PCRE semantics permit several regex queries:

1. **Full match**, i.e., **recognition**: A Boolean result indicating whether or not the *entire* candidate string is in the language of the regex.
2. **Partial match**, i.e., **partial recognition**: A Boolean result indicating whether or not some *substring* of the candidate string is in the language of the regex.
3. **Matched string**: For a partial match, indicates the substring and offset of the candidate string that matched the pattern.[14]
4. **Capture groups**, i.e., **parse**: If the candidate string is in the language of the regex, and if the regex contains sub-patterns enclosed in capture groups, then the sub-strings that they matched and their positions are disclosed. This requires that the regex engine act as a parser (building a parse tree) rather than a recognizer (returning a Boolean result).

The results from these queries are dictated by the semantics of the regex match, i.e., the matching behavior that a regex will exhibit when it is applied to a string. Perl-Compatible Regular Expressions follow a *leftmost-greedy rule*. This rule can be stated in three parts as follows [189]:

1. **Leftmost match:** "*A pattern is matched against a subject string from left to right.*" In the more precise words of the POSIX specification, "*The search for a matching sequence*

[14]In addition to being reported to the caller, if a regex is being evaluated in "global" mode, this offset may be tracked by the regex engine as the starting point in a subsequent match.

Table 2.5: ***Leftmost-greedy match semantics.*** *This table illustrates the leftmost-greedy match semantics followed by all mainstream regex engines. The syntax follows Table 2.3.*

Pattern	**Candidate string**	**Matching substring(s)**	**Rule illustrated**
	Default semantics		
a+	**a**baa	a	Leftmost match
(a|aa)	**a**a	a	Left-to-right disjunctions
(a+)(a+)	**aa***a*	(aa), (a)	Greedy pattern
	Non-greedy quantifiers		
(a+?)(a+)	**a***aa*	(a), (aa)	Non-greedy pattern

starts at the beginning of a string and stops when the first sequence matching the expression is found, where 'first' is defined to mean 'begins earliest in the string' " [202].

2. **Left-to-right disjunctions:** "*The matching process tries each alternative in turn, from left to right, and the first one that succeeds is used.*" In other words, the disjunction operation is not commutative [284]; `/(a|aa)/` and `/(aa|a)/` will match different substrings.
3. **Greedy patterns:** "*Quantifiers are 'greedy', that is, they match as much as possible (up to the maximum number of permitted times), without causing the rest of the pattern to fail.*"[15]

The behavior of a few illustrative cases is illustrated in Table 2.5. Expanding on these cases in more detail:

- In the first example, the "*Leftmost match*" semantic causes the quantifier to only match the first character, even though there is a longer matching substring later in the string.
- In the second example, the "*Left-to-right disjunctions*" semantic gives the first of the two disjunction patterns priority over the second.
- In the third example, the "*Greedy patterns*" semantic causes the first quantified group to consume as many characters as possible while still permitting a match to occur. Hence, the first two a's are consumed in the first group, and the third a is consumed in the second group.
- In the fourth example, the use of non-greedy repetition (indicated at the bottom of Table 2.3) is illustrated. The first quantified group is non-greedy, reversing its behavior from "*matches the longest possible string*" to "*matches the shortest possible string.*" Hence, the first group (non-greedy) consumes only the first a, and the remaining a's are consumed in the second (greedy) group.

[15]This rule can be inverted using the "non-greedy" quantifiers proposed by Friedl [163], also known as "lazy" or "reluctant" quantifiers. Such quantifiers consume the shortest viable sequence, not the longest one.

Table 2.6: ***Sample reductions from K-compatible E-regexes to pure K-regexes.*** *The syntax follows Table 2.3.*

Abbreviation	Feature	Notation	Reduced to K-regex
		Additional quantifiers	
ADD	One-or-more repetition	$P+$	a+ → aa*
QST	Zero-or-one repetition	$P?$	a? → a|ε
LWB	At-least-m repetition	$P\{m,\}$	a{2,} → aaa*
SNG	Exactly-n repetition	$P\{n\}$	a{3} → aaa
		Shorthand for a subset of Σ	
CCC	Custom character class	[*aeiou*]	[aeiou] → a|e|i|o|u
RNG	CCC with range	[*a-z*]	[a-c] → a|b|c
NCCC	Negated CCC	[ˆ*aeiou*]	[ˆa] → The disjunction over $\Sigma - a$
WSP	Built-in class: whitespace char.	\w	\w → \t||\t|\n|\r|\v

2.4.3 Relationship between K-regexes and E-regexes

Clearly E-regexes (Table 2.3) offer far more features than Kleene's regular expressions (Grammar 2.1). The features of E-regexes can be divided into three categories with respect to K-regexes.

First, K-regexes are a subset of E-regexes, as indicated in the topmost section of Table 2.3.

Second, some of the extended regex features can be trivially reduced to K-regexes. These *K-compatible* features do not increase the expressiveness of the string language that can be encoded using a regex, but rather permit more concise or readable notation. These features appear above the double-line in Table 2.3, in the sections labeled "Additional quantifiers" and "Shorthand". Sample reductions from regexes using K-compatible features to K-regexes are given in Table 2.6. The expansions of the quantifiers in the upper portion of Table 2.6 are particularly instructive; because the only way for an NFA to maintain memory is to move to an appropriate state, bounded repetition can only be encoded in an NFA by introducing additional states, once for each repetition of the sub-pattern. Nested bounded repetition will thus yield a geometric increase in states (i.e., a very large $|Q|$). This problem can be combated by departing from a pure NFA model, e.g., by introducing counters into the automaton model [79, 217].

In the third category (below the double line in Table 2.3) are the remaining extended regex features. Some extended regex features offer engineers expressiveness beyond traditional regular expressions (e.g., backreferences are NP-hard [62]), others are regular but exponentially concise (e.g., atomic groups [88]), and still others have not yet been formally studied. There is an active line of formalization work that provides specifications for such features and shows whether or not they are truly regular (i.e., expressible with an equivalent K-regex) [84, 85, 88, 107, 108, 162, 294].

2.4.4 Implementing PCRE semantics in a regex engine

The regex engines in mainstream programming languages support PCRE semantics using automaton simulation [132, 163]. They employ the algorithms given in §2.3.1. In particular, to support leftmost-greedy semantics in the context of various regex queries, they must guarantee not only that regexes match properly, but also that (1) The patterns of a disjunction are prioritized in left-to-right-order; and (2) The greedy/non-greedy behavior of each quantified sub-pattern operates properly. In addition, to support capturing sub-patterns, they must monitor the offsets at which the corresponding sub-string is matched. These requirements cannot be implemented using pure NFAs or DFAs, but can be modeled through minor extensions of automata theory.

To formally describe disjunction and greedy semantics, Berglund et al. have extended the notion of an NFA to a *prioritized NFA* [85]. The changes required in the automaton simulation are not complex. In a Spencer-style backtracking-based NFA simulation, edges are *prioritized* such that $\delta(q, \sigma)$ returns a list instead of a set. For example, the NFA in Figure 2.9 would be prioritized to consider the upward $q_1 \rightarrow q_2$ edge (first clause in the disjunction) before the downward $q_1 \rightarrow q_3$ edge (second clause). In a Thompson-style lockstep NFA simulation, all valid edges are processed in each step, and so the current state-set Φ_{curr} is ordered to prioritize the appropriate states [133]. In a DFA construction, the set of states in each multi-state would likewise be ordered.

To support capture queries, NFAs can be extended to Laurikari's notion of *tagged automata* [221]. This extension is orthogonal to the introduction of prioritization, and essentially amounts to performing bookkeeping operations each time the edges into and out of a capture group are traversed during automaton simulation.

The extended regex features (below the double-line in Table 2.3) are straightforward to support in a Spencer-style backtracking simulation, and can be implemented by extending the definition of an edge in the NFA. To implement a *zero-width assertion*, an ε-edge can be introduced that can only be taken if the associated test passes, e.g., for `\b` confirming that the preceding and subsequent characters are a mix of "word" or "non-word" characters. Lookaround assertions can be implemented using recursion, applying the regex engine on the associated sub-pattern. To implement a *backreference*, an NFA edge can be introduced that will (1) confirm that the contents of the corresponding captured group appear following the current index into the candidate string, and (2) advance the index that many characters. To implement *non-greedy repetition*, the prioritization of the corresponding back-edge is inverted from highest- to lowest-priority. To implement *atomic groups* and *possessive quantifiers*, which are the practical equivalent of *cuts* [86, 249], traversing an edge that leaves the corresponding sub-automaton results in an appropriate manipulation of the backtracking stack.

Some of these behaviors are difficult to implement under a Thompson-style NFA simulation. The authors of the production-grade Thompson-style regex engines (Google's RE2, and the engines used in Go and Rust) have chosen not to support any of these features. Backreferences are particularly problematic to support in a Thompson-style engine, because a backreference requires that the regex engine record the *path* through the automaton during simulation [63]. In Spencer's backtracking algorithm these paths are straightforward to track, because only one path is being considered at a time; in Thompson's lockstep algorithm, all paths are collapsed into the current state frontier to save space, and recording the distinct paths leading to the state frontier would require exponential space in the worst case.

2.4.5 (The failure of) Alternative regex semantics

Given its longstanding support in programming languages and systems utilities (§2.4.2), the PCRE syntax and semantics for regexes has acquired technical inertia. Alternative pattern matching semantics have not captured a significant share of the pattern matching market.

Given a regex and a candidate string, the leftmost-greedy rule is not the only semantic. The POSIX committee proposed "*leftmost-longest*" match semantics [201], which were adopted by the Boost regex engine [89]. Clarke and Cormack proposed "*shortest-match substring*" semantics, which they argue is more suitable for structured text such as is commonly encountered in HTML or XML documents [124]. The PCRE syntax and semantics are, however, fairly entrenched, and these alternative semantics have not been adopted by any mainstream programming language.

Researchers have also considered the problem of approximate matching, in which the outcome of a regex match is defined not as a Boolean condition but on a continuum of degree of fit. This problem arises in the context of text with errors, e.g., computational biology, signal processing, or information retrieval tasks [258]. Solutions support a range of pattern encodings, including single strings [320, 340], sets of strings [254], and Kleene regular expressions [255, 257, 340]. Despite the potential utility of approximate pattern matching, it has not seen widespread adoption in software engineering practice.

I believe the primary lesson to be learned from these lines of work is that, as a string matching tool for software engineering, regexes are here to stay. Regex-based string matching solutions have been estimated to occur in over 40% of software projects (§2.6.2), regexes are widely used by professional software engineers (§2.6.1), and regexes are supported in all mainstream programming languages. Studies of alternative matching problems and semantics have their applications, but the software engineering community appears to have settled on E-regexes-style regexes as one of their tools of choice.

2.5 Regular expression denial of service (ReDoS)

We have now covered the necessary background to define and discuss Regular expression Denial of Service (ReDoS). In this section I will define the conditions for a ReDoS vulnerability (§2.5.1), summarize the analyses that identify regexes with super-linear worst-case behavior (§2.5.2), and describe researchers' preliminary findings on the incidence of ReDoS in practice (§2.6.3).

2.5.1 Conditions for a ReDoS vulnerability

Software engineers who wish to use a regular expression in their computer programs will typically turn to a regex engine. For example, an engineer working in JavaScript might use the regexes built into the language specification. These regexes will be evaluated using the regex engine of the JavaScript interpreter — e.g., in Google's V8 JavaScript runtime, their regexes would be evaluated by the Irregex regex engine [131]. When it evaluates a regex match, a regex engine employs one or more of the algorithms described in §2.3. Unfortunately, some regex engines rely on algorithms that expose their users to ReDoS vulnerabilities.

In §2.3.1 I described two regex matching algorithms that exhibit best-case linear performance but worst-case exponential performance in either the length of the pattern or the candidate string (Table 2.2). If a regex engine is implemented with one of these algorithms, an application that relies on it may be exposed to a denial of service attack [60, 247] called ReDoS [135, 290], which is a type of *algorithmic complexity attack* [136]. A ReDoS attack requires four conditions:

1. **Multi-client service**: The victim operates a service that handles requests from multiple clients, permitting a malicious client to impact the experience of other clients.
2. **Super-linear regex engine**: The victim application uses a regex engine for which certain regex matches exhibit super-linear time complexity.
3. **Super-linear regex on untrusted input**: The victim application uses a regex that exhibits super-linear worst-case behavior in its regex engine, and the candidate strings on which the regex is used are not adequately sanitized.
4. **No safeguards**: The victim does not have appropriate safeguards in place to cap a client's resource usage.

If these conditions are met, then an attacker can submit input designed to trigger the super-linear regex behavior. Because the service has insufficient safeguards, the super-linear regex evaluation will divert resources from legitimate clients to this malicious one. If the malicious client can divert enough resources, they can reduce the quality of service offered to legitimate clients. At the extreme, the attacker diverts enough resources to completely deny service to legitimate clients.

ReDoS Conditions 1 and 2 are commonly met. Prior research has shown that engineers commonly use regexes to parse user input as part of a web service [115, 238] (ReDoS Condition 1). As Cox observed anecdotally [132] and we show empirically in Chapter 5, many of the regex engines built into programming languages use a Spencer-style backtracking algorithm to perform regex matches (ReDoS Condition 2).

The remaining conditions to be considered are ReDoS Condition 3 (whether the regexes are super-linear), and ReDoS Condition 4 (whether there are safeguards in place). In the remainder of this section I will discuss techniques to evaluate the super-linearity of a regex (Condition 3). The existence of safeguards varies by application context. In Chapter 9 we discuss the lack of a safeguard in many web applications.

2.5.2 Identifying super-linear regexes

As noted in §2.5.1, many of the regex engines built into mainstream programming languages perform a Spencer-style backtracking NFA simulation to answer regex queries [132]. To understand the potential for ReDoS Condition 3 to be met, engineers must determine whether their user-facing regexes could exhibit worst-case super-linear behavior in a Spencer-style backtracking NFA simulation. Because the phenomenon involves a substantial amount of backtracking through the search space, this super-linear behavior is known colloquially as *catastrophic backtracking* [163]. Despite the maturity of regex theory, the conditions for a *super-linear regex*, i.e., one for which a Spencer-style backtracking regex evaluation may exhibit worst-case performance that is polynomial or exponential the length of the input string, have only recently been formalized.

One of the necessary conditions for a regex to exhibit worst-case super-linear behavior is that it be ambiguous. In §2.5.2.1 I define regex ambiguity and provide examples. Then I consider the various means of identifying worst-case super-linear regexes. Researchers have investigated this problem using both dynamic (§2.5.2.2) and theoretical (§2.5.2.3) analyses. These approaches come with the typical trade-offs for their species. The dynamic analyses have no fidelity issues, but struggle to explore the full input space. On the other hand, the theoretical analyses offer proofs of correctness, but only insofar as their model of a regex engine squares with reality. Practitioners typically do not apply formal analyses, but instead follow heuristics to identify super-linear regexes (§2.5.2.4).

2.5.2.1 Ambiguity of regular expressions

The *ambiguity* of a regular expression has been described in several equivalent ways. In syntactic terms, a regular expression is unambiguous if there is exactly one parse tree for every string in its language [97, 317]. In terms of the behavior of its equivalent automaton, a regular expression is unambiguous if there is exactly one way in which it matches every string in its language [92]. Conversely, if a regular expression is *ambiguous*, then there are strings that occur in its language in more than one way. The definition of ambiguity in regular expressions is analogous to that of ambiguity in a context-free grammar [98, 151].

The ambiguity of a regular expression can be determined using direct analysis of the language of the expression [97, 317] or through analysis of the corresponding automaton [69, 92, 119]. The *degree of ambiguity* of a regular expression can be expressed in terms of the maximum number of distinct derivations for any string in the language of the regular expression. In terms of the corresponding NFA, the degree of ambiguity is the maximum number of distinct accepting paths through the automaton for any string in its language. Some regular expressions are unambiguous, while others have ambiguity that is fixed, polynomial, or exponential in the length of the string [69, 310]. I will show examples of each type.

Unambiguous regular expressions Informally speaking, unambiguous regular expressions have the property that for every string in its language, each character can only be mapped to one component of the expression. In terms of a backtracking NFA simulation, a regular expression is unambiguous if, during simulation on a string, non-deterministic choices never reach the same point. The regular expression $R_1 = a$ with $L(R_1) = \{a\}$ is unambiguous, as are the regular expressions $R_2 = ab^*c$ with $L(R_2) = \{ab^kc \mid k \geq 0\}$, and $R_3 = a(b|c)$ with $L(R_3) = \{ab, ac\}$. The automaton representation of each of these regexes is deterministic (i.e., a DFA), or can easily be made so through the treatment of ε-edges. However, an unambiguous regular expression may still result in a non-deterministic finite automaton representation (i.e., an NFA), e.g., $R_4 = (ab|ac)$ with $L(R_4) = \{ab, ac\}$.[16] In this example, the candidate string 'ad' would require an NFA simulation to traverse two different branches before a mismatch is determined. However, there is no suffix that would permit the branches to join again, and so the regular expression remains unambiguous.

Finitely-ambiguous regular expressions Some regular expressions are k-ambiguous — they have a maximum degree of ambiguity that results from the structure of the expression, and that is independent of the input string. For example, no matter the length of the input string, the regular expression $R_5 = (a|a)$ with $L(R_5) = \{a\}$ is 2-ambiguous. The string "a" is in the language of both halves of the disjunction, and so an input string can be parsed in 2 ways, but no more. The degree of ambiguity may still be quite large as a function of the length $|R|$ of the regular expression. For example, Figure 2.10 illustrates the automaton corresponding to an $\mathcal{O}(2^{|R|})$-ambiguous regular expression $R_6 = (a|a)(b|b)(c|c)$. As another example, a regex of the form $R_7 =$ `/a?a?a?a?a?.../` is $\binom{|R|}{|R|/2}$-ambiguous. To see this, consider the number of ways to parse the input string $w = a^{\frac{|R|}{2}}$. Each of the k a's can be matched to any of the $|R|$ slots, left to right.

[16]Cox distinguishes between R_3 (whose "NFA" is deterministic) and R_4 using the term "one-pass NFA" [133].

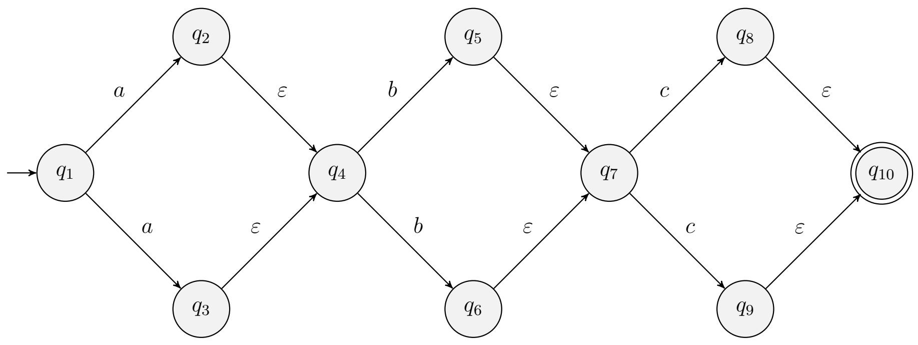


Figure 2.10: ***A regular expression with finite ambiguity.*** *This figure shows the finitely ambiguous NFA produced by Thompson's construction for the regular expression $R = /(a|a)(b|b)(c|c)/$. The string "abc" is in $L(R)$, and can be parsed eight ways — there are two ways for it to reach q_4, four ways to reach q_7, and eight ways to reach q_{10}.*

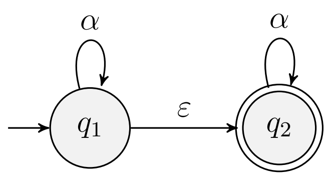
Infinitely-ambiguous regular expressions By applying unbounded repetition to finitely ambiguous structures, regular expressions can become *infinitely ambiguous*. If a regular expression is infinitely ambiguous, then there are infinitely many ambiguous strings in the language of the regular expression, and the degree of ambiguity is a function of the length of these ambiguous strings. The degree of ambiguity can be either polynomial or exponential in the length of the input string.

There are two ambiguous structures in an NFA that can lead to infinite ambiguity. These structures are depicted generically in Figure 2.11a (polynomial ambiguity) and Figure 2.11b (exponential ambiguity), with details in the figure captions. In the language of Wüstholz et al. [341], in each figure the node labeled q_1 is a *pivot node* because from this node an NFA simulation can "pivot" along one path or the other.

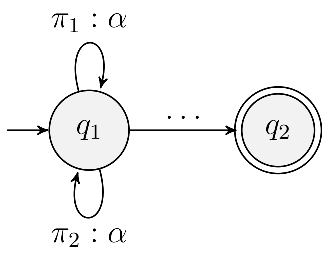
This dissertation is largely concerned with ReDoS through infinitely ambiguous regular expressions. Although finitely-ambiguous regular expressions may have an exponential degree of ambiguity in $|R|$ [132], I believe such regexes are uncommon in practice. By comparison, regular expressions with infinite ambiguity are trivial to construct and are commonly used in practice. As a result, most chapters only consider regexes with infinite ambiguity. Only Chapter 8 treats both finite and infinite ambiguity.

2.5.2.2 Detecting super-linear regexes using dynamic analyses

Several researchers have evaluated dynamic analyses to identify a regex's worst-case match complexity. Given a regex, these analyses evaluate its behavior in the regex engine of interest on a variety of inputs. Their goal is to identify inputs that take a particularly long time to evaluate, and from this to infer the regex's worst-case match complexity. Sullivan proposed a classical fuzzing approach in the manner of Miller et al. [245], re-purposing a Microsoft product's regex input generator to test a regex's match time on a range of randomly-generated



(a) This NFA has a simple polynomially-ambiguous structure. Two nodes q_1 and q_2 both have an unbounded quantifier, both consume the string α, and are connected by an ε-edge. As an example, the regex /a ∗ a ∗ / would have such a structure. Beginning from q_1 on the string $w = \alpha^k$, the automaton can advance from q_1 to q_2 after consuming any prefix α^i, $0 \le i \le k$. There are thus $k = \Theta(|w|)$ distinct accepting paths to q_2. By extending this automaton with additional similar states (e.g., $R = /a{*}a{*}a{*}/$), the number of paths increases geometrically in $|k|$.



(b) This NFA has exponentially-ambiguous structure. As an example, the regex /(a|a) ∗ / would have such a structure, as would the regex /(a∗) ∗ /. Beginning from q_1 on the string $w = \alpha^k$, the automaton can return to q_1 along either path π_1 or path π_2. Thus, if w is accepted, then there will be $|2^k| = \Theta(2^{|w|})$ distinct accepting paths to q_2.

Figure 2.11: ***Regular expressions with infinite ambiguity.*** *Regular expressions with infinite ambiguity.*

input strings [314, 315]. Petsios et al. [279] and Shen et al. [297] both proposed genetic search algorithms to discover particularly expensive input strings for a given regex, with fitness measured by the number of instructions performed by the regex engine during evaluation. Due to the enormous string search space, these dynamic approaches are not able to identify regexes with quadratic super-linear behavior. They are more successful at identifying regexes with highly super-linear worst-case behavior, viz. exponential behavior and polynomials that are cubic or higher. They can also identify regexes with large finite ambiguity, e.g., the regex from Figure 2.10.

2.5.2.3 Detecting super-linear regexes using theoretical analyses

Other researchers have pursued static analyses of a regular expression to determine whether it could exhibit worst-case super-linear behavior. These analyses are based on the model of a backtracking regex engine as described in §2.3.1.2, and define the necessary and sufficient conditions for super-linear matching in this model: (1) that the regex have an infinitely ambiguous sub-pattern $R_{super-linear} = P_{prefix} \cdot P_{ambig} \cdot P_{suffix}$; and (2) that there be some strings $w_{super-linear} = \alpha\beta^k\gamma$ such that $\alpha \in L(P_{prefix})$, β^k is ambiguous in $L(P_{ambig})$, and $\gamma \notin L(P_{suffix})$, so that $w_{super-linear} \notin L(R)$. If these conditions are met, then a backtracking search will traverse the sub-string β^k as many times as the sub-pattern is ambiguous. For example, in the case of the NFA depicted in Figure 2.11a, the backtracking search will traverse the input $w = a^k$ once for each of the $k = \Theta(|w|)$ paths through the string, at a cost of $|w| * \Theta(|w|) = \Theta(|w|^2)$.

Researchers have considered several theoretical approaches by which to identify regexes that have worst-case super-linear matching behavior. Sugiyama et al. [313] and Kirrage et al. [209] both considered an abstract model of the evaluation tree of a regex. Others have modeled the regex engine's behavior using NFAs [284, 335, 336, 341] and reduced the problem to graph reachability. Some analyses identify solely regexes with exponential worst-case complexity [87, 209, 284], and others will identify any super-linear behavior, whether polynomial or exponential [313, 335, 336, 341].

All of these analyses are equivalent to searching for paths through the NFA derived from the regex. In keeping with the necessary and sufficient conditions defined above, the desired paths have three components: a *prefix* to reach an ambiguous sub-automaton; a *pump string* exploiting the NFA's ambiguity that can be repeated to increase the amount of backtracking in the NFA simulation; and a *suffix* that will cause a mismatch.[17] For example, on the regex `/^b(a+)+$/`, the prefix would be "b", the pump would be "a" (each "a" increases the backtracking by a factor of two), and a suffix would be any other character (to trigger backtracking).

Many of these analyses include proofs of soundness and completeness for regexes that are written in the regex language they consider and that are evaluated under their model of a regex engine. However, they still have sources of false positives and false negatives for the E-regexes that are supported in most regex engines. False negatives may arise for two reasons: (1) they consider only the behavior of K-regexes and K-compatible E-regexes, ignoring super-linearity that may arise from the use of extended regex features like backreferences; and (2) they consider only regexes with infinite ambiguity, ignoring large finitely ambiguous regexes, e.g., the regex from Figure 2.10. False positives may arise where their models are inconsistent with real regex engine implementations, e.g., ignoring prioritization [209] or not considering the impact of optimizations like Aho-Corasick [64], Boyer-Moore [96], and Knuth-Morris-Pratt [212].[18] Lastly, these analyses ignore the effect of *flags* on the semantics of a regex match, which can affect properties like case sensitivity, the character set, and the interpretation of certain operators. Flags can lead to both false positives and false negatives for these analyses.

[17] Some analyses consider richer PCRE semantics, and show that super-linear behavior can manifest even when the candidate string is in the language of the regex, e.g., when non-greedy quantifiers are used to de-prioritize the accepting path [336].

[18] These techniques are used to short-circuit evaluations or identify feasible starting and stopping points based on regex sub-patterns that are sequences of symbols. For example, any string that can match the regex $/(c+)+d/$ must contain at least one 'd', and the only viable starting points for a search can be identified by working backwards from each instance of a 'd' in the candidate string.

2.5.2.4 Regex ambiguity anti-patterns

Several regex reference texts make informal recommendations to avoid super-linear behavior [156, 177, 179]. They suggest that software engineers should avoid nested quantifiers ("*star height*") [163, 177], and more generally that software engineers should "*watch out when…[different] parts of the [regex] can match the same text*" [179]. These recommendations are an imprecise warning against writing regexes that contain ambiguity, and therefore we will refer to these anti-patterns as *ambiguity anti-patterns*. We evaluate this advice in Chapter 6.

If an engineer cannot avoid ambiguity, these texts suggest that it could be refactored using atomic grouping or possessive quantifiers. Such advice is applicable in the languages that support either of these features — Java, Perl, .NET, Ruby, and PHP [5, 47]. The viability of such refactorings has not yet been studied scientifically.

2.6 Research on the use of regexes in practice

The previous sections described the mathematical theory behind regexes (K-regexes) §2.2, gave common algorithms for language membership testing §2.3, stated their syntax and semantics in practice (E-regexes) §2.4, and discussed the risk of ReDoS when super-linear regexes are deployed in user-facing contexts §2.5. This section summarizes the empirical research literature on regexes from a software engineering perspective.

2.6.1 Qualitative research on the use of regexes

Several research teams have examined the ways in which software engineers use regexes in practice. At a high level, these works have told us three things: (1) That regexes are commonly used by practitioners; (2) That regexes are frequently used in critical places, e.g., input sanitization; and (3) That many engineers struggle to use regexes correctly.

To the best of my knowledge, Singer et al. were the first to document the use of regexes in practice. During an ethnographic study, they reported that software engineers spent much of their time on program comprehension tasks, and that engineers commonly applied regex tools like grep to search for relevant sections [299, 300]. Their findings align with Goebelbecker's popular introduction to regexes, which describes searching files as a common application [174].

Chapman and Stolee performed the next studies of regex use in practice [115]. In 2016, they surveyed 18 software engineers at a small software firm to understand their use of regexes. The average respondent wrote regexes frequently, typically at least weekly. They used regexes in a range of contexts: within text editors, on command lines, as part of an SQL query, and as part of various computer programs. These engineers commonly reported using regexes for parsing user input, searching files, and summarizing data (e.g., counting lines that match a pattern). With Wang, Chapman and Stolee later studied the comprehensibility of regexes [116], identifying more and less comprehensible regex synonyms (e.g., `/a|b/` and `/[ab]/`) with an eye towards determining regex anti-patterns [213].

The most recent studies of regex use in practice were conducted by Michael [237] and Donohue [149], summarized together in [238].[19] These studies surveyed a combined 279 software engineers from many companies, and interviewed 17 of them for further insight. They reported that regexes are hard, and that engineers find search, validation, and documentation tasks particularly difficult.

None of these qualitative studies have included a comparison of regexes to other tools, whether specifically for pattern matching (e.g., string functions or parsing expression grammars [236]) or more broadly to software engineering tools, e.g., compilers, query languages, version control systems, or collaboration tools. Although learning of engineering difficulties with regexes is valuable, it would be fruitful to understand their place in a hierarchy of difficult software engineering tasks.

2.6.2 Quantitative research on the use of regexes

Although qualitative research has told us about how regexes are used in practice, we know relatively little about regex practices in the wild. On a small scale — a few thousand websites or applications — researchers have reported on regex repetition practices, the most commonly used regex features, and the existence of relatively synonymous regexes (overlapping languages).

In their 2010 work, Hodován et al. reported that many websites use the same (JavaScript) regexes, commonly for identifying web browsers via their user-agent strings [191]. They collected a ten-year longitudinal sample of regexes from 100 websites, extracting about $200,000$ regexes ($5,000$ unique) over 10 years. I conjecture that the high degree of repetition they report might be due to widespread adoption of the same client-side JavaScript libraries, or perhaps due to infrequent regex evolution [332].

Chapman and Stolee examined regex usage in $4,000$ Python projects [115]. They found that regexes were widely used, appearing in about 40% of the projects. Analyzing their corpus $14,000$ distinct regexes, they reported on the relative popularity of the ~ 35 regex features in Python. About half of these regexes contained repetition constructs like $+$ (ADD) and $*$ (KLE), while only 0.5% contained rarer constructs like backreferences (BKR). Chapman et al. leveraged this corpus to guide their study of synonymous regex constructs [116].

Wang and Stolee went on to consider regex test practices, grounded in a corpus of $15,000$

[19] I helped conduct these studies.

unique regexes extracted from 1, 225 Java projects [332]. They reported that the test suites for these projects successfully covered the *lines* on which regexes occurred, but provided poor coverage of the regexes themselves. Depending on the coverage metric, the non-diverse input strings led to average regex coverage of only 59% or 29%. Troublingly, they also reported that the state-of-the-art input generation tool, Rex [328], did not prioritize generating diverse inputs and would thus not be a particularly useful tool for improving on the relatively poor regex test coverage. One shortcoming of this work was a lack of comparison between regex test practices and broader test practices. It would be helpful to know whether the projects were typically well tested using, say, line coverage, and that regexes were abnormally under-tested. Conversely, if the projects had poor test quality in general, it would be unsurprising that the regexes were also poorly tested.

Finally, Wang et al. further considered regex evolution, i.e., the way in which a regex in a software project changes over time. Building on the same set of Java projects as [332], Wang et al. reported that regexes do not typically evolve once they enter a software project's source code [333]. When they do evolve, regexes are typically *expanded* in two ways: to accept a wider variety of strings, and using a larger set of regex features than before.

Both qualitative data and these quantitative measurements suggest that regexes are widely used in practice. These quantitative measurements have provided regex corpora that could guide the development of future tools or regex engines. But given the relatively small scope of these quantitative studies, the generalizability of the findings as well as the characteristics of interesting populations of regexes remain unknown. From the characteristics of regexes in a few hundred or a few thousand projects in JavaScript, Python, and Java, we cannot safely generalize to, e.g., the population of regexes in Python projects or the population of regexes across many programming languages.

2.6.3 Studies of super-linear regexes and ReDoS in practice

Michael et al. found among 279 professional software engineers from many companies, all used regexes but only 38% were aware of ReDoS attacks [238]. As a result, it is unsurprising that many examples of super-linear regexes have been reported in small-scale studies.

While evaluating their super-linear regex detectors, several researchers have measured the number of super-linear regexes within a corpus of regexes used in practice. Sugiyama et al. found super-linear regexes in the five PHP projects they examined [313]. Rathnayake and Thielecke and Weideman et al. observed super-linear regexes in the Snort WAF rules and the RegExLib library [284, 335]. Shen et al. found hundreds of super-linear regexes in the Snort WAF rules, the RegExLib library, and Chapman and Stolee's Python regex corpus [297].

Rather than considering regexes in isolation, two works have considered regexes in their application context. Wüstholz et al. combined their super-linear regex analysis with taint analysis to establish that in 27 out of the 150 Java web applications they considered, super-linear regexes were evaluated on untrusted input [341]. From a black-box approach, Staicu et al. leveraged insights into modern software engineering practices in the Node.js community. They identified super-linear regexes in popular JavaScript libraries from the npm ecosystem, conjectured how they might be deployed within a Node.js application, and successfully exploited ReDoS vulnerabilities in 339 out of the 2,846 popular web services they attacked [307].

2.6.4 Tools for working with regexes

Given the value of regexes and the difficulty that engineers report when working with them, it is unsurprising that researchers have proposed a range of tools to support software engineers in regex engineering tasks. These tools consider regexes in two distinct modalities: the regex pattern itself, and its language (the set of strings it matches). Tools have been proposed to aid engineers under both of these interpretations of a regex, and a long line of work has considered automatic regex composition.

Some research on regex maintenance has focused on diagnosing issues within the regex pattern. Regexes are included in programming languages, but not incorporated into type systems. This omission can cause syntax errors to manifest late in the development cycle, at run-time instead of compile-time. Spishak et al. described a regex type system that can identify a range of errors earlier in the development cycle [306]. The approach of Beck et al. applies syntax highlighting concepts from general-purpose programming languages to regexes [80], and others have considered automaton-based visualizations [104]. For syntactically correct regexes, Casias et al. recently proposed early steps towards regex debugging assistance [110].

Other work focuses on understanding the set of strings that a regex will match. Some researchers have done so to support program analysis and testing, e.g., understanding the string constraints encoded in a regex in order to permit deeper path exploration [223, 227, 328]. Others have considered the regex's language from a testing perspective. These researchers hypothesized common regex errors, and proposed interactively probing developers to determine whether a regex exhibits an error [73, 219, 220]. For comprehension, Blackwell proposed a visualization intended to support regex comprehension based on the strings that it matches [91].

Rather than supporting regex maintenance tasks, many researchers have investigated the problem of automatic regex composition. Their goal has been to learn a regex pattern from examples, to induce an automaton (and/or its corresponding regex) from inputs that it accepts or rejects. This line of work has typically been as a stepping stone towards higher-order learning, targeting a relatively simple problem first. Researchers have induced automata from a variety of starting points, including positive examples [145], positive and negative examples [70], noisy (biological) examples [164], and examples plus additional guidance [76, 225]. Although these tools have not always been developed with an eye towards practical adoption by software engineers, this seems like a natural application of such work.

Since the empirical literature does not contain large-scale regex corpora, it is not clear whether these tools support engineers in solving the regex problems they actually face. To ensure that their tools to be relevant to practitioners, regex tool builders would benefit from evaluating their tools on a representative regex corpus. Many of these studies would also benefit from evaluation on a population of software engineers.

2.7 What we don't yet know

In this section I described the mathematical foundations of regexes (§2.2), the common algorithms for regex language membership testing (§2.3), and the syntax and semantics of regexes in software engineering practice (§2.4). Combining this background, in §2.5 I explained the risk of ReDoS: the denial-of-service implications of the conjunction of (1) software engineers' common use of regexes to process untrusted user input; and (2) the use of inefficient algorithms to perform regex matching. Despite recent exploratory studies (§2.6), there are still many gaps in scientific knowledge related to ReDoS. These gaps can be summarized into two categories:

- To what extent is ReDoS a problem in practice? (Part II)
- How viable are the various solutions to ReDoS? (Part III)

In Part II, I report that ReDoS is a widespread problem in practice. In Part III I systematically evaluate the gamut of solutions to ReDoS.

Part II

Is ReDoS a Problem in Practice?

Outline and summary

"If it ain't broke, don't fix it."

–*Anon.*

In §2.5 I defined ReDoS and discussed the existing research into this security vulnerability. These studies have been restricted either to evaluating the effectiveness of super-linear regex detectors on a small corpus of regexes [209, 284, 297, 313, 335], or to searching for exploitable ReDoS vulnerabilities in a handful of popular software projects [307, 341]. Although these studies have suggested that ReDoS may be a problem in practice, they have not been conducted with a broad enough scope to give a sense of how widespread the problem is. Without large-scale measurements, regex engine developers have little reason to concern themselves with ReDoS. In particular, we don't know:

- *ReDoS Condition 3*: The extent to which small-scale estimates of super-linear regex frequency will generalize to full software ecosystems; and
- *ReDoS Condition 2*: The extent to which real-world regex engines evaluate some regexes in super-linear time.

In this part of the dissertation, I will present my research into these questions. I begin with a series of ReDoS case studies (Chapter 3). This collection of anecdotes guides the design and interpretation of the experiments that follow. Next, Chapter 4 describes the results of my ecosystem-scale empirical study. I report that thousands of software modules are potentially exploitable in ReDoS attacks, including vulnerabilities in the core libraries of Python and Node.js. Chapter 5 shows that the findings of Chapter 4 generalize to new contexts: super-linear regexes are common in software written in many programming languages, and the regex engines in many programming languages commonly exhibit worst-case super-linear behavior.

Briefly put, up to 10% of real-world regexes exhibit super-linear worst-case behavior on most of the production regex engines we considered. This suggests that ReDoS is a threat to real software on a larger scale than might previously have been imagined. Motivated by these findings, in Part III I evaluate the solution space for ReDoS.

Chapter 3

Case studies of problematic super-linear regex behavior

3.1 Summary

Super-linear regex evaluations have led to several well-documented actual or potential service outages and performance issues. This section summarizes four such issues in the form of case studies. From these studies we can learn several lessons (§3.6): that regexes are used in many contexts, that regexes can be problematic in many contexts, and that even weakly super-linear regex behavior can cause outages in production.

The case studies of Cloudflare and Stack Overflow describe production service outages. These outages appear to have occurred accidentally, rather than being induced by a malicious actor. However, they illustrate the potential impact of a ReDoS attack in practice.

The performance characteristics of the regexes in each case study are summarized in Table 3.1.

Statement of Attribution The material presented in this section has not previously been published.

Table 3.1: ***Super-linear regexes in ReDoS case studies.*** *This table summarizes the super-linear regexes from the ReDoS case studies. The performance measurements are on a desktop-class machine using the typical Spencer-style regex engine used by Node.js (v12). If a match took more than 60 seconds, it was deemed too expensive to complete. For the Atom case study, the worst-case input corresponds to the minimal example from Listing 8.* TTM: *Time to match.*

Case study	**Worst-case behavior**	**Worst-case input**	**TTM 250 chars.**	**TTM 100K chars.**
MediaWiki (§3.2)	Attacker's choice	Varies	*Too expensive*	*Too expensive*
Cloudflare (§3.3)	Quartic	+ + ...+	14 seconds	*Too expensive*
Stack Overflow (§3.4)	Quadratic	!\\$t$\\$t$...\\$t$!	0 seconds	6.5 seconds
Atom (§3.5)	Exponential	ababab...!	*Too expensive*	*Too expensive*

Listing 5 Attacker-controlled regex pattern from MediaWiki vulnerability. This listing shows the PHP code surrounding the attacker-controlled regex pattern that led to MediaWiki CVE 2015-6736. The contents of the variables `value` and `possibility` are under the control of the attacker, and the `value` used as the regex pattern is neither quoted nor escaped. Thus the attacker can craft an arbitrary regex pattern and combine it with an input that triggers the regex's worst-case behavior.

```
// $value is unsanitized and is used as a regex
if ($value == $possibility
   || (preg_match( '`^' . $value . ' \(i\)$`i', $possibility))
   || (!$this->mCaseSensitive && preg_match( '`^' . $value . '$`i', $possibility)) )
{
      ...
}
```

3.2 CVE 2015-6736 at MediaWiki

MediaWiki is a software package that supports building scalable wiki-style websites [10]. This package is used by Wikipedia and the other Wikimedia Foundation websites [12]. In 2016 the MediaWiki engineering team disclosed a denial of service security vulnerability that was assigned CVE 2015-6736 [45].

The regex The code snippet at the root of this vulnerability is given in Listing 5. Because the attacker controls both the regex pattern and the input, they can craft a super-linear regex and accompany it with a worst-case input of their choice.

The deployment context MediaWiki supports extensions that a site administrator can use to customize their deployment [9]. These extensions include software that runs on the server side. The problematic code snippet used this regex as part of the Quiz extension [11]. The extension used a regex to determine whether an input (`possibility`) matched one of a set of prescribed values (`value`) in a case-insensitive manner. The regex match would be performed on the server side, so a malicious user could force a MediaWiki server to perform an effectively arbitrary amount of work to evaluate it.

The resolution The engineer Marius Hoch repaired this vulnerability by escaping regex metacharacters prior to evaluating the user input as a regex pattern [190].

Listing 6 Super-linear regex from Cloudflare outage.

```
regex = /(?:(?:\"|'|\]|\}|\\|\d|(?:nan|infinity|true|false|null|undefined|symbol|math)
 |\`|\-|\+)+[)]*;?((?:\s|-|~|!|{}|\||\||\+)*.*(?:.*=.*)))/
  // Minimal failing example (quartic): /a*b?c?a*a*a*=/
  // Minimal failing example (quadratic): /.+.+=/
```

3.3 July 2019 service outage at Cloudflare

Cloudflare is an Internet infrastructure company. Their primary offerings revolve around caching and delivering content for their customers, who own a reported 20 million web properties [4]. One industry expert estimates that they serve 5-10% of internet traffic [261], including services by Uber, OKCupid, and Peloton [167, 168]. In July 2019 they experienced a 27-minute service outage caused by a super-linear regex evaluation, in turn affecting the availability of their clients' services. The summary given here is based in part on their post mortem analysis [180].

The regex The regex that led to the Cloudflare outage is given in Listing 6. This regex has worst-case quartic behavior in Spencer-style engines due to the four adjacent loops that can all match a "+" character.[1] It will exhibit this behavior on any line containing a relatively short sequence of "+" characters that does not contain an "=", because the sequence of "+" characters can be parsed against the ambiguous sub-pattern in many ways. This regex will also exhibit weaker quadratic behavior on any line containing a long sequence of non-newline characters that does not contain an "=".

The deployment context Cloudflare used this regex as part of a Web Application Firewall (WAF) security rule. WAF rules are intended to filter malicious traffic in HTTP conversations, e.g., cross-site scripting (XSS) or SQL injection attacks [270]. This regex was intended to identify XSS attacks in network traffic, and was therefore being evaluated on untrusted input.

The outage Shortly after Cloudflare's engineers deployed this regex worldwide, the regex encountered network traffic that triggered its worst-case behavior. There was enough network traffic of this form that all of Cloudflare's servers that handled HTTP/HTTPS traffic began to spend the majority of their CPU cycles performing regex matches. For the next 27 minutes, Cloudflare's servers did not route traffic but rather matched regexes, leading to outages to their customers' services.

[1]The regex is quartic under full-match semantics, but quintic under partial-match semantics for the reasons discussed in §3.4.

The resolution After 27 minutes, Cloudflare's engineering team disabled the WAF rule suite, causing traffic to flow (albeit with a lower level of protection). Cloudflare's engineers then performed root cause analysis and rolled back the errant rule. Within the next month they stated that they planned to switch from the Lua regex engine to the RE2 or Rust regex engines, both of which guarantee better performance.

Remarks Surprisingly, the analysis from Cloudflare's engineering team actually understated its worst-case behavior. They described it as a quadratic regex rather than a quartic one. Their analysis focused on the `/.*.*=/` portion and omitted the two preceding groups that could also match a +. This regex can exhibit quadratic worst-case behavior under a different class of inputs, so it may be that the network traffic that triggered the super-linear behavior at Cloudflare caused the regex engine to exhibit quadratic behavior.[2] This seems likely, because any long packet that did not contain an "=" would trigger quadratic behavior, while only a packet containing a long sequence of "+" characters would trigger the quartic behavior.

3.4 July 2016 service outage at Stack Overflow

Stack Overflow is an online question and answer forum [22]. It is part of the Stack Exchange Network and focuses on supporting computing practitioners. In July 2016 they experienced a 34-minute service outage caused by a super-linear regex evaluation, preventing users from posing questions or answering them. In early 2020, Stack Overflow received about 10 million visits per day [58], so a rough estimate suggests that this outage affected over 200,000 possible visits.[3] The summary given here is based in part on their post mortem analysis [153].

The regex The regex that led to the Stack Overflow outage is given in Listing 7. This regex has worst-case quadratic behavior in Spencer-style engines due to the use of partial-match semantics on the right-hand operand of the disjunction. It will exhibit this behavior on any line containing a long sequence of whitespace surrounded by non-whitespace character(s).

The deployment context This regex was used on the Stack Overflow server side to clean (and reduce the size of) all posts prior to delivering them to users. Its specific purpose was to trim whitespace from the beginning and end of the lines of posts.

[2]Specifically, the inputs might trigger the first and third loop and end in a newline, or they might trigger the two "`DOTALL`" loops and end without a "=".

[3]Assuming that visits are distributed evenly throughout the day, $10M * \frac{34}{24*60} = 236,111$.

Listing 7 Super-linear regex from Stack Overflow outage. This listing shows the super-linear regex that led to the Stack Overflow outage. Note that disjunction has higher precedence than the anchors ^ and $, so this regex is correctly read "A string beginning with whitespace, or a string ending with whitespace."

```
regex = /^[\s\u200c]+|[\s\u200c]+$/
  // Minimal failing example: /\s+$/
```

The outage On July 20, 2016, post #384884433 appeared on the home page. At the time, this post contained a line with a sequence of 20,507 whitespace characters that did not end in whitespace [46], triggering the quadratic worst-case behavior for all visitors.[4] Once post #384884433 appeared on the home page, all visitors to Stack Overflow triggered about 200 milliseconds' worth of unanticipated compute-intensive regex matching on the server side. Given the popularity of the website, Stack Overflow's servers were presumably unable to sustain this load and serve requests for the home page. Although other pages might still have been accessible, Stack Overflow's load balancing system determined their servers' health by requesting them to provide the home page, and no backend server could serve requests for this page in a timely manner. Thus, the load balancing system declared each server unhealthy one by one. Eventually no servers were available to serve any client requests, for the home page or otherwise.

The resolution It took Stack Overflow's engineers 34 minutes to discover the outage, identify the problematic regex, and replace it with equivalent linear-time string functions. After the outage, their engineering team took ameliorative steps including auditing their regexes and their validation workflow for posts, and modifying how the health of their servers was measured.

3.5 Performance problem in the Atom editor

Atom is a popular text editor [2]. It was released as open-source software in 2014, and as of 2015 had about 300,000 monthly users. In June 2016, a user identified benign-looking file contents that caused a local Atom session to exhibit a 30-minute lag before handling a keystroke. The summary of the performance problem given here is based in part on an investigation by David Galbraith [165].

The regex The regex that caused the Atom performance issue is given in Listing 8. This regex has worst-case exponential behavior in Spencer-style engines due to the presence of

[4]For unclear reasons, the post has since been edited to remove the whitespace characters.

Listing 8 Super-linear regex from Atom performance issue. This listing shows the super-linear regex that caused the Atom performance issue.

```
regex = /^\s*[^\s()}]+(?<m>[^()]*\((?:\g<m>|[^()]*)\)[^()]*)*[^()]*\)[,]?$/
  // Simplified by Galbraith: /^([^()]*\(\)[^()]*)*\)$/
  // Minimal failing example: /^(a*ba*)*$/
  // Repaired by Galbraith (quadratic):
  //    /^\s*[^\s()}]+([^()]*\((?:\1[^()]*|[^()]*)\))*[^()]*\)[,]?$/
```

"book-end" identical quantified sub-groups within a quantified group.

For the "`Minimal failing example`" in Listing 8, consider the behavior of the NFA simulation on a candidate string of the form "*abab...!*". Once the simulation passes the first `/a*/` and reaches the second `/a*/`, the next *a* it sees can be consumed either in the second `/a*/` or the first one. Since both sub-patterns are optional, either consumption is valid, creating ambiguity. The subsequent *b* will always be consumed and the NFA simulation will then return to the second `/a*/`. The effect is that for every *ab*, there is a compounding of two paths to explore, leading to an exponential number of paths to try in the worst case. Additional *a*'s between each *b* would linearly increase the number of paths to explore in that "round", with each subsequent *b* incurring the compounding effect.

Based on this analysis, for the full regex the compounding would occur for each additional function call in a line. Any text between the "()" of each function call could be consumed in either of the "book-end" quantified groups, increasing the effect of the compounding.

The deployment context Atom used this regex to determine whether or not to reduce the level of indentation when a user entered a new line. The regex matches lines with unbalanced parentheses, specifically those with more ")" than "(". When Atom observed such a line, it would decrease the indentation by one level on the next line, supporting "pretty printing" of a coding convention within Google's Go programming language.

The performance problem At the time of Galbraith's report, this exponential behavior would be triggered any time that a user pressed "Enter" under the following circumstances: the cursor was at the end of a line of Go code that contained a large number of function calls and did not end in a closing parentheses.

The resolution Galbraith worked with the Atom development team to replace the regex with a semantically equivalent one that does not exhibit exponential worst-case behavior.

Remarks The exponential regex in Atom would not cause a service outage in typical use cases. Atom is usually used on a developer's machine, and so this performance problem would only affect users that had the relevant file contents. Were Atom offered as a service, this regex could be used by one user to deny access to others.

As in the Cloudflare case study, in this case study the worst-case behavior can be triggered by legitimate and typical-seeming input. In contrast, in the Stack Overflow case study the super-linear worst-case behavior would only be triggered by abnormal input. It had presumably been used without issue until the unfortunate appearance of post #384884433 on the home page.

The repaired regex introduced by Galbraith in his pull request [166] is still super-linear, but its worst-case performance was improved from exponential to quadratic. Galbraith's failure to produce a linear-time regex is perhaps unsurprising, given that he described never having heard of super-linear regex evaluations until after identifying that a regex was the cause of the performance problem [165]. However, in the context of a text editor, quadratic worst-case performance on a single line of code is acceptable. According to my measurements, under Galbraith's repaired regex a line of code would need to be about 2,000 characters long before its worst-case match time exceeded 5 milliseconds. This is well below the threshold of human perception [137]. Galbraith's improvement thus moves the input space that will trigger the performance issue, from the realm of function-heavy Go code to the realm of the esoteric.

3.6 Lessons learned

There are three lessons to be learned from these case studies.

First, worst-case behavior notwithstanding, it is apparent that regexes are a useful and flexible tool. These case studies showed regexes being used at many points in the computing stack, for diverse purposes. These software projects had successfully used regexes (including several of the problematic regexes) without an issue in the *common case*, and only experienced problems under exceptional circumstances. As a general rule, it appears that the typical space and time complexity of a Spencer-style backtracking regex engine is acceptable in production deployments.

Second, because of their utility, we see that regexes can be problematic in many contexts. In these case studies they caused performance and security issues while being applied: to test user input by the wiki platform MediaWiki; as part of packet inspection by the CDN Cloudflare; to reduce network costs by the discussion forum Stack Overflow; and while performing pretty-printing in the text editor Atom. Unless they rely on a linear-time regex engine, engineers should certainly not permit users to define their own regexes to be evaluated on the server side, and they should remain wary whenever they deploy a regex on untrusted input.

Third, even weakly super-linear regex behavior can be problematic. It is unsurprising that worst-case exponential behavior can cause performance problems (as in Atom) or security vulnerabilities (as at MediaWiki). But even low-polynomial behavior, e.g., the quadratic performance experienced at Cloudflare and Stack Overflow, can have serious repercussions.

Chapter 4

Measuring the use of super-linear regexes in practice

4.1 Summary

In Part II I observed that the risk of ReDoS in practice was unknown because no large-scale measurement study had been performed. This chapter summarizes the results of my ecosystem-scale study of super-linear regexes in the JavaScript and Python ecosystems.

Methodology In this work, my collaborators and I performed the first large-scale empirical study to understand the extent of super-linear regexes in practice as well as the mechanisms that could be used to identify them. We analyzed the ecosystems of two of the most popular programming languages to understand the incidence of super-linear regexes. Our study covers the Node.js (JavaScript) and Python core libraries, as well as 448,402 (over 50%) of the modules in the *npm* [56] and *pypi* [57] module registries. For each software project we studied, we statically extracted its regexes and applied an ensemble of super-linear regex detectors to determine whether they might be super-linear. We then evaluated the degree of super-linearity in real regex engines, and estimated the application domains in which they were employed.

Findings We found that super-linear regexes occur in practice and in prominent places: they appear in the core Node.js and Python libraries as well as in thousands of modules in the npm and pypi registries, including popular modules with millions of downloads per month. We found over 4,000 unique super-linear regexes across npm and pypi, covering a wide range of application domains. Furthermore, nearly 300 of these regexes are high-risk because they have exponential complexity. Super-linear regexes comprise approximately 1% of the unique regexes we examined. In Chapter 5, I will improve on the methods presented here to show that super-linear regexes are approximately an order of magnitude more frequent under partial-match semantics.

Statement of Attribution The material presented here is excerpted from a paper that I presented at ESEC/FSE 2018 [139].

4.2 Study design and research questions

Our goal in this study is to understand the extent of potential ReDoS vulnerabilities in practice. In particular, we focus our investigation on studying super-linear regexes, which can be exploited to cause ReDoS. We consider three research questions:

RQ1: How prevalent are super-linear regexes in practice?
RQ2: How strongly vulnerable are the super-linear regexes?
RQ3: Which application domains do super-linear regexes affect?

4.3 RQ1: How prevalent are super-linear regexes in practice?

In answering this research question, our goal is to obtain large-scale measurements of the incidence of super-linear regexes in practice.

4.3.1 Methodology

In brief, this is how we measured the incidence of super-linear regexes in the wild. We used static analysis to extract all the regexes used in the Node.js and Python core libraries as well as more than half of the modules in the npm (JavaScript) and pypi (Python) registries. We applied super-linear regex detectors to filter for potentially-super-linear regexes, and concluded with a dynamic validation phase to prove that a regex was actually vulnerable.

Software We chose to study the largest software ecosystem as measured by number of open-source projects, that associated with *JavaScript* [144]. To gauge the generality of our results, we also studied the *Python* software ecosystem.

The source code in software ecosystems can be divided into the language core ("platform"), 3rd-party libraries, and applications [229]. We studied the regexes used in two portions of these ecosystems: the language core and in 3rd-party libraries. While super-linear regexes in a language's core modules and in 3rd-party libraries are not always exploitable in ReDoS attacks (e.g., because an application might never call a vulnerable API), they have a much broader potential impact. In the worst case, all applications using a module could be affected, including both open-source and closed-source applications [286]. As a practical consideration, production applications are often closed-source. In contrast, in modern ecosystems the language core and 3rd-party libraries are generally open-source, and 3rd-party libraries are conveniently organized in a registry that tracks metadata like where to find the module's source code.

For each language's core, we tested each supported version. For 3rd-party libraries, we examined the master branch of every module listed in the npm and pypi registries that had a URL on which we could run `git clone`. We chose not to use the packaged version of modules provided by the registries because these are sometimes packed, minified, or otherwise obfuscated in ways that complicate analysis, attribution, and vulnerability reporting.

Extracting regexes After cloning each module, we statically extracted its regexes. We cloned the latest master branch, with no history to minimize the impact on the VCS hosting service. Then we scanned it for source code based on file extensions (`.js` or `.py`). We built an abstract syntax tree (AST) from each source file, using `babylon` [49] for JavaScript files and the Python AST API for Python files. Walking the ASTs, we identified every regex declaration and extracted the pattern, skipping any uses of dynamic patterns. Excluding these dynamic patterns means our results provide lower bounds on the number of super-linear regexes.

Identifying super-linear regexes After extracting the regexes used in each module under study, we created a mapping from unique patterns to the modules using them. We then analyzed these unique patterns.

Our super-linear regex identification process has a static detection phase and a dynamic validation phase. For the static *detection* phase, we queried an ensemble of three super-linear regex detectors: `rxxr2` [284], `regex-static-analysis` [335], and `rexploiter` [341].[1] These detectors use different algorithms to report whether or not a regex may exhibit super-linear behavior, and if so will recommend malign input to trigger it. Our static phase collects each detector's opinion and produces a summary. The detectors, most frequently `regex-static-analysis`, may consume excessive time or memory in making their decision, so we limited the detectors to 5 minutes and 1 GB of memory on each regex and discarded unanswered queries. These super-linear regex detectors are research prototypes, so they do not support all regex features nor guarantee correctness.

Our dynamic *validation* phase uses this summary to test the accuracy of each detector's prediction for the regex engine of the language of interest. The detectors follow different algorithms based on assumptions about the implementation of the regex engine, and these assumptions may or may not hold in each language of interest. To validate a detector's predicted malign input, our validator tests this malign input on the possibly-super-linear regex in small Node.js and Python applications we created.

This is how we identified *super-linear regexes*. To permit differentiating regexes by their degree of vulnerability (§4.4), we measured how long each regex took to match a sequence of malign inputs with varying numbers of pumps. We began with one pump and followed

[1]cf. §2.5.2. Sullivan's SDL regex fuzzer [315] is no longer available, Sugiyama et al. [313] did not publish their analysis tool, and Shen et al. [297] had not yet presented their work at the time of our study.

a geometric sequence with a factor of 1.1, rounding up. We tested 100 inputs, the last with 85,615 pumps, and marked the regex super-linear if the regex match took more than 10 seconds on a match, as this is far longer than a linear-time regex match would take. We stopped at 85,615 pumps for two reasons. First, this number was sufficient to cause super-linear complexity to manifest without being attributable to the overheads of enormous strings. Second, this many pumps results in malign inputs 100K-1M characters long, long enough to become potentially expensive for attackers to exploit. We distributed this analysis and ran multiple tests on each machine in parallel, dedicating one core to each test with `taskset` [42] to remove computational interference between co-located tests.

4.3.2 Results

We found that super-linear regexes are surprisingly common in practice. The Node.js and Python core libraries both contained super-linear regexes, and about 1% of all unique regexes in both npm and pypi were super-linear regexes. In all, 3% of npm modules and 1% of pypi modules contained at least one super-linear regex.

Language Core We found one super-linear regex in the core libraries of Node.js (server-side JavaScript). At the time of this study, the supported versions of Node.js were v4, v6, v8, and v9. We scanned the core libraries (`lib/`) of each of these versions. In v4 we identified and disclosed two super-linear regexes used to parse UNIX and Windows file paths. These regexes had been removed for performance reasons in v6 so the other versions of Node were not affected.[2] This vulnerability was published as CVE-2018-7158 and fixed by the Node.js core team.

We found three super-linear regexes in the core libraries of Python. At the time of this study, the supported versions of Python were v2 and v3. We scanned the core libraries (`Libs/`) of each of these versions. Both versions shared two super-linear regexes, one in `poplib` and one in `difflib`. We identified an additional vulnerability in the v2.7.14 `fpformat` library. These vulnerabilities were published as CVE-2018-1060 and CVE-2018-1061; we authored the patches.

Third-party modules Table 4.1 summarizes the results of our registry analysis. We were able to clone 66% of npm (375,652 modules) and 58% of pypi (72,750 modules). In this sample of each registry we found that about 1% of the unique regexes were super-linear regexes (3,589 in npm, and 704 in pypi).

Figure 4.1 summarizes two different distributions in the npm and pypi datasets using Cumulative Distribution Functions (CDFs). The dotted lines show the distribution of the number

[2]They were replaced with a custom parser that optimized matching for non-malicious input. The Node.js development team was not aware that these regexes could lead to ReDoS under malicious input.

Table 4.1: ***Summary of our measurements of super-linear regexes in the npm and pypi module registries.*** *Troublingly, 1% of unique regexes were super-linear regexes, affecting over 10,000 modules.*

Registry	Total modules	Scanned mod.	Unique regexes	Super-linear regexes	Affected mod.
npm	565,219	375,652 (66%)	349,852	3,589 (1%)	13,018 (3%)
pypi	126,304	72,750 (58%)	63,352	704 (1%)	705 (1%)

of unique regexes in each module. Consistent with prior small-scale empirical measurements [115, 341], observe that more than 30% of npm and pypi modules use at least one regex. The npm modules tend to contain more distinct regexes than the pypi modules do. The solid lines show the distribution of the number of modules in which each super-linear regex appears: in the npm registry some super-linear regexes appear in hundreds or thousands of modules, while in the pypi registry the most ubiquitous super-linear regexes are only used in about 50 modules. This suggests that regex re-use may be more common in JavaScript than in Python, something we investigate in a subsequent chapter.

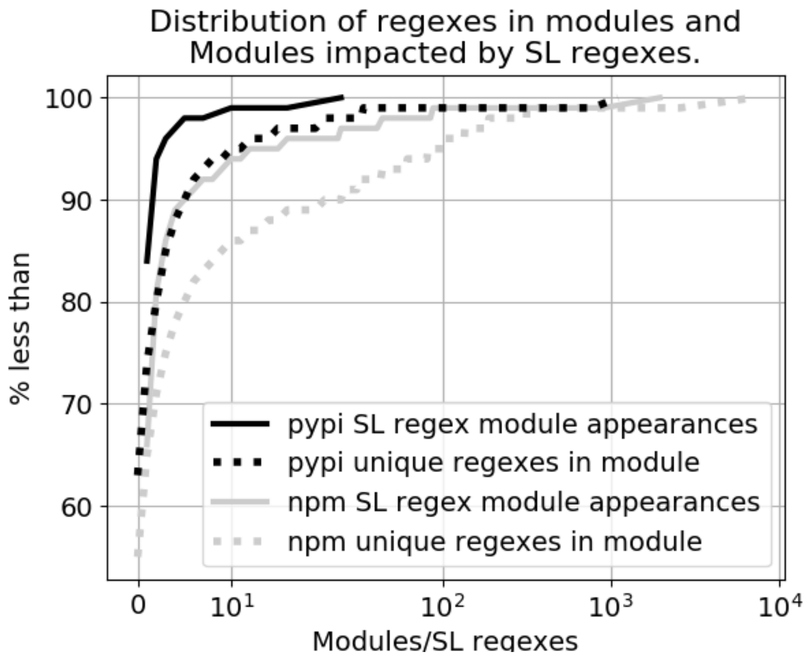


Figure 4.1: ***Distribution of regexes and super-linear regexes in npm and pypi modules.*** *This figure shows two CDFs. The dotted lines indicate the distribution of the number of unique regexes used in modules, while the solid lines show the distribution of the number of modules affected by super-linear regexes. Note the log scale on the x-axis.*

To give a sense of how impactful these super-linear regexes might be, for each module we obtained the popularity (registry downloads/month) and computed the project size based on the source files we scanned [55]. Modules with super-linear regexes are indicated in black

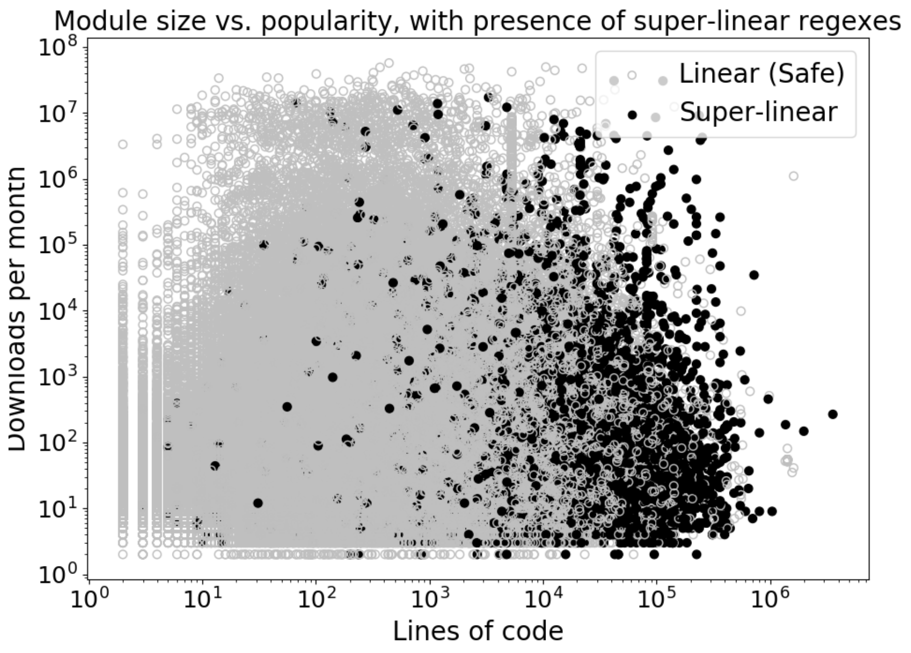


Figure 4.2: ***The code size and popularity of npm modules, marked if they contain super-linear regexes.*** *npm modules by size and popularity (log-log). The 13,018 modules with super-linear regexes are in black. Note the "trivial packages" on the left side [59].*

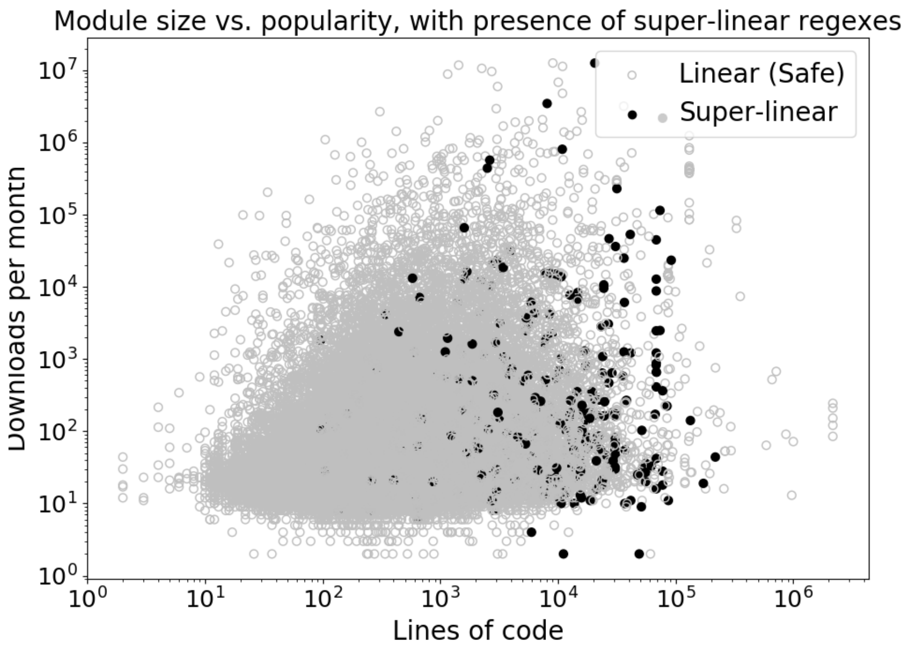


Figure 4.3: ***The code size and popularity of pypi modules, marked if they contain super-linear regexes.*** *pypi modules by size and popularity (log-log). The 705 modules with super-linear regexes are in black.*

in Figure 4.2 (npm) and Figure 4.3 (pypi). In both registries, larger modules are more likely to contain super-linear regexes, and super-linear regexes are slightly more common in modules with lower download rates.

4.4 RQ2: How strongly vulnerable are the super-linear regexes?

From an engineering perspective, some super-linear regexes are worse than others. Super-linear regexes whose super-linear behavior manifests on shorter malign inputs are of greater concern than those only affected by longer malign inputs. Longer malign inputs could be prevented by other parts of the software stack (e.g., limits on HTTP headers), while short malign inputs may only be prevented by modifications to the vulnerable software itself. In this section we refine our measure of super-linear regexes, differentiating between *exponential* and *polynomial* vulnerabilities.

4.4.1 Methodology

While the degree of vulnerability of a super-linear regex can be predicted by theoretical analyses, we are not confident of the accuracy of such predictions because they are based on a regex engine model that may not hold on production regex engines. Thus, we used curve fitting to differentiate between exponential and polynomial super-linear regexes, and between different degrees of the polynomial. As discussed in §4.3, our dynamic validation step tests the match time of the appropriate regex engine (JavaScript-V8 or Python) on a sequence of malign inputs with a geometrically increasing number of pumps. We measured the time that it took to compute each match. We then fit the time taken for different numbers of pumps against both exponential ($f(x) = ab^x$) and polynomial (power-law: $f(x) = ax^b$) curves, and chose the curve that provided the better fit by r^2 value. When the malign inputs from the different super-linear regex detectors resulted in different curves (e.g., the regex `/^a*a*(b*)*$/` has inputs that will induce quadratic or exponential worst-case behavior), we used the steepest, deadliest curve. As in §4.3, we distributed the work across multiple machines. As result, the multiplicative factors of the curves are not comparable, but the bases or exponents are.

This analysis allows us to create a hierarchy of vulnerabilities. Exponential super-linear regexes are more vulnerable than polynomial super-linear regexes, because the number of pumps (length of malign input) required to achieve noticeable delays is smaller. For the same reason, among polynomial super-linear regexes, those with larger b values are more vulnerable than those with smaller b values. The curve type and the b values influence the degree of vulnerability more strongly than the a values.

Table 4.2: ***Measured worst-case behavior of super-linear regexes.*** *This table shows the degree of vulnerabilities in the npm and pypi datasets. The polynomial vulnerabilities are further broken down by the degree of the polynomial, b, which we rounded to the nearest integer. This excluded some regexes whose polynomial degree rounded down to 1.*

Degree of vulnerability	**npm (3,589 vulns)**	**pypi (704 vulns)**
Exponential $O(2^n)$	245 (7%)	41 (6%)
Polynomial $\mathcal{O}(n^{b>4})$	100 (3%)	15 (2%)
Polynomial $\mathcal{O}(n^4)$	44 (1%)	5 (1%)
Polynomial $\mathcal{O}(n^3)$	535 (15%)	107 (15%)
Polynomial $\mathcal{O}(n^2)$	2,638 (74%)	534 (76%)

4.4.2 Results and Analysis

A breakdown of the regexes by their degree of vulnerability is in Table 4.2. Exponential super-linear regexes were rare in both registries: only 7% of the super-linear regexes from npm, and 6% of those from pypi were exponential. The majority of the super-linear regexes in both registries were polynomial, tending to $\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$. This finding has implications for super-linear regex detectors as well as for software developers.

Contribution of the ensemble members The contribution of each ReDoS detector is shown in Table 4.3. `regex-static-analysis` [335] was the most valuable detector in our ensemble. It was the only detector to identify 87% of the super-linear regexes in npm and 92% of the super-linear regexes in pypi. `rxxr2` found relatively few vulnerabilities because it only searches for regexes with exponential vulnerabilities, while the majority of super-linear regexes were polynomially vulnerable. Both `rexploiter` and `regex-static-analysis` were more ambitious, searching for both exponential and polynomial vulnerabilities. `rexploiter`'s low success rate may be due to a bug in its output generation which prevented us from automatically parsing some of its reports. Unfortunately, `rexploiter` is closed source and the authors would not share their source code with us, so we could not investigate further. `regex-static-analysis` initially had a similar bug. As it is open-source, we improved its success rate by patching this bug.

Implications for software engineering with regexes The super-linear behavior of polynomial regexes typically manifests for malign inputs on the order of many hundreds or thousands of characters long. Such strings are often longer than any legitimate strings, as is the case for strings with many of the semantic meanings listed in Table 4.4 (§4.5). Thus, rejecting too-long strings before testing them against a regex would be a cheap and effective defense approach and should be considered as a best practice when writing regexes. We

Table 4.3: ***Performance of each ReDoS detector in our ensemble.*** *This table estimates the areas of strength and value added by each ReDoS detector in our ensemble.* Exp *and* Poly *indicate the number of exponential and polynomial super-linear regexes found by this detector, respectively.* Solo *is the number of "solo" finds — regexes that only that detector correctly identified as super-linear.*

Detector	**npm**			**pypi**		
	Exp	**Poly**	**Solo**	**Exp**	**Poly**	**Solo**
`regex-static-analysis` [335]	150	3,227	3,122	25	667	649
`rexploiter` [341]	34	248	157	10	24	4
`rxxr2` [284]	165	107	35	27	19	6

measure the popularity of this defense in Chapter 6.

4.5 RQ3: Which application domains do super-linear regexes affect?

Regexes are used in a variety of application domains. From our own experience in writing regexes, and from a manual analysis of 400 regex uses in npm modules, we posit that developers often write regexes with one of the semantic meanings listed in Table 4.4. These semantic meanings may be of interest in some application domains but not others. For example, we conjecture that identifying source code or naming conventions is the domain of linters and compilers, that webservers are more interested in identifying HTML and user-agent strings, and that servers or scripts may be prepared to change their behavior based on the error messages that they encounter.

4.5.1 Methodology

In this section, we describe our techniques to automatically categorize regexes into these semantic groups. We began by manually labeling the semantic meaning of 400 regex usage examples based on inspection of the regex itself as well as how it was used in the project(s) in which we found it. Although some of the regexes we encountered were obscure and their purpose could only be identified by looking for comments and other clues in the surrounding source code, it became clear to us that many regexes with the semantic meanings listed in Table 4.4 could be automatically classified. There were 200 unique regexes among these 400 examples, and we found that the duplicated regexes were always used with the same semantic meaning in different modules.

We developed an automatic labeling scheme that uses a combination of parsing and "meta-

Table 4.4: ***Proposed common semantic meanings for regexes, with results from automatic labeling.*** *Proposed common semantic meanings for regexes. The examples are automatically-labeled (SL) regexes from our npm dataset. The last two columns are the number of regexes labeled with each semantic meaning in our npm and pypi datasets.*

Meaning	**Example**	**npm**	**pypi**
Error messages	`/no such file '.+[/\\](.+)'/`	22,197	881
File names	`/[a-zA-Z-0-9_\/-]+\.json/`	10,151	497
HTML	`/href="(.+\.css)(\?v=.+?)?"/`	8,786	2,504
URL	`/^.+:\/\/[^\n\\]+$/`	6,986	2,048
Naming convention	`/^[$_a-z]+[$_a-z0-9-]*$/`	4,096	1,056
Source code	`/function.*?\(.*?\)\s*\{\s*/`	3,941	105
User-agent strings	`/Chrome\/([\w\W]*?)\./`	3,135	124
Whitespace	`/(\n\s*)+$/`	2,016	441
Number	`/^(\d+|(\d*\.\d+))+$/`	762	238
Email	`/^\S+@\S+\.\w+$/`	444	97
Classification rate	—	18%	13%

regexes" to label regexes based on the proposed semantic meanings. For example, here is a simplified version of our meta-regex to label regexes as describing whitespace:

```
/^\^?(\\s|\\n|\\t| |[\|\*\+\[\]\(\)]|)+\$?$/
```

This simplified regex looks for a string (regex pattern) containing only whitespace characters, as well as meta-characters that might be used to anchor the pattern ('^' and '$') or to encode varying quantities of whitespace ('+', '*', etc.).

We iteratively improved our regex labeler. In each iteration, we labeled a randomly selected subset of 10,000-30,000 regexes from our npm regex dataset. We manually examined 100 of the regexes assigned to each semantic meaning. One or more representatives of any mis-labeled regexes were added to a test suite, and the iteration was complete once the regex labeler correctly identified all the regexes in the suite.

This process resulted in a precise regex labeler for regexes that are reasonably specific. As you might expect based on how we derived it, our labeler works well for "easy to classify" regexes that restrict the input to something close to the expected language. Our labeler's recall is unknown because it is difficult to know what semantic meaning a developer *intended* for a regex. An overly-permissive regex will not give many clues as to its intended language because it will capture its target language as well as a broader set of strings.

We refined our labeler through 17 iterations. At the conclusion of this process our test suite contained 358 regexes, and we were reasonably confident in its precision. We then applied it to our npm and pypi datasets. Irrespective of whether our list of semantic meanings for regexes is complete, it serves the goal of studying how different domains may be affected by ReDoS. We leave the search for a complete list of regex semantic meanings to future work.

4.5.2 Results

I will highlight two results from this experiment. First, as summarized in Table 4.4, we found regexes in all of these domains in both npm and pypi. Second, some semantic meanings are more prone to being expressed with super-linear regexes than others. As can be seen in Figure 4.4, developers should be cautious when writing regexes for emails, user-agent strings, source code, and HTML.

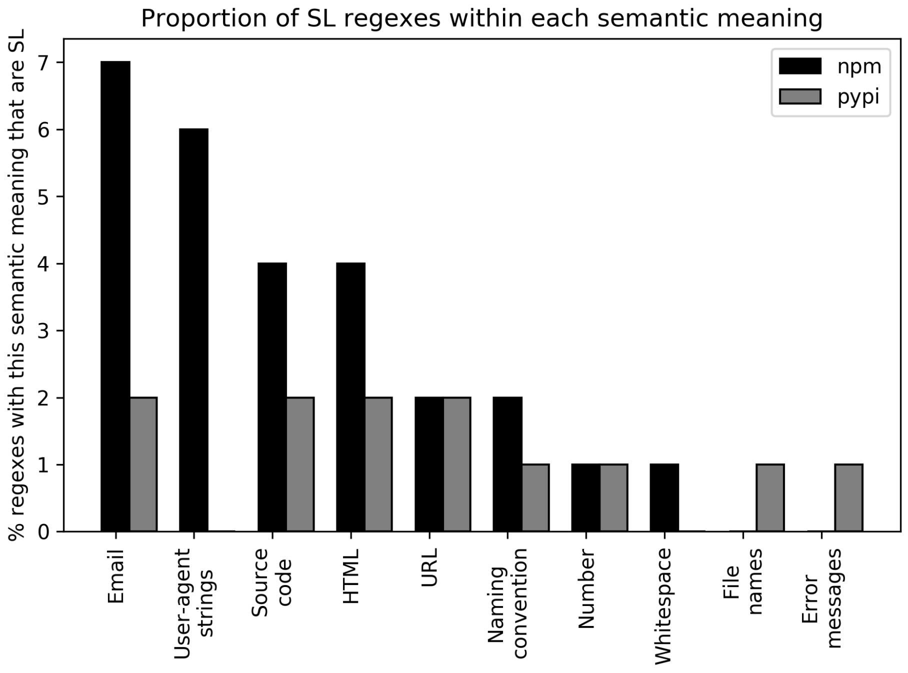


Figure 4.4: ***Proportion of super-linear regexes from the npm and pypi datasets that bear each semantic meaning.***

4.6 Discussion

Many regexes are super-linear In RQ1 we found that about 1% of unique regexes exhibit super-linear worst-case behavior; later we update this finding to a proportion of about 10%. This is a substantial fraction of regexes, and we believe the implication is that ReDoS vulnerabilities are surprisingly common in practice.

Most super-linear regexes are quadratic In RQ2 we showed that most of these super-linear regexes exhibit polynomial worst-case behavior, typically quadratic. Although this behavior is less concerning than exponential behavior or higher-order polynomials, several of the ReDoS case studies described in Chapter 3 involved quadratic polynomials. The risk of ReDoS should not be dismissed on these grounds.

Not all super-linear regexes are ReDoS vulnerabilities To be a ReDoS vulnerability, a super-linear regex must be used in an attacker-triggerable manner (ReDoS Condition 3 from §2.5.1). For example, super-linear regexes used in test suites to validate a program's output are unlikely exploit candidates. This study does not distinguish between regex usage contexts. However, smaller-scale studies have shown that many super-linear regexes are user-facing [341], which aligns with qualitative reports about engineering practices [115, 237].

4.7 Threats to validity

Internal Validity. In §4.6 we discussed the primary internal threat to the validity of this study — that not all super-linear regexes are ReDoS vulnerabilities

External Validity. A threat to external validity concerns whether our findings will hold for other ecosystems and scenarios. We partially addressed this threat by studying two popular programming languages with large ecosystems. As the general theme of our findings was consistent across these ecosystems, we expect our results to generalize to other ecosystems as well. Chapter 5 examines this threat in more detail.

In this study we did not investigate the proportion of regex *usages* that involve super-linear regexes, rather the proportion of *unique regexes* that are super-linear. Weighting our findings by regex popularity might appear to affect our results. Later in this work, however, it becomes clear that most regex usages involve unique regexes, so weighting usage would not affect our conclusions. An independent analysis of our data can be found elsewhere [205].

Construct Validity. In RQ1 we reported on the proportion of regexes that exhibit super-linear worst-case behavior. Our results may be affected by inaccuracies in the super-linear regex detectors. We address false positives by dynamically confirming the report from the super-linear regex detectors. But false negatives are also possible: we report only the super-linear regexes that can be detected by existing techniques (e.g., none of them considers the use of inherently super-linear features like backreferences). This means that the super-linear regexes we identified represent a *lower bound* on the number of such regexes in practice.

In RQ3, our method of labeling regex application domains was ad hoc, and we do not claim to have identified all application domains in which regexes could be applied. Our goal in studying RQ3 was to understand whether super-linear regexes appear across application domains, and whether different application domains are affected differently by them. Our precise but otherwise potentially incomplete set of application domains still allowed us to answer these questions in the affirmative.

Finally, a super-linear regex is only one of the criteria for a ReDoS attack (§2.5). This study is focused on identifying super-linear regexes, and did not confirm that they are exploitable. We did not perform taint analysis to confirm that malign input could reach these regexes, nor did we attempt to filter out modules used on the client side.

Chapter 5

Generalizing regex measurements

" There are more things in heaven and earth, Horatio, than are dreamt of in your philosophy "
–Hamlet

5.1 Summary

In Chapter 4 I conducted experiments to determine whether super-linear regexes were common enough in practice. The initial findings were troubling – super-linear regexes appear to be fairly common in practice, with denial-of-service implications via ReDoS. However, those experiments were restricted in two ways: (1) They did not consider the regexes that a program dynamically constructs; and (2) They considered regexes only two programming languages. This chapter summarizes the results of a many-language regex comparison to test the generalizability of those and other prior findings. If the findings from Chapter 4 generalize, then super-linear regexes (and ReDoS) may be a widespread problem that the web services community should address. If they do not generalize, then perhaps only smaller engineering communities need concern themselves.

Methodology This chapter describes the first large-scale many-language comparison of regexes. Generalizing the findings from existing research depends on validating two hypotheses: (1) Various regex extraction methodologies yield similar results; and (2) Regex characteristics are similar across programming languages. To test these hypotheses, we collected a corpus of regexes from software written in eight programming languages, and compared these regexes across eight regex metrics to capture the dimensions of regex representation, string language diversity, and worst-case match complexity.

Findings These two generalizability hypotheses generally held. Our findings support a nuanced notion of universal regex practices. In our first experiment, we show that the regex extraction methodology does not produce significantly different regex corpuses. In our second experiment, we found that the regexes from different programming languages are not significantly different on four of our eight metrics, and on the other metrics only a few languages are outliers. Because regexes appear to be similar across programming languages, we were able to replicate many findings from prior research in new programming languages on a larger regex corpus. Thus, our regex corpus is a reasonable approximation of the ways software engineers use regexes. Measurements on this corpus can guide data-driven designs of a new generation of regex tools and regex engines. We apply the corpus for this purpose in Chapter 8.

Statement of Attribution The material presented here draws primarily from work presented at ASE 2019 [142]. Where noted, relevant material is excerpted from work presented at ESEC/FSE 2019 [141].

5.2 Motivation

The better we understand how and why software engineers use regexes, the better tools we can build to support them. This dissertation is specifically focused on the ReDoS problem, and there are many other opportunities for support as well. To guide this endeavor, empirical regex researchers have sought to understand the characteristics of real-world regexes. As described in §2.6, the efforts of these researchers have provided many hints about how engineers use regexes in practice. Regexes are *widely used*, reportedly appearing in 30–40% of software projects with applications like input sanitization, error checking, document rendering, linting, and unit testing [115, 139, 332]. Software engineers may rely more heavily on some *regex features* than others, possibly tied to the relative comprehensibility of different features [116]. Features like quantifiers, capture groups, and character classes are commonly used in Python, while backreferences and lookaround assertions rarely appear in practical regexes [115]. Engineers may *under-test* their regexes, perhaps relying on line coverage instead of automaton graph coverage [332]. Most regexes may go unmodified after entering version control [333]. And many prominent software modules and web services rely on *super-linear regexes* and are vulnerable to ReDoS [139, 297, 341].

If these preliminary empirical regex findings generalize, they can guide research into more fruitful directions and nip others in the bud [102]. For example, if regexes are as widely used as is thought, then visualization and input generation tools can be valuable aids for many developers. And if super-linear worst-case time complexity is as common as has been estimated, then addressing this behavior by overhauling regex engines seems natural. Conversely, if regexes do not change after entering version control [333], then regex-specific differencing tools (e.g., for code review) may not have great utility. And if non-regular regex extensions like backreferences and lookaround assertions are as rare universally as initial results suggest, then they should be a low priority for tool support and regex engine optimizations.

Empirical regex research depends on two generalizability hypotheses. Generalizing this research will permit us to guide the design of future research prototypes and engineering systems. As summarized in Table 5.1, the corpuses used in prior empirical regex research were

Table 5.1: ***Comparison of existing regex corpuses by extraction method, programming language, and scale.*** *This table compares existing regex corpuses by extraction method, programming language, and scale. No comparison has been made between extraction methods (static analysis vs. program instrumentation). Regex characteristics have been studied in only three programming languages (Python, JavaScript, and Java).*

Corpus	Extraction method	Languages	# Projects
[297]	Static analysis	Python, JavaScript	50
[341]	Static analysis	Java	150
[115]	Static analysis	Python	4K
[139] (Chapter 4)	Static analysis	JavaScript, Python	375K, 72K
[332]	Program instrumentation	Java	1.2K
This work	Validated static analysis	Eight languages	200K

created using one of two regex extraction methodologies, and cover only three programming languages.

Comparing regex extraction methodologies When a software engineer matches a string against a regex, they must specify the regex pattern and construct a Regex object. The regex pattern can be provided as a static string to the Regex constructor. Or the developer might wish to supply a variable string, e.g., to build a complex regex by concatenating its constituent parts. Rasool and Asif suggest that this practice of regex templating, which they call "abstract regexes," may make regexes easier to debug [283]. For a real-world example, the widely-used `marked` Markdown parser relies heavily on regex templating.[1] We illustrate these concepts in Listing 9.

Regex corpuses have been constructed using either static analysis or program instrumentation (Table 5.1). These approaches have familiar tradeoffs. Using static analysis, researchers can analyze an entire software project, but may not be able to extract dynamically defined regex patterns like those in Listing 9 without intra- and inter-function dataflow analysis. In contrast, runtime analysis can extract both statically and dynamically defined regex patterns so long as the relevant call sites are evaluated during execution. It is not clear whether a regex corpus based on one extraction methodology would be comparable to a regex corpus based on the other.

Regex variation by language? Regex research has provided hints about how engineers use regexes in practice, but these works have been isolated to practices in three programming languages. Engineers may choose a programming language based in part on their task [268, 280] ("the right tool for the job"), and some tasks may have greater call for pattern matching.

[1]See `https://github.com/markedjs/marked`.

Listing 9 Code snippet illustrating regexes more and less amenable to different extraction methodologies. Regex corpuses based on static analysis or program instrumentation may yield different results. For example, the regex used to match the `emailStr` is of varying complexity depending on the `regexType` flag. Static analysis might only be able to retrieve the simplest regex pattern, while an instrumented application or runtime might identify all three patterns if the software can be exercised thoroughly.

```
def isEmail(emailStr, regexType, externalRegex):
  if regexType == "SIMPLE_REGEX":
    reg = Regex(".+@.+")
  elif regexType == "COMPLEX_REGEX":
    NAME_REGEX = "[a-z0-9]+"
    DOMAIN_REGEX = "[a-z0-9]+(\.[a-z0-9]+)+"
    regex = NAME_REGEX + "@" + DOMAIN_REGEX
    reg = Regex(regex)
  else:
    reg = Regex(externalRegex)
  return reg.match(emailStr)
```

It is not unreasonable to suppose that the characteristics of the regexes used to solve these problems may likewise vary by programming language. For example, in our own professional software engineering experience, we used complex regexes in scripting languages like Perl but rarely did so in "systems" contexts (C++).

5.3 Study design and research questions

Empirical software engineering research has taught us much about the characteristics of real regexes. But empirical software engineering shares the properties of any other experimental discipline. We hypothesize, sample, measure, and analyze. Then we generalize from our sample to a larger population. However, generalization takes care — it requires understanding how well the sample represents the population, and sometimes this relies on hypotheses about the relationship between the sample and the population.

In this work we formulate and test two generalizability hypotheses underlying prior research (including that presented in Chapter 4). *First*, we test whether prior results may have been biased by following different regex extraction methodologies (§5.5). Researchers have extracted regexes using static analysis or runtime instrumentation, and to generalize from one methodology to the other we must show that the extracted regexes are similar. *Second*, we test whether prior results generalize to other programming languages.

We therefore propose two *regex generalizability hypotheses*: the Extraction Methodology (EM) and Cross-Language (CL) hypotheses. These hypotheses must be tested before prior regex research can be generalized to other software and other programming languages.

H-EM It does not matter whether a regex corpus is constructed using static analysis or program instrumentation. At scale, using either extraction methodology will yield a corpus with similar distributions of regex metrics.

H-CL Regex characteristics are similar across programming language. The distributions of regex metrics will be similar for software from different programming languages.

It is not clear what inferences can be drawn from existing regex corpuses until we have tested these hypotheses. Until they are tested, generalization is tempting but unsound. For example, the recently-published contention that "*over 80% of regular expressions written in GitHub projects are not tested*" [333] generalizes from Wang et al.'s findings [332], which as shown in Table 5.1 considered only 1.2K projects written in only one programming language. Although we agree that [332] raised the interesting prospect of regex under-testing, we think it is premature to generalize from a small number of projects written in one programming language to millions of projects written in hundreds of programming languages.

With all this in mind, our research questions can be summarized as follows:

Theme 1: Hypothesis testing
RQ1: Does the Extraction Methodology Hypothesis hold?
RQ2: Does the Cross-Language Hypothesis hold?

Theme 2: Replicating prior results
RQ3: Does super-linear behavior generalize to other regex engines?
RQ4: Can we replicate other previous regex research?

5.4 Regex metrics for use in hypothesis testing

In this section we introduce our comprehensive collection of regex metrics (Table 5.2). These metrics are used to compare the regexes considered while testing H-EM and H-CL in subsequent sections.

We selected metrics to characterize a regex in three dimensions: its representation, the diversity of the language it describes, and the complexity of various algorithms to solve its membership problem. These metrics fulfill two purposes. First, they include most regex metrics considered in prior research, allowing us to evaluate generalizability. Second, our metrics include those of particular interest to the developers of regex tools and regex engines. In testing these hypotheses, we characterize the largest extant regex corpus in support of data-driven tool and engine designs.

Table 5.2: ***Regex metrics organized by representation, language diversity, and worst-case match complexity.*** *The final column references previous studies that measure or apply this metric. *: no prior scientific measurements.*

Dimension	Metric	Description	Implications	Prior Studies
Representation	Pattern length	Characters in the regex (C# translation)	Length affects visualization, comprehension	*
	Feature vector sparseness	Number of distinct features used	More features: harder to comprehend	*
	# NFA vertices	Number of vertices in an epsilon-free NFA	Size affects visualization, comprehension	[332]
Lang. diversity	# simple paths	Num. of representative matching strings	Comprehension; Test suite size	[220] (basis)
Complexity	DFA blow-up	Ratio of DFA vertices to NFA vertices	Feasibility of static DFA-based algorithm	*
	Mismatch ambiguity	Worst-case match time for backtracking NFA simulation	Feasibility of Spencer's algorithm	[139, 341]
	Average outdegree density	Average completeness of outgoing edge set	Cost of Thompson's algorithm	*
	Has super-linear features	Whether regex relies on super-linear regex features (backreferences, lookaround assertions)	Unavoidable super-linear match complexity	[115, 139, 332]

5.4.1 Metrics for Regex Representation

We measure the representation of a regex in terms of the pattern and its corresponding automata. The features and structural complexity of a regex may impact regex comprehension [116], affecting areas like code re-use and code review. These metrics may also influence the design of visualization tools ("Which features does my visualization need to support? How will typical regexes look in my visualization?").

A regex's pattern representation is the face it shows to engineers. Measures on the pattern representation give some sense of the impression an engineer has when examining the regex. We first measure the *length* of this representation in terms of the number of characters in the string encoding of the pattern. Then we measure its *Chapman feature vector* [115], counting the number of times each regex feature is used. For example, for the regex `/(a+)\1+/` we would compute a regex length of 7 and report two uses of the `+` feature and one use of the capture group and backreference features.

The pattern representation of an (automata-theoretic) regex corresponds to an NFA and DFA representation used by a regex engine to answer regex language membership queries. As we discuss during our analysis, measures of the automata complexity can inform the design of a regex engine. We apply a Thompson-style construction [321] to generate an (epsilon-free) NFA: a graph with vertices corresponding to NFA states connected by labeled edges indicating the character to consume to transition from one state to another.[2] We measure *the number of vertices in the NFA graph.*

[2]There are other NFA constructions optimized for fewer vertices or fewer edges, and a rich literature on the automata minimization problem [192]. We considered using minimized NFAs but found the algorithmic complexity was too great to handle the longer regexes in our corpus.

5.4.2 Metrics for Regex Language Diversity

A regex pattern encodes a string language, i.e., the family of strings that its corresponding automaton will match. We measure the diversity of each regex's language with an eye to its testability. The larger and more diverse a regex's language, the larger the variety in the strings the regex accepts, and the more difficult it is to completely test and validate it.

We operationalize the notion of diversity by measuring *the size of a set of representative matching strings* for that language. We do this by measuring the number of distinct paths from the start state to the accept state that use each node at most once (i.e., the automaton graph's simple paths [14]). Each of these paths corresponds to a string in the language of the regex and is distinct in some way from each other path. In particular, for every optional node there is a simple path that does and does not take it; for every disjunction `/a|b/` there are separate sets of simple paths exploring each option. This family of strings is illustrated in Figure 5.1.[3]

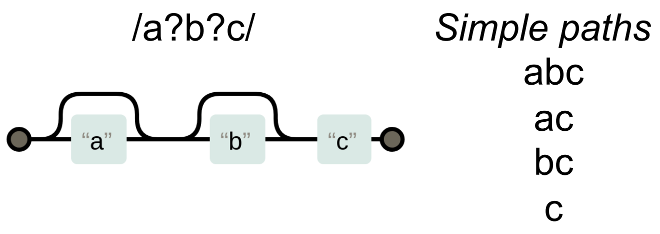


Figure 5.1: ***Illustration of the "simple paths" regex metric.*** *This figure shows the simple paths for the regex* `/a?b?c/`.

Using simple paths to measure language diversity is similar in spirit to using basis paths as proposed by Larson and Kirk [220]. However, their goal was to obtain a manageable set of test strings. We believe succinctness comes at the cost of reduced comprehension. Basis paths can be used to ensure node coverage, but may not fully illustrate the range of "equivalence classes" in the regex's language the way that simple paths will.

5.4.3 Metrics for Regex Worst-Case Complexity

The worst-case time complexity of a regex match depends on the algorithm used to solve it, and several regex membership algorithms have been proposed with complexity ranging from linear to exponential. Our metrics in this dimension are intended to inform the design

[3]This family can also be thought of as the (finite) set of strings in the language of the regex $r_{\text{loop-free}}$ after removing all loops from an original regex r. The size of this family can be determined recursively from the regex representation using rules like: $|characters| = 1$; $|A*| = |A?| = |A\{0,\}| = |A| + 1$; $|A \vee B| = |A| + |B|$; $|AB| = |A| * |B|$.

and application of regex engines based on these different algorithms. We described these algorithms in detail in §2.3, and reiterate relevant aspects here to clarify our choice of metrics.

First, we consider algorithms that have super-linear worst-case complexity as a function of the regex (and input). Regex engines based on these algorithms are reportedly easier to implement and maintain [305], and so there is a tension between language designers' desires and the needs of software engineers who rely on "pathological" regexes in practice. If a high-complexity algorithm is used, pathological regexes become security liabilities — they can [135, 139] and have [140, 307] led to denial of service exploits (ReDoS).

Complexity in static DFA engines One super-linear regex match algorithm statically converts the NFA to an equivalent DFA, offering linear time matches in the size of the input and the DFA. A DFA representation, however, is well known to have worst-case exponentially more states than its corresponding NFA representation [301]. If regexes with enormous DFA representations are common, this kind of algorithm is impractical; if they are rare, then it could be used alone or as the first approach in a hybrid regex engine.

To inform static DFA-based regex engines, we compute the following metric. Using the machinery from the representation metrics, we convert each regex NFA to a DFA. We compute *the ratio of DFA to NFA states* to evaluate how frequently this conversion results in an exponential state blow-up.

Complexity in Spencer engines The Spencer algorithm [305] is a super-linear matching algorithm that relies on a backtracking-based NFA simulation. Spencer's algorithm is used in most programming languages, including JavaScript, Java, and Python [132, 141]. Each time this algorithm has a choice of edges, it takes one and saves the others to try later if the first path does not lead to a match. Several researchers have formalized the conditions for super-linear Spencer-style simulation due to NFA ambiguity [284, 335, 341], and shown that the worst-case simulation cost for a regex on a pathological input may be classified into linear, polynomial, or exponential as a function of the input string.

To inform Spencer-style regex engines, we compute the following metric. We measure a *regex's worst-case partial-match complexity in a Spencer-style engine*. For this measurement we use Weideman et al.'s analysis [335].[4] In our measurements we report the proportion of regexes that this analysis marks as polynomial and exponential among those it successfully analyzes. If super-linear regexes are common in software written in programming languages that use Spencer-style regex engines, the designers of those programming languages may wish to consider an alternative algorithm to reduce the risk of ReDoS vulnerabilities.

[4]Weideman et al.'s analysis was the most successful among those we applied in Chapter 4.

Complexity in Thompson engines The Thompson algorithm [321], popularized by Cox [132], uses a dynamic-DFA based NFA simulation. It forms the basis of the Go and Rust regex engines. Each time a Thompson-style matching algorithm has a choice of edges, it simulates taking all of them, tracking the current set of possible NFA vertices and repeatedly computing the next set of vertices based on the available edges in the NFA transition table. In effect, a Thompson-style engine computes the DFA dynamically, not statically, and only computes the state-sets that are actually encountered on the input in question. It offers worst-case $\mathcal{O}(|Q|^2 * |w|)$ complexity for a candidate string w on a regex whose NFA has $|Q|$ vertices, with the cost of each transition bounded by the number of outgoing edges that must be considered for each vertex in the current state-set. Note that each vertex may have outgoing edges to between zero and all m of the vertices in the graph, and the cost of each step of the Thompson algorithm depends on the number of outgoing edges from the current state-set.

We use the following metric to inform the design of a Thompson-style engine: *the average vertex outdegree density*, $\frac{1}{|Q|}\sum_{1}^{|Q|}\frac{\text{deg}_{\text{vertex}_i}}{|Q|} = \frac{|E|}{|Q|^2}$, where E is the edge set of the automaton. This is a $[0,1]$ metric, 0 for completely unconnected graphs and 1 for completely connected graphs. For a Thompson-style engine, for the current state-set Φ_{curr}, it will cost an average of $|\Phi_{curr}|$ times this metric to compute the next state-set.[5]

Unavoidable super-linear complexity Most regex engines support a feature set beyond traditional automata-theoretic regular expressions. Of particular note are backreferences, a self-referential construct proved to be worst-case exponential in the length of the input [63], and lookaround assertions, which are typically implemented with super-linear complexity. To round out our complexity metrics, we measure *the proportion of regexes that rely on these super-linear features*, through reference to the feature vector computed as part of the regex representation metrics. Understanding the popularity of these features may guide future regex engine developers in deciding whether or not to support these features. The most recent programming languages to gain mainstream adoption, Rust and Go, decided not to support these features, and it is not clear whether this decision will impose significant portability problems on engineers transitioning software from other languages to these ones.

5.4.4 Implementation of metric measurements

We built our measurement instruments on Microsoft's Automata library [241], which underlies the Rex regex input generation tool [328]. To the best of our knowledge this is the most advanced open-source regex manipulation library. Our fork extends the Automata library in several ways:

[5]Assuming that vertices are visited with equal probability.

- We fixed several bugs in its automaton manipulations, eliminating long-running computation and memory exhaustion.
- We added support for generating the Chapman feature vector of a regex.
- We added support for collapsing certain expensive portions of a regex to facilitate simple path computation.
- We added support for emitting an automaton's graph in a format suitable for subsequent analysis.
- We introduced a command-line interface for automation.

The Automata library only supports .NET-compliant regexes. We therefore implemented an ad hoc syntactic regex conversion tool to translate regexes from other languages into a semantically equivalent .NET regex before measuring them. To reduce bias, we converted at least 95% of the regexes originating in each language. These translations sufficed:

1. We replaced Python-style named capture groups and backreferences, `(?P<name>A)...(?P=<name>)`, with the .NET equivalent, `(?<name>A)...\k<name>`.
2. .NET only permits curly brackets to indicate repetition, while some other languages interpret curly brackets with non-numeric contents as a literal string. We escaped any curly bracket constructions of this form.
3. .NET does not support the `/\Q...\E/` escape notation. We removed the Q-E bookends and escaped the innards.
4. .NET does not support certain inline flags. We replaced the Unicode support flag with the "case insensitive" flag to preserve the presence of the feature while ensuring .NET compatibility.

The Automata library does not support simple path measurements, so we analyzed the NFA graph it produced using the NetworkX library [183].

The Automata library can parse all .NET-compliant regexes, but it can only produce NFAs for regexes that are regular (i.e., K-regexes, e.g., no support for backreferences). We therefore omit automata measurements when necessary. We also omit automata measurements when the Automata library took more than 5 seconds to generate them.

5.5 RQ1: Does the Extraction Methodology Hypothesis hold?

Here we test the H-EM hypothesis: "It does not matter whether a regex corpus is constructed using static analysis or program instrumentation." *We found no reason to reject this hypothesis in the software we studied.*

We tested H-EM in the context of open-source software modules (libraries). Lacking access to closed-source software, we studied open-source software out of necessity. We opted to study modules rather than applications by choice. In our experience, modules have less variability in design and structure than do projects randomly sampled from GitHub, facilitating automated analysis. In addition, the ecosystem of most popular programming languages includes a large module registry, and so modules were a convenient target for our cross-language comparison experiment (H-CL). Using modules to test H-EM as well unified our methodology.

5.5.1 Methodology

Summary Our methodology is summarized in Figure 5.2, and Table 5.3 provides the details. We targeted software modules written in the three most popular programming languages on GitHub: JavaScript, Java, and Python [169]. We extracted regexes using both static analysis and program instrumentation. After creating a regex corpus, we used statistical tests to determine whether there were significant differences between the regexes extracted using each methodology in any programming language.

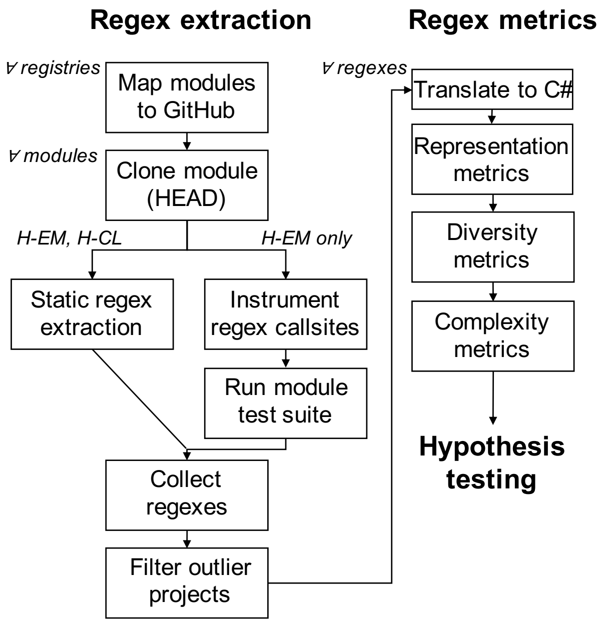


Figure 5.2: ***Methodology followed in our study of regex generalizability.*** *This figure shows our analysis flowchart. We performed regex extraction for H-EM, and for H-CL we leveraged an existing corpus derived using similar methodology.*

Software Modules were chosen by identifying the most prominent module registry for each language, mapping its modules to GitHub, and examining approximately the most important 25,000 modules from each. For JavaScript we used npm modules [32], for Python we used pypi modules [34], and for Java we used Maven modules [28]. Because software engineers commonly star modules that they depend on, we used a module's GitHub stars as a proxy for importance [95]. We extracted regexes from the entire module source code, both production code (e.g., `src/`) and test code (e.g., `test/`). We considered only source code written in the language appropriate for the module registry (e.g., only Python files for *pypi* modules, as determined by the `cloc` tool [55]).

Extraction through static analysis We followed the methodology described in [115, 139, 141]. In each language, we used an AST builder to parse the module source code and visit the regex-creating call sites. We extracted statically-defined regex patterns from each such call site. We did not perform any dataflow analysis: we extracted string literals used as the regex pattern, and did not attempt to resolve non-literal arguments. For example, this extraction would only retrieve the "SIMPLE_REGEX" from Listing 9.

We examined the documentation for each programming language to learn the regex-creating call sites. Generally, regexes can be created directly through the language's Regex type and indirectly through methods on the language's String type. For example, in JavaScript you can create a regex directly using a regex literal, `/pattern/`, or the RegExp constructor, `new RegExp(pattern)`, or indirectly using a String method like `s.search(pattern)`. The AST libraries and regex-creating call sites we identified for each language are listed in Table 5.3.

Extraction through program instrumentation We followed a methodology similar to that of Wang and Stolee [332], but repaired one of their threats to validity. We targeted the same regex-creating call sites as we did in the static analysis. We applied a program transformation to instrument the (potentially variable) regex pattern argument at these call sites. Our instrumentation consisted of an inline anonymous function to log the pattern and return it, avoiding side effects. We then executed the test suites for the modules and collected the regexes that reached our instrumentation code. For example, this extraction would retrieve each of the regexes from Listing 9, provided the test suite covered each path.

We automatically executed the test suite for each module that used one of the common build systems for its registry (Table 5.3). We identified these build systems using a mix of Internet searches and iterative analysis of modules from each registry. Because our source code-level instrumentation did not follow the coding conventions of the projects, some build attempts initially failed during an early linting stage. We configured our builds to skip or ignore the results of linting.

We found that many modules did not have test suites [59], and others failed to build due to external dependencies. We took several measures to increase the number of successful test executions. In Java, we installed all Android SDKs and Build Tools using Google's `sdkmanager`, permitting us to build many modules intended for use on Android. In Python, we attempted to run test suites under Python 2.7 and Python 3.5/6 using many different build systems. However, these ad hoc approaches may have caused us to miss projects with

Table 5.3: ***Details of the regex extraction techniques used to compare the regexes extracted using static analysis and program instrumentation.*** *Regex extraction details for H-EM. For static analysis, we extracted any constant regex patterns used at these call sites. For program instrumentation, we wrapped these call sites with a call to a log routine.*

Language	Regex call sites	AST modules	Build systems	Sample invocation
JavaScript	*RegExp*: RegExp literals, RegExp constructor *String methods*: match, matchAll, search	Babel [3]	npm [33]	npm install-build-test
Java	*java.util.regex.Pattern*: compile, matches *String methods*: matches, replaceFirst, replaceAll, split	JavaParser [6]	Maven [29], Gradle [26]	mvn clean-compile-test
Python	*re module*: compile, escape, findall, finditer, fullmatch, match, search, split, sub, subn	ast, astor[1]	Distutils [24], Tox [39], Nox [31], Pytest [35], Nose [30]	python3 setup.py test

other build systems or dependencies.

Collecting regexes via source code instrumentation ensured that we captured only the regexes created within each module, permitting direct comparison of the regexes extracted through the two different methodologies. This approach counters one of the threats to [332], which instrumented the language runtime and attempted to filter out third-party regexes.

Constructing the regex corpus After extracting regexes from each module using the two methods, we combined the results into a corpus of unique regex patterns based on string equality of the regex pattern representations. We then noticed that some projects contributed orders of magnitude more regexes to the corpus than others did. The median number of unique regexes in regex-using projects was 1–3 in our experiment, while a few outlier libraries defined hundreds or thousands of distinct regexes — enough to bias statistical summaries of the regex corpus.[6] We therefore omitted regexes from projects at or above the 99th percentile of the number of unique regexes per project.

The regex corpus used to test the H-EM hypothesis is summarized in Table 5.4. Several elements of this corpus are worth noting. The corpus contains a moderate number of regexes extracted using static analysis and program instrumentation, ranging from around 15K (Java) to around 80K (JavaScript). We found that 30–50% of the modules in each

[6]For example, the most prolific regex producers were pypi's `device_detector` module, which has 4,953 distinct regexes to match user-agent strings, and Maven's `recursive-expressions` module, which creates 3,398 regexes to test its extended regex APIs.

Table 5.4: ***Summary of the corpuses resulting from the two regex extraction methodologies.*** *Summary of corpus used to test H-EM. For each cell "X (Y)", we obtained X unique regexes across Y regex-using modules. The final row gives the regex intersection. This corpus contains 124,800 unique regexes.*

Extraction method	**JavaScript**	**Java**	**Python**
Static	71,799 (13.1K)	10,237 (8.2K)	27,641 (9.1K)
Instrumentation	21,759 (4.4K)	9,236 (3.1K)	11,514 (3.7K)
Static ∩ Inst.	13,633	3,463	5,690

language used at least one regex, supporting previous estimates [115, 139, 332]. We were able to extract regexes from 3K–4K modules using program instrumentation, or about one third of the number from which we obtained regexes through static analysis.[7] Lastly, as you can see in the final row of Table 5.4, about half of the regexes obtained through program instrumentation were *not* obtained through static analysis and would thus not have been captured by a static-only extraction methodology.

Threats and considerations Our approach is best-effort, neither sound nor complete. JavaScript and Python are dynamically typed, which could lead to *false positives* (non-regexes entering our corpus). For example, our JavaScript instrumentation relies on method names and signatures to find regexes, and for example may emit non-regexes if a class has a "match" method that shares the signature of the corresponding String method (Table 5.3). Our analysis is also subject to *false negatives*, through modules that could not be parsed or built by our analyses (e.g., unsupported language versions or unfamiliar build systems), and through modules that create regexes via third-party APIs (e.g., using an "escape special chars and return a Regex" API). We appeal to the scale of our dataset to ameliorate concerns about corpus validity.

For each module, we limited the static and dynamic phases of regex extraction to 10 minutes, and included in our corpus all regexes extracted during this time limit. Regex extraction and metric calculation were performed on a 10-node cluster of server-class nodes: Ubuntu 16.04, 48-core Intel Xeon E5-2650 CPU, 256 GB RAM.

5.5.2 Statistical Methods

We used statistical methods to determine whether there is evidence to reject H-EM — whether the regexes extracted using these two methodologies exhibited significant differences along any of our regex metrics. The statistical tests we chose were influenced by the

[7]We attribute this proportion to a combination of our failure to run the test suite, and poor code coverage within successful test suites.

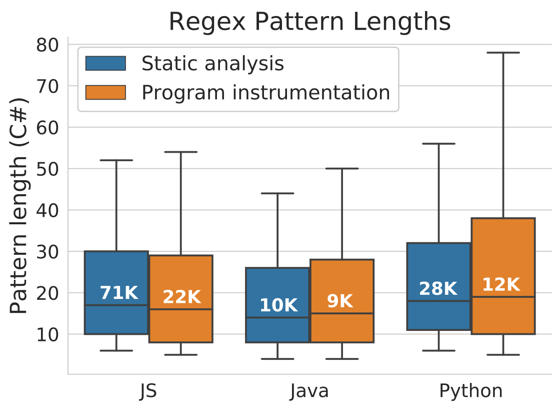


Figure 5.3: ***H-EM: Regex pattern lengths by programming language.*** *Lengths of regexes extracted statically and dynamically, grouped by language. Whiskers indicate the (10, 90)th percentiles. Outliers are not shown. The text in each box shows the total number of regexes included in that group. This figure resembles the figures for the other metrics. We found negligible-to-small effect sizes for intra-language regex extraction comparisons across all metrics.*

distribution of the regex characteristics. Tests such as the Analysis of Variance (ANOVA) are typically used to evaluate such hypotheses. However, these tests require normality and homogeneity of variance, and none of the regex metric distributions met these assumptions. Therefore, we instead used the nonparametric Kruskal-Wallis test [215], with language and extraction mode as the treatment variables and our metrics as the dependent variables.

We found that hypothesis tests alone did not usefully describe our data. The scale of our regex corpus gave us tremendous statistical power, causing hypothesis tests to detect statistically significant but practically irrelevant differences in the data. So, after performing the Kruskal-Wallis hypothesis test, we calculated effect sizes for pairwise differences between groups. Because the distributions of regex characteristics do not meet the conditions assumed by parametric statistical tests, we applied a nonparametric difference effect size measurement d_r derived from the commonly used Cohen's d [126]. The d_r measure is a scaled robust estimator of Cohen's d proposed by Algina et al. [67], and shown to be robust to non-normal and non-homogeneous data [222]. It takes on scaled values indicating the size of the difference between two samples, ranging from 0 (no difference) to 1 (large difference).

5.5.3 Results

As indicated in Table 5.4, to test H-EM we split the regex corpus into two subsets: those extracted using static analysis, and those extracted using program instrumentation. Regexes found using both techniques were included in both subsets.

We compared the two subsets in terms of the metrics described in §5.4. For all metrics, we found negligible-to-small effect sizes ($d_r <= 0.3$) between the static and dynamic subsets within each language. Figure 5.3 is illustrative: the similarity of regex lengths between the two subsets in each language is visually apparent. Other metrics look similar.

Therefore, we are unable to reject the null hypothesis H-EM. This conclusion supports the generalizability of prior empirical regex findings — from regexes declared using string literals to those generated dynamically, and vice versa.

5.6 RQ2: Does the Cross-Language Hypothesis hold?

Here we test the H-CL hypothesis: "Regex characteristics are similar across programming languages." The H-CL hypothesis held for many characteristics. However, we identified several metrics on which there were moderate to large effect sizes between programming languages. *Not all regex characteristics span programming languages. Some differ significantly.*

5.6.1 Methodology

5.6.1.1 Experimental Design

As we reported in §5.5, we did not reject the H-EM hypothesis in any of the three programming languages we studied. We used this finding as a basis for our methodology for testing H-CL: under the assumption that H-EM holds more broadly, we evaluated the regex characteristics for software in many languages based on regexes obtained solely through static analysis. For this comparison, we developed a polyglot regex corpus (§5.6.1.2).[8] This corpus contains 537,806 unique static regexes extracted from 193,524 popular software modules written in eight programming languages: JavaScript, Java, PHP, Python, Ruby, Go, Perl, and Rust. These regexes were obtained statically using extraction methods similar to those described in §5.5.1.

After collecting this corpus, we followed the same measurement and statistical approach for H-CL that we did for H-EM. We measured the characteristics of the regexes in the polyglot regex corpus and again found that the distributions did not meet the conditions of normality and homogeneity of variance. Again the large sample size caused nonparametric Kruskal-Wallis hypothesis tests to yield uniformly significant differences. Thus, we report programming languages with a moderate ($d_r > 0.5$) or large ($d_r > 0.7$) pairwise effect size.

[8]This corpus was originally published in my ESEC/FSE 2019 work [141].

*Table 5.5: **Summary of the polyglot regex corpus.** Our regex corpus was derived from software written in 8 programming languages. The first five languages are ranked by the most available libraries (ModuleCounts [144]) and popularity in open-source (GitHub). We also studied Go, Perl, and Rust out of scientific interest. The two final columns show the contribution to our corpus. **JavaScript***: We also extracted regexes from TypeScript source code, by transpiling it to JavaScript.*

Lang. (Registry)	**Libs.**	**GH**	**# mod. anal.**	**Unique regexes (avg.)**
JavaScript* (npm)	1	1	24,997	150,922 (6.0)
Java (Maven)	2	3	24,986	19,332 (0.8)
PHP (Packagist)	3	5	24,995	44,237 (1.2)
Python (pypi)	4	2	24,997	43,896 (1.8)
Ruby (RubyGems)	5	4	24,999	153,334 (6.1)
Go (Gopm)	9	9	24,997	22,105 (0.9)
Perl (CPAN)	7	—	31,827 (all)	142,777 (4.5)
Rust (Crates.io)	10	—	11,724 (all)	2,025 (0.2)
		Sum:	*193,524*	*578,628*

5.6.1.2 Polyglot Regex Corpus

In order to answer our remaining research questions we needed a *polyglot regex corpus*: a set of regexes extracted from a large sample of software projects written in many programming languages. The existing regex corpuses are small-scale [115, 341] or include only two programming languages [139]. As summarized in Table 5.5, our corpus covers about 200,000 projects in 8 programming languages.

Programming languages We are interested in studying common regex practices, and as a result we focus our attention on "major" programming languages defined by two conditions: (1) The language has a large module ecosystem; (2) The language is widely used by the open-source community. We operationalized these concepts by consulting the ModuleCounts website [144] and the GitHub language popularity report [169]. We also considered Go, Perl, and Rust for scientific interest; Perl popularized the idea of regexes as a first-class language feature, and Go and Rust are relatively new mainstream languages. The languages we used are listed in Table 5.5.

Software projects Within these languages, we chose to study the software modules published in each language's primary *module registry* for two reasons. First, it permits a relatively fair cross-language comparison, since we observe that many modules fill equivalent ecological niches, e.g., logging or schema validation. Second, we feel that modules are of greater general interest than applications. Modules are published, maintained, and used by a mix of open-source and commercial software developers, and bugs and security vulnerabilities in modules have a significant ripple effect.

Our goal was to analyze the most important modules in each language's primary module registry. Borges and Valente recently showed that GitHub star count is a reasonable proxy for importance [95]. To have a uniform measure of importance across languages and registries, we filtered each registry for the modules available on GitHub, sorted those by the number of stars, and analyzed the most-starred modules.

Software modules on GitHub: Most registries offer an API for module metadata, which includes the location of a module's source code. Where such an API was available, we considered the modules whose source code was hosted on GitHub. CPAN (Perl) and Maven (Java) were exceptions to this rule. CPAN does not consistently list project URLs, but it does offer a way to enumerate source code artifacts; we mirrored the entire registry and analyzed all 30K of the modules therein. Maven does not consistently list project URLs, and does not permit easy enumeration of source code artifacts. Hence, we performed a *reverse mapping* from GitHub projects to Maven artifacts. We enumerated the Java projects on GitHub using the GitHub search API,[9] and used heuristics on their READMEs to identify those that listed Maven artifacts. We manually determined that the README files for GitHub projects with corresponding Maven artifacts commonly denote this in one of two ways — they display a Maven badge, or they include a snippet for a dependent's `pom.xml` file to include their project as a dependency. For each of the 184,099 projects enumerated by our search (93% of all such projects), we visited the README on GitHub and used regular expressions to filter for those that include one of these elements, reducing the set to 29,268 projects that advertise a Maven artifact.

Most important modules: As Figure 5.4 shows, the distribution of GitHub stars was similar for the modules in each programming language. We chose to analyze the top 25,000 modules by GitHub stars, which in most languages captured all but the (very long) tail of modules with 0-2 stars. Perl and Rust had relatively few modules in their registries, and we analyzed all of their modules.

Regex extraction Following the methodology in Chapter 4, for each module we cloned the HEAD of its default branch from GitHub and extracted any statically-declared regexes. We extracted regexes declared in regex evaluations as well as regexes compiled and stored in variables for later use. In each module we extracted regexes only from source files in the programming language corresponding to the registry (e.g., JavaScript and TypeScript for npm, Perl for CPAN, etc.), omitting regexes in places like build scripts written in another language.

[9] On 7 January 2019, about 198,595 GitHub projects matched the query `maven in:readme language:java`. Due to GitHub's limit of 1000 responses per query, we partitioned this space by project size and creation date to increase the number of projects we captured.

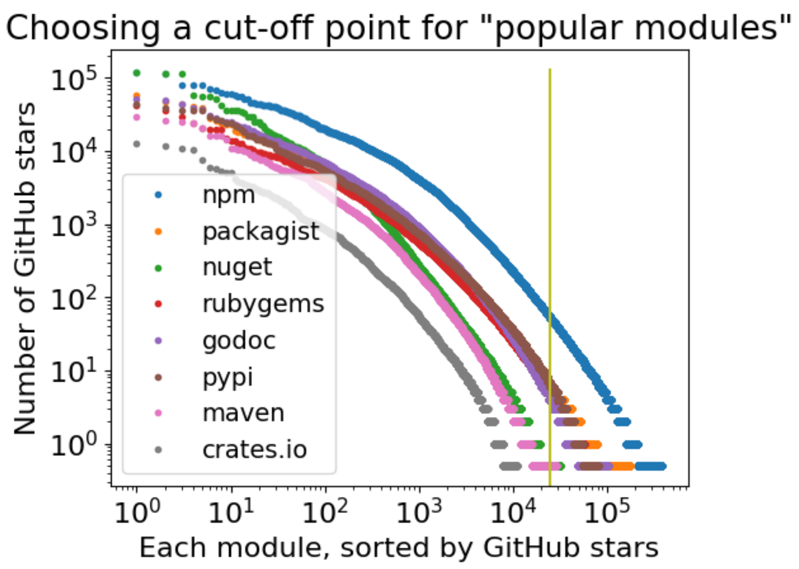


Figure 5.4: ***Distribution of module stars by programming language.*** *This figure shows the top modules in a registry sorted by star count. Note the log-log scale, on which projects with 0 stars are represented as having 0.5 stars. The vertical bar denotes the 25,000 position in this ranking, which was the cutoff point that we selected for analysis.*

Summary of polyglot regex corpus Our corpus contains 537,806 unique regexes extracted from 193,524 projects written in 8 programming languages. Each language's contributions are listed in Table 5.5. Average regex use varies widely by language, from 0.2 regexes per module (Rust) up to 6.1 regexes per module (Ruby). The total unique regexes by language exceeds 537,806 due to inter-language duplicates.

5.6.2 Results

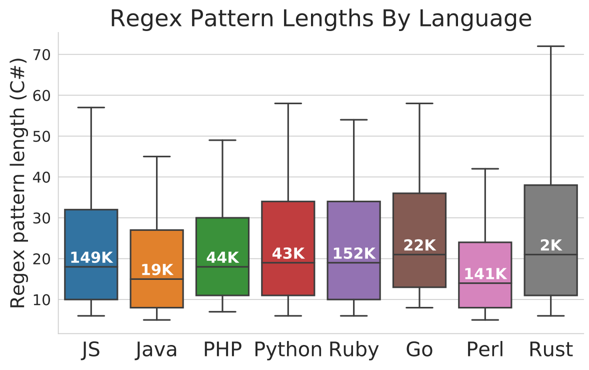
Having collected and measured the polyglot regex corpus, we applied statistical tests as described in §5.6.1. Table 5.6 summarizes the results for each metric. We report the details for the metrics with significant effect sizes below. In §5.9 we discuss some of the implications of these and other measurements.

- *Pattern length.* **Perl** regexes tend to be shorter than those in Go and Rust, with moderate effect sizes (Figure 5.5a).
- *Features used.* Regexes in **Ruby** (large effects) and **JavaScript** (moderate) tend to use fewer features than regexes in PHP, Python, Go, and Rust (Figure 5.5b).
- *# NFA vertices.* Regexes in **Ruby** tend to have more NFA vertices than those in Java and Perl (moderate) (Figure 5.5c).
- *Average outdegree density.* Regexes in **Ruby** have a significantly smaller outdegree density

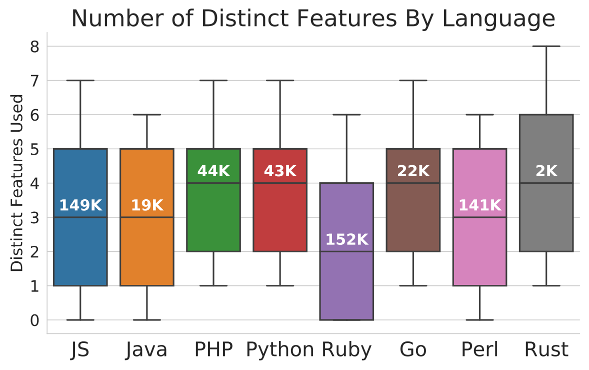
Table 5.6: ***Measurements of regexes extracted from different programming languages.*** *Metrics for each programming language in the H-CL experiment. The second column gives the range of the median or the observed percentage, and the third notes programming languages with significant differences from other languages.*

Metric	**Low / High**	**Unusual langs.**
Length	Perl: 14 / Go: 21	Perl
Feat. vect. sparseness	Ruby: 2 / Go: 4	Ruby, JS
# NFA vertices	Java: 7 / Ruby 14	Ruby
# simple paths	Go: 1 / Python: 2	–
DFA blow-up	Perl 1.1 / Ruby 1.7	–
Mismatch ambiguity	Ruby: 19.1% / Python: 38.4%	–
Avg. outdegree density	Ruby: 0.08 / Java: 0.19	Ruby
Has super-linear features	Perl: 2.3% / JS: 4.3%	–

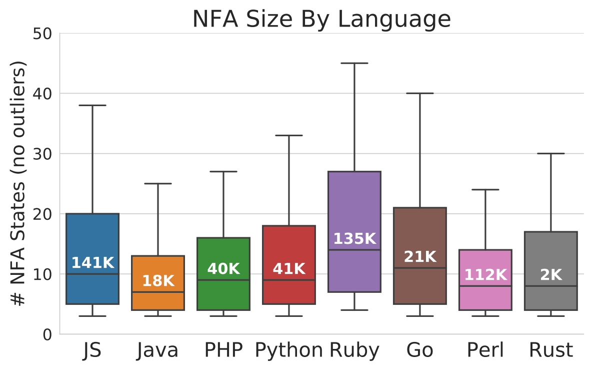
than those in Perl, PHP, and Rust (moderate), and Java (large) (Figure 5.5d).



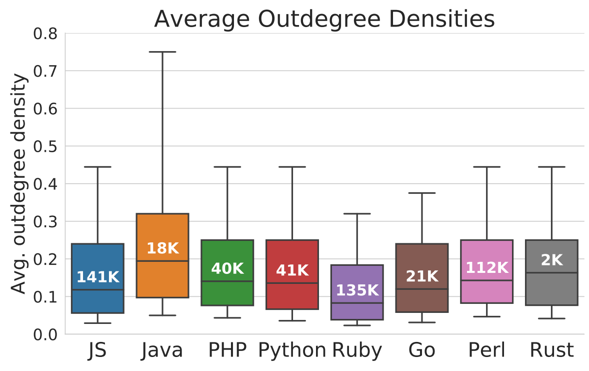
(a) Regex lengths.



(b) Number of distinct features used.



(c) Regex NFA size (# vertices).



(d) Average outdegree density.

Figure 5.5: ***H-CL: Cross-language regex comparisons on various metrics.*** *Whiskers are (10, 90)th percentiles. Outliers are not shown.*

5.7 RQ3: Does super-linear behavior generalize to other regex engines?

The regex engines in different programming languages may employ different algorithms, optimizations, and defenses. Prior work has estimated the proportion of regexes that exhibit super-linear (typically polynomial) behavior in JavaScript and Python at 1% [139] (Chapter 4, and in Java at 20% [341]. Cox has presented anecdotal examples of super-linear behavior in other regex engines [132], but without an empirical study to better characterize the state of practice it should not be assumed that ReDoS Condition 2 (use of a super-linear regex engine) affects typical regexes.

The documentation of most regex engines does not describe their worst-case performance. This section therefore measures the degree to which regexes exhibit super-linear behavior in many programming languages, due to those languages' use of a Spencer-style regex engine.[10] It thus replicates and updates prior findings in many programming languages.

5.7.1 Methodology

We generally followed the methodology described in §4.3. For each regex we (1) query an ensemble of state-of-the-art super-linear regex detectors, and then (2) evaluate any predicted super-linear regex behaviors in each language of interest. We enhanced this methodology to address two causes of false negatives.

Experimental parameters We allowed each of the detectors to evaluate a regex for up to 60 seconds using no more than 2 GB of memory. If a detector predicted that a regex would be super-linear, we evaluated its proposed worst-case input in each of the 8 languages in our study using input strings intended to trigger exponential or polynomial behavior[11]. If a regex match took more than 10 seconds in some language, we marked it as super-linear.

Techniques to reduce false negatives We extended our previous methodology in two ways to reduce the number of false negatives (i.e., super-linear regexes marked as linear-time). First, we added Shen et al.'s dynamic super-linear regex detector [297] to their ensemble ([284, 335, 341]). Shen et al.'s detector was published after the earlier experiments were completed. Second, and more critically, we introduce a new technique that identifies both polynomial and exponential super-linear regexes that their detector ensemble would not detect. The static detectors in the ensemble: (1) assume full-match semantics, and (2) do

[10] This experiment was published in [141].

[11] We used 100 pumps for exponential and 100,000 pumps for polynomial. This methodology causes some high-polynomial regexes to be counted as exhibiting exponential worst-case performance.

not scale well to regexes with large NFAs. We combat these problems by querying detectors with the original regex as well as *regex variants* that they can more readily analyze.

The first query variant addresses an *unrealistic assumption* in the analysis performed by some of the detectors in the ensemble ([284, 335, 341]). Although these detectors assume that the regex engine is using full-match semantics, regex engines generally default to partial-match semantics. For example, some detectors predict linear behavior for `/a+$/`, but it is quadratic in many languages when used with a partial-match API. To address this assumption, we query the detector ensemble with an (anchored) full-match variant of unanchored regexes, e.g., `/^[\s\S]*?a+$/`. This modification thus assumes that regexes are being used with a more expensive regex match query. We expect the presence of anchors to cause the detectors to identify more polynomial regexes, but not additional exponential regexes; the concatenation of "`[\s\S]*`" and an existing regex should add no more than polynomial ambiguity.

The second query variant addresses *inefficient implementations* in the detector ensemble. Some of the detectors cannot complete their analysis within our time limit on regexes with large NFA representations. For example, they time out on the (exponential) regex `/(a{1,1000}){1,1000}$/` because its NFA explodes in size. To account for this inefficiency, we query the detector ensemble with variants that replace bounded quantifiers with unbounded ones, e.g., `/(a+)+$/`.

In our experiments, these variants reduce the rate of false negatives without introducing false positives. Although some of these variants may be more vulnerable than the original, we always test any worst-case input on the original regex (dynamic validation). The first variant may unmask polynomial regexes that would otherwise go undetected, and the second may identify both polynomial and exponential regexes.

5.7.2 Results

The detector ensemble estimated that more than 20% of the regex corpus would exhibit super-linear behavior in a Spencer-style backtracking regex engine. Figure 5.6 illustrates the extent to which the regexes in our polyglot corpus actually exhibited worst-case super-linear behavior in each of the 8 languages under study.

Figure 5.6 indicates that super-linear regexes may be more common — by up to an order of magnitude! — than was reported in Chapter 4. It is worth noting that Figure 5.6 does not provide a direct comparison to [139]. We have a different corpus and are testing regexes from multiple origin languages. However, the same larger proportions occur when considering the subset of our corpus derived from JavaScript and Python (as theirs was). Our results are of the same order of magnitude as Wüstholz et al.'s small-scale estimate in Java [341]. The majority of the newly-discovered regexes were identified through our variant testing technique; as expected, the new detector by Shen et al. [297] identified only

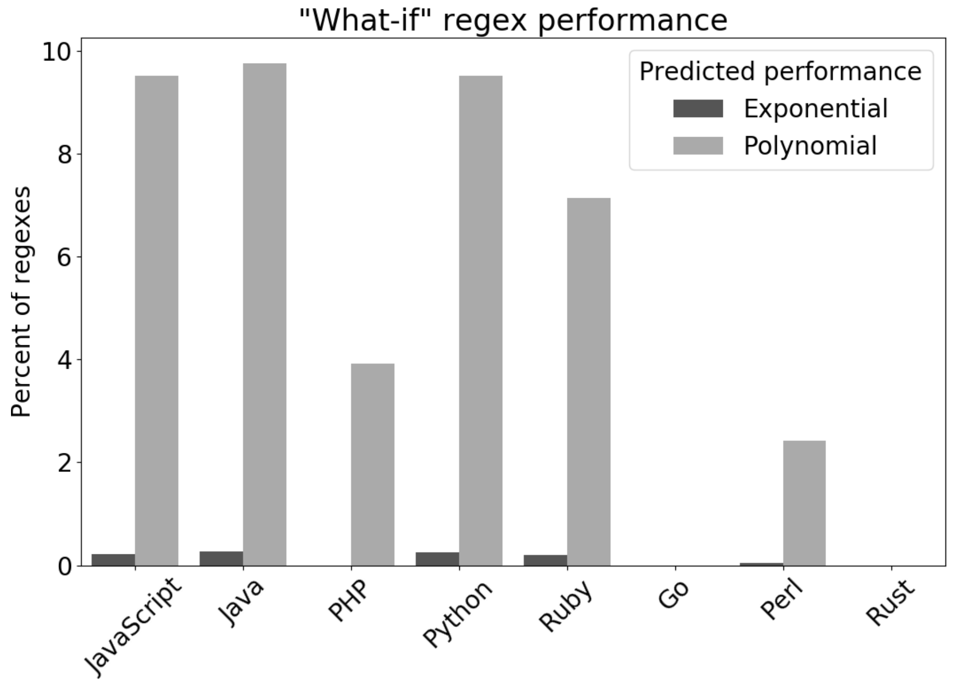


Figure 5.6: ***Frequency of super-linear regex behavior in eight programming languages.*** *There are three distinct families of worst-case regex performance. We identified no regexes with exponential behavior in Go and Rust, and only 6 regexes had polynomial behavior in those languages. Regexes with exponential behavior are rare in PHP and Perl (Perl – 227; PHP – 0), but polynomial behavior still occurs. In contrast, over 1,000 regexes have exponential behavior in Ruby, Java, JavaScript, and Python, and polynomial behavior is also more common in those languages.*

exponential regexes (1,421 of them). Manual examination of these regexes suggests that they have exponentially ambiguous sub-patterns that the other detectors might in principle have found, but that the patterns contained extended features (e.g., backreferences) not supported by the other detectors.

5.7.3 Analysis

Two findings are clear from this experiment. First, there is a gap between the models used by several of the super-linear regex detectors, and the real Spencer-style regex engines used in production. Second, not all Spencer-style regex engines are created equal.

This experiment illustrates a significant gap between theoretical models of Spencer-style regex engines and their actual implementation. The detector ensemble estimated that at least 20% of these regexes would exhibit super-linear behavior. For three of the four detectors, this prediction was based on their model of a perfectly naive Spencer-style regex engine, as described in §2.3.1.2. However, Figure 5.6 indicates that no more than 10% of these regexes actually exhibited super-linear behavior in real regex engines. The implication is that real-world regex engines do not perfectly follow the idealized backtracking model. Manual analysis suggests that this deviation can be explained by these engines' use of well-known optimizations like the Aho-Corasick [64], Knuth-Morris-Pratt [212], and Boyer-Moore [96] string searching algorithms. These optimizations permit them to short-circuit some queries based on the properties of the regex and/or the contents of the candidate string, without paying the full cost of a backtracking search. These optimizations cut both ways; some regexes may be incorrectly (statically) classified as super-linear although all queries against them can be resolved in linear time using such optimizations, while others may be incorrectly (dynamically) classified as linear-time if the attack inputs proposed by the super-linear ensemble can be trivially rejected, even if some other attack input might require super-linear behavior. Improving the accuracy of super-linear regex detection analysis by refining the regex engine model may be a profitable line of future work.

The proportion of regexes that exhibit exponential and polynomial worst-case behavior varies widely by language. The regex engines in these languages fall into three families: (1) *Slow* (JavaScript, Java, Python, Ruby); (2) *Medium* (PHP, Perl); and (3) *Fast* (Go, Rust). Through an investigation of regex engine internals, we have attributed the causes of these families to two of the defenses against ReDoS that are evaluated in Part III. The regex engines used by Go and Rust (*Fast*) do not use a Spencer-style backtracking regex engine, but rather a Thompson-style engine [146, 176] (Chapter 7). The remaining six languages all employ a Spencer-style backtracking regex engine, but the engines of PHP and Perl (*Medium*) apply run-time caps on resource utilization to short-circuit certain long-running evaluations (Chapter 9).

5.8 RQ4: Can we replicate other previous regex research?

Using the H-CL corpus, we attempted to replicate and generalize many of the findings described in §2.6.

Regex use In agreement with prior estimates of regexes in 30–40% of modules (Python, JavaScript [115, 139]), regex use is common in the modules that contributed to the H-CL regex corpus, ranging from 23% (Go) to 71% (Perl). This finding did not generalize to **Rust**; only 5% of Rust modules contained regexes.

Regex feature popularity Feature usage rates in Python regexes were in agreement with findings from Chapman and Stolee [115]. The relative popularity ranking of different regex features is approximately similar across all programming languages in our corpus. For example, across languages, capture groups like `/(a)/` are a popular feature, while inline flag changes like `/(?i)CaSE/` are relatively rarely used.

Use of extended regex features Prior researchers have reported that developers do not commonly use extended regex features (backreferences, lookaround assertions), with rates below 5% reported in JavaScript, Python, and Java [139, 332]. This rate holds in all programming languages we studied that support those features (i.e., excepting Go and Rust).

Automaton sizes We were not initially able to replicate findings from Wang and Stolee's work describing automaton sizes [332]. They reported that the (Java) regexes in their corpus, obtained using program instrumentation, had much larger DFAs than we found, with a 75th percentile of 70 nodes and 212 edges. The Java regexes in the corpus we analyzed (obtained using static analysis) have a 75th percentile of only 10 nodes and 70 edges. We first confirmed that our measurement instrument could replicate their results on their corpus. We then wondered if a few atypical projects might dominate their corpus, as in our corpus prior to our filtering step (§5.5.1). Indeed, we found that 19 source files in their corpus sat at or above the 99th percentile of unique regexes, and contributed more than half of the unique regexes in their corpus. After filtering out these files, our two corpuses had similar DFA measures.

This comparison emphasizes the importance of considering outlier projects during regex corpus construction. In both corpuses, a few projects contained enough regexes to bias the statistics derived from analyzing thousands of projects. We believe filtering out the regexes from these outlier projects offers a more accurate perspective on the population of "average" regex-using projects. However, this may be a matter of preference; perhaps major users of regexes deserve a greater voice in corpuses. We are grateful to Wang and Stolee's commitment to open science, permitting us to confirm that this phenomenon occurred in both sets of software and that the same filtering approach was effective on both sets.

5.9 Discussion

Previous empirical research on regex characteristics focused on statically-extracted regexes in software written in a small number of programming languages. This focus was not myopic: based on our suite of eight metrics we found that regex corpuses are similar whether they follow a regex extraction methodology based on static analysis or program instrumentation, and some characteristics of regexes are similar across many programming languages. However, some regex characteristics do not generalize across programming languages, and we encourage future empirical regex researchers to design their studies accordingly. We hope our methodological refinements and our efforts to validate generalizability hypotheses lay the foundation for further empirical regex research. We look forward to a new generation of regex tools and regex engines inspired by our measurements.

In the preceding sections we applied our measurements to test the validity of the H-EM and H-CL hypotheses. In those experiments we compared the *relative* values of the measures in different subsets of regex corpuses. However, the *specific* values of our measurements may be of interest to regex tool designers (cf. §2.6.4) and regex engine developers. Though there are outliers in each category, appropriate percentiles are useful for reasoning about the common case of regexes encountered by regex tools and engines.

Tools can be tailored to real regexes Measures of regex representation (pattern, automaton) may be the most relevant for regex visualization and debugging tools. The (25,75)th percentile lengths of regexes in every language are between 5 and 40 characters, with medians of between 15 and 20 characters. Pattern-based regex tools (e.g., syntax highlighters [37], match/mismatch aids [36]) should ensure that they perform well on regexes of these lengths. Similarly, NFA-based regex tools (e.g., railroad diagrams [37]) should accommodate NFAs with between 5 and 30 NFA states, which will cover the (25,75)th percentile range in every language.

Comprehensive regex testing appears feasible The 90th percentile of simple path family size for regexes in every language is at most 10. This means that the vast majority of regexes have at most ten simple paths through their NFA representation, so a covering set of at most ten inputs is sufficient to enumerate the “equivalence classes” of these regexes. Larson and Kirk’s basis path-based approach [220] would yield even fewer inputs. Thus, exhaustive representative input generation is quite feasible for most regexes. This is not currently a feature in existing popular regex tools, and we recommend that they incorporate it as a cheap but potentially valuable feature.

Changing regex engines for ReDoS Our findings in this dimension can inform the design of the next generation of regex engines. We apply some of these findings in Part III.

First, we report that *a static DFA-based matching algorithm is feasible for the vast majority of regexes* (Figure 5.7). The bottom 90% of regexes have a blow-up factor of 2.5–3.75 in every language, implying that constructing and storing the DFA will not cost much more than the NFA would. A naive DFA approach would offer a guaranteed linear-time solution in the size of the original regex for 90% of regexes.

Second, it appears that *super-linear regexes are common* in any programming language that uses a Spencer-style engine. Encouragingly, because we found that fewer than 5% of regexes use super-linear features (backreferences, lookaround assertions) in any programming language, Thompson’s algorithm might be applied to almost all regexes in every programming

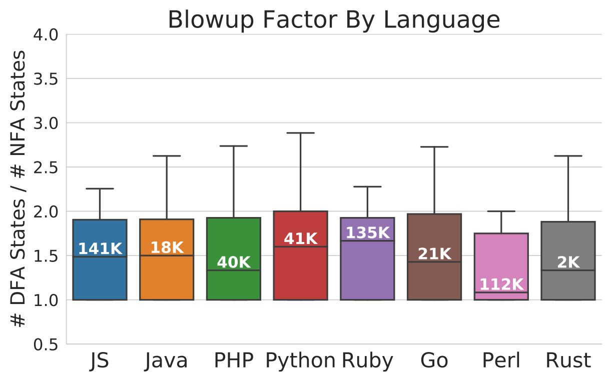


Figure 5.7: ***Distribution of DFA blowup from each programming language.*** *Whiskers are (10, 90)th percentiles. Outliers are not shown.*

language. This change would address most ReDoS vulnerabilities, albeit with potential portability problems (Chapter 7). Eliminating support for super-linear features seems infeasible in languages that already support them, but a hybrid engine that uses Thompson's algorithm where possible might be effective. This approach has previously been taken by `grep` [163].

Lastly, were regex engine designers to incorporate Thompson's algorithm, as have the designers of Rust and Go, they should consider its average cost. This cost depends on the number of transitions that must be considered as the algorithm updates its current state-set. In most programming languages the 90th percentile NFA outdegree density is no larger than that of the regexes in Rust (0.38) and Go (0.44), so lessons learned in Rust and Go may be applicable to other programming languages. However, in Java the 90th percentile NFA outdegree density is much higher, roughly 0.75. Thus, many languages can adopt a Thompson-style engine by referencing the approach in Rust and Go, but in Java more careful consideration may be required (Figure 5.5d).

5.10 Threats to validity

Internal validity As noted in §5.5.1, our regex extraction methodologies were liable to both false positives (non-regexes included) and false negatives (real regexes excluded). Although regrettable, we do not believe that these inaccuracies systematically biased our corpus.

When the modules we studied accepted external regexes (e.g., the third case in Listing 9), our program instrumentation approach would capture any regexes specified through API calls in the test suite. These "test" regexes might resemble the other regexes in the module not because they would be similar in production usage, but because they were authored by the same developer(s) who wrote the other regexes in the module. Within a given module, the regexes extracted through static analysis and program instrumentation might have similar characteristics not because of intrinsic similarities but rather because of developer biases. We hope, however, that we sampled a diverse enough set of modules to observe many different developers' styles for regexes.

We considered all unique regexes equally, deduplicating the regexes by their pattern (string representation). Focusing on the characteristics of subsets of our corpus, e.g., popular regexes, user-facing regexes, or regexes used in software testing, could be a topic for future study.

External validity Part of the purpose of our study was to address threats to external validity in prior research, by testing whether the regex extraction methodology biased regex corpuses (§5.5) and whether previous empirical regex findings generalized to regexes written in other languages (§5.6). We performed our experiments in the context of open-source software modules. The generalizability of this approach to other software — e.g., applications or closed-source software — is yet to be determined. Given the many programming languages considered in our analyses, we would be surprised if our findings did not generalize to regexes in other general-purpose programming languages. However, it is not clear whether they will apply to other pattern-matching contexts, e.g., to the regexes used in firewalls and intrusion detection systems [127, 288].

At each stage in our analysis (Figure 5.2), some regexes "leaked out." For example, we could not translate some regexes into the C# syntax. The losses were generally acceptable — for all but one of the metrics our measurements included at least 90% of the regexes. The exception was the worst-case Spencer analysis, for which we could measure only about 80% of the regexes. The missing regexes might have different characteristics, e.g., relying on unusual features.

Construct validity Table 5.2 summarizes the metrics we used to characterize regexes. Most of these metrics, or their relatives, have been applied in prior work, and measure fundamental aspects of regexes. The new metrics we introduced are based on factors considered by existing regex engines.

We considered but omitted two metrics considered by prior work [139, 332]. First, we do not generate mismatching strings for the language, although these may be of similar interest for testing purposes. These could be generated in a similar way by first taking the complement of the regex. Second, we do not attempt to label a regex based on the set of strings that it matches, e.g., "a regex for emails". The specific string language that a regex matches is an application concern. Our metrics are instead intended to characterize the components with which an engineer chose to construct a regex.

Performance portability. Our results here assume that our super-linear regex detector ensemble is effective. These detectors were designed with the naive Spencer-style regex engines in mind ("Slow family") and might miss super-linear behavior for regexes in the Medium and Fast families. For example, it is not clear whether the defenses of PHP and Perl are sound or simply effective against these detectors' inputs. See Chapter 9 for more details.

Part III

Evaluating Approaches to Address ReDoS

Outline and summary

"But if it is broke..."

–Anon.

In Part II of the dissertation, I presented empirical results showing that super-linear regexes are commonly used in practice. The implication of these findings is that ReDoS may be a widespread security vulnerability. In Part III of the dissertation, I evaluate a range of solutions that engineers can use to address ReDoS in their server-side applications.

To make this evaluation systematic, let us recall the ReDoS Conditions from §2.5:

1. **Multi-client service**: The victim operates a service that handles requests from multiple clients, permitting a malicious client to impact the experience of other clients.
2. **Super-linear regex engine**: The victim application uses a regex engine for which certain regex matches exhibit super-linear time complexity.
3. **Super-linear regex on untrusted input**: The victim application uses a regex that exhibits super-linear worst-case behavior in its regex engine, and the candidate strings on which the regex is used are not adequately sanitized.
4. **No safeguards**: The victim does not have appropriate safeguards in place to cap a client's resource usage.

I assume that software engineers will continue to use regexes to process user-defined input (ReDoS Condition 1). Thus, the solution space is determined by addressing one of the three remaining conditions. As a solution to ReDoS, engineers can attempt to eliminate *ReDoS Condition 2* by improving the worst-case complexity of the regex engine; or *ReDoS Condition 3* by sanitizing input or eliminating super-linear behavior in their user-facing regexes; or *ReDoS Condition 4* by adding safeguards to cap a client's resource usage.

The subsequent chapters in this section consider each of these solutions in detail. Chapter 6 shares the results of the first empirical study of super-linear regex refactoring (ReDoS Condition 3). Chapter 7 evaluates the feasibility of improving the worst-case complexity of a Spencer-style regex engine by the simple expedient of replacing it with a linear-time regex engine (ReDoS Condition 2). Chapter 8 examines optimizations to the problematic Spencer-style backtracking NFA simulation that improve its worst-case complexity through memoization (ReDoS Condition 2). Finally, Chapter 9 considers the effectiveness of various caps on resource utilization that can be introduced within a regex engine or software runtime (ReDoS Condition 4).

Based on this analysis, Table 5.7 summarizes the points in the solution space that we evaluate. These solutions offer varying degrees of practicality and ReDoS-proofing. Some involve application-level changes, others require changing the regex engine in some way, and some

Table 5.7: This table presents the points in the ReDoS solution space that we evaluate. The first approach examines the feasibility of application-level changes. The second and third approaches focus on fundamental changes to the regex engine. The final approach considers changes to the regex engine or application framework/runtime to cap client resource usage.

Approach	**ReDoS Condition**	**Analyzed in**
Refactor application	Condition 3	Chapter 6
Alternative match algorithm	Condition 2	Chapter 7
Optimized match algorithm	Condition 2	Chapter 8
Introduce resource caps	Condition 4	Chapter 9

call for both. After considering these solutions individually, Part IV compares them and provides the conclusions of this dissertation.

Chapter 6

Application-level refactoring

6.1 Summary

An engineer might address a ReDoS vulnerability in their software by (1) identifying it, and then (2) repairing it. This approach addresses ReDoS Condition 3: exposing a super-linear regex to untrusted input. In this chapter I discuss experiments on the effective identification and repair of ReDoS regexes.

Methodology In Chapter 4, I described the collection of a large corpus of regexes extracted from JavaScript and Python modules. Here I use that data in two experiments. First, I empirically evaluate the effectiveness of the regex ambiguity anti-patterns that software engineers currently use to *identify* super-linear regexes (§2.5.2.4). Then, I disclosed 284 potential ReDoS vulnerabilities and observed the strategies that software engineers followed to *repair* these super-linear regexes in their software.

Findings Using automated implementations of engineering heuristics, we found that the conventional wisdom embedded in ambiguity anti-patterns is prone to false positives. Many regexes fit the mold of an ambiguity anti-pattern but fail other criteria for super-linearity. In terms of repair strategies, we found that engineers fix super-linear regexes using one of three techniques: trimming the input, revising the regex, or replacing it with alternative logic. Among these techniques, revising the regex was the most common, regardless of whether engineers were previously aware of the others. Fixing super-linear regexes is relatively difficult: many of the engineers' first attempts were unsuccessful, and not all the applied fixes fully resolved the ReDoS vulnerability as the engineers intended.

Statement of Attribution The material presented here is excerpted from [139]. It was performed in conjunction with the work described in Chapter 4, which influenced the experimental design.

6.2 Study design and research questions

In this chapter I consider research questions along two themes.

First, we study whether regex ambiguity anti-patterns are a useful signal for super-linear regexes. As we discussed in §2.5.2.4, avoiding regex ambiguity anti-patterns is a technique used in practice to prevent ReDoS, but the effectiveness of this approach has not been evaluated. In this investigation we check the validity of this conventional wisdom.

Theme 1: Detecting ReDoS.
RQ1: Do ambiguity anti-patterns signal super-linear regexes?

Second, we study how ReDoS vulnerabilities are repaired in practice. The scientific and engineering communities lack an understanding of the preferred strategies to repair super-linear regexes. An empirical understanding will guide other engineers when they fix ReDoS vulnerabilities, and can be applied by researchers to automatically propose palatable patches.

Theme 2: Fixing ReDoS.
RQ2: How have software engineers repaired ReDoS vulnerabilities?
RQ3: What ReDoS repair strategies do software engineers prefer?
RQ4: How effective are software engineers' manual repairs?

6.3 RQ1: Do ambiguity anti-patterns signal SL regexes?

As discussed in §2.5.2.4, software engineers have conventional wisdom about what makes a regex super-linear. The ambiguity anti-patterns that engineers use describe a *necessary* condition for super-linear behavior, but not a sufficient one. They may have false positives, regexes that match the anti-pattern but will not exhibit super-linear behavior. But if their false positive rates are low, then they may be a useful heuristic for software engineers during composition and code review.

In this experiment, we operationalize these anti-patterns and then measure their false positive rate.

6.3.1 Methodology

Three regex ambiguity anti-patterns We know of three regex ambiguity anti-patterns. The first, nested quantifiers or "star height", is discussed in many places including [163, 177, 312]. The second is rather more vague: "watch out when...[different] parts of the [regex] can match the same text" [179]. These anti-patterns are a primitive and informal description of regex ambiguity (§2.5.2.1), which is itself a necessary condition for super-linear regex behavior.

We manually identified three distinct ways that such ambiguity arose in the super-linear regexes from our corpus. The three ways all include a quantifier so that they will have unbounded ambiguity, and rely on the use of one of the three other primitives of regular languages to create an ambiguous sub-pattern (Grammar 2.1). One involves the combination of a quantifier and nesting, i.e., the "star height" anti-pattern. The second and third both involve "matching the same text", by means of (a) the combination of a quantifier and disjunction; or (b) the combination of a quantifier and concatenation.

The first anti-pattern is *star height > 1*, i.e., nested quantifiers. This leads to super-linear behavior when the same string can be consumed by an inner quantifier or an outer one, as is the case for the string "a" in the regex `/(a+)+/`. In this case, the star height of two results in two choices for each pump, with worst-case exponential behavior on a mismatch. This anti-pattern is commonly used in practice through the `safe-regex` tool [312].

The second anti-pattern is a form of ambiguity that we call *Quantified Overlapping Disjunction* (QOD). An example of this anti-pattern is `/(\w|\d)+/`. This quantified disjunction has two nodes that overlap in the digits, 0-9. On a pump string of a digit there are two choices of which group to use, with worst-case exponential behavior on a mismatch.

The third anti-pattern is a form of ambiguity that we call *Quantified Overlapping Adjacency* (QOA). For an example of this anti-pattern, consider the regex `/\s*\s*/`. The two quantified `\s*` nodes overlap and are adjacent. This anti-pattern has worst-case polynomial behavior on a mismatch. See also the description given in Figure 2.11a.

Star height is the most well-known anti-pattern in the ecosystems we studied. The npm module `safe-regex`, which tests for this anti-pattern, is used millions of times each month. Software engineers frequently referenced it in our email conversations, and it is used in the well-known `eslint-plugin-security` plugin for eslint. The other anti-patterns appear in reference works on regexes, but we do not know how well known they are in practice.

Operationalizing the anti-patterns We implemented tests for the presence of these anti-patterns using the `regexp-tree` regex AST generator [50].[1]

- To measure *star height* we traverse the AST and maintain a counter for each layer of nested quantifier: +, *, and ranges where the upper bound is at least 25.[2]
- To detect *QOD* we search the AST for quantified disjunctions. When we find them we enumerate the Unicode ranges of each member of the disjunction and test for overlap.
- To detect *QOA* we search the AST for quantified nodes. We test for pairs of quantified nodes that have an overlapping set of characters.

False positives: There are two ways in which these operationalizations may incorrectly flag

[1]The *safe-regex* tool [312] is implemented incorrectly, so we used our own implementation. We provided a patch to the author of *safe-regex*.

[2]Groups with lower quantifications do not readily exhibit super-linear behavior.

a linear-time regex as super-linear.

1. Ambiguity is a necessary but not sufficient condition for super-linear behavior. In keeping with the descriptions of the anti-patterns, none of our tests includes a check for the second condition, a mismatch-triggering suffix. For example, if a "." means "match any character", then the regex `/.*.*/` is ambiguous but cannot be made to mismatch. It will run in linear time.
2. Again in keeping with written descriptions, our implementations may mark an unambiguous regex as ambiguous. For example, the regex `/(ab*c)*/` has star height 2 but is unambiguous.

False negatives: For QOD and QOA our prototypes only consider AST nodes containing individual quantified characters (e.g., `/\d+/` or `/.*/`, not `/(ab)+/`). Some ambiguous regexes will thus not be detected.

6.3.2 Results and analysis

The results of applying our anti-pattern tests to our npm and pypi regex datasets are shown in Table 6.1. We discuss this table in light of the sources of the ambiguity anti-patterns' true and false positives, as well as their false negatives.

True positives Columns 2 and 3 show that our operationalizations of the ambiguity anti-patterns were able to identify the super-linear regexes. We found at least one anti-pattern in most of the super-linear regexes (81-86%). The anti-pattern proportions are also roughly accurate relative to the measured worst-case behavior (Chapter 4). Super-linear regexes with the (polynomial) QOA anti-pattern are much more common than the (exponential) QOD and Star Height anti-patterns.

False positives Columns 4 and 5 show that the ambiguity anti-patterns exhibited many false positives — linear-time regexes that contain ambiguity anti-patterns according to the definitions we used. This false positive rate seems too high for these ambiguity anti-patterns to be useful without further refinement.

False negatives As expected, our operationalizations of the anti-patterns did not identify all super-linear regexes. We manually inspected a random sample of 70 of the unlabeled super-linear regexes from npm modules, and confirmed that 65 of them contained one or more of these anti-patterns. They involved ambiguous constructions too complex for our current anti-pattern test tools to detect. For example, our tools do not identify as QOD the regex `/(\d|\d\.|\.\d)+/`, which is ambiguous when the elements of the disjunction are

*Table 6.1: **Utility of the super-linear regex anti-patterns**, as measured by our operationalizations. For each anti-pattern, we present the number of super-linear regexes that had this pattern in each ecosystem, and then the false positive rate. For the false positive rate we rely on the super-linear regex detectors (§4.3.1) as ground truth. For example, in npm 12% of the super-linear regexes had star height > 1, but 94% of the regexes with star height > 1 were linear-time. As some regexes have multiple anti-patterns, the final row eliminates double-counting.*

Anti-pattern	**Number of super-linear regexes**		**False positive rate**	
	npm	**pypi**	**npm**	**pypi**
Star height > 1	443 (12%)	62 (2%)	94%	98%
QOD	40 (1%)	6 (1%)	97%	95%
QOA	2,548 (71%)	555 (79%)	90%	94%
Totals	2,901 (81%)	604 (86%)	91%	95%

repeated. To identify this, our algorithm could be extended to apply "quantifier unrolling" to capture ambiguity.

6.4 RQ2: How have software engineers repaired ReDoS vulnerabilities?

Here we provide the first characterization of the fix approaches engineers have taken when addressing ReDoS vulnerabilities. This study tells us which fix strategies engineers currently choose when left to their own devices. This effort is a preliminary step towards our larger study of preferred fix strategies, described in §6.5.

6.4.1 Methodology

We were interested in thorough reports describing super-linear regexes and how engineers fixed them. We thus searched for ReDoS in security databases using the keywords "Catastrophic backtracking", "REDOS", and "Regular expression denial of service". We searched both the CVE database and the Snyk.io database.[3]. We used any reports with two properties: (1) the report used the definition of ReDoS given in §2.5.1; and (2) the vulnerability was fixed and the report included a link.

For each vulnerability report, we manually categorized the fix strategy the engineers took. If a fix used more than one strategy (e.g., both Trim and Revise), we counted it under each

[3]Snyk.io's database tracks vulnerabilities in popular module registries, including npm and pypi.

Table 6.2: ***The three super-linear regex repair strategies.*** *Examples of the fix strategies for a super-linear regex for emails that we reported in the Django web framework (CVE-2018-7536). The engineers chose to fix this ReDoS vulnerability using the algorithm described in "Replace".*

	Example
Original	`/^\S+@\S+\.\S+$/`
Trim	if 1000 < input.length: throw error
	else: test with existing regex
Revise	`/^[^@]+@([^\.@]+\.)+$/`
Replace*	Custom parser:
	(1) Exactly one @ must occur, at neither end of the string, and
	(2) There must be a '.' to the right of the @, but not adjacent.

of the used strategies.

6.4.2 Results and analysis

ReDoS Reports We identified 45 unique historic ReDoS reports (condition 1) across the CVE and Snyk.io databases. The earliest report was from 2007 and the most recent from 2018. 37 of these reports included fixes (condition 2). Three of these reports were unique to the CVE database, 27 were unique to the Snyk.io database, and 7 appeared in both databases.

Fix strategies Three fix strategies were typical in these reports.

Trim Leave the regex alone, but limit the size of the input to bound the amount of backtracking.
Revise Change the regex.
Replace Replace the regex with an alternative strategy, e.g., writing a custom parser or using a library.

Table 6.2 gives an example of each fix strategy. Only the Revise strategy was discussed in any of the reference texts on regexes we reviewed [156, 163, 177, 179].

Table 6.3 summarizes the results from this study, as well as the subsequent studies on new fixes (§6.5) and on fix correctness (§6.6). In the first row, we can see that engineers in the historic dataset commonly Trimmed, Revised, or Replaced, each more than 20% of the time.

Table 6.3: ***Frequency of repair strategies for historic and new vulnerabilities.*** *Examples of each super-linear repair strategy are given in Table 6.2. Some of the new fixes used more than one strategy.*

		Trim	Revise	Replace	Total
Historic	*Fix approach*	8	18	11	37
	Unsafe fixes	1	2	0	3
New	*Fix approach*	3	35	15	48
	Unsafe fixes	0	0	0	0

6.5 RQ3: What ReDoS repair strategies do software engineers prefer?

In §6.4 we described the three common fix strategies engineers used in the historic ReDoS reports. However, we do not know whether these engineers knew every strategy, and thus we cannot be sure that they preferred one strategy over another. In this experiment, we describe the fix strategies taken by engineers who were informed of all of the strategies.

6.5.1 Methodology

To learn what fix strategies engineers would take if they knew all of the options, we needed to convince a sizable group of engineers to fix super-linear regexes. Because we felt that the maintainers of popular modules would be more likely to fix problems therein, we examined the use of super-linear regexes in all npm and pypi modules downloaded more than 1000 times per month. See Figures 4.2 and 4.3 for the effect of this filter. We filtered these modules for those whose super-linear regex(es) were clearly a ReDoS vector based on a manual inspection, and then contacted the maintainers of those modules by email with a vulnerability disclosure.

In our disclosures, we included the following information:

1. The super-linear regex(es) and the files in which they lay;
2. The degree of vulnerability (§4.4) for each regex;
3. Each malign input, with prefix, pump, and suffix;
4. The length of an attack string leading to a 10-second timeout on a desktop-class machine; and
5. A description of the three fix strategies we observed in the historic data (Table 6.2), with links to two example repairs of each type.

6.5.2 Results

After applying our two-stage filter, we disclosed 284 vulnerabilities across both ecosystems to the module maintainers. 48 (17%) of our disclosures have resulted in fixes so far. Prominent projects that applied fixes based on our reports include the Hapi and Django web frameworks and the MongoDB database.

The fix strategies the maintainers chose are shown in Table 6.3. Compared to the historic fix strategies, engineers exposed to examples of all three fix strategies still preferred Revise to Trim. The use of Revise rose from 49% to 73%, while the use of Trim fell from 22% to 6%. The use of Replace remained around 30%. Clearly these engineers preferred Revise when they considered all three choices. Anecdotally, in our discussions with engineers they indicated that trimming seemed like a “hack” when compared to the other two strategies.

6.6 RQ4: How effective are software engineers’ manual repairs?

Any one of fix strategies in Table 6.2 can go awry. To **Trim**, engineers must solve a Goldilocks problem: trim too short and valid input will be rejected, trim too long and the vulnerability will remain. To **Revise**, engineers must craft a linear-time regex that matches a language close enough to the original that their APIs continue to work. Lastly, to **Replace**, engineers must write a parser for the input that matches an equivalent or related language.

In this study we examine the correctness of engineers’ fixes.

6.6.1 Methodology

Here is the fix safety classification scheme that we used.

- We called a **Trim** fix unsafe if the maximum allowed input length can still trigger a noticeable slowdown.
- We compared the input limit to the lengths of malign inputs derived using the super-linear regex identification procedure from Chapter 4.
- We called a **Revise** fix unsafe if it was labeled vulnerable by our super-linear regex identification procedure.
- We called a **Replace** fix unsafe if the replacement logic was super-linear in complexity based on manual inspection.

6.6.2 Results

Our findings for the effectiveness of the historic and new fixes are summarized in Table 6.3. Several of the historic fixes were incorrect. The new fixes were uniformly correct (nearly all engineers asked us to review their fixes before publishing their changes).

Trim 1 of the 8 historic Trim fixes was unsafe. The initial choice of length limit was too generous and the regex remained vulnerable for two years before this was discovered and the length limit lowered.

Revise 2 of the 18 historic fixes resulted in a revised, but still super-linear, regex. One of these was replaced before our study. We discovered the other in Chapter 4 and disclosed it in §6.5 before performing this portion of our study.

Replace We manually inspected the fixes that used the Replace strategy to gauge their complexity. All appeared sound, relying on one or more linear scans of the input.

Testing their fixes Regardless of the fix strategy, engineers did not usually include test cases for their changes. In the historic dataset, 8 of the 37 fixes included tests. In the new dataset, 18 of the 48 fixes included tests. We did not discuss the lack of tests with engineers, but we conjecture two possible reasons for the lack of tests. First, a regex is an implementation detail, so a test tailored to the regex might be viewed as "testing the implementation" rather than "testing the interface" of a function. Second, introducing tests that validate a fix would simultaneously also publicize an attack exploit affecting the dependents of older versions.

6.7 Discussion

Ambiguity anti-patterns may offer explanatory power In RQ1, we found that the ambiguity anti-patterns are common in practice, and these anti-patterns often occur without meeting the remaining conditions for super-linear worst-case behavior. They should not be used alone as a super-linear regex detection tool.

However, we believe that the regex ambiguity anti-patterns may have value when used in conjunction with the accurate super-linear regex detectors proposed by researchers. Many of the existing super-linear regex detectors analyze automata, not regexes. When these tools report super-linear behavior caused by ambiguity in the NFA, this may not help engineers diagnose the cause in the right modality — the pattern that the engineer must modify.

This is where the anti-patterns might be useful: once a true positive has been identified by an automaton-based analysis, the anti-patterns may assist engineers in diagnosing and repairing the original regex. We believe that understanding the usefulness of the ambiguity anti-patterns during regex repair is a promising direction for future work. Recent theoretical descriptions of regex ambiguity in the modality of the regex pattern might also prove useful during regex repair [97, 317].

Automatic refactoring as an alternative to manual repair In RQ2 and RQ3, we determined the ReDoS repair strategies that engineers have taken and prefer to take. The general trend was towards *revising* the regex and *replacing* the regex, and away from *truncating* or otherwise sanitizing the candidate string. But we found that manual repair was difficult, with several examples of incorrect repairs. In light of the findings from Mischa et al. cited earlier [238], this is not too surprising. If most software engineers are unfamiliar with ReDoS— and anecdotally, those we emailed typically were — then dealing with a ReDoS vulnerability would be an unfamiliar task, and revising a super-linear regex would thus be difficult.

Automatic refactoring strategies, focused on regex revision or replacement, may be a useful direction to address this problem. Automated refactorings could be applied to change the regex within the application, or they could intercept regex queries prior to the regex engine and dynamically refactor super-linear regexes.

There is recent work on ReDoS amelioration that has focused on automatically revising regexes. For example, van der Merwe et al. [326] proposed to apply the flow algorithm [92] to revise a K-regex R into a semantically equivalent regex R' (i.e., $L(R) = L(R')$) with the property that the NFA corresponding to R' is unambiguous (i.e., Thompson's construction yields a DFA).[4] The approach of Cody-Kenny et al. [125] is to mutate the original regex until a new regex is identified that preserves the tested subset of the regex's language but lowers its worst-case complexity. This scheme does not guarantee lowered complexity, and may struggle with under-tested regexes [332].

These regex revision approaches have focused solely on linear-time behavior, ignoring engineering considerations like readability and maintainability. For approaches like van der Merwe et al., which are language-preserving, I am not confident that readability can be maintained. The strategy of Cody-Kenny et al. does not require that the full language of the regular expression be preserved, and so their fitness function could be tuned for readability [116]. These efforts have also considered only regex equivalence up to recognition (i.e., the same regex language), ignoring the additional concern of equivalence up to parsing (e.g., preserving capture groups). Perhaps the family of revised regexes from the experiment

[4]These automatically-generated regexes may be exponentially larger than the original regexes, which merely moves the exponential behavior from the NFA simulation to the NFA construction. In light of our finding that most NFAs grow linearly when converted to a DFA in our corpus, this may not be a serious shortcoming in practice (Figure 5.7).

discussed in this chapter could be used to guide the development of future automatic repair algorithms. In particular, some of the fixes described in this chapter were not language-preserving, suggesting that van der Merwe et al. may be setting too high a bar.

Towards the automatic replacement of super-linear regexes, Medeiros et al. [236] propose to convert (all) K-regexes into Parsing Expression Grammars (PEGs), which are an unambiguous cousin of CFGs [158]. Using PEGs instead of regexes is conjectured to improve readability and testability [203], and PEGs can be parsed in linear time [158]. However, perhaps such a significant shift is unnecessary. In many of the *replace* examples we observed that a small set of string functions sufficed to match the same language of strings or a sufficiently-close one for practical purposes. Perhaps equivalent functions encoding similar string constraints as a regex can be generated automatically by drawing on techniques from the string constraints community [208].

Our own experiences fixing super-linear regexes In addition to the 48 fixes from module maintainers, we submitted 9 fixes when maintainers asked us for help. Our own experiences may illuminate some of the factors that engineers will consider when selecting a fix strategy.

The fix strategy we selected (1 Trim, 9 Revise, 2 Replace, with 3 overlaps) depended on both whether the super-linear regex was exponential or polynomial, and how identifiable the language of the regex was. When a regex was exponential or was polynomial with a large degree, the vulnerability would manifest on short malign input. We fixed these by Revising, aided by visualizations from the `regexper` tool to understand the original language and study the source of the super-linear behavior.[5] When the super-linear behavior was less severe (e.g., quadratic), we considered both Revise and Trim. When we could discern the language described by the regex, we favored Revise. When the regex's language was unclear or many regexes were applied to the same input (e.g., parsing a user agent string), then Trim became an attractive alternative. We felt an aversion to Replace because it felt overly verbose. As Wang and Stolee predicted [332], when we changed the language of the regex we found it difficult to confirm that we had not done so for the subset of strings it might receive. These projects had limited test suites, and did not always contain representative sets of strings $S_1 \subseteq L(R)$ in the language of the regex R, and $S_2 \cap L(R) = \emptyset$ outside of it.

[5]The regexper tool can be found at `https://regexper.com/`.

Chapter 7

Replacing the regex engine

7.1 Summary

An engineer might address a ReDoS vulnerability in their software by evaluating the super-linear regex using a regex engine with improved worst-case complexity. Substituting a faster regex engine for a slower one would address ReDoS Condition 2, improving all regexes' worst-case complexity by applying an alternative match algorithm. For example, in Chapter 5 (Figure 5.6) we showed that there are regex engines with distinct classes of worst-case performance. Moving from a slower engine to a faster one is a reasonable strategy for ReDoS amelioration, and is precisely the remedy that Cloudflare adopted after the incident discussed in our case study (§3.3).

Such a substitution can be effected through application-level changes or through programming language-level changes. At the application level, an engineer could adopt a 3rd-party regex engine by accessing it as a library. Within the programming language, the language maintainers could replace their regex engine implementation with that of another regex engine. If regex engines are compatible with one another, this approach is straightforward; if they vary in their syntax or semantics, then correctness concerns (i.e., backwards compatibility) may outweigh the potential security benefits.

Many regex engines support identical or similar syntaxes, and it would be natural to assume that regexes can be moved from one to another without modification. Indeed, prior work surveying 158 professional software engineers found that many engineers believe that regexes share a universal specification [237]. If true, this would make changing regex engines a natural ReDoS amelioration technique — simply switch to an alternative regex engine with improved worst-case performance. But can regexes be copy/pasted across regex engine and retain their semantic characteristics? We have anecdotal evidence that this may not be the case. Chapman and Stolee reported that some developers struggle with "[regex] inconsistencies across [programming languages]" [115]. Inconsistent regex behavior has even been reported between different implementations of the same Perl version [107]. We are not the first to observe regex portability issues, but we are the first to provide evidence of the extent and impact of the phenomenon. We use a systematic approach to identify syntactic and semantic differences between many different regex engines, including documented, undocumented, and defective regex engine behaviors.

Methodology This chapter evaluates the efficacy and risks of replacing the regex engine in three dimensions. *First*, we clarify the potential worst-case benefits of moving from one regex engine to another, refining the performance measurements presented in Chapter 5. *Second*, we experimentally identify semantic differences between regex engines, clarifying the need for a translation layer before this ReDoS amelioration approach can be safely applied. To do this, we combine the polyglot regex corpus presented in Chapter 5 with an ensemble of input generators, and compare the behavior of these regex-input pairs across many regex engines. *Third*, we empirically measure the extent to which software engineers currently use identical regexes in different regex engines. To do this, we measure regex duplication within the polyglot regex corpus, and compare those regexes to regexes sourced from the two most common Internet sources of regexes. If identical regexes are commonly used in different regex engines, this may suggest that the semantic incompatibilities we identified are sufficiently unusual that they can be deemed "acceptable risk" by engineers.

Findings We found that changing regex engines is an effective ReDoS amelioration approach for many regexes, but may introduce correctness concerns due to under-specified and incompatible differences in regex engine behavior. Though most regexes (92%) compile in different regex engines, 15% exhibit semantic differences from one engine to another. While some of these differences were documented in the corresponding regex engine documentation, others were not and were only exposed through our experiments. Beyond legitimate differences, we also identified incorrect behavior in the regex engines of JavaScript-V8, Python, Ruby, and Rust. However, despite the risks of regex non-portability, re-using regexes across regex engine boundaries appears to be a common practice.

Statement of Attribution The data presented here is largely excerpted from a paper that I presented at ESEC/FSE 2019 [141]. Some of its interpretation is new to this dissertation, as is the material in §7.5.

7.2 Study design and research questions

Like other code snippets [343], regexes may flow into software from Internet forums or other software projects. Unlike most code snippets, however, regexes can flow unchanged across programming language (and regex engine) boundaries. Regex engines (mostly) have compatible syntax, potentially lulling engineers into a false sense of semantic compatibility.

In §2.4.2 I described the PCRE semantics to which most programming languages attempt to adhere. There has been no successful specification of regex syntax and semantics; Perl-Compatible Regular Expressions (PCRE) [189] and POSIX Regular Expressions [202] have influenced but not standardized the various regex dialects that programming languages support, leading to manuals with phrases like "these aspects...may not be fully portable" [43].

No prior systematic effort had been made to evaluate these engines' actual compatibility. Consequently, measuring the extent to which regexes are truly portable from one regex engine to another is the primary burden of this chapter.

This study measures the extent to which regexes are portable across different regex engines, as measured by their different behavior in different programming languages. Applying the polyglot regex corpus described in §5.6.1.2, we explore the *regex portability problems*: the problems that developers may face when they take a regex written for one regex engine and use it in another. The most pernicious of these problems take the form of *false friends*: regexes that compile in both regex engines (syntactic compatibility) but match different strings (semantic incompatibility). When moved to a linear-time regex engine, formerly super-linear regexes might change from being ReDoS vulnerabilities to being logical errors.

With these ideas in mind, this study investigates three research questions.

RQ1: To what extent does moving from one regex engine to another offer consistent performance benefits?

RQ2: To what extent do syntactically-compatible regexes exhibit semantic differences between regex engines, and why?

RQ3: Despite these differences, to what extent are a common core of regexes used across regex engine boundaries?

7.3 RQ1: To what extent does moving from one regex engine to another offer consistent performance benefits?

In §5.7 we identified three distinct families of regex engines in terms of their worst-case performance: Fast, Medium, and Slow. We made this determination based on the proportion of all regexes that (compiled and) exhibited super-linear behavior in the regex engines under consideration. However, merely measuring the proportion of regexes may not suffice. The overall proportion alone does not indicate whether the improvement is *uniform* — in other words, it is not clear whether some regexes may exhibit worse performance in one of the "faster" regex engines.

7.3.1 Methodology

We answered this question by leveraging the per-language performance information collected for each regex as part of the study described in Chapter 5. We performed a pairwise comparison of each regex's performance in each language, determining whether the match had the same worst-case complexity, completed more quickly, or completed more slowly. If regex performance improves monotonically from the "Slow" to "Medium" to "Fast" regex engine families, we can conclude that the ReDoS amelioration of changing regex engines is suitable from a security standpoint. Conversely, if regex performance degrades, changing regex engines may be unsuitable.

7.3.2 Results

In brief, we found that regex performance does improve monotonically when moving from the "Slow" to "Medium" to "Fast" regex engine families. Figure 7.1 shows the frequency with which regexes exhibit *worse* behavior in one of a pair of languages. Lighter cells in the heatmap correspond to no performance degradation, while shaded cells indicate that some regexes performed worse from the source language (columns) to the destination language (rows). For example, in this figure we can see that the ~10% of regexes that are super-linear in Java and in JavaScript (cf. Figure 5.6) are the *same* regexes.

To interpret this figure in more detail, recall that JavaScript, Java, Python, and Ruby were deemed "Slow", PHP and Perl were "Medium", and Go and Rust were "Fast". The analysis then proceeds as follows:

- Comparing regex performance within the "Slow" family (JavaScript, Java, Python, and Ruby), regex performance was nearly identical between JavaScript, Java, and Python, and Ruby appears to offer slight performance benefits compared to its siblings.
- Comparing regex performance within the "Medium" family (PHP and Perl), we have the interesting result that 1-3% of regexes worsen in each direction. This implies that the resource caps and/or optimizations applied in these regex engines are non-equivalent and succeed on different groups of super-linear regexes. See Chapter 9 for further analysis.
- Comparing regex performance within the "Fast" family (Go and Rust), we see no performance degradation.
- When moving regexes from the "Slow" family to the "Medium" family, nearly all regexes exhibited no performance degradation – regex evaluations retained their prior performance or ran more quickly.
- Similarly, when moving regexes from the "Slow" or "Medium" families to the "Fast" family, all regexes exhibited no performance degradation – regex evaluations retained their prior performance or ran more quickly.

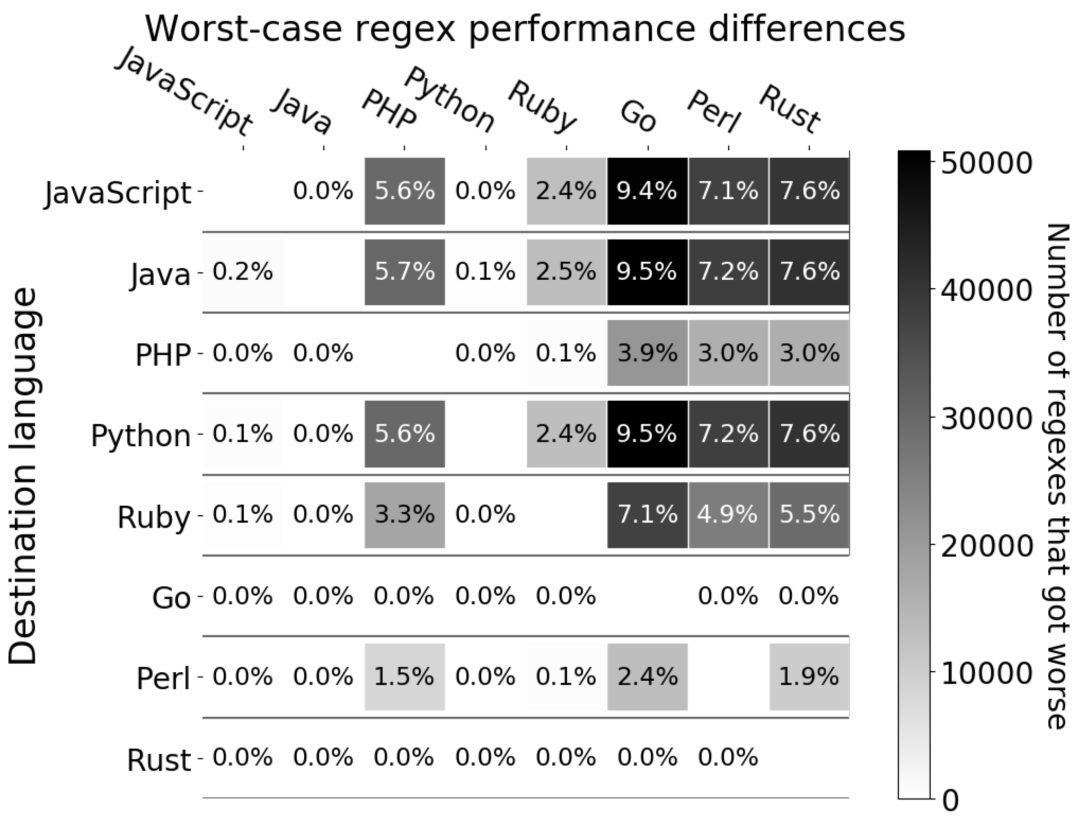


Figure 7.1: ***Faster regex engine families provide uniform performance improvement.*** *Pairwise view of regex performance differences between regex engines. The asymmetry in the chart is due to the regex engine performance families. Cells are colored according to the number of regexes that exhibit worse behavior in the destination (row) than the hypothetical source (column). Lighter rows are safer destinations; the individual cells contain the percent of the regexes supported in that language pair whose worst-case performance is worse in the destination. For example, regexes do not perform any worse in JavaScript than Java, but 8% of regexes perform worse when moved from Rust to JavaScript.*

7.4 RQ2: To what extent do syntactically-compatible regexes exhibit semantic differences between regex engines, and why?

In answer to RQ1, we found that engineers will obtain uniform performance improvement when moving from a slower family of regex engines to a faster one. This movement may, however, be accompanied by correctness risks. Although the regexes considered in that experiment were *syntactically* compatible between two regex engines (i.e., they could be used without modification), and although changing engines yielded *performance* improvements relevant to ReDoS, it is possible that the *semantics* of the regex matches changed.

In this experiment we measure the extent to which regex semantics are preserved when moving from one regex engine to another. These regex engines all purportedly follow PCRE's leftmost-greedy semantics, but the extent to which they deviate has not previously been evaluated.

7.4.1 Methodology

The general methodology of this experiment was as follows: We took every regex, generated candidate strings for it, recorded its match behavior on each string in each regex engine, and performed a pairwise comparison of the match behavior for each regex-input pair in each programming language. The details of our methodology are given next.

Regexes considered in each programming language pair When we compare a regex's behavior in a pair of programming languages, we use the subset of the regex corpus that is syntactically valid in that pair. Most comparisons are on the majority of the corpus — 76% of the corpus was syntactically valid in every language, and 88% were syntactically valid in all but Rust.

Evaluating regex behavior We tested the behavior of a regex in each regex engine by means of a small program written in each of the 8 programming languages from which the corpus was derived. Each of these programs accepts a regex pattern and candidate string, queries the regex engine for a match via the appropriate programming language API, and reports the result. We used the most comprehensive of the match queries: a parse of the first matching substring, i.e., a partial match including the matched substring and capture groups. On a match, we recorded (1) the substring that matched, and (2) the contents of all capture groups. We used the default match flags in each programming language. Table 7.1 lists the programming language versions used in our tests.

Table 7.1: ***The language versions used in our regex portability experiments.*** *Summary of programming language (and thus regex engine) versions and documentation used in our experiments and analysis. Most versions are the default for Ubuntu 16.04.*

Language	Version information	Documentation
JavaScript	Node.js v10.9.0 (V8 v6.8)	[147, 148]
Java	Oracle JDK 8	[130]
PHP	PHP 7.4.0-dev (cli)	[181]
Python	Python 3.5.2	[160, 216]
Ruby	Ruby 2.3.1p112	[99]
Go	Go v1.6.2	[176]
Perl	Perl v5.22.1	[20, 218, 323]
Rust	Rust v1.32.0 (nightly)	[146]

Input generation In search of an interesting set of inputs, we created an ensemble of five state-of-the-art regex input generators: Rex [328], MutRex [73], EGRET [220], ReScue [297] and Brics [252]. These generators produce either matching strings (Rex, Brics) or both matching and mismatching strings (MutRex, EGRET, ReScue). We used Rex, MutRex, and EGRET unchanged. We modified ReScue to use the strings it explores in its search for SL inputs. We modified Brics to generate random input subsets, not infinitely many inputs.

We wanted these inputs to provide good regex automaton coverage. Wang and Stolee showed that 100 Rex-generated inputs yield about 50% regex coverage [332], so we requested 10,000 inputs from each input generator with a time limit of 10 seconds. Table 7.2 summarizes the number of unique inputs generated for each regex. Most regexes were tested with more than 1,000 unique inputs. These generators perform well on K-regexes (about 95% of regexes), but do not consistently support regexes that use extended features (e.g., backreferences or lookaround assertions). The effect of this in our experiment is that E-regexes may not be tested on a meaningful set of inputs, e.g., inputs that are in the language of a regex with a backreference.

Additional experimental parameters These experiments were performed on a 10-node cluster of server-class nodes running Ubuntu 16.04.

7.4.2 Results

Table 7.2 summarizes our results. About 15% of regexes participated in at least one difference witness, and among the language pairs we observed all three classes of witnesses. In Table 7.2 and Figure 7.2 we report the number of distinct regexes participating in the difference witnesses rather than the number of distinct witnesses themselves, because we expect that many of the witnessing inputs for a given regex are members of an equivalence

Table 7.2: **Summary statistics for the semantic portability experiment.**

Metric	Value
Percentile inputs per regex: 25th – 50th – 75th	1,057 – 2,410 – 2,510
Regexes with any difference witnesses	15.4% (82,582)
Regexes with any match witnesses	8.1% (43,417)
Regexes with any substring witnesses	4.2% (22,597)
Regexes with any capture witnesses	7.5% (40,457)

class on which a difference manifests.

A more detailed description of the semantic differences between languages is presented in Figure 7.2 as a heatmap. The cells are colored proportional to the number of regexes that have any witness of a difference between that pair of languages. The three numbers in the cell denote the percent of regexes with match, substring, and capture witnesses for that pair of languages. As can be seen in Figure 7.2:

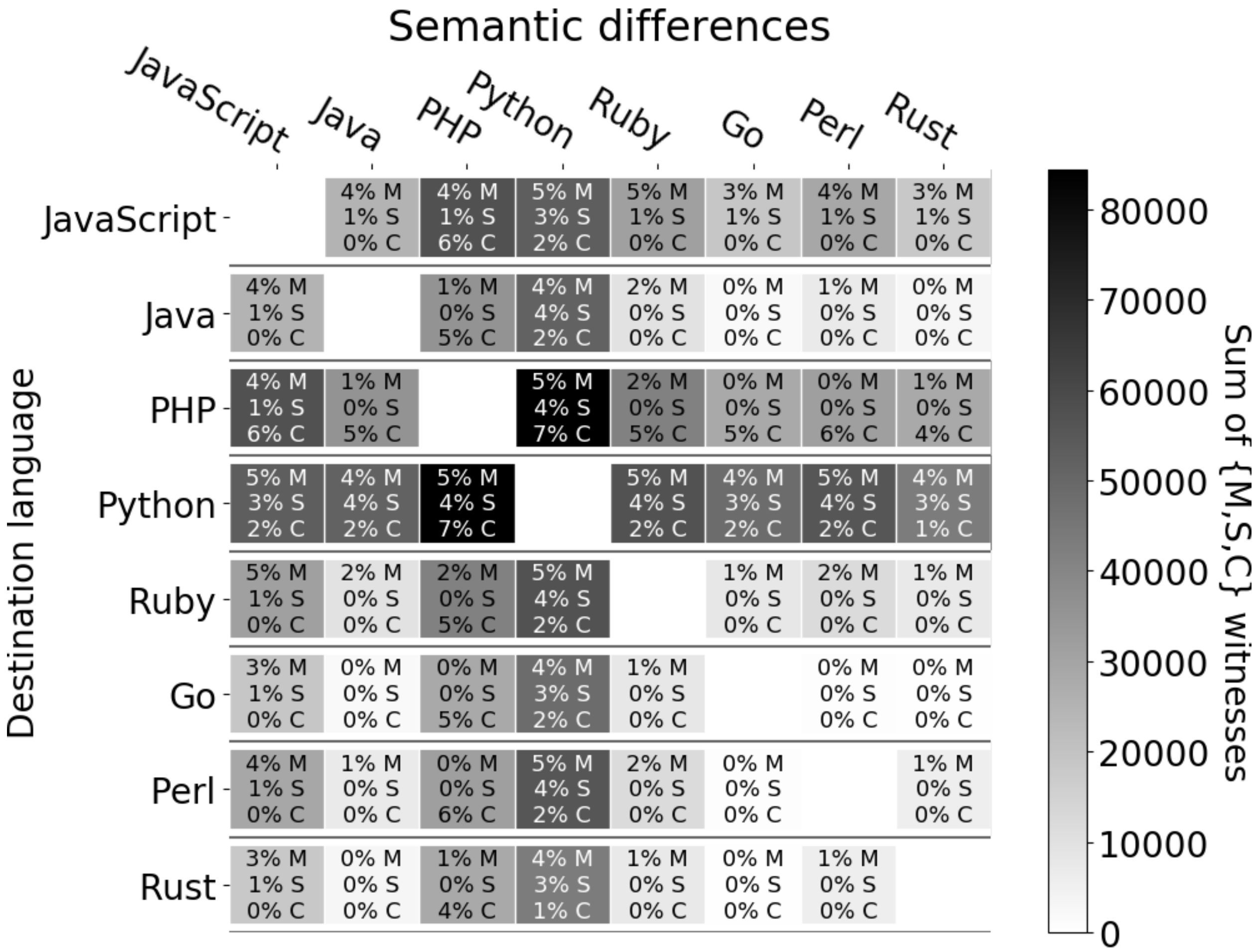
- There are many language pairs with match witnesses.
- PHP and Python are the primary sources of substring witnesses.
- PHP is the primary source of capture witnesses.

The behavior between trios of languages is not always directly comparable in Figure 7.2. This difficulty is caused by slight variation in the sets of regexes considered in each programming language pair, due in turn to our experimental design. Recall that we compared the regexes that were syntactically valid in each language pair, rather than considering solely the regexes that were valid in every programming language.

7.4.3 Analysis

Implications for regex engine replacement as a ReDoS amelioration In answering RQ1, we found that changing regex engines will improve worst-case regex performance. Our investigation of RQ2 reveals that doing so may cause thousands of regexes to exhibit different semantics (Figure 7.2). Although these various regex engines follow PCRE syntax, they vary sufficiently that careful porting or a dedicated translation layer would be necessary in order to follow this ReDoS amelioration strategy.

Some regex engines are closer in semantics than others. For example, between the "Slow" Java and the "Fast" Go and Rust regex engines, *all* regexes that are syntactically compatible will improve in performance without exhibiting different semantics (Figure 7.2). In contrast, moving from the "Slow" JavaScript regex engine to the "Fast" regex engines will cause up to 3% of the regexes to exhibit different semantics behavior. Our data permit similar conclusions to be drawn from each of the origin regex engines of interest.



	JavaScript	Java	PHP	Python	Ruby	Go	Perl	Rust
JavaScript		4% M 1% S 0% C	4% M 1% S 6% C	5% M 3% S 2% C	5% M 1% S 0% C	3% M 1% S 0% C	4% M 1% S 0% C	3% M 1% S 0% C
Java	4% M 1% S 0% C		1% M 0% S 5% C	4% M 4% S 2% C	2% M 0% S 0% C	0% M 0% S 0% C	1% M 0% S 0% C	0% M 0% S 0% C
PHP	4% M 1% S 6% C	1% M 0% S 5% C		5% M 4% S 7% C	2% M 0% S 5% C	0% M 0% S 5% C	0% M 0% S 6% C	1% M 0% S 4% C
Python	5% M 3% S 2% C	4% M 4% S 2% C	5% M 4% S 7% C		5% M 4% S 2% C	4% M 3% S 2% C	5% M 4% S 2% C	4% M 3% S 1% C
Ruby	5% M 1% S 0% C	2% M 0% S 0% C	2% M 0% S 5% C	5% M 4% S 2% C		1% M 0% S 0% C	2% M 0% S 0% C	1% M 0% S 0% C
Go	3% M 1% S 0% C	0% M 0% S 0% C	0% M 0% S 5% C	4% M 3% S 2% C	1% M 0% S 0% C		0% M 0% S 0% C	0% M 0% S 0% C
Perl	4% M 1% S 0% C	1% M 0% S 0% C	0% M 0% S 6% C	5% M 4% S 2% C	2% M 0% S 0% C	0% M 0% S 0% C		1% M 0% S 0% C
Rust	3% M 1% S 0% C	0% M 0% S 0% C	1% M 0% S 4% C	4% M 3% S 1% C	1% M 0% S 0% C	0% M 0% S 0% C	1% M 0% S 0% C	

Figure 7.2: ***Pairwise view of potential semantic portability problems****. This chart is symmetric, and distinguishes regexes by language and type. The individual cells indicate the percent of the regex corpus with at least one (M)atch, (S)ubstring, and (C)apture witnesses in that language pair, and darker cells indicate that regexes more commonly have difference witnesses in that pair of languages. For example, Java, Go, and Rust generally agree on regex behavior. At the scale of our corpus, each percentage point represents about 5,000 regexes.*

Root cause analysis We investigated the root causes of the many ways in which changing regex engines may affect regex semantics. We developed an automatic tool, the `Cross Examiner`, to estimate the causes of the difference witnesses identified through our experiment. We iteratively examined unclassified witnesses, referenced the regex documentation for the disagreeing languages (Table 7.1), understood the reason for the different behaviors where documented, and encoded heuristics to classify witnesses as due to this behavior. The causes we identified are summarized in Table 7.3. Approximately 98% (80,736/82,582) of witnesses could be explained by one or more of these causes.

Table 7.3 differentiates the witnesses by type. The first group of witnesses are cases where some languages support a feature that others do not. In the second group, languages use the same syntax for different features. The third group are cases where languages use the same syntax for the same features but exhibit different behavior. The final group are defects we identified in a regex engine's behavior, described below.

Comparison to documented semantics We studied each language's regex documentation (Table 7.1) to see if these witnesses could be easily explained. Comparing the grey cells and boldfacing in Table 7.3, we note that more than half of the "unusual" behaviors were unspecified in that language's documentation. *Testing, not reading the manual, is the only way for software engineers to learn these behaviors.*

Table 7.3: ***The semantic differences identified during our semantic portability experiment.*** *Each row indicates a witness regex, the expected behavior(s), and each language's interpretation. The first three groups describe different classes of valid but semantically distinct behavior. The final group describes the bugs we found; E- means Engine, D- means Docs. Boldface indicates what we believe to be potentially-surprising behavior. "–" indicates languages where a feature causes syntax errors. The behavior in the grey cells was not specified in the documentation.*

Witness	**Description**	**JavaScript**	**Java**	**PHP**	**Python**	**Ruby**	**Go**	**Perl**	**Rust**
False friends 1: Regex notation describes a feature in one language and no feature in another.									
`/\Qa\E/`	Quote directive ; "QaE"	**"QaE"**	Quote	Quote	**"QaE"**	**"QaE"**	Quote	Quote	–
`/\G/`	Match assertion ; "G"	**"G"**	Assertion	Assertion	**"G"**	Assertion	–	Assertion	–
`/\Ab\Z/`	Anchors ; "AbZ"	**"AbZ"**	Anchors	Anchors	Anchors	Anchors	–	Anchors	–
`/a\z/`	End of line ; "az"	**"az"**	EOL	EOL	**"az"**	EOL	EOL	EOL	EOL
`/\K/`	Match reset ; "K"	**"K"**	–	Reset	**"K"**	Reset	–	Reset	–
`/\e/`	ESC ; "e"	**"e"**	ESC	ESC	**"e"**	ESC	–	ESC	–
`/\cC/`	ctrl-C ; "cC"	ctrl-C	ctrl-C	ctrl-C	**"cC"**	ctrl-C	–	ctrl-C	–
`/\x{41}/`	"A" (hex) ; "x...x"	**"x...x"**	"A"	"A"	–	–	"A"	"A"	"A"
`/(a)\g1/`	Backref notation ; "ag1"	**"ag1"**	–	Backref	**"ag1"**	**"ag1"**	–	Backref	–
`/(a)\g<1>/`	Backref notation ; "ag⟨1⟩"	**"ag⟨1⟩"**	–	Backref	**"ag⟨1⟩"**	Backref	–	–	–
`/\p{N}/`	Unicode digit ; "pN"	**"p{N}"**	1	1	**"p{N}"**	1	1	1	1
`/\pN/`	Unicode digit ; "pN"	**"pN"**	Digit	Digit	**"pN"**	**"pN"**	Digit	Digit	Digit
`/[[:digit:]]/`	Digit ; Custom Char. Class (CCC)	**CCC**	**CCC**	Digit	**CCC**	Digit	Digit	Digit	Digit
False friends 2: The same regex notation describes different features.									
`/^a/`	^: Beginning of input or line	Input	Input	Input	Input	**Line**	Input	Input	Input
`/a++/`	Possessive quantifier ; regular	–	Possessive	Possessive	–	Possessive	–	Possessive	**Regular**
`/(a)\1/`	Backref ; octal	Backref	Backref	Backref	Backref	Backref	–	Backref	**Octal**
`/\h/`	Horz. whitespace; Hex; "h"	**"h"**	Whitespace	Whitespace	**"h"**	**Hex**	–	Whitespace	–
Nuanced: The same regex notation describes the same feature, but engines exhibit subtly different behavior.									
`/(a)(?<b>b)/`	Named and unnamed capture groups?	Both	Both	Both	–	**Named only**	–	Both	–
`/[]]/`	CCC of "]" ; empty CCC + "]"	**Empty**	"]"	"]"	"]"	"]"	"]"	"]"	"]"
`/((a*)+)/`	Diff. capture of \2 on "aa"	**\2: "aa"**	\2: empty	\2: empty	\2: empty	\2: empty	**\2: "aa"**	\2: empty	**\2: "aa"**
`/((a)|(b))+/`	Diff. capture of \2 on "ab"	**Empty**	"a"	"a"	"a"	"a"	"a"	"a"	"a"
Bugs we found in regex engines.									
E-Python: `/(ab|a)*?b/`	Diff. capture of \1 on input: "ab "	"a"	"a"	"a"	**Empty**	"a"	"a"	"a"	"a"
E-Rust: `/(aa$)?/`	Matched substring on "aaz"	Empty	Empty	Empty	Empty	Empty	Empty	Empty	**"aa"**
E-Rust: `/(a)\d*\.?\d+\b/`	Matched substring on "a0.0c "	"a0"	"a0"	"a0"	"a0"	"a0"	"a0"	"a0"	**"a0.0"**
E-JavaScript: *Complicated*	Input order matters?	**Yes**	No	No	No	No	No	No	No
D-OracleJava: `/$\s+/`	$ matches before final `\r`?	No	**Yes**	No	No	No	No	No	No
D-Ruby: `/a{2}?/`	Lazy "aa" ; optional "aa"	Lazy	Lazy	Lazy	Lazy	**Optional**	Lazy	Lazy	Lazy

Regex engine testing Though in this experiment we assumed that the regex engines were trustworthy, our methodology can be viewed as a mix of fuzz [117] and differential [232] testing of the regex engines themselves. Researchers have searched for defects in regex engines using similar techniques [66, 94] and automatically-generated regexes. Since the space of possible test regexes grows combinatorially, our "testing" contrasts from theirs insofar as ours is prioritized by the regexes that engineers use in practice, and driven by more thorough synthetic input generation.

During our examination of difference witnesses, we identified five cases where one regex engine disagreed with the others *and* its behavior was inconsistent with its documentation. We opened defect reports based on the behaviors sketched in the third section of Table 7.3. So far these bugs have been confirmed in V8-JavaScript, Python, Ruby, and Rust.

Semantic errors in applications Although engineers may identify some semantic regex problems during testing, others may cause unexpected regex behavior in practice. To estimate the frequency of semantic problems in practice, we developed linter-style tools to identify regexes that use features that are unavailable in their language. For example, in JavaScript the anchor notation `/\Ab\Z/` is interpreted literally as AbZ, but engineers who use this notation in JavaScript projects probably intend anchors. Among the JavaScript (npm) modules from which we derived our corpus, we identified 31 modules that used this notation. In total we identified hundreds of modules containing potential semantic regex bugs. We have begun opening bug reports against these modules.

It is possible that these regexes were derived from copy/paste practices. However, engineers might introduce such bugs even when designing regexes from scratch, since they may design them based on a (supposed) regex *lingua franca* that does not extend to the language in which they are developing.

7.5 RQ3: To what extent are a common core of regexes used across regex engine boundaries?

In answer to RQ2, we found many ways in which regex engines differ semantically, which may discourage engineers from adopting the ReDoS defense of changing regex engines. However, it may be that the semantic differences we identified are not relevant in practical usage. A regex's string language may vary in the universe of all possible strings (which we sampled in §7.4), but if the universe of input strings in the program context is incomplete then some of these semantic differences may not be important. For example, whether the anchor `/^/` indicates the beginning of the input string or of a line within it is only relevant if the regex is applied on input strings that can contain more than one line.

Measuring regex re-use practices across regex engine boundaries will shed light on this question. To give nuance to our quantitative findings of semantic incompatibilities, in this experiment we will measure the extent to which software engineers use a common core of regexes across regex engine boundaries. To the best of our knowledge, this is the first measurement of the re-use of common regexes in the literature. Prior work has considered this question only qualitatively [237, 238], reporting on the regex re-use practices of professional software engineers through surveys and interviews. Software engineers did say they re-used regexes, most commonly from Internet resources like Stack Overflow and from other software. They did not frequently describe memorable encounters with regex incompatibilities. This suggests, at least qualitatively, that in the common case regex engine replacement may be viable.

7.5.1 Methodology

The general methodology of this experiment was as follows: we identified a set of "common" regexes, and then determined how often those regexes were used in multiple programming languages.

Unique regexes We define a unique regex by its string representation (e.g., the regex `/ab*c/` is represented by the string "ab*c"). We use string equality to determine the module registries (thus, which programming languages) in which we found that regex. We rejected alternative measures of regex equivalence like string language [333] or behavioral [115] similarity, because regexes that match on these measures may already have been ported by engineers to address semantic differences across regex engines.

Common regexes We defined a *Common Regex* as one that met two conditions:

General task: It appeared in more than one software module, implying that it was not overly customized (e.g., not a regex tailored to a particular project's error messages).

Generally available: It appeared in one of the Internet sources mentioned by software engineers as a typical re-use source.

Mining Internet sources We extracted regexes from the two Internet sources most commonly cited by the engineers that Michael et al. surveyed: RegExLib and Stack Overflow. In both of these relatively-unstructured Internet regex sources, our extraction effort was best-effort.

Extracting regexes from RegExLib. We exfiltrated the RegExLib database by performing an empty search, which returns a paginated set of all of their regexes. We traversed and scraped the resulting web pages.

Extracting regexes from Stack Overflow. For Stack Overflow, we relied on the "regex" tag to identify regexes. Through manual analysis we found that questions and posts with the "regex" tag commonly denote regexes using code snippets. Using all Stack Overflow posts as of September 2018,[1] we found all questions tagged with "regex" as well as the answers to those questions and automatically extracted code snippets from those posts. To filter, we then removed snippets that contained no regex-like characters based on the PCRE regex syntax given in Table 2.3.

7.5.2 Results

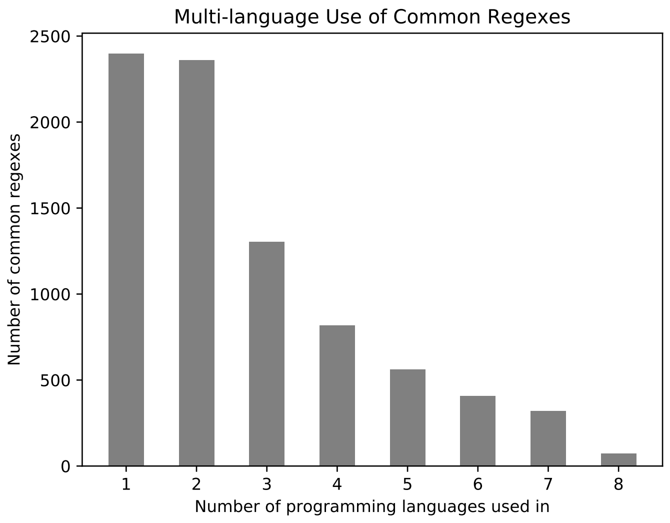


Figure 7.3: ***Multi-language use of common regexes.*** *This figure shows the frequent multi-language use of Common Regexes, defined according to our methodology. Among the 8,234 Common Regexes, 71% are used in multiple languages.*

Common regexes Within our regex corpus, most regexes are customized to their usage context and do not appear in more than one software module. Among the 537,806 unique regexes, only 123,108 (23%) met the *General Task* criterion by appearing in more than one module. Within these, we identified 8,234 regexes that were also *Generally Available*, i.e., also found on an Internet source.

Crossing regex engine boundaries We found that the 8,234 Common Regexes were frequently used across programming language boundaries, and therefore across regex engine boundaries. As Figure 7.3 shows, 5,837 (71%) of the common regexes are used in multiple regex engines. Among the regexes that met only the *General Task* criterion, the proportion

[1]We used an archived image of Stack Overflow, available at `https://archive.org/download/stackexchange/stackoverflow.com-Posts.7z`.

falls; 24,411 of these regexes (19%) are used in more than one regex engine. This supports the qualitative findings of Michael et al.: that Internet sources are a common distribution channel from which software engineers re-use regexes across programming language boundaries.

7.6 Discussion

The results of RQ1 and RQ2 demonstrate that the ReDoS remedy of wholesale regex engine replacement is promising but comes at the risk of backwards incompatibility. Triangulating these findings with RQ3, however, it appears that there is a large class of common regex use cases for which the existing incompatibilities are not (known to be) problematic.

Performance improvement In answering RQ1, I found that moving from a "Slow" to "Medium" to "Fast" regex engine will uniformly improve worst-case performance, and in Rust and Go I found that nearly all supported regexes would perform in linear time (cf. Figure 5.6). Thus, changing regex engines offers engineers an easy fix to the performance aspect of their ReDoS problems. But engineers must consider factors beyond performance improvement. Specifically, they should be concerned about *semantic changes* and *feature support*.

Semantic changes In answering RQ2 I considered semantic changes, and found a substantial number of regexes whose behavior would change if used in a Fast-er regex engine. A *translation layer* could address the syntactic compatibility issues, but it is unclear how to build a translation layer for the cases I identified in which equivalent features follow different semantics in different regex engines. The implementation of such a translation layer is further complicated by the lack of formal semantics describing production regex engine behavior. As discussed in §2.4.3, there is ongoing work on formalizing the behavior of regex engines. My experimental comparisons may guide theorists towards under-specified aspects of regex engine behavior. The existence of these incompatibilities may also be symptomatic of under-specification and under-testing within major programming languages, which if true suggests another line of work.

Feature support In addition to semantic differences on supported regexes, the regex engines in the "Fast" family simply do not support extended regex features like backreferences and lookaround assertions [146, 176]. In §5.8 I reported that these features are used in only about 5% of the regex corpus. Most applications do not depend on those features and could adopt a Fast-er regex engine after careful regex porting. For the applications that do need them, they might obtain some ReDoS relief from shifting to the "Medium" family of regex engines, but this would be an incomplete solution because those regex engines still exhibit super-linear worst-case performance on many regexes.

Programming language documentation Each programming language's regex documentation currently focuses only on its own syntax and semantics. We recommend that regex documentation additionally describe its deviations from external specification(s), e.g., PCRE or POSIX. Explicitly discussing incompatibilities will inform developers of "gotchas", and it will have the indirect effect of reminding them that regexes are (currently) not a *lingua franca*. Longer term, explicitly considering each language's divergence from specification(s) will help designers reach agreement on a next-generation universal regex specification.

We also recommend that programming language maintainers document the worst-case match performance of their regex engines. At the moment, only Rust and Go describe their worst-case (linear-time) performance. Documenting the potential for ReDoS is particularly important for the members of the "Slow" family, who have put software engineers at the greatest risk.

7.7 Threats to validity

Internal validity *Super-linear behavior:* Our conclusions about uniform improvement from Slow-er to Fast-er regex engine families are based on the accuracy of the super-linear regex detectors. It is possible that some of these improvements can be attributed instead to defenses and optimizations that might be circumvented by cannier detectors. If so, improved super-linear regex detectors might change our findings. This is most relevant to the Medium-performance family, as the regex engines in the Fast family use Thompson's lockstep NFA simulation which can guarantee linear-time performance.

Regex flags: Our analyses deliberately omit the use and effect of regex flags to avoid complicating our fundamental questions. However, regex flags may affect syntax, semantic, and performance portability. The (in?)equivalence of regex flags across languages might complicate regex porting for syntax and semantics. In addition, flags can actually affect worst-case regex performance — e.g., a case-insensitive flag permits the regex engine to traverse paths that are not considered with a case-sensitive match, potentially increasing the number of super-linear regexes. Our ensembles of input generators and super-linear regex detectors also ignore the implications of flags.

Internet regexes: Our methodology for identifying regexes in Stack Overflow and RegExLib was best-effort, and we may have omitted regexes with unusual formatting.

External validity Our findings on semantic compatibility are specific to the regex corpus collected in our work, which was derived from popular open-source software projects. It is possible that our results would change if tailored to the regexes in a particular context, e.g., the Snort regex corpus. We believe, however, that the scale of our corpus is sufficient to permit generalization of our findings in typical software engineering contexts.

Construct validity *Programming language = regex engine:* In our experiments, we assumed that programming languages accurately reflect the behavior of their regex engines. It is possible that the regex engines themselves behave more consistently, and that the variation we observed is due instead to interposition in the programming language's runtime. For example, internal wrappers around the regex engine might perform regex rewriting or replace trivial regex operations with string functionality, affecting semantics. To the best of our knowledge, no such interposition exists. We have examined the libraries surrounding the JavaScript-V8, Python, Perl, PHP, and Rust regex engines, and other researchers have examined those for the Java regex engine [336].

Chapter 8

Optimizing a regex engine through memoization

8.1 Summary

"Those who cannot remember the past are condemned to repeat it."

–George Santayana

The ReDoS amelioration considered in Chapter 7 addressed ReDoS Condition 2 by *substituting* a super-linear regex engine for a linear-time one. In this chapter, we consider an alternative approach to addressing ReDoS Condition 2: *optimizing* a backtracking-based regex engine. In particular, we consider the application of *memoization* to eliminate the redundancy that causes super-linear regex evaluations in backtracking regex engines. Unlike regex engine substitution (or equivalent overhauls of engine internals), memoization can be easily incorporated into existing backtracking-based regex engines. We are not the first to propose memoization in this context, but we are the first to analyze space-efficient selective memoization schemes, and to evaluate the technique empirically. After introducing related work in §8.2, we proceed to an analysis (sections 8.4 and 8.5) and evaluation (§8.6) of memoization for K-regexes, and then extend our results to E-regexes (§8.7).

Our primary interest is *not* to improve on the *state of the art* algorithms for the recognition and parsing problems for truly regular expressions (K-regexes). The Thompson NFA simulation and Brzozowski derivatives offer the best known time and space complexities (Table 8.3). None of the algorithms we present improves on them. Instead, our goal is to improve on the *state of practice* algorithms. We present theoretical analyses that demonstrate exponential improvements in time complexity at what we believe are acceptable increases in space complexity, and evaluate our approach on a large-scale regex corpus in a practical implementation. All this is true for K-regexes; for E-regexes, in §8.7 we improve on previous theretical results for the complexity of two extended regular expression features (lookaround assertions and backreferences).

Methodology The investigation in this chapter considers four research questions. *First*, we employ a theoretical analysis to determine the expected effect of memoization on K-regexes. *Second*, in light of concerns about the space costs of memoization (§8.2), we apply theoretical knowledge and empirical lessons to propose potential space optimizations for K-regex memoization. *Third*, we experimentally evaluate the performance of the various K-regex memoization techniques we propose. *Fourth*, we examine the extension of memoization to E-regexes.

Findings In answer to RQ1, we report that the super-linear behavior of Spencer-style backtracking regex engines on K-regexes is caused by redundant exploration. Using memoization, a backtracking regex engine can guarantee linear-time matches. In answer to the unattractive space penalty incurred by memoization, in RQ2 we describe several space optimizations with the potential to reduce or eliminate any increased space complexity caused by memoization. These optimizations are based on theoretical analysis and empirical observations. In our experiments for RQ3, we showed that most super-linear regexes can be evaluated in linear time. More importantly, with an appropriate encoding scheme the space cost for most regexes is *constant* with respect to the length of candidate string. In answer to RQ4, we show that (1) regexes with lookarounds entail *super-exponential* complexity in existing regex engines, but can be performed in linear time with memoization; and (2) regexes with backreferences can be evaluated in a lower exponential than has previously been reported, and for typical regexes cost only polynomial time and space.

Statement of attribution The material presented here is being prepared for publication. C. Coghlan assisted with the study of the Perl regex engine. S.A. Hassan analyzed the typical contents of the capture groups referenced by backreferences.

8.2 Related work

Regex engine optimizations for the average case Regex evaluation has been a longstanding optimization target. Researchers have proposed a diversity of optimizations to improve the *average* performance of a regex match: minimizing the automaton [112, 155, 184, 193, 195, 204, 303, 331, 342]; pursuing more efficient automaton encodings [155, 259, 347]; extending the automaton model [79, 256, 260]; leveraging properties of constant subpatterns [64, 96, 199, 212, 345]; introducing supporting data structures [114, 276]; optimizing the use of I/O libraries [198]; relaxing match requirements [118, 344]; applying specialized hardware [100, 109, 217, 291, 348, 350]; and parallelizing the match process [263, 346]. Unfortunately, none of these optimizations is suitable for improving the *worst-case* performance of a regex match based on a Spencer-style simulation. We propose to apply memoization techniques to address this problem.

Memoization Memoization is an optimization technique that spends space to save on time. The key idea, proposed by Michie, is to record a function's known input-output pairs in order to avoid evaluating a function more than once per input [239]. If evaluating the function is expensive, the time savings can be substantial if a function is commonly called with the same inputs. From an engineering perspective, memoization has been argued to permit the use of more "natural, straightforward" algorithms than other approaches [319, 325] Memoization has been widely applied in functional programming, logic programming, and dynamic programming.

In functional programming, memoization can be applied verbatim. Functions that are side-effect free can be implemented by combining a memo (hash) table with the original implementation [325]. In this context, the set of a function's dependences (e.g., the input domain) is potentially enormous, and the memo table may grow unbounded. Researchers have explored various strategies for reducing space costs, including partial memoization via symbolic analysis of a function's dependences [71, 349], automated approaches such as garbage collection [128, 197], and selective memoization schemes [61] that track a subset of the possible memoizable states. As the memo table need not be complete, caching with a replacement policy is also a viable option, although this has the downside of decreasing the predictability of an algorithm's behavior [128]. Memoization has also been used as a solution to search problems arising in logic programming and dynamic programming. For example, the search process for a satisfying assignment of variables can be expedited by memoizing the failed "no-good" solutions considered during a backtracking search [82, 282, 319]. Likewise, dynamic programming algorithms combine a careful evaluation order with memoization to reduce the overall time complexity of a problem [83, 129].

Memoization in string parsing The literature most relevant to this chapter is the work on optimizing parsing problems using memoization. Most researchers have focused on using memoization to parse context-free grammars (CFGs) and parsing expression grammars (PEGs). Birman and Ullman showed that context-free languages could be parsed in linear time through backtracking and memoization, or, equivalently, a dynamic programming approach [90]. This fact was re-discovered by Norvig [264] and popularized by Ford under the name "packrat parsing" [157]. As an alternative to a backtracking approach, Might et al. have extended Brzozowski derivatives (§2.3.2) to CFGs and applied memoization to such a parser [244]. All of these approaches entail $\mathcal{O}(|G| * |w|)$ space complexity, where G represents the rules in a grammar and w is the candidate string [81]. Ford argues that, as CFGs are typically used to parse software projects, this cost is acceptable; he expects that $|w| \leq 100$ KB [157].

Memoization in regex evaluations Memoization has been considered but rejected for regular expression matching by the research community. Various researchers have remarked on its potential application, but not implemented it out of concern over the expected space complexity [85, 87, 132, 295]. Although these authors did not explain why they felt space was so critical, we conjecture the crux is that some regex contexts involve strings much longer than the 100 KB assumed by packrat parsers (e.g., regexes for biological applications or natural language processing).

Within the engineering community, to the best of my knowledge only Google's RE2 regex engine and the Perl regex engine employ some form of memoization. RE2 generally uses Thompson's lockstep NFA simulation, but to resolve certain match-with-capture queries, RE2 memoizes a backtracking NFA simulation. In particular, it memoizes this simulation if the full search space can be encoded in a bitmap that is less than 32 kilobytes [133]. This will protect against high-complexity behavior for small regexes and short input strings, but would fail on the long inputs associated with the polynomial regexes described in our case studies (Chapter 3). We discuss this approach in more detail in §8.4.

The memoization scheme of the Perl regex engine is more convoluted. As it has not previously been described in the scientific literature, I introduce it here. The Perl regex engine has used a selective memoization scheme [61] since 1999.[1] The scheme has not been substantially changed since its introduction. We describe the implementation as of commit 34667d08d3b in February 2020.

General description of the scheme: The Perl regex engine's memoization scheme records the backtracking algorithm's visits to a subset of the "*complex*" engine states. Specifically, it records visits to the first k states associated with "complex" unbounded repetitions of the form `/A*/` where the language of the sub-pattern A contains strings of varying lengths.[2] As an example, the regex engine will memoize the sub-pattern for R_1 = `/(a+)*/`, where $L(A) = L(a+) = \{a, aa, aaa, \ldots\}$. As another example, the regex engine will memoize the sub-pattern for R_2 = `/(a?)*/`, where $L(A) = L(a?) = \{\varepsilon, a\}$.

What is not memoized: Additional complex states after the first k are not memoized. In addition, Perl regex engine distinguishes several other "*simple*" engine states associated with bounded repetition `/A{m,n}/` or unbounded repetition `/A*/` when the language of the sub-pattern A contains only strings of a fixed width; these are not memoized.

The contents of the memo table are also affected by the use of extended Perl features, as defined in §2.4.2.1. The contents of the memo table are *reset* ("voided", in the engine's parlance) if an engine state is encountered that uses a backreference or a conditional. The use of memoization is *disabled entirely* if an engine state is encountered that uses callouts (embedded code), which can change the regex pattern or the candidate string being evaluated.

Unprotected super-linear behavior using this scheme: With this selective memoization scheme in place, a regex engine can still exhibit super-linear worst-case behavior on several classes of super-linear regexes. Here they are, ordered according to my understanding of their practical

[1]The Perl regex engine's memoization scheme was introduced by the engineer J. Hietaniemi in commit 2c2d71f56, which shipped in Perl v5.6.

[2]These complex engine states are referred to as `CURLYX-WHILEM` states in the Perl regex engine. The regex engine memoizes the first $k = 16$ complex states, resulting in space complexity $\mathcal{O}(16 * |w|)$.

importance:

1. Some K-regexes will be exponential, if there is exponential ambiguity involving only "simple" states. For example, the regex `/(a|a)+/` has exponential ambiguity with a "simple" state; the sub-pattern $A = a|a$ has the fixed-width language $L(A) = \{a\}$.
2. Some K-regexes will be polynomial, viz. those that involve bounded repetition of complex or simple sub-patterns, and/or the unbounded repetition of simple sub-patterns. For example, the quadratic regex `/.*.*/` and the high-polynomial regex `/(.*){100}/` are both unprotected.
3. Some E-regexes will be super-linear (exponential or polynomial), namely those for which super-linear backtracking crosses a conditional or a backreference. For example, the regex `/('|")(a+)+\1/` is unprotected, because the exponential backtracking will repeatedly test the backreference and reset the cache.
4. As Cox noted [132], regexes with large finite ambiguity will not be protected, such as is illustrated in Figure 2.10.
5. Ambiguous regexes in which a super-linear evaluation crosses an inline Perl snippet through `eval` will not be protected.
6. Regexes with more than k complex states may be super-linear by omission in the memo-table. For example, one of the exponentially ambiguous sub-patterns of the regex `/((a*)*){k+1}/` would not be protected by memoization.[3]

The first three families of super-linear regexes appear in real-world regexes. It is not clear whether the final three families are of similar practical interest. We leave this investigation to future work.

Although these families are not protected by memoization in the Perl regex engine, other aspects of the engine may still prevent super-linear behavior for regexes of these forms. For example, Perl has a separate optimization that eliminates exponential behavior in the first family, whose members are exponentially ambiguous K-regexes with a fixed-width sub-pattern constructed from a disjunction.[4] The engine developers observed that if all paths through a disjunction are the same length, and any path matches, it is not necessary to test the other paths — they will all reach the same search state (i.e., the same vertex, and by the fixed-width property, the same index into w). This "local memoization" reduces the time complexity of the evaluation from exponential to linear. As another example, Perl employs optimizations to identify feasible starting offsets in the input based on fixed sequences $\Sigma*$ within the regular expression. It is not always possible to craft input that triggers super-linear behavior while avoiding this optimization.

In light of these gaps, although we cannot find records of the origin of the memoization scheme, we conjecture that Perl's selective memoization scheme may have been designed to protect regexes containing the *nested quantifiers anti-pattern* (Chapter 6). For example,

[3]To trigger the exponential behavior in Perl, manually unroll the bounded quantifier.

[4]The Perl regex engine identifies these fixed-width sub-patterns during the pattern-to-automaton conversion.

the regex `/(a*)*/` run in linear time under the Perl cache. Such regexes clearly have an unbounded "complex" state to protect.

Some additional notes on the Perl regex engine's implementation are given in Chapter A.

8.3 Study design and research questions

In the preceding section I summarized the related work on memoization for regexes and sundry parsing problems. Although several researchers have mentioned the potential benefits of memoization in regex evaluation, their treatment of it has been brief and solely qualitative out of concerns about large storage costs. In practice, two production-grade regex engines employ limited forms of memoization, although neither protects users against the forms of typical super-linear regexes. Their space costs have not been evaluated experimentally.

In light of the real risk of ReDoS posed to practitioners, and the modest expected engineering cost of incorporating sound memoization into an existing regex engine, we believe that memoization should be re-considered as a practical ReDoS solution. We have two advantages over prior work in this space: (1) *Theory:* Existing approaches were implemented prior to the identification of the conditions for super-linear regular expression behavior, so these approaches could not leverage the contributions of that research. (2) *Data:* We have a large corpus of real-world regexes, and can design and evaluate our solution appropriately.

In this chapter we investigate four research questions of concern to the maintainers of Spencer-style backtracking regex engines:

RQ1: What is the expected effect of memoization on K-regexes?
RQ2: How might the space costs of K-regex memoization be reduced?
RQ3: Experimentally, what are the space and time costs of K-regex memoization?
RQ4: How might memoization be extended to E-regexes?

8.4 RQ1: What is the expected effect of memoization on K-regexes?

The class of K-regexes is a natural first target, because in Chapter 5 we reported that 95% of the regexes in our regex corpus fall into this category. Guided by the formal definitions of the root causes of super-linear worst-case behavior, we can protect K-regexes in various ways.

Although prior work has stated that memoization can benefit regular expressions, we are not aware of a careful treatment of the problem. In answering this research question, we provide missing formalizations and introduce concepts useful in the subsequent sections.

8.4.1 Super-linear regex evaluations are caused by redundancy

First, let us observe that the size of the *search space* for K-regex matching is the product of the size of the automaton and the length of the input string w: $|Q|*|w|$. This property follows from the observation that whether a K-regex matches a candidate string can be expressed recursively in terms of the current positions in the automaton and the input string, and not on how these positions were reached. K-regex matches are path-independent and side-effect free. Starting from a given matching algorithm search state $\langle q, i \rangle$, the match can be determined solely based on whether any of the subsequent automaton nodes $\delta(q, w[i])$ will match from the string position $i+1$. As there are $|Q|$ starting positions in an automaton, and $|w|$ positions in the input string, there are thus at most $|Q| * |w|$ distinct matching algorithm search states, bounding the search space.

If a complete exploration of the search space is required, then it would be natural for the worst-case match time complexity to equal the size of the full search space: $\mathcal{O}(|Q|*|w|)$.[5] Of course, under some inputs the behavior will be much faster, e.g., due to an input that leads quickly to a sink state. But if a regex match algorithm requires more than $\mathcal{O}(|Q|*|w|)$ in the worst case, it can be improved. The production implementations of the Spencer backtracking algorithm far exceed the worst-case match time complexity of $\mathcal{O}(|Q| * |w|)$. Proofs of the exponential worst-case behavior of the Spencer NFA simulation (i.e., $\mathcal{O}(|Q|^{|}w|)$) are based on counting the number of paths through the NFA graph on a mismatch. Such an argument is commonly framed [85, 87], but omits the key observation that many of these paths are *redundant*.[6] The following theorem explains the Spencer algorithm from the perspective of redundancy.

Theorem 8.4.1. Suppose the Spencer-style backtracking algorithm (Listing 3) is used to determine whether a candidate string w matches a pathological completely-connected NFA with $|Q| \geq 2$. Then for all $q \in Q$, the BacktrackingPoint $\langle q, w_j \rangle$ is pushed onto the stack $|Q|^{j-1}$ times for each j, $1 \leq j < |\text{str}|$.

Proof. The argument proceeds by weak induction.

Base case: $j = 1$. Spencer's simulation begins by computing $\delta(q_0, w_0)$ to obtain `possibleStates` $= Q$. It pushes onto the stack the BacktrackingPoints $\langle q_i, 0+1 = 1 \rangle$ once apiece, or $|Q|^{j-1} = |Q|^{1-1} = |Q|^0 = 1$ time each.

Inductive step. Assume the theorem holds for index $k < |w|$: the backtracking stack contains $|Q|^{k-1}$ instances of the BacktrackingPoint $\langle q, k \rangle$ for each $q \in Q$. Now consider the number of times we push onto the stack the BacktrackingPoint $\langle r, k+1 \rangle$ for an arbitrary $r \in Q$. Because the algorithm terminates, we eventually pop and process each of the BacktrackingPoints

[5] More precisely, the worst-case cost of the match should be written as $\mathcal{O}(|Q| * |w| * X)$, where X is an upper bound on the cost of evaluating each state under a given match algorithm.

[6] Indeed, it is not surprising that there must be redundancy in the Spencer algorithm, or else it too would run in linear time.

$\langle q, k \rangle$. When we process each, on the first iteration of the inner quantifier we push onto the stack the BacktrackingPoint $\langle r, k+1 \rangle$ when we apply the transition function. Let us count: for each $q \in Q$, *each* of the corresponding $|Q|^{k-1}$ instances of the BacktrackingPoint $\langle q, k \rangle$ pushes onto the stack *one* instance of the BacktrackingPoint $\langle r, k+1 \rangle$. There are $|Q|$ such $q \in Q$, so we push onto the stack the BacktrackingPoint $\langle r, k+1 \rangle$ a total of $|Q| * |Q|^{k-1} = |Q|^k = |Q|^{(k+1)-1}$ times. □

Since it takes exponential time to enqueue and dequeue an exponential number of BacktrackingPoints, we obtain an exponential bound on the time complexity of the Spencer algorithm. But consider what the theorem says: we push onto the stack each BacktrackingPoint $\langle q, |w| \rangle$ a total of $|Q|^{|w|}$ times for later processing. But because regex evaluations are path-independent and side-effect free, any subsequent processing of this BacktrackingPoint would be redundant.

8.4.2 Memoized NFAs

Using memoization, a backtracking regex engine can record the areas of the search space that it explores. If it reaches those regions again, it can short-circuit a redundant exploration. The state space has size $|Q| * |w|$, and can be expressed using a matrix with Q columns and w rows.

Although this *full memoization* approach is a standard technique, we will discuss several *selective memoization* schemes in §8.5 whose properties are less obvious. To permit an accurate characterization of their soundness, we will extend the definition of an NFA to incorporate a memoization scheme. We refer to this entity as a *Memoized NFA* (M-NFA). Defining an M-NFA permits us to reason about the behavior of arbitrary NFA simulation algorithms when applied to an M-NFA; Listing 10 is one embodiment.

The M-NFA extensions are summarized in Table 8.1.

M: The memo function M of an M-NFA is stateful, and is updated as the backtracking simulation proceeds. The memo function initially returns 0 to all queries. When a backtracking search state $\langle q, i \rangle$ is marked, the memo function returns 1 for all future queries to that search state. The memo function state space has size $|Q| * |w|$. It can be tracked using a matrix with Q columns and w rows, though other implementations are possible.

δ_M: An M-NFA's memoized transition function δ_M accepts the typical arguments to δ, plus a candidate string index i in the range $\mathbb{N}^{|w|}$. It is defined as:

$$\delta_M(q, \sigma, i) = \{r \in Q \mid r \in \delta(q, \sigma) \wedge M(r, i+1) = 0\}$$

In other words, δ_M uses the memo function M to dynamically eliminate redundant transitions during the simulation.

Table 8.1: ***Components of a memoized finite automaton.*** *Components of a Memoized Non-deterministic Finite Automaton (M-NFA) derived from a finite automaton $A = \langle Q, q_0 \in Q, F \subseteq Q, \Sigma, \delta, \rangle$. The components of A are listed above the mid-rule (cf.Table 2.1). The components of the M-NFA for A include those, as well as the additional components below the mid-rule: M and δ_M.*

Component	**Meaning**
Q	The (finite) set of states of the automaton: $Q = \{q_1, q_2, \ldots, q_m\}$, with $\|Q\| = m$
$q_0 \in Q$	The initial state of the automaton
Σ	The input alphabet for strings: $w \in \Sigma*$
$\delta : Q \times \Sigma \cup \{\varepsilon\} \to \mathbb{P}(Q)$	The original transition function of the automaton, i.e., its graph "edges"
$F \subseteq Q$	The accepting (Final) states of the automaton
$M : Q \times \mathbb{N}^{\|w\|} \to \{0,1\}$	The memo function of the automaton
$\delta_M : Q \times \Sigma \cup \{\varepsilon\} \times \mathbb{N}^{\|w\|} \to \mathbb{P}(Q)$	The memoized transition function of the automaton

Simulation: An M-NFA can be simulated on a candidate string w beginning from q_0, by repeated application of the memoized transition function δ_M. If the simulation ends in a state $q \in F$, the candidate string is accepted by the M-NFA.

Memoization scheme: During the simulation, the choice of which search states to memoize is determined by a *memoization scheme* (policy). For example, in Listing 10 the scheme is to memoize every search state. We propose schemes that memoize all search states associated with a subset Φ of the automaton vertices, i.e., all search states $\langle q \in \Phi, i \in \mathbb{N}^{|w|} \rangle$.

Ambiguity: With these changes, we can define an ambiguous M-NFA analogous to an ambiguous NFA (§2.5.2.1). An M-NFA is *ambiguous* if there exists a string w such that when it is simulated from q_0, there is more than one path to the accept state q_F.

8.4.3 Applying Memoization

To convert an NFA into an M-NFA, we must provide a memoization policy by which to update the memo function M. In this section we consider the *full memoization* scheme: memoize every search state $\langle q, i \rangle$ that we visit.

Listing 10 illustrates how to incorporate this memoization scheme into the Spencer-style backtracking NFA simulation from Listing 3. Here, memoizing a BacktrackingPoint indicates that it is either scheduled to be explored (i.e., on the backtracking stack), or has already been explored (and did not lead to a success). Once a BacktrackingPoint $\langle q, i \rangle$ is memoized, a subsequent visit to that point would be redundant. Due to the memoization scheme used here, the memo function would return the empty set instead of the original destinations from $\langle q, i \rangle$, eliminating redundancy in the search.

Listing 10 Memoized version of Spencer's backtracking-based algorithm on K-regexes (cf. Listing 3). The added lines are indicated with a +.

```
class BacktrackingPoint:
  def __init__(self, state, stringIx):
    self.state, self.i = state, stringIx

def isInLanguage(regexPattern, candidateString):
  # Build the automaton and initialize the backtracking stack
  NFA = buildNFA(regexPattern)
  stack = Stack() # Contains BacktrackingPoints
  initialState = BacktrackingPoint(NFA.q0, 0)
  stack.push(initialState)

+ # Memo table: a (Q x m) 2D array. True means "already seen"
+ memoTable = [[False for i in range(len(candidateString)] for j in range(len(NFA.Q))]

  # Backtracking quantifier to evaluate choices we haven't yet considered
  while not stack.empty():
    # Simulate the next backtracking point to completion,
    # appending to the stack at each non-deterministic choice.
    bPt = stack.pop()
    currState = bPt.state
    i = candidateString[bPt.i]

    for j, nextChar in enumerate(candidateString[i:]):
        # Where might nextChar lead?
        possibleStates = NFA.deltaTransitionFunction(currState, nextChar)

+       # Memoize: Filter the BacktrackingPoints we've seen already, mark the new ones
+       candidateBacktrackingPoints = [BacktrackingPoint(q, j+1) for q in possibleStates]
+       newBacktrackingPoints = [bp for bp in candidateBacktrackingPoints
+                                 if not memoTable[bp.currState][bp.i]]
+       if not newBacktrackingPoints: break # Failed to find new BPs
+       for bp in newBacktrackingPoints: # Mark these BacktrackingPoints as visited
+         memoTable[bp.currState][bp.i] = True

        # DFS as before
        possibleStates = [bp.state for bp in newBacktrackingPoints]
        currState, others = possibleStates[0], possibleStates[1:]
        newBacktrackingPoints = [BacktrackingPoint(q, j+1) for q in others]
        stack.push( newBacktrackingPoints )

    # End of candidateString. Check if we ended in one of the FA's accept states.
    finalState = currState
    if NFA.acceptStates.contains(finalState):
      return MATCH

  return MISMATCH # None of the paths reached an accept state.
```

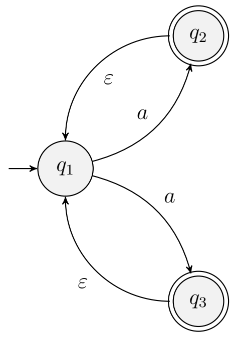


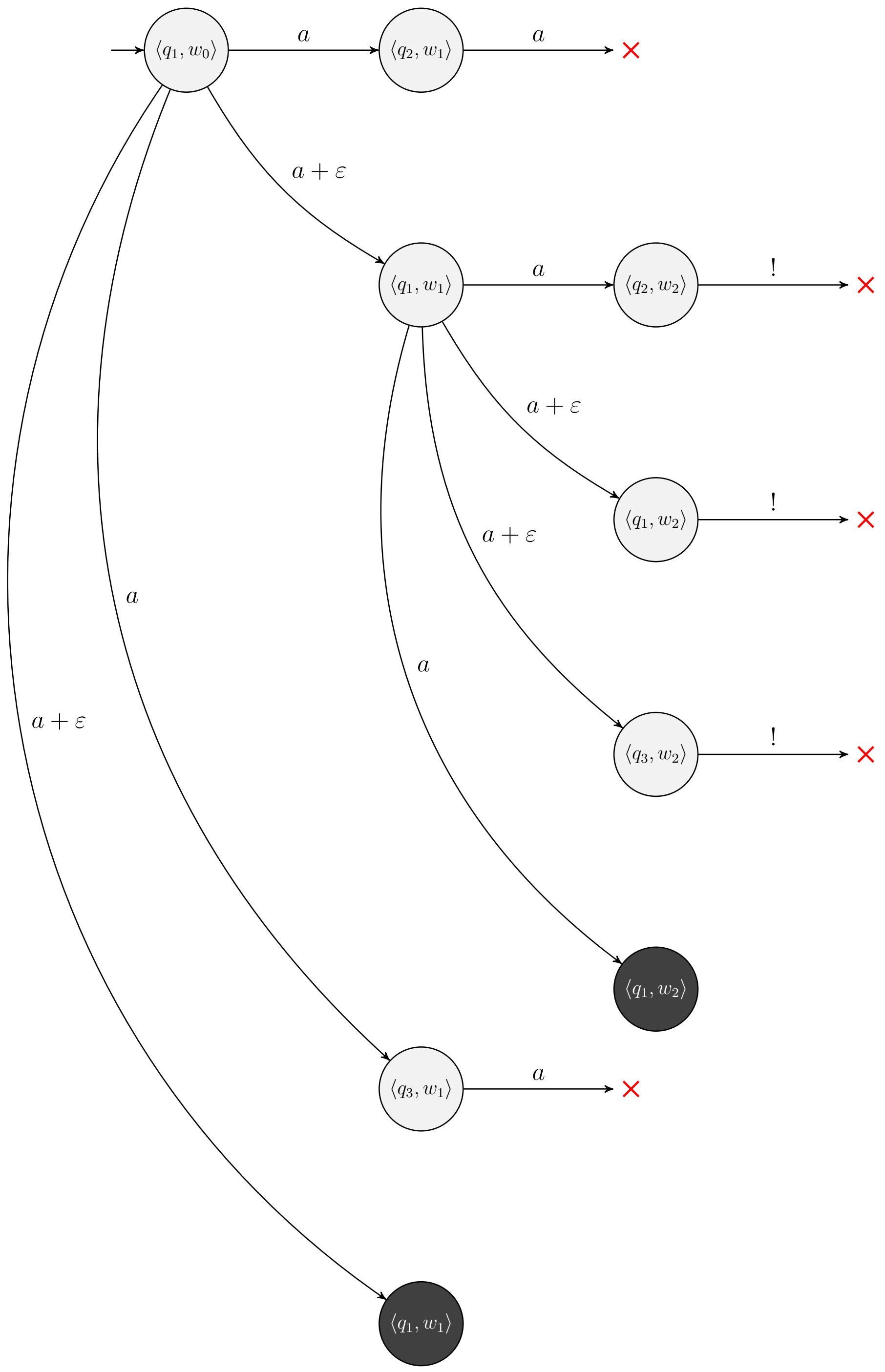
Figure 8.1: ***Automaton used to illustrate the effect of memoization.*** *This figure shows the NFA produced by Thompson's construction for the regular expression /(a|a)+/, slightly reduced for clarity of presentation.*

Table 8.2: ***Example memoization table.*** *The $|Q| \times |w|$ memoization table at the conclusion of the search described in Figure 8.2. A ✓ indicates that the search state has been visited (`True`).*

Indices of w = "$aa!$"	q_1	q_2	q_3
0 ('a')	✓	–	–
1 ('a')	✓	✓	✓
2 ('!')	✓	✓	✓

To illustrate the behavior of this algorithm, let us consider its operation on the NFA resulting from the regex `/(a|a)+/`. The corresponding NFA is given in Figure 8.1, and the *search tree* of the memoized backtracking search is illustrated in Figure 8.2. Table 8.2 presents the final memoization table. The nodes of the search tree are BacktrackingPoints indicating the current search state $\langle q, i \rangle$. The search proceeds through the tree left-to-right, top-to-bottom to follow the PCRE leftmost-greedy search prioritization semantics. Observe that the search states $\langle q_1, w_1 \rangle$ and $\langle q_2, w_1 \rangle$ both appear twice in the tree, because these states are reachable from both q_2 and q_3; the second instance of each is shaded. With memoization, we only consider each distinct $\langle q, w_i \rangle$ at most once. Without memoization, the Spencer algorithm would redundantly evaluate the search from those states twice, doubling the total number of search states explored for each additional 'a' in the candidate string.

The algorithm we have described improves the worst-case performance of the *recognition* problem (regex match) for Spencer-style backtracking regex engines. It also improves the worst-case performance of the other regex match problems, e.g., *parsing* (capture groups). The use of memoization eliminates redundant path exploration, but does not otherwise affect the behavior of the regex engine. The regex engine is free to populate capture groups during the simulation, enabling it to answer parse queries in linear time.



$\langle q_1, w_0 \rangle$
a
$\langle q_2, w_1 \rangle$
a
$a+\varepsilon$
$\langle q_1, w_1 \rangle$
a
$\langle q_2, w_2 \rangle$
!
$a+\varepsilon$
$\langle q_1, w_2 \rangle$
!
$a+\varepsilon$
$\langle q_3, w_2 \rangle$
!
a
$\langle q_1, w_2 \rangle$
a
$\langle q_3, w_1 \rangle$
a
$a+\varepsilon$
$\langle q_1, w_1 \rangle$

Figure 8.2: ***Example search tree using memoization.*** *Backtracking search tree on the NFA from Figure 8.1 on the string "aa!", with memoization in place. An X indicates rejection (moving to the sink state). A shaded node is skipped because the outcome has been memoized. The NFA exploration is prioritized, so nodes higher on the page are explored before nodes lower on the page. The delta transition function returns the ε-closure when processing a character.*

Under the memoization approach, a relationship between the Spencer backtracking algorithm and the Thompson lockstep algorithm becomes clear. Thompson's lockstep algorithm avoids exponential search time complexity by removing duplicates from its state frontier. A memoized version of Spencer's backtracking algorithm likewise avoids exponential search time complexity by marking search state nodes to avoid searching them more than once. The result is that Thompson's algorithm performs a non-redundant *breadth*-first search through the state space, and Spencer's algorithm performs a non-redundant *depth*-first search through the same state space. By naming and eliminating the redundant work in Spencer's backtracking algorithm, we can realize the duality of the two algorithms. Thompson's scheme remains more space efficient, partly because the breadth of the search tree (if visualized to be rooted at q_0 and growing downwards) is bounded by $|Q|$, while its height is bounded by $|w|$; typically $|w| \gg |Q|$

8.4.4 Arguments of correctness and cost

In this section I discuss the correctness and cost of a backtracking NFA simulation algorithm applied to an M-NFA using the full memoization scheme.

Correctness For this approach to be *correct*, the memoized version of Spencer's backtracking algorithm (Listing 10) and the non-memoized version (Listing 3) must have an identical input-output function. Observe that the effect of memoization is to ensure that each search state $\langle q, w_i \rangle$ is added to the backtracking stack at most once. If that state is reachable along more than one path through the NFA as a result of ambiguity, subsequent visits to that state are short-circuited using the memoization table. As discussed in §8.4.1, the result of a K-regex regex match starting from a given search state does not depend on the path by which that search state was reached. Consequently, any additional visits to a search state are redundant, and eliminating them does not change the input-output behavior of the search algorithm.

Time complexity This algorithm will process at most $\mathcal{O}(|Q| * |w|)$ search states. The δ function must be applied to each of these, at a cost of $\mathcal{O}(|Q|)$ each time. Thus, this algorithm will run in $\mathcal{O}(|Q|^2 * |w|)$ time. With reference to Table 2.2, the worst-case time complexity of this memoized Spencer's algorithm is: (1) Far lower than the current Spencer NFA simulations; and (2) Equal to that of Thompson's algorithm.

Space complexity The memoization scheme in Listing 10 instantiates a two-dimensional array of size $|Q| \times |w|$. Thus this algorithm will have $\Theta(|Q| * |w|)$ space complexity.

With reference to Table 2.2, the *worst-case* space complexity of this memoized Spencer's algorithm is: (1) Equal to that of the current Spencer NFA simulations; and (2) A factor of

$|w|$ larger than that of Thompson's algorithm. Thus, in a worst-case complexity analysis, memoization is a "free lunch" for a Spencer-style regex engine: it uses the same amount of space, yet yields far lower time complexity. However, in the *average* case this algorithm may exhibit larger space costs than is desirable. We consider improvements in the next section.

8.5 RQ2: How might the space costs of K-regex memoization be reduced?

Despite its potential for a worst-case "free lunch", it would be preferable to have an effective memoization scheme with lower space costs. Although the space cost of memoization is no worse (in the worst case) than that of the backtracking stack already used in many PCRE regex engines, from a practical perspective its cost can be substantial.

For example, consider the space cost of a $|Q| \times |w|$ memo table in the Stack Overflow case study (§3.4). Using the measurement instruments from Chapter 5, Stack Overflow's quadratic regex has an NFA with 7 states, which was evaluated on a problematic input string containing approximately 20,000 characters. Maintaining a memo table for this evaluation would require an order of magnitude more storage than the string itself. As the size of regex NFAs commonly exceeds 30 states (e.g., that is below the 90th percentile for JavaScript, Python, Ruby, Go, and Rust regexes, cf. Figure 5.5c), a $|Q| \times |w|$ memo table could exceed practical memory limitations.

To reduce the space costs of the full memoization discussed in §8.4, this section proposes two classes of improvements. In the first class are two selective memoization schemes [61] (§8.5.1). These schemes reduce space costs by memoizing different subsets of the search states associated with interesting classes of NFA vertices. In the second class of improvements are two alternative encodings for the memo table (§8.5.2). The first scheme eliminates negative table entries using hashing, and the second compresses the table using run-length encoding.

We compare our proposals to the state of the art in §8.5.3. The benefits of the different selection schemes and encodings is considered in our evaluation (§8.6).

8.5.1 Selective memoization schemes

In this section we present two selective memoization schemes intended to reduce the storage costs of memoization. Like the full memoization scheme just presented, these memoization schemes can be applied to the M-NFA derived from the NFA of interest.

Our presentations characterize the time complexity of these schemes by bounding the number of times that a search state $\langle q, i\rangle$ might be visited for any candidate string w. The cost of an NFA simulation can be expressed as (# search states visited) $\times$ (cost per state). As discussed previously in this dissertation, the cost per visited search state in a Spencer-style NFA simulation is $\mathcal{O}(|Q|)$. Therefore, if we bound the number of visits per search state, we bound the time complexity for the NFA simulation on an arbitrary candidate string.

These selective memoization schemes vary in their ability to eliminate super-linear behavior in $|w|$, the length of the candidate string. The first scheme eliminates *all* redundant visits during NFA simulation, whether caused by finite or infinite ambiguity (§8.5.1.3); each of the $|Q| \times |w|$ search states is visited at most once. Building on prior analyses, we show that the second scheme eliminates redundant visits caused by infinite ambiguity (§8.5.1.4). However, it will not eliminate super-linear behavior due to finite ambiguity, which can be as large as $\mathcal{O}(2^{|Q|})$.

Each of these schemes is expected to have lower space costs than the one that precedes it. This decrease is because the NFA vertices selected by the full memoization scheme from §8.4 are a superset of those selected in the first selective scheme (§8.5.1.3), which in turn are a superset of those of the second selective scheme (§8.5.1.4).

The selective memoization schemes proposed in this section each target a set of vertices that is more than is needed, i.e., an over-approximation of the vertices actually necessary to memoize to achieve the desired reduction in redundancy. The over-approximation is used because it is expensive to accurately select the necessary vertices. The larger vertex-sets can be identified at much lower computational cost.

Comparison to the theoretical state of the art The expected performance of these schemes is summarized in Table 8.3. These algorithms improve the time complexity of Spencer's algorithm from exponential to polynomial in $|Q|$, with varying space complexities.[7] However, comparing Table 8.3 to Table 2.2, it is clear that these schemes do not improve the state of the art for the K-regex recognition problem. Using memoization, Spencer's algorithm can match the best known time complexity for this problem, but at the cost of multiplying the best known space complexity by a factor dependent on the structure of the automaton. The value of these schemes is that our techniques can improve the performance of the Spencer algorithm — which is the state of practice — without requiring practitioners to introduce major changes in their regex engine.

8.5.1.1 Preliminary definitions

To prove the guarantees of these schemes, we will make use of the following definitions.

Definition 8.5.1 (Simulation position)**.** We define a *simulation position* $\pi = \langle q \in Q, i \in \mathbb{N}^{|w|} \rangle$ as one of the possible simulation positions (search states) encountered during the backtracking search algorithm. Two simulation positions are *different* if they differ in the

[7]The practical space costs can be lowered using encoding schemes (§8.5.2).

Table 8.3: ***Space and time complexity of selective memoization schemes.*** *Predicted performance of the selective memoization schemes, ordered from highest to lowest space cost. All space complexities describe only the cost of storing the memoization records; regex engines must also track the backtracking stack at a cost of $\mathcal{O}(|Q| * |w|)$. The time complexity of the final approach includes the term $f(Q)$, which depends on the ambiguous structure of the regex's NFA.*

Selected vertices	**Fixes inf. ambig.**	**Fixes fin. ambig.**	**Time cxty.**	**Space cxty.**
None: Spencer's algorithm	✗	✗	$\mathcal{O}(\|Q\|^{\|w\|})$	—
$\Phi_{all} = Q$: All vertices	✓	✓	$\mathcal{O}(\|Q\|^2 * \|w\|)$	$\mathcal{O}(\|Q\| * \|w\|)$
$\Phi_{in-deg>1}$: in-degree > 1	✓	✓	$\mathcal{O}(\|Q\|^2 * \|w\|)$	$\mathcal{O}(\|\Phi_{in-deg>1}\| * \|w\|)$
$\Phi_{quantifier}$: loop destinations	✓	✗	$\mathcal{O}(\|Q\|^2 * \|w\| * f)$	$\mathcal{O}(\|\Phi_{quantifier}\| * \|w\|)$

automaton vertex q or the candidate string index i. If a simulation position is subscripted π_i, we may denote its automaton vertex as q_i.

Definition 8.5.2 (Simulation path)**.** We define a *simulation path* of simulation positions, denoted $\Pi = \pi_0\pi_1 \ldots \pi_n$. This represents a valid sequence of simulation positions (search states) visited by the backtracking algorithm. In a simulation path, π_0 is the position $\langle q_0, 0\rangle$, and each π_i is in $\delta(\pi_{i-1})$. Two simulation paths are *different* if they are of different lengths, or if at some index i they contain different simulation positions, i.e., are at different automaton vertices.

Definition 8.5.3 (Bounded ambiguity)**.** Let A be an ε-free NFA. We define its *bounded ambiguity* as:

$$\begin{gathered} \mathit{boundedAmbig}(A) = \max_{0 \le i \le |Q|} \Big(\max_{s,t \in Q} \Big(\\ \#\text{ distinct simulation paths } s \rightsquigarrow t \text{ of length } i \\ \Big) \Big) \end{gathered}$$

Note that $\mathit{boundedAmbig}(A)$ differs from the ambiguity of A. An automaton can be infinitely ambiguous, i.e., increasingly-long candidate strings can be defined whose ambiguity is larger than any finite bound. In contrast, our definition of $\mathit{boundedAmbig}(A)$ captures the maximum possible ambiguity for strings of length no more than $|Q|$.

8.5.1.2 Assumptions (M-NFA pre-processing steps)

In our theorems and proofs, we assume that the M-NFAs involved have two additional properties: having one accepting state, and being ε-free. These properties are standard proof tactics for automata [194, 301].

First, we assume that the M-NFAs are modified to have a single accepting state q_F. This ensures that if a candidate string is ambiguous, then the ambiguous paths all terminate at the same vertex q_F. Any M-NFA can be converted with no change in its language: introduce q_F, direct ε-edges to it from the vertices in F, and update F to $F = \{q_F\}$.

Second, we assume that the M-NFAs are ε-free. This has the convenience of ensuring that the string index i increases for consecutive simulation positions in a simulation path, i.e., every step consumes a character from w. Any M-NFA can be converted with no change in its language: δ must be defined as δ_ε, computing the ε-closure such that every transition consumes a character from w. However, for the standard Thompson NFA construction, our proofs hold with minor modifications.

8.5.1.3 Select vertices with in-degree > 1

Intuition This scheme selects for memoization the vertices $\Phi_{in-deg>1} \subseteq Q$ that have in-degree greater than one. Memoizing this subset of states suffices to eliminate redundant visits, making the M-NFA unambiguous. Why? In order for there to be redundancy in the M-NFA simulation, we must reach the same search state $s = \langle q, i \rangle$ twice. To reach the same search state twice, there must have been a non-deterministic choice (a fork in the simulation path), and both legs of this fork must have reached the same search state. To reach the same search state, the two paths must at some point have converged. In order for two paths to merge, there must have been some node with in-degree > 1. If we memoize visits to this node, we prevent more than one visit to the search state s.

An illustration is given in Figure 8.3.

Soundness The following theorem formalizes this intuition.

Theorem 8.5.4. Suppose an M-NFA is simulated with a memoization policy targeting $\Phi_{in-deg>1}$. Then every simulation position $\pi = \langle q, i \rangle$ will be visited at most once, and the M-NFA is unambiguous.

Proof. The proof proceeds by contradiction. Suppose the $\Phi_{in-deg>1}$ memoization scheme is employed but some search position π_k is visited twice. For this to be the case, there must be a "fork in the road", as illustrated in Figure 8.4. More formally, this means the M-NFA simulation traverses different simulation paths Π_a and Π_b,

$$\Pi_a = \pi_{a_0} \dots \pi_{a_i} \dots \pi_{a_j} \dots \pi_{a_k} \dots, 0 \leq k \leq |w|$$
$$\Pi_b = \pi_{b_0} \dots \pi_{b_i} \dots \pi_{b_j} \dots \pi_{b_k} \dots, 0 \leq k \leq |w|,$$

such that:

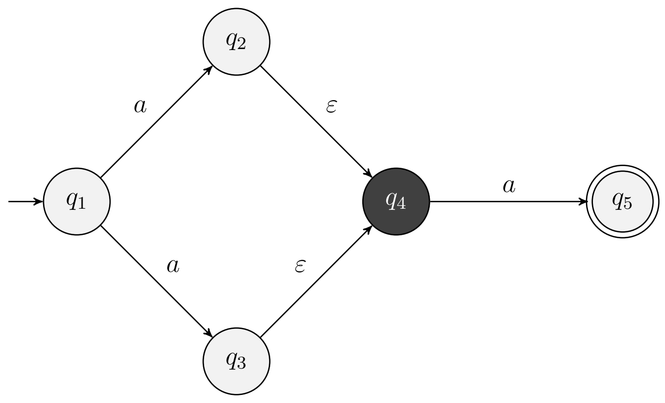


Figure 8.3: ***Automaton used to illustrate memoizing*** $\Phi_{in-deg>1}$*, the vertices with in-degree* > 1*. This figure shows the NFA produced by Thompson's construction for the regular expression* $/(a|a)a/$*, slightly reduced for clarity of presentation.*

In this memoization scheme, we will only memoize visits to the shaded vertex, $\Phi_{in-deg>1} = \{q_4\}$*. This memoization is sufficient to prevent redundant visits to* q_4 *and* q_5 *on the input "aaa!"; we take the upper route first, but then the lower route is short-circuited once we backtrack. We need not memoize visits to* q_1*,* q_2*,* q_3*, nor* q_5*, to achieve this short-circuiting.*

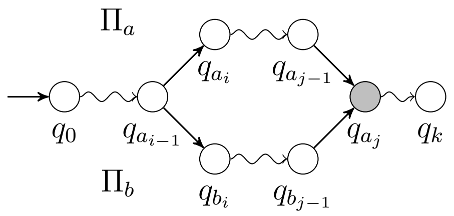


Figure 8.4: ***Illustration for the proof of theorem 8.5.4.*** *Vertex* $q_{a_j} = q_{b_j}$ *must have in-degree* > 1 *because it is the first point of convergence after the split at vertex* $q_{a_{i-1}} = q_{b_{i-1}}$*.*

1. The paths diverge. At some $i < k$, $\pi_{a_i} \neq \pi_{b_i}$, e.g., at a point of non-determinism.
2. The paths converge. There is some smallest j, $i < j \leq k$, such that $\pi_{a_j} = \pi_{b_j}$, and so $q_{a_j} = q_{b_j}$.

The paths must diverge, else they would not be different and π_k would be visited only once. They must converge, else $\pi_{a_k} \neq \pi_{b_k}$. Now, because the paths converge, the vertex in $\pi_{a_j} = \pi_{b_j}$ must have in-degree > 1. In more detail, since j was the earliest point of convergence after $i-1$ on the two paths, it must be that $q_{a_j} \in \delta(q_{a_{j-1}}, w_{j-1})$ and likewise $q_{b_j} \in \delta(q_{b_{j-1}}, w_{j-1})$. Since $\pi_{a_{j-1}} \neq \pi_{b_{j-1}}$, we have $q_{a_{j-1}} \neq q_{b_{j-1}}$, and so the in-degree of $q_{a_j} > 1$.

But if $q_{a_j} \in \Phi_{in-deg>1}$, the M-NFA simulation will not traverse both Π_a and Π_b. We are memoizing all visits to automaton vertices with in-degree > 1. Without loss of generality, suppose we first visit q_{a_j} via the simulation path Π_a, thus marking π_{a_j} in the memo function M. When we backtracked to $\pi_{a_{i-1}}$ and evaluated the alternative path Π_b, at $\pi_{b_{j-1}}$ we would have found the path eliminated by the memo function: $q_{a_j} \notin \delta_M(q_{b_{j-1}}, w_{j-1})$. So we cannot reach the search position $\pi_{a_j} = \pi_{b_j}$ more than once along these paths, because Π_b would terminate at $\pi_{b_{j-1}}$.

$\square$

Time complexity The vertices in $\Phi_{in-deg>1}$ can be identified as part of the NFA construction, at constant additional cost. This memoization scheme then guarantees that every search state $\langle q, i \rangle$ will be visited at most once. Thus, by the same argument as in §8.4, the NFA simulation will complete in $\mathcal{O}(|Q|^2 * |w|)$ steps.

Space complexity This memoization scheme requires memoizing visits to the vertices with in-degree > 1. It thus requires space $\Phi_{in-deg>1} \times |w|$.

Unnecessary memoization Only some of the vertices in $\Phi_{in-deg>1}$ are actually necessary to memoize to avert redundant visits. To illustrate this, suppose an automaton such that $F = \{q_{in-deg>1} \in \Phi_{in-deg>1}\}$. If this NFA is unambiguous, then memoizing visits to $q_{in-deg>1}$ is unnecessary; by definition, there can only be one unique path to it for any candidate string, and so no search states will be visited redundantly. Consider, for example, the NFA associated with the regex `/a(bc)a/|`. This NFA has a vertex with in-degree > 1, but it is clearly unambiguous. However, identifying the ambiguous vertices in an NFA is computationally expensive (§8.5.1.4), while identifying $\Phi_{in-deg>1}$ is cheap.

In this and the next memoization scheme, the unnecessary memoization occurs because the vertices are selected based on the automaton's "skeleton". The vertex selection considers the existence of edges (potential ambiguous paths), but ignores the labels on the edges.

Why $\Phi_{in-deg>1}$? The choice of $\Phi_{in-deg>1}$ may strike the reader as arbitrary. Why not $\Phi_{in-deg=1}$, or $\Phi_{out-deg=1}$, or $\Phi_{out-deg>1}$?

We observe that vertices with in-degree 1 or out-degree 1 are "dominated" [334] by the vertices with larger degrees. There is no opportunity for non-deterministic choices at the degree-1 vertices.

Now consider the memoization of the vertices $\Phi_{out-deg>1}$. These are the states from which non-deterministic choices are possible. Let us take for example the M-NFA that memoizes $\Phi_{out-deg>1} = \{q_1\}$ from Figure 2.11a, when simulated on the input $w =$ "aaa!". The memoization strategy will not eliminate redundant visits to search states. Each of the search states $\langle q_1, i \rangle$ will be visited once, but the search state $\langle q_2, |w| \rangle$ will still be visited $|w|^2$ times. In terms of the proof of theorem 8.5.4, ambiguity became *possible* when the paths π_a and π_b diverged at $q_{a_{i-1}} = q_{b_{i-1}}$ ($q_{a_{i-1}} \in \Phi_{out-deg>1}$), but it was only *realized* when the paths converged again at $q_{a_j} = q_{b_j}$. Both paths must be explored until q_{a_j} is reached; it is only when two paths converge that the exploration becomes redundant.

8.5.1.4 Select quantifier destinations

In the previous scheme we showed that redundant visits can be eliminated without memoizing all NFA vertices. In this scheme we will further reduce the set of memoized vertices, at the cost of permitting some redundancy during M-NFA simulation.

Intuition This scheme selects for memoization the vertices $\Phi_{quantifier} \subseteq Q$ that are *quantifier destinations*, i.e., the automaton vertices to which the "back-edges" introduced by a * or + are directed. These vertices are illustrated in Figure 2.5. Why is this memoization scheme effective? If the NFA is ambiguous, then there may be multiple cycle-free simulation paths of the same length that end in the same search state. Without a cycle, there cannot be more such paths; with a cycle, the ambiguity can compound. A cycle can only be introduced in a simulation path by visiting one of the vertices in $\Phi_{quantifier}$. By memoizing these vertices, the compounding of the ambiguity is prevented.

However, some redundancy is still possible, as illustrated in Figure 8.5.

Soundness The following theorem formalizes this intuition.

Theorem 8.5.5. Let the memo function track simulation positions involving the M-NFA vertices that are "quantifier destinations", $\Phi_{quantifier}$, i.e., the automaton vertices to which any back-edges in a topological sort from q_0 are directed (cycle ancestors). Then the M-NFA is finitely ambiguous. A simulation position involving a vertex t will be visited at most (1) Once if $t \in \Phi_{quantifier}$; (2) *boundedAmbig*(A) times if t is not reachable from a cycle ancestor; and (3) $|\Phi_{quantifier}| \times |Q| \times$ *boundedAmbig*(A) times otherwise.

Put simply, this theorem states that when back-edges can be taken at most once from any simulation position, then ambiguity in the simulation cannot compound. The simulation will retain any ambiguity in its "cycle-free" analog (i.e., a variant that has the back-edges removed). The ambiguity may increase as the result of back-edges taken at different offsets, but it remains bounded.

Proof. Choose a target simulation position $\pi = \langle t \in Q, i \rangle$. We will show the visit bound for each case.

Case $t \in \Phi_{quantifier}$: If $t \in \Phi_{quantifier}$, then the memo function ensures that π is visited at most once.

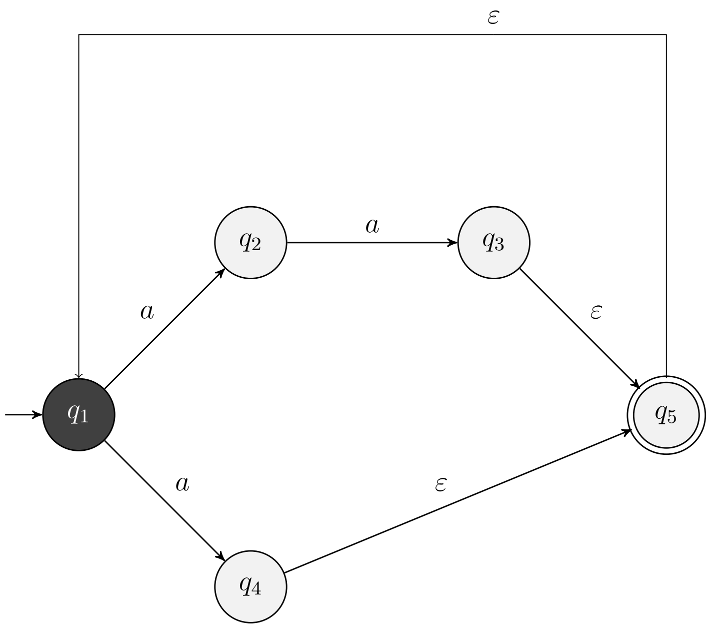


Figure 8.5: ***Automaton used to illustrate memoizing*** $\Phi_{quantifier}$*, the "loop" vertices that are the destinations of unbounded quantifiers. This figure shows the NFA produced by Thompson's construction for the regular expression /(aa|a)*/, slightly reduced for clarity of presentation. In this memoization scheme, we will only memoize visits to the shaded vertex,* $\Phi_{quantifier} = \{q_1\}$.

On the candidate string $w = aaa$, there are two M-NFA simulation paths that visit the simulation state $\langle q_5, 3\rangle$. From $\langle q_1, 0\rangle$, one takes the longer path through the disjunction and the other takes the shorter, reaching $\langle q_1, 2\rangle$ and $\langle q_1, 1\rangle$, respectively. From either of these distinct simulation states, $\langle q_5, 3\rangle$ can be reached by using the shorter or longer path through the second disjunction.

This redundancy is not possible using the previous memoization policy targeting $\Phi_{in-deg>1}$, which would also memoize visits to q_5.

In terms of theorem 8.5.5, $|Q| = 5$, $|\Phi_{quantifier}| = 1$, and boundedAmbig(A) $= 8$*, for a limit of 40 visits to any simulation state. The upper bound is not tight in this example because there are not* boundedAmbig(A) *distinct paths from the vertex in $\Phi_{quantifier}$ to each other vertex for each string length $0 \leq |w|$. Instead, many of these paths in* boundedAmbig(A) *are obtained via cycles through q_1 and are eliminated by the memoization scheme.*

Case $q \in \Phi_{quantifier} \not\rightsquigarrow t$: If there is no path from a cycle ancestor to t, then every simulation path reaching π must be cycle-free and thus contain $\leq |Q|$ positions. The bound then follows from the definition of *boundedAmbig*(A). This result also covers the case when $\Phi_{quantifier} = \emptyset$.

Case $q \in \Phi_{quantifier} \rightsquigarrow t$: We partition the space on i.

Clearly, if $i \leq |Q|$ then π can be visited at most *boundedAmbig*(A) times. So suppose $i > |Q|$. Consider two observations. First, any simulation path containing more than $|Q|$ positions must include a cycle. Second, for the same reason, after a simulation path makes its final visit to a simulation position involving some $q \in \Phi_{quantifier}$, that simulation path must terminate within $|Q|$ steps. This is because the back-edges to $\Phi_{quantifier}$ are the only means of introducing a cycle, and without further cycles a simulation path must come to an end.

Now then, as we have supposed that $i > |Q|$, so the distinct simulation paths to π must all include some "cycle" simulation position $\pi' = \langle q \in \Phi_{quantifier}, j \rangle$ at most $|Q|$ steps beforehand. Recall that the memo function assumed in this theorem will prevent more than one simulation path through each such π'. There are $|\Phi_{quantifier}| \times |Q|$ possible cycle positions π', so all of the distinct simulation paths to π must share at most $|\Phi_{quantifier}| \times |Q|$ distinct simulation path prefixes. From these π', each simulation prefix may diverge up to *boundedAmbig*(A) times to reach π. Multiplying, we obtain an upper bound of $|\Phi_{quantifier}| \times |Q| \times boundedAmbig(A)$ distinct simulation paths that can reach π. □

The use of an ε-free M-NFA simplifies the argument, but the claim holds with minor modifications for the standard Thompson NFA. In particular, δ must be defined as δ_ε, computing the ε-closure such that every transition consumes a character from w.

Time complexity The vertices in $\Phi_{quantifier}$ can be identified as part of the NFA construction, at constant additional cost. This memoization scheme then guarantees that under M-NFA simulation, every search state $\langle t, i \rangle$ will be visited at most $|\Phi_{quantifier}| * |Q| * ambig(A)$ times. There are $\mathcal{O}(|Q| * |w|)$ such search states, and it costs $\mathcal{O}(|Q|)$ to visit each of them. We therefore have a time complexity bound for this M-NFA simulation of:

$$\mathcal{O}(\ (|\Phi_{quantifier}| * |Q| * ambig(A)) * (|Q|^2 * |w|)\ ) = \mathcal{O}(|Q|^3 * |w| * |\Phi_{quantifier}| * ambig(A)).$$

As is clear from the bound, this memoization scheme eliminates super-linear behavior caused by infinite ambiguity. The time complexity of the simulation grows only linearly with $|w|$. This memoization scheme will *not* eliminate finite ambiguity, nor super-linear behavior caused by finitely-ambiguous sub-structures, because such sub-structures do not involve loops (else they would be infinitely ambiguous).

Space complexity This memoization scheme requires memoizing visits to the vertices with in-degree > 1. It thus requires space $|\Phi_{quantifier}| \times |w|$. We note that all quantifier vertices have in-degree > 1, so $\Phi_{quantifier} \subseteq \Phi_{in-deg>1}$.

Unnecessary memoization To achieve the same time complexity reduction, we conjecture that only the NFA's Wüstholz pivot nodes need be memoized. Based on the Thompson construction, these vertices $\Phi_{pivot} \subseteq \Phi_{quantifier}$. To see this, recall that a Spencer-style backtracking regex engine first uses the Thompson construction (§2.2.2.3) to build an NFA, and then simulates it on the input (Listing 3). To be infinitely ambiguous, this NFA must permit simulation paths that visit a vertex more than once, i.e., a cycle. In the Thompson construction, cycles can only arise from the use of an unbounded quantifier (cf. Figure 2.5), which introduces a back-edge into the automaton. Therefore, *an automaton's Wüstholz pivot nodes are always a subset of the destinations of its loops.*

This memoization scheme targets a super-set of Φ_{pivot} because identifying pivot nodes is computationally expensive. Weideman et al. show that the 90% of regexes that do not contain pivot nodes can be identified with $\mathcal{O}(|A|_\delta{}^3) = \mathcal{O}(|Q|^6)$ time complexity [335]. They also show that identifying the pivot nodes for the remaining 10% of super-linear regexes requires $\mathcal{O}(2^{|Q|})$ time complexity. This time complexity may be acceptable for small regexes, but might be problematic in a general-purpose regex engine because regexes with larger automata (e.g., $|Q| \geq 30$ states) are not unusual.

8.5.2 Efficient encodings

The selective memoization schemes discussed in §8.5.1 identify a subset of automaton vertices for which memoization will be performed. For each selected automaton vertex, the approaches so far maintain a *visit vector* denoting at which of the $|w|$ candidate string indices the vertex has been visited. Thus far we have considered using bitmaps to record the search states visited during the backtracking NFA simulation, at a cost of $|w|$ for each visit vector.

Here we consider how to reduce the cost of storing the visit vectors using more efficient encodings. These approaches do not affect the soundness of the memoization scheme to which they are applied.

8.5.2.1 Eliminate negative entries

Intuition and Approach The $|w|$ cells of each of the k visit vectors are initialized to 0 (unvisited) and updated to 1 when visited. Instead of tracking all of these cells, we can instead track only the visited cells; a *negative entry* in the visit vector implies that it has not been visited.

If we use a data structure with efficient random access and update times for sparse data, we can store only the visited cells in the now-sparse visit vector. A hash table keyed on the tuple $\langle q, i\rangle$ is a natural implementation choice. As discussed in §8.2, a memo table based on hashing is the memoization approach used in functional programming contexts.

Time complexity This approach has the same worst-case time complexity as the full $\Theta(|Q| * |w|)$ memoization table. However, it may exhibit larger constants. A pre-allocated bitmap, i.e., a two-dimensional array, can be accessed in constant time. A dynamic structure that can mark search states on demand may not share these characteristics. For example, a hash table on the search state tuples $\langle q, i\rangle$ will have amortized $\mathcal{O}(1)$ insertions and lookups, but is expected to have higher constant costs than a pre-allocated array.

Space complexity This encoding scheme has input-dependent space costs. If every search state must be explored, it will have the same worst-case state complexity as full memoization: $\mathcal{O}(|Q| * |w|)$. However, in the case of a successful match or a non-pathological mismatch, a much smaller subset of search states may need to be explored. For example, if the NFA is unambiguous, it will determine whether it accepts a candidate string in $\mathcal{O}(|w|)$ steps. In such a case, this encoding scheme will require at most $\mathcal{O}(|w|)$ space.

8.5.2.2 Compressed memoization for bitmap memo tables

Intuition and Approach The memo tables used so far are a potential target for compression. A high compression ratio may be achievable, because the table has a small alphabet and the visit vectors in the table are expected to have low entropy.

Small alphabet: When memoization is applied in the context of arbitrary functions in functional programming, the range of these functions is unpredictable [197]. In the context of an NFA simulation for K-regexes, the range of the function we memoize is small: $\{0, 1\}$ (Boolean). Because the alphabet of a visit vector is small, many table entries must be repeated (i.e., will be either 0 or 1), making the memoization table a natural target for compression if it has low entropy.

Expected low entropy: We expect the backtracking search strategy necessitated by PCRE's leftmost-greedy semantics to yield visit vectors with low entropy. Because it follows a top-to-bottom, left-to-right search for automaton nodes with quantifiers, the simulation explores nearby string indices before the further ones. This has the consequence that any runs within these automaton nodes will *accrete* as characters are processed, yielding consistently low entropy within each column. This observation has separately been described in terms of the locality of NFA traversals [78].

As a result of these two properties, it is reasonable to expect that the memo tables in this context will be compressible. Many states may be visited at only a few distinct indices of the input string, resulting in intervening compressible runs of 0's due to unvisited states. Other states — those that are quantifier destinations — may be visited at many indices. Some of these visits will be especially compressible. For unbounded monadic quantifiers like `/a+/` or `/.*/`, once an NFA simulation reaches this quantifier, it will attempt the regex matches upon consuming all available *adjacent* characters, accreting a compressible run of 1's within the bitmap. Such quantifiers are common — in the polyglot regex corpus described in Chapter 5, among the 253,216 regexes that use an unbounded quantifier, fully 103,664 (41%) include the catch-all unbounded quantifier `/.*/` or `/.+/`.

As a compression strategy, we propose to use run-length encoding (RLE) [287]. We implemented a standard RLE scheme using fixed-width runs of length 1. Our implementation uses a separate binary search tree for each automaton vertex being memoized. The tree elements are runs keyed by their offsets [250]. This structure offers $\mathcal{O}(\log k)$ accesses for a vertex with k runs.

Time complexity In the worst case, this approach may increase the time complexity of the memoization scheme to which it is applied. If the search pattern is pathological, there will be $\mathcal{O}(|w|)$ distinct runs for each automaton vertex that is memoized, incurring a cost of $\mathcal{O}(\log |w|)$ each time a search state is memoized or queried. If k automaton vertices are memoized, the time complexity *due to compression* will be $\mathcal{O}(|k| * |w| * \log |w|)$. In the best case, this approach may achieve perfect compression, requiring merely $\mathcal{O}(1)$ to memoize or query a search state. In this case the worst-case time complexity of the original scheme is preserved.

Space complexity For each automaton vertex being memoized, applying an RLE encoding to its visit vector requires between $\Omega(1)$ and $\mathcal{O}(|w|)$ space. If the encoding scheme is ineffective, there will be no change in the space complexity. If it is effective, the space complexity of the underlying memoization scheme may be significantly reduced.

This encoding approach may have lower space costs than eliminating negative entries does. The initial $|w|$ negative entries in the visit vectors can be compressed to a single run. If the assumption of locality holds in typical cases, then instead of filling a memoization table with positive entries, RLE can just extend existing runs.

8.5.3 Comparison to existing memoization schemes

The full memoization scheme described in §8.4 is an exhaustive version of the scheme followed by RE2. RE2 only applies such a table when $|Q| \times |w| \leq 32$ KB [134], ensuring a constant space bound but failing to protect queries on long regexes or long candidate strings. For example, this scheme would not have prevented the outage discussed in our Stack Overflow case study (§3.4), where the input string was over 20 KB long and the regex's automaton had around 10 vertices.

The approach of memoizing quantifier destinations (§8.5.1.4) is a stronger version of the approach taken by the Perl regex engine. The Perl regex engine memoizes a subset of the quantifier destinations, namely those associated with "complex" sub-patterns. The $\Phi_{quantifier}$ scheme we described extends the Perl regex engine's approach to all quantifier destinations. We have provided the first argument of the soundness of the Perl regex engine's approach in terms of finitely and infinitely ambiguous regexes, and we have provided the first upper bound on the time complexity of a rigorous version of its memoization scheme. The Perl regex engine's greater-than-$\mathcal{O}(|Q| * |w|)$ worst-case behavior under memoization may come as a surprise to its maintainers and others who have claimed that it achieves a lower bound [85, 87, 132].

We are not aware of selective memoization schemes that target the automaton vertices $\Phi_{in-deg>1}$ with in-degree > 1. We are not aware of evaluations of alternative encoding schemes in this context. We are not aware of previous applications of RLE to a Boolean memoization table.

8.6 RQ3: Experimentally, what are the space and time costs of K-regex memoization?

The memoization schemes described in §8.5 offer attractive worst-case performance benefits. They incur varying degrees of space complexity, which may be offset by efficient encodings of the memo table.

In this section we consider two aspects of the space and time costs of K-regex memoization:

Selective memoization The size of the memoization targets for the selective memoization schemes, relative to full memoization; and

Encodings The space benefits of the encoding schemes relative to each selective memoization scheme, contrasted with the additional time costs thereof.

First I introduce the prototype regex engine in which we will measure these costs (§8.6.1). Then, in §8.6.2 we statically measure the size of the memoization targets, and in §8.6.3 we dynamically measure the impact of the encoding schemes on the memoization tables in each approach.

8.6.1 Prototype regex engine and supported regexes

We prototyped the selective memoization and encoding schemes on top of a Spencer-style backtracking regex engine published by Cox for experimental purposes [132]. We will refer to this engine as the *baseline regex engine*. The baseline regex engine supports K-regexes as well as some E-regex extensions: capture groups (CG), non-greedy quantifiers (LZY), and the ANY character class. The baseline regex engine was further limited to a pre-allocated backtracking stack of at most 1,000 search states. To increase the number of supported regexes, we added support for several additional features; anchors (STR, END), various commonly-used built-in character class shorthands (WSP, DEC, WRD, NWSP, NWRD, NDEC), and special handling for whitespace notation (\n, \t). To support longer input strings in our evaluation, we made the backtracking stack dynamically sized, with no bound.

In these experiments we are concerned with properties of typical regexes that are independent of their match or parse behavior (semantics). Therefore we used syntactically supported regexes from the polyglot corpus collected in Chapter 5, regardless of their origin programming language(s). Without modification, the baseline regex engine supports an estimated 191,006 of the regexes (35%) from this regex corpus. We will refer to these as the *supported regexes*.

We analyze the space and time costs of memoization on the supported subset of the corpus.

8.6.2 Evaluation of selective memoization

8.6.2.1 Methodology

The space costs of the selective memoization schemes depends on the size of the sets of vertices selected for memoization: $\Phi_{in-deg>1}$ and $\Phi_{quantifier}$. To determine the viability of the schemes, we measured the typical sizes of these sets for the supported regexes. These measurements were collected by constructing the NFAs, then counting $|Q|$ and the appropriate vertices in each subset.

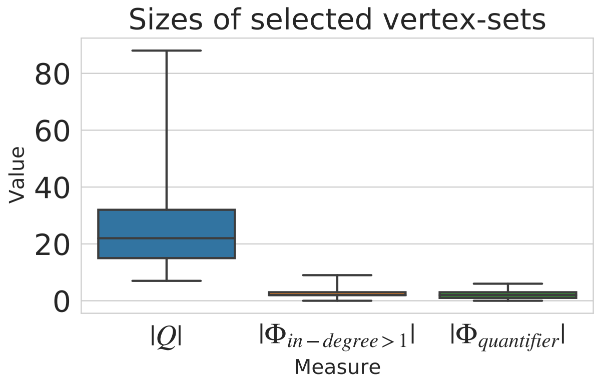
8.6.2.2 Results

We evaluated the supported regexes on our prototype engine. It was able to measure 152,110 (29%) of the regex corpus, and 14,944 (24%) of the super-linear regexes.

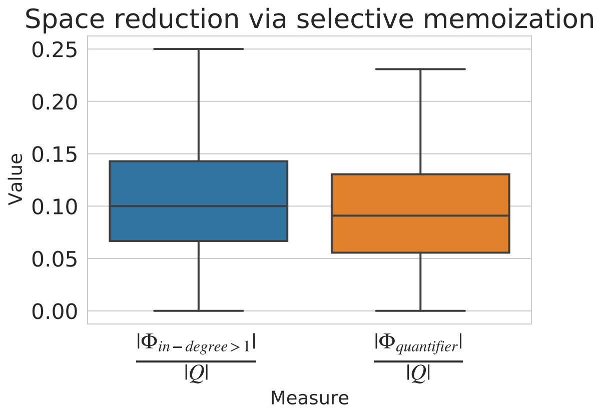
Figure 8.6 shows our findings for the measured regexes from these two groups. Figure 8.6a shows the distribution of the sizes of the vertex-sets targeted by the various selective memoization schemes. Figure 8.6b shows the ratios of the sizes.

8.6.2.3 Analysis

From Figure 8.6b, we note that the selective memoization schemes offer significant space savings over the full memoization scheme. The 75th percentile of both all regexes and only



(a) Vertex-set sizes.



(b) Vertex-set size ratios.

Figure 8.6: ***Raw sizes and relevant ratios of various vertex-set sizes.*** *Whiskers indicate the (1, 99)th percentiles. Outliers are not shown. The figures for the supported subset (shown) and the supported super-linear subset (not shown) are similar.*

the super-linear regexes is between 30 and 40 (31 and 38, respectively), and on long inputs the ($|Q| \times |w|$) cost of full memoization would be significant. The 75th percentile value of $|\Phi_{in-degree>1}|$ is much lower, only 3 across all regexes and 4 for the measured super-linear subset. The value for $|\Phi_{quantifier}|$ is similar or equal in both cases.

From Figure 8.6b, we observe that both schemes offer *similar* space reductions.

In brief, for the measured regexes we found that $|Q| \gg |\Phi_{in-deg>1}| \approx |\Phi_{quantifier}|$. We therefore recommend that the maintainers of backtracking regex engines incorporate a $|\Phi_{in-deg>1}|$-based memoization scheme, as it has similar space costs but stronger time complexity guarantees. However, the outlier values of $\Phi_{in-degree>1}$ are an order of magnitude larger than those of $\Phi_{quantifier}$, so regex engine developers might use $\Phi_{quantifier}$ as a back-up option.

8.6.3 Evaluation of encodings for various memoization schemes

The prototype regex engine supports all memoization schemes (1 full, 2 selective) and all encoding schemes (bitmap, hash table, RLE compression). This yields nine distinct memoization approaches, plus two baselines: the non-memoized engine, and the performance of the Perl regex engine's selective memoization scheme. The following methodology and results describes our comparison of these memoization approaches.

8.6.3.1 Methodology

Regexes and inputs From the full set of 51,224 distinct super-linear regexes from the corpus described in Chapter 5, 14,944 were in the supported subset of our corpus. However, due to a clerical error, we evaluated only 13,260 in this experiment. For each of these regexes, we used the input that produced the largest growth in worst-case behavior in our prototype regex engine among those recommended by the super-linear regex detectors.

Metrics We measured the amount of space taken by each combination of selection and encoding scheme. In detail, the space measures we report are:

- For the bitmap encoding scheme, we report the space cost for a given regex as $|\Phi| \times |w|$, where Φ is the set of vertices selected for memoization.
- For the negative encoding scheme, we report the space cost for a given regex as the number of distinct search states $\langle q \in \Phi, i \rangle$ that are visited during the simulation. This value is maximized at the conclusion of the simulation.
- For the RLE encoding scheme, we collect the maximum number of runs at any point in the simulation for each vertex in Φ. Note that these values may be maximized mid-simulation, e.g., if distinct runs arise but are eventually merged. We report the sum of these counts.

We did not measure the running time. All of the memoization approaches we apply reduce the worst-case performance from super-linear to linear; the reduction in complexity is of greater import than the precise (linear) matching time involved. The time costs may be estimated by multiplying the space costs (number of entries in the data structure) by the cost of an update. The bitmap encoding scheme offers $\mathcal{O}(1)$ updates, negative encoding offers amortized $\mathcal{O}(1)$ updates, and RLE offers $\mathcal{O}(\log k)$ updates where k is the number of runs for a vertex.

Configuration For consistency with the experiments in Chapter 5, we measured space costs using 100,000 pumps of the input for each regex. We used the same number of pumps for Perl, and used a threshold timeout of 5 seconds for Perl's timeout. We permitted our prototype regex engine to run for up to 180 seconds, since our interest is not to evaluate its (non-optimized) running time but rather the space costs of memoization on a realistic corpus of super-linear regexes.[8]

In our experiments we used the simplest RLE encoding, with runs of length 1. We discuss opportunities for improvement on this approach in §8.8.

[8]With 100,000 pumps, 75 of the regex evaluations timed out in our prototype. We attribute this to the impact of instrumentation and logging on such long inputs. With $10,000$ pumps instead of 100,000 pumps, all evaluations finish within the 180 second time limit (most were far faster).

No linear-time regexes We omitted measurements on linear-time regexes, as well as linear-time inputs on super-linear regexes. We did this for two reasons. First, by definition a regex match that completes in linear time will visit a linear number of search states in the length of the input string. Thus, the full bitmap will cost the usual $|Q| \times |w|$ space, and the negative or RLE encoding schemes will cost $\mathcal{O}(|w|)$ space. Second, no memoization scheme is necessary until the simulation exceeds a minimum threshold. For example, the Perl regex engine's memoization scheme is only enabled after the vertices in $|\Phi_{quantifier}|$ have been visited a combined $\Phi_{quantifier} \times |w|$ times. Perl's deferral of memoization does not affect its time complexity guarantees, but reduces the time and space costs in the common case of a linear-time evaluation.

8.6.3.2 Results

Absolute space costs The absolute space costs of the combinations of selection and encoding schemes are illustrated in Figure 8.7. The bitmap ("No encoding") approach costs a constant $|\Phi| \times |w|$ regardless of the content of w. First, compare this to the space cost of the negative encoding. Despite the logarithmic scale, the two costs are comparable; a significant fraction search states are explored by the simulation during a super-linear regex evaluation. This is unsurprising; with 10^5 pumps, we expect approximately 10^5 visits to the search states associated with "pivot" vertices, as well as to any intervening vertices dominated by the pivots. Second, compare the cost of these approaches to the RLE scheme. As suggested by our analysis in §8.5.2.2, RLE offers substantial compression benefits.

Relative space costs Next, we compared the relative costs of these schemes. Figure 8.8 compares the costs of each scheme to the largest possible cost: a $|Q| \times |w|$ bitmap (the leftmost blue bar of Figure 8.7). For the median super-linear regex, depending on the selection scheme the cost of a negative encoding scheme ranges from around 40% to around 10% of a full bitmap. The RLE scheme's performance is better, with a 95th percentile of 12 for $\Phi_{in-deg>1}$ and 10 for $\Phi_{quantifier}$ — in other words, RLE offers a *constant* cost for either of the selection schemes across most of our benchmark suite.

Comparison to Perl Using Perl's regex engine, 17% of these regexes still exhibited super-linear worst-case behavior. Some combination of Perl's selective memoization scheme and its other optimizations still protected the remainder. In contrast, in our prototype regex engine all of the super-linear regex exhibited linear-time behavior under all of the memoization schemes.

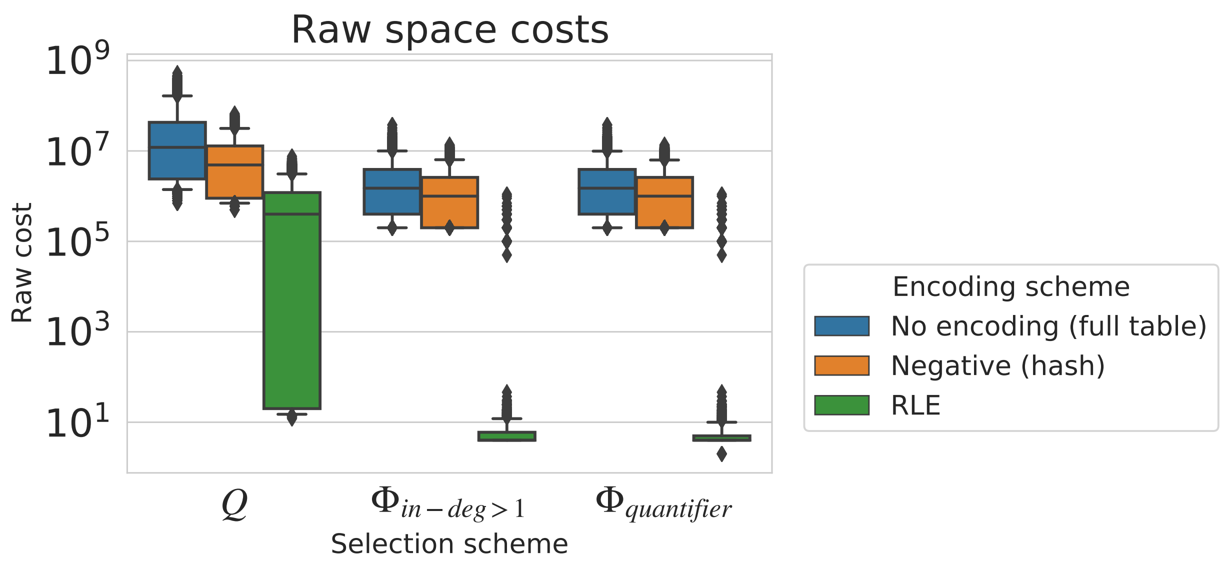


Figure 8.7: ***Absolute space costs of memoization schemes.*** *Absolute space costs with 100,000 (10^5) pumps. The x-axis distinguishes between the selection schemes, and the encoding schemes for each selection scheme are distinguished by colors. Whiskers indicate the (1, 95)th percentiles. Outliers are shown. The units on the y-axis are the absolute costs in terms of allocated storage objects, which vary by selection scheme (§8.6.3.1). Note the log scale on the y-axis.*

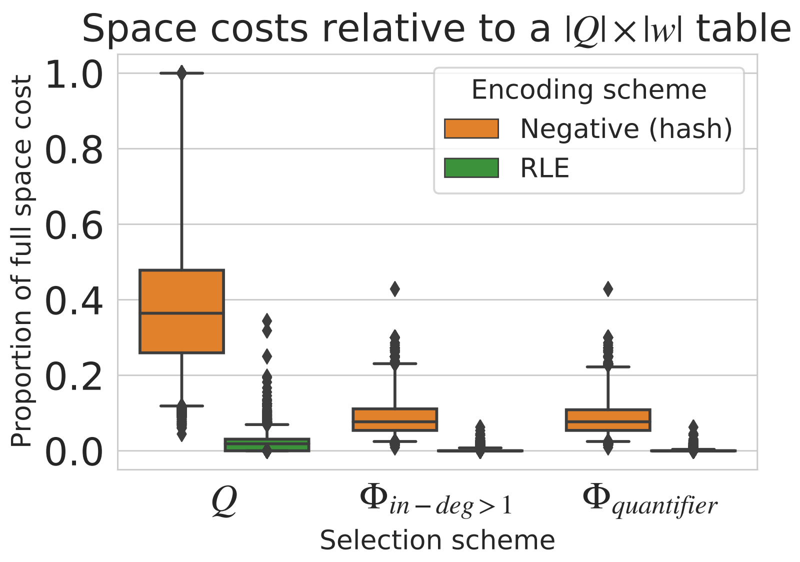


Figure 8.8: ***Relative space costs of memoization schemes.*** *Relative space costs with 100,000 (10^5) pumps. The relative costs are calculated as a proportion relative to the baseline of a $|Q| \times |w|$ bitmap. Whiskers indicate the (1, 99)th percentiles. Outliers are shown.*

8.6.3.3 Analysis

Our results show that selective memoization is a promising approach to address ReDoS. The time and space complexity cost of the various selection schemes was described in Table 8.3. The measurements of Figure 8.6 showed that selective memoization would benefit many regexes from the supported subset of our regex corpus. Figure 8.7 upheld this promise on the supported subset of super-linear regexes from our regex corpus.

Our results also show that RLE may be an effective encoding scheme for the memoization table in the context of automaton simulations. Negative encoding offers some space improvement over a bitmap, but still incurs linear space costs. Figure 8.8 shows that RLE achieves a high compression ratio, offering constant-time space costs for 95% of regexes using either selective memoization scheme.

8.7 RQ4: How might memoization be extended to E-regexes?

The memoization schemes discussed so far have only applied to K-regexes. In this section we will discuss their application to E-regexes. We show the theoretical time and space bounds of E-regexes in their current implementations, and the time and space bounds that can be obtained by extending our memoization techniques to the extended features of interest. We do not evaluate the bounds experimentally.

As noted in §2.4.3, E-regexes can be divided into three classes: K-regexes, K-compatible, and E-regexes. In sections 8.4 and 8.5 we described memoization schemes for the classes of K-regexes and K-compatible regexes. We have not yet considered worst-case behaviors and memoization techniques for E-regexes, the three groups below the double-line of Table 2.3: zero-width assertions, backreferences, and prioritization/backtracking controls. We discuss prior work on these topics in §8.7.1. Then we discuss memoization for zero-width assertions in §8.7.3, and for backreferences in §8.7.4. Memoization is orthogonal to prioritization/backtracking controls, so we treat those E-regex features briefly here.

The prioritization and backtracking controls (LZY, ATM, POS) affect match semantics, and therefore the rules that an NFA simulation must follow. Prioritization (LZY) will change the order in which simulation states are explored, and backtracking controls (ATM, POS) will remove some paths from the simulation. None of these features will increase the cost of a simulation in terms of the number of times each search state is visited. The memoization techniques discussed earlier can be applied to regexes that support these extended features.

8.7.1 Related work on E-regexes

Most prior work on regexes has focused on K-regexes.

Zero-width assertions To the best of our knowledge, §8.7.3 presents the first analysis of the worst-case complexity of regexes with lookaround assertions, and the first application of memoization to reduce this complexity to linear in $|w|$. *Backreferences* Regexes with backreferences (REWBR) have been studied by several researchers. Aho first showed that REWBR problems are NP-hard [63]. Câmpeanu and Santean analyzed a non-memoized REWBR automaton simulation, and showed that it has polynomial space and exponential time complexity [107]. Namjoshi and Narlikar described an extended NFA model to evaluate REWBR, and parameterized the cost of the simulation in terms of the number of backreferences [256].

We introduce a more general f-NFA model that can be used to analyze both zero-width assertions and backreferences (§8.7.2), and in §8.7.4 we refine their worst-case parameterization, evaluate common parameter values in our regex corpus, and consider the cost of supporting REWBR in a memoized NFA backtracking simulation. Using our regex corpus, we report that for *typical* backreference usage the worst-case memoized space and time complexity is polynomial in $|Q|$ and $|w|$.

8.7.2 Generalizing from NFAs to f-NFAs

The typical implementations of zero-width assertions and backreferences can be modeled with minor extensions to the NFA model. In particular, we will extend δ by introducing an additional type of edge, which we will call the f-edge. We will refer to the extended NFA as an f-NFA.

In the original NFA model, the δ function encoded edges $q_a \overset{\sigma \in \Sigma \cup \{\varepsilon\}}{\rightarrow} q_b$. Each edge represented a transition from q_a to q_b. Each transition consumed one character from the candidate string w, or 0 characters in the case of ε-edges.

f-edges differ from typical edges in two ways. First, each f-edge encodes an arbitrary function, not just single-character comparison. Second, an f-edge consumes a variable number of characters. We can denote this $f(w,i) \rightarrow \{\text{True}|\text{False}\} \times \mathbb{N}^{|w|}$ where w is the candidate string and i the current index. An f-function returns `True` if the edge can be taken, else `False`. If the edge can be taken, it also returns an integer corresponding to the number of characters to consume. Note that tracking i also requires adding a counter to the f-NFA, similar to the counters used to identify matching substrings in tagged automata.

Clearly all NFAs are also f-NFAs, with each of the edges corresponding to a simple matching function and consuming 1 or 0 characters. Although the definition of an f-NFA permits complex functions, simple functions suffice to implement zero-width assertions and backreferences. The cost of these functions affects the worst-case time complexity of the f-NFA simulation.

8.7.3 Memoization for zero-width assertions

The typical implementations of zero-width assertions can be modeled with f-edges. Similar to the construction rule for concatenation (Figure 2.7), each of these assertions introduces an additional automaton vertex q_f with an f-edge whose function (1) confirms the corresponding condition is matched, and (2) consumes no characters.

8.7.3.1 Modeling and costs for fixed-width assertions

The functions for start-of-string (STR) or end-of-string (END) return `True` depending on whether the w index is 0 or $|w|$, respectively. The function for word boundaries (WNW) compares $w[i]$ and $w[i+1]$ and returns True if there is indeed a word boundary. The function for non-word boundaries (NWNW) is similar. These functions can be evaluated in constant time, so they add no cost to the original NFA simulation.

8.7.3.2 Modeling regular expressions with lookaround assertions (REWLA)

Typical regex engine implementations To understand the time complexity of evaluating a Regular Expression With Lookaround Assertions (REWLA), we consider their typical implementation in production regex engines. We examined the engines of Perl, PHP, Python, and JavaScript-V8. These regex engines implement REWLA through recursion, supporting some or all regex features within the sub-pattern.[9] The recursion is implicit: these regex engines expand the sub-pattern P into the automaton, match it like a typical sub-pattern with the appropriate orientation for look-behind, but omit advancing the offset into w because the assertions are zero-width.

Production regex engines impose different limits on the sub-pattern P permitted in lookahead and lookbehind assertions. Of the programming languages considered in Chapter 5:

- Rust and Go do not support lookaround assertions.
- For lookahead assertions: all six of the other programming languages support sub-patterns that are K-regexes and some E-regex features (e.g., nested lookaround assertions and backreferences).
- For lookbehind assertions: (a) all six of the other programming languages support sub-patterns that test for a constant string; (b) JavaScript, Java, Ruby, and PHP support

[9] As an optimization, constant sub-patterns can be implemented instead using direct string comparison, as is done in Perl.

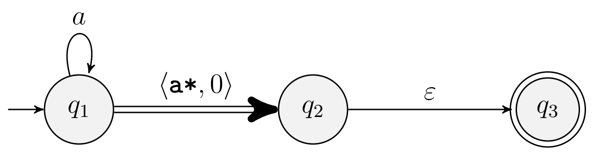


Figure 8.9: ***Automaton used to illustrate the complexity of zero-width assertions.*** *This figure shows the f-NFA produced by Thompson's construction for the regular expression /(a*)(?=a*)/, slightly reduced for clarity of presentation. The f-edge $q_1 \to q_2$ describes the zero-width lookahead assertion. This assertion tests whether the subsequent characters match the pattern /a*/, but consumes no characters.*

variable-length constant strings, e.g., `/(?<=a|aa)/`; and (c) JavaScript and Java support arbitrary K-regexes, e.g., `/(?<=a*)/`.

Modeling As with fixed-width assertions, we can model lookaround assertions using an f-NFA. Positive lookaround assertions request the regex engine to determine whether a sub-pattern P is matched, beginning from $w[i]$ and looking earlier in reverse string order (PLB) or rightward in standard order (PLA). Negative lookaround assertions (NPLB, NPLA) pose the same question but invert the outcome. Thus, these functions can be determined by evaluating the sub-pattern on the appropriate substring of w.

In light of the varied support for the expressivity of the sub-pattern P, and to simplify our analysis, we will model the sub-pattern P in a lookahead or lookbehind assertion as a K-regex or K-compatible. We will refer to such regular expressions as REWLA ("Regular Expressions With Lookaround Assertions").

Example Figure 8.9 shows an example f-NFA for the REWLA `/a*(?=a*)/`.

8.7.3.3 Time complexity of evaluating REWLA with existing algorithms

Under our model, the f-NFA for a REWLA will have a set of normal NFA states Φ_{normal} and a set of f-vertices Φ_f. With each of the f-vertices $q_{f_i} \in \Phi_f$, there is an associated sub-automaton with states Q_{f_i}. We will denote the union of these states as $Q_{R_f} = \Phi_{\text{normal}} \cup \Phi_f \cup \bigcup_i Q_{f_i}$.

Time complexity using backtracking Since typical regex engines use a Spencer-style backtracking NFA simulation for K-regexes, and since each lookaround sub-pattern P may be a K-regex in our model, the time complexity of evaluating the f-NFA associated with an REWLA in such regex engines is *super-exponential.* This complexity arises from two properties:

1. A K-regex may have exponentially many paths for a backtracking NFA simulation to explore; and
2. Zero-width assertions permit some f-NFA simulation steps to have exponential time complexity. By contrast, when simulating a K-regex each simulation step costs $|Q|$ (viz. a character comparison and then evaluation of the transition function δ). Critically, zero-width assertions consume no characters, and thus do not eliminate any paths when they are evaluated.

For example, the REWLA `/^(a|a)*(?<=^b(a|a)*)$/` has time complexity $\Omega(2^{|w|^2})$ on the input $w = a^k b$. To see this, observe that the final offset will be reached $2^{|w|}$ times, and that the look-behind assertion will cost $\Theta(2^{|w|})$ to test each time. Thus we have a cost of $\Omega(2^{|w|} \times 2^{|w|}) = \Omega(2^{2|w|}) = \Omega(2^{|w|^2})$: super-exponential in $|w|$.[10]

Time complexity using a linear-time NFA simulation algorithm Although we are not aware of any implementations in practice, the time complexity of REWLA is lower if a linear-time NFA simulation algorithm is used to evaluate the f-NFA and to perform sub-automaton simulations.[11] Using Thompson's simulation or either our full or $\Phi_{in-deg>1}$ memoization schemes, each of the vertices in $\Phi_{normal} \cup \Phi_f$ is visited at most once for each of the $|w|$ associated search states, at a cost of $|Q|$ per search state. Now, the cost of evaluating each of the f-edges (i.e., simulating an NFA) is $\mathcal{O}(|Q_{f_i}|^2 * |w|)$. This cost is applied each time one of the f-edges is considered by the δ of the f-NFA, which in turn depends on the number of times that search states associated with the f-automaton vertices Φ_f are visited. Since this count is at most $|w|$ per vertex $q_f \in \Phi_f$, we have a time complexity bound of:

$$\mathcal{O}(|\Phi_{\text{normal}}| * |Q_{R_f}| * |w|) \quad + \quad \mathcal{O}(|\Phi_f| * |Q_{R_f}| * |w| * (\Sigma_i |Q_{f_i}|^2 * |w|)).$$

Since $\Phi_{\text{normal}} \subseteq Q_{R_f}$ and $Q_{f_i} \subseteq Q_{R_f}$, this can be bounded by $\mathcal{O}(|Q_{R_f}|^4 * |w|^2)$ — super-linear in $|w|$.

Comparison to a *flattened regex* For comparative purposes, we will also define a *flattened regex* R_{flat} derived from a regex R that contains lookaround assertions. R_{flat} is a K-regex whose NFA representation is of a similar size to the f-NFA representation for R. R_{flat} is the result of replacing the lookaround assertions within the original pattern R with concatenations, resembling how REWLA are implemented in practice. As an example flattening, if $R = /A(?{<}{=}P_1)(?{=}P_2)/$, then $R_{\text{flat}} = /A \cdot P_1 \cdot P_2/$. With this definition, $|Q_{R_{\text{flat}}}| = \Theta(|Q_{R_f}|)$. The cost of simulating the f-NFA for R_f is thus substantially more than the cost of simulating the NFA for R_{flat}, bearing an extra term of $|Q_{R_f}|^2 * |w|$. Intuitively, the additional

[10]We have experimentally confirmed this growth rate in JavaScript-V8.

[11]The regex engines that use Thompson's linear-time algorithm (Rust, Go) do not support lookaround assertions.

cost is incurred because the zero-width assertions do not consume characters; they do not eliminate any simulation paths.

8.7.3.4 Optimizing REWLA using memoization

Using memoization, we can decrease the time complexity of match and parse queries for regexes with zero-width assertions by a factor of $|w|$ compared to the cost using existing linear-time NFA simulation algorithms. Specifically, we can reduce the cost to $\mathcal{O}(|Q^4_{R_f}|*|w|)$, which is $\Theta(|Q_{R_{\text{flat}}}|^4 * |w|)$. This means that although answering queries for regexes with lookaround assertions has a higher time complexity than for similarly-sized K-regexes and K-compatible regexes, the time complexity is still linear in the length of the candidate string w.

This reduction in time complexity follows from a simple observation: for each $q_f \in \Phi_F$, the $\mathcal{O}(|w|)$ simulations of the corresponding sub-automaton will all operate on the same sub-automaton and on some substring of w. Rather than treating them independently, we can benefit from applying what we learned on one simulation to the next one. Suppose we memoize the sub-automaton (i.e., convert it to an M-NFA), and denote the sub-automaton search states in terms of w (and not the sub-string of w beginning at $w[i]$). Suppose further that we preserve the M-NFA's memoization table across the entire simulation of the higher-level f-NFA, i.e., we remember the simulations that began at the $|w|$ different indices i. This is akin to preserving the memoization table across different match and parse queries for the f-NFA itself.[12] If the full or $\Phi_{in-deg>1}$ memoization scheme is used for the sub-automaton, then the cost of testing each lookaround assertion is $\mathcal{O}(|Q_{f_i}|^2 * |w|)$ *amortized over all of the* $\mathcal{O}(|w|)$ *simulations.* If the $\Phi_{quantifier}$ memoization scheme is used, the cost increases following Table 8.3.

Time complexity Under this approach, the time complexity of the full f-NFA simulation becomes

$$\mathcal{O}(|\Phi_{\text{normal}}| * |Q_{R_f}| * |w|) + \mathcal{O}(|\Phi_f| * |Q_{R_f}| * (\Sigma_i |Q_{f_i}|^2 * |w|)).$$

Applying the same reasoning as before, we obtain a time complexity of $\mathcal{O}(|Q_R|^4 * |w|)$.

Space complexity The space complexity of such a simulation is

$$\mathcal{O}(|Q| * |w|) + \mathcal{O}(|w| * \Sigma_i |Q_{f_i}|) = \mathcal{O}(|Q_R| * |w|) = \mathcal{O}(|Q_{R_{\text{flat}}}| * |w|).$$

[12]One might do this if the regex were expected to be queried multiple times for the same candidate string w, e.g., if a user must choose from a fixed set of candidate strings as in a drop-down menu.

This is the same as would be required to simulate a similarly-sized K-regex regex. The space requirements for each sub-automaton could be improved using the techniques described in §8.5.

Remarks The time complexity we have achieved using memoization is unsurprising from an automata-theoretic perspective, because our REWLA model does not add expressive power to a K-regex. The set of regular languages is closed under intersection [281], and lookaround assertions are a means of encoding the intersection of two regular languages — the language of $R_1 = /A(? <= B)/$ is the same as the language of $R_2 = A \cap B$. The Rabin-Miller Cartesian product construction for the intersection of two regular languages with vertices Q_A and Q_B yields an automaton with $|Q_A| \times |Q_B|$ states [281], and an efficient simulation of this automaton would take $\mathcal{O}((|Q|^2)^2 * |w|) = \mathcal{O}(|Q|^4 * |w|)$ steps, just as can be obtained through our memoization approach. However, we believe that our algorithm for a linear-time evaluation expressed in terms of the f-NFA used by existing regex engine implementations, without requiring the calculation of an explicit automaton intersection, may be of use to practitioners. This is especially true for real-world lookaround implementations, some of which support a more expressive lookaround sub-pattern P.

8.7.4 Memoization for backreferences

8.7.4.1 Modeling regular expressions with backreferences (REWBR)

Typical regex engine implementations To understand the time complexity of evaluating a Regular Expression With BackReferences (REWBR [84, 108]), we consider their typical implementation in production regex engines. Regex engines implement REWBR in a straightforward manner: they track the contents of each capture group using a pair of indices $\langle j, k \rangle$ into w, and when they encounter a backreference at w offset i they compare the subsequent characters from $w[i:]$ to the current contents of the appropriate capture group.

Modeling These implementations can be modeled with f-edges. Each instance of a backreference introduces an additional automaton vertex q_f with an f-edge. The condition tested by this f-edge function is whether the following characters in w match the current contents of the indicated capture group. This function costs $\mathcal{O}(|w|)$ to evaluate, and consumes the corresponding number of characters if it is successful.

Example Figure 8.10 shows an example f-NFA for the REWBR `/(a)\1/`.

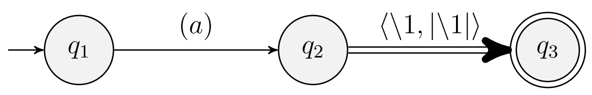


Figure 8.10: ***Automaton used to illustrate the f-NFA for backreferences.*** *This figure shows the f-NFA produced by Thompson's construction for the regular expression /(a)\1/, slightly reduced for clarity of presentation. The f-edge $q_1 \to q_2$ describes the backreference. This edge tests whether the subsequent characters match the current contents of the first capture group, (a), and consumes that many characters on success.*

8.7.4.2 Time complexity of evaluating REWBR with existing algorithms

Prior work has shown that evaluating a REWBR is NP-hard [63, 84, 108]. In terms of our f-NFA model, evaluating an REWBR necessitates the exploration of up to an exponential number of paths in an automaton.

Time complexity using backtracking A Spencer-style backtracking algorithm already explores up to an exponential number of paths, so in this regard the time complexity for REWBR is no worse than it was before. However, each simulation step may now cost $|w|+|Q|$ instead of $1+|Q|$ to accommodate the $\mathcal{O}(|w|)$ cost of testing the backreference. We therefore obtain a rough bound of $\mathcal{O}(|w| * |Q| * |Q|^{|w|})$.

Time complexity using Thompson's linear-time NFA simulation algorithm If a REWBR must be evaluated, it invalidates the observation that permits Thompson's algorithm to offer linear time complexity. Thompson's algorithm leverages the path-independence of recognition using a K-regex, and collapses all paths that reach a search state $\langle q, i \rangle$ into a single entity to track. The semantics of REWBR, however, state that two paths that reach the same search state *but populate a capture group differently* are no longer equal.[13] Were Thompson's algorithm extended with this new definition of path equality, the time complexity becomes the same as that of Spencer's backtracking approach, but the space complexity becomes exponential.

8.7.4.3 Optimizing REWBR using memoization

Summary We show that REWBR can be evaluated using memoization, offering some time complexity improvement for backtracking-based approaches. The time complexity remains exponential in the recognition query $\langle \text{Regex}, w \rangle$, but the exponent is reduced. In particular, we can reduce the time complexity from exponential in $|w|$ to exponential in $|Q|$ — from $\mathcal{O}(|Q|^{|w|})$ to $\mathcal{O}(|w|^{|Q|})$. Since we typically expect $|w|$ to dominate $|Q|$, this is a substantial reduction.

[13]See Figure 8.11, also used to discuss memoization for REWBR.

Approach Because the contents of a capture group depend on the path taken through the automaton, REWBR disrupt our path-independent memoization scheme just as they impact Thompson's algorithm (§8.7.4.2). The outcome of the f-edge function is no longer determined solely by the search states that have been reached. This problem is illustrated in Figure 8.11.

The difficulty illustrated by Figure 8.11 suggests two solutions: either erase the memoization table whenever a backreferenced f-edge is tested; or extend the memoization scheme to track not only the search states but also the contents of the capture groups at those search states. The Perl regex engine takes the former approach;[14] we will consider the latter. Extending the memoization scheme requires further modifying the M-NFA described in Table 8.1 in two ways.

Capture groups We must track a vector of capture group contents, CG, where CG_i denotes the contents of the i^{th} capture group. Each capture group can be represented with a pair (a, b) of indices into w. We discussed this extension in §2.4.4.

Memoized transition function The memoized transition function, δ_M, must now consider the contents of the capture groups. So δ_M is now $\delta_M : Q \times \Sigma \cup \{\varepsilon\} \times \mathbb{N}^{|w|} \times CG \to 2^Q$.

The memoized transition function now depends on the current search state $\langle q, i \rangle$ as well as the *path state* $\langle CG_1, CG_2, \ldots, CG_{|Q|} \rangle$ corresponding to the capture group contents. There are $|Q| * |w|$ distinct search states. Each capture group can take on $|w|^2$ different values (the number of contiguous substrings of w), yielding $(|w|^2)^{|Q|}$ possible distinct path states.

Now we would like to determine the time and space complexity of the M-NFA simulation used to answer regex match and parse queries. It is well known that if we use $|Q|$ as the upper bound of the number of backreferences and backreferenced capture groups, the time complexity of a match or parse operation is exponential [63, 84]. But $|Q|$ is too loose of a bound for typical regexes. Only the contents of the backreferenced capture groups (i.e., the path state) can affect the simulation result. If two simulations take different non-captured paths that reach the same search state, the simulation result will be the same for each path. We will therefore parameterize this complexity in terms of *the number of backreferences*, which we denote $|BR|$, and *the number of backreferenced capture groups*, which we denote $|BR_{\text{uniq}}|$. For example, the regex $R_1 = /(a)\backslash 1/$ has $|BR| = |BR_{\text{uniq}}| = 1$, while the regex $R_2 = /(a)(b)\backslash 1\backslash 1\backslash 2/$ has $|BR| = 3$ and $|BR_{\text{uniq}}| = 2$,

Regexes typically contain few distinct backreferenced capture groups and few backreference uses. For example, in our polyglot regex corpus, there are $\approx 3,500$ backreference-using regexes. Of these, 98% of these regexes contain at most three backreferences ($|BR| \leq 3$), and 98% of these regexes contain backreferences to at most two distinct capture groups ($|BR_{\text{uniq}}| \leq 2$).

[14] As discussed in §8.2, this leads to unprotected exponential behavior (case 3).

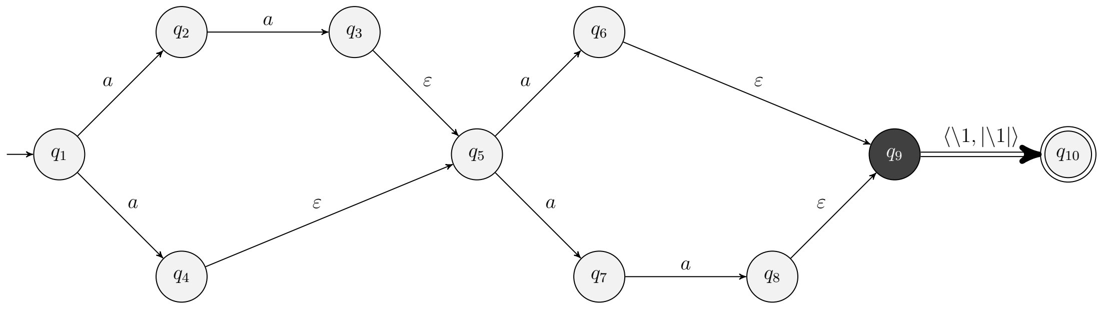


Figure 8.11: ***Automaton used to illustrate the difficulty of memoization with backreferences.*** *This figure shows the NFA produced by Thompson's construction for the regular expression* /(aa|a)(a|aa)\1/, *slightly reduced for clarity of presentation. The paths between q_1 and q_5 determine the contents of capture group 1. The paths between q_5 and q_9 determine the contents of capture group 2. The f-edge $q_9 \to q_{10}$ describes the backreference, which matches the contents of capture group 1 and which consumes the same number of characters.*

Suppose we apply the full memoization scheme to the candidate string w = aaaa *(and in particular that we are memoizing visits to the shaded node, q_9). This candidate string can be accepted in one way: "go down, then down", i.e., with \1 = a and \2 = aa. This path will reach the simulation state $\langle q_9, 3\rangle$. But it will only reach this state* after *that state is reached by the path "up, then up" — the path with \1 = aa and \2 = a, which mismatched because the $q_9 \to q_{10}$ edge could not match \1. The up-up path has higher precedence than the down-down path according to PCRE's left-to-right disjunction rules (§2.4). Our existing memoization scheme will incorrectly cause the down-down path to be short-circuited, leading to an incorrect rejection of the candidate string.*

In other words, because these two paths differ, they populate \1 differently. The outcome from the up-up path does not dictate the outcome for the down-down path. But the memoization schemes discussed in sections 8.4 and 8.5 incorrectly assume it does. The difficulty is again ambiguity in the automaton.

By applying the extensions discussed in §8.7.4 in place, the difficulty can be resolved. The up-up path and the down-down path still both reach the search state $\langle q_9, 3\rangle$, but the memoization function δ_M now distinguishes between these search states based on their values for \1 and \2. One does not short-circuit the other.

Space complexity We will suppose a full memoization scheme for simplicity. There are $|Q| * |w|$ search states from the previous model. There are $|BR_{uniq}|$ capture groups that can affect the evaluation of an f-edge, each of which can take on $\mathcal{O}(|w|^2)$ distinct values which we must also track. For each search state, therefore, there are $\mathcal{O}((|w|^2)^{|BR_{\text{uniq}}|})$ distinct path states. The memoization table may thus contain at most

$$\mathcal{O}(|Q| * |w| * |w|^{2*|BR_{\text{uniq}}|}) = \mathcal{O}(|Q| * |w|^{1+2*|BR_{\text{uniq}}|})$$

distinct entries. For typical regexes, $|BR_{\text{uniq}}| \leq 2$, and the space cost becomes a large polynomial of $|w|$. But since this term appears as an exponent, any use of backreferences significantly increases the size of the memoization table.

This space complexity can be decreased using selective memoization or efficient encodings.

Time complexity With this memoization scheme, each combination of simulation state and path state (capture group contents) will be visited at most once. Unlike the regular NFA vertices, which cost $\mathcal{O}(1) + \mathcal{O}(|Q|)$ to test and traverse, the f-edges associated with the $|BR|$ backreference vertices cost at most $\mathcal{O}(|w|) + O(|Q|)$ to test and traverse. This cost can be expressed as the sum of the (cheaper) cost of visits to normal vertices, Φ_{normal}, and to the (more expensive but typically fewer) backreference f-vertices, Φ_{BR}. This sum is:

$$\mathcal{O}(\ (|w|^{2*|BR_{\text{uniq}}|}) * |w| \ * \ (|\Phi_{\text{normal}}| * (1 + |Q|) \ + \ |\Phi_{\text{BR}}| * (|w| + |Q|))\ ).$$

If we suppose that all vertices are f-vertices, and that $|w| + |Q| \approx |w|$, then we can obtain the simplified form $\mathcal{O}(|Q|^2 * |w|^{2+2*|BR_{\text{uniq}}|})$. Since it is typical for $|BR_{\text{uniq}}| \leq 2$, for practical purposes this bound can be treated as a large polynomial in $|w|$.

Remarks Both the space and time complexities are exponential in the number of distinct backreferenced groups. This complexity emerges from the number of distinct capture groups that it is necessary to track during the automaton simulation. The memoization still eliminates redundant visits to all search states, but must be more cautious in order to safely eliminate these visits (by also tracking path states).

An additional parameterization can further characterize the costs of typical backreference-using regexes. If a backreferenced capture group has unbounded width (i.e., they use a * or +), then it adds a factor of $|w|^2$ to the time and space complexities. Many backreferenced capture groups, however, are fixed-width. 78% of these regexes reference only fixed-width capture groups, most commonly to find a matching single or double quote to a string. For example, patterns like `/('|")\w+\1` are common. Regexes that use only fixed-width capture groups can remove the factor of 2 in the exponent, reducing the size of the polynomial in typical cases.

Our measurements of typical parameters suggests that most regex usage contexts do not need a large number of backreferences. Regex engine maintainers should consider documenting the worst-case performance parameterized by the use of backreferences. They may also consider limiting the number of distinct backreferences ($|BR_{\text{uniq}}|$) permitted in a regex, reasoning that the additional expressive power is not worth the additional cost. If this number is limited, then the worst-case space and time complexity for all backreferences becomes polynomial in $|Q|$ and $|w|$.

8.8 Discussion

Improving the performance of the RLE encoding scheme In §8.6, we reported that an RLE encoding scheme with runs of length 1 was effective in most cases. For 95% of the supported subset of super-linear regexes, there were at most 12 runs over all vertices at any point in each simulation. However, as can be seen in Figure 8.7, our corpus contains some super-linear regular expressions whose simulation requires $\Theta(|w|)$ runs. An expression-specific choice of run lengths might similarly reduce these costs.

A simplified example of such a super-linear regular expression is `/^(aa|aa)*$/`, shown in Figure 8.12. This expression has infinite ambiguity resulting from the quantified ambiguous disjunction. When this automaton is simulated on input that triggers its super-linear behavior, the vertex q_1 will be visited at every second candidate string index, i.e., the search states $\langle q_1, 0\rangle$, $\langle q_1, 2\rangle$, $\langle q_1, 4\rangle$, $\ldots$, $\langle q_1, |w|\rangle$. The visit vector for this vertex will thus be $101010\ldots1010$. On such a visit vector, a length-1 RLE encoding will obtain no compression, while a length-2 RLE encoding can achieve perfect compression.

An effective length for the RLE encoding scheme could be determined dynamically or statically. Dynamically, the length might be guessed from the initial sequence of updates. A variable-length encoding could also be employed, although this might impose larger runtime costs. Statically, we believe the general principle for the choice of length is to use the least common multiple of the lengths of the simple path(s) to the vertex, e.g., $\text{LCM}(2, 2) = 2$ for the example in Figure 8.12. Although we have not yet proved this property, the intuition is that any repeated visits to the vertex can only occur at multiples of the simple paths. The lengths of the simple paths can be calculated during the expression-to-NFA conversion.

Developing worst-case inputs for E-regexes Our analyses of the worst-case behavior of zero-width assertions and backreferences may guide the development of worst-case inputs for regexes that use these features. For zero-width assertions, worst-case inputs will repeatedly exercise expensive lookaround assertions, e.g., by targeting complex lookaround assertions that can be reached at many distinct indices of w. For backreferences, worst-case inputs should search for ambiguous sub-structures that can populate capture group(s) with different values. Both of these approaches can be built on top of existing automaton-based

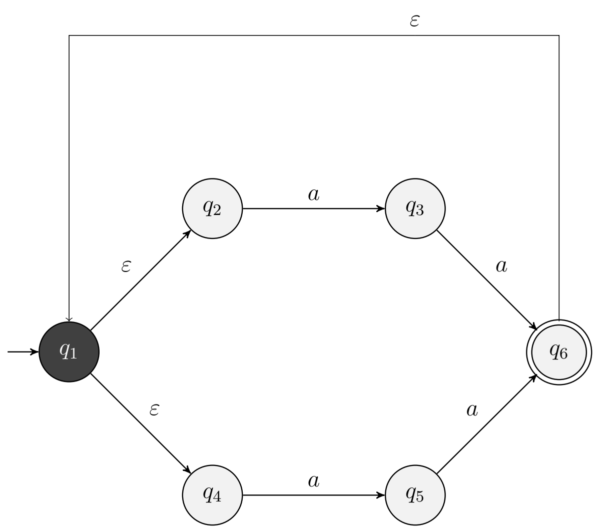


Figure 8.12: ***Automaton used to illustrate the effect of tuning the RLE run length.*** *The vertex q_1 will be memoized by any of the selection schemes we have proposed. However, on problematic input — e.g., $w = a^k \cdot b$ — this state will never be visited at consecutive offsets. A length-1 RLE scheme offers no compression, while a length-2 RLE scheme offers perfect compression.*

analyses (§2.5.2) that have been extended to f-NFAs.

The peril of local optimality There is an engineering lesson to be learned from the shortcomings of the Perl regex engine's memoization scheme. Since its introduction, the Perl regex engine's memoization scheme has only protected "complex" quantified sub-patterns, ignoring "simple" ones. These "simple" ones instead follow an optimized path that benefits behavior in the average case. For example, simpler sub-patterns can be tested using primitive comparison operations like `memcmp` rather than the more abstract machinery needed to handle a complex sub-pattern. These optimizations permit the Perl regex engine to perform quickly on matching or blatantly-mismatching inputs, but due to its insufficient memoization it struggles on ambiguity-exploiting input for such sub-patterns. As in many other contexts, the engineering lesson here is this: *locally optimal behavior may lead to global performance degradation.*

8.9 Threats to validity

Internal validity We believe our prototype regex engine is implemented correctly. As a sanity check, our prototype confirms that each search state is visited at most the number of times predicted by our theorems. Although our results are consistent with our theorems, implementation errors could bias our findings.

External validity The measurements discussed in §8.6.2 and §8.6.3 are on a subset of the full regex corpus. It is unclear whether our findings generalize to regexes that use additional features that are not supported by our prototype. Comparing regexes within the sets that use different features was not among the generalization experiments performed in Chapter 5.

Construct validity We compare different size measurements in Figure 8.7, but believe we measured appropriate values for each type of encoding.

Chapter 9

Techniques to cap per-client resource utilization

9.1 Summary

The final ReDoS amelioration approach considered in this dissertation is that of capping the amount of resources that a client may use. This approach resolves ReDoS Condition 4; if a malicious client cannot over-consume the web server's resources, then the client cannot conduct a ReDoS attack.

There are two classes of ReDoS defenses of this form. They can be described as *algorithm-oriented* defenses and *time-oriented* defenses. Both of these defenses measure the cost of a regex match, and if that cost exceeds a threshold then the evaluation is short-circuited with an exception. These defenses vary in how this cost measure is defined. An *algorithm-oriented* defense measures resource utilization in terms of the long-running algorithm, in this case the regex engine implementation. A *time-oriented* defense measures resource utilization in terms of wall clock time.

Methodology In this chapter we consider three aspects of the ReDoS amelioration of resource caps. *First*, we assess the effectiveness of the existing algorithm-oriented and time-oriented ReDoS defenses used in mainstream regex engines. We test the proportion of a corpus of super-linear regexes whose runaway behavior is detected and short-circuited by these defenses. *Second*, we measure the extent to these defenses have been adopted by software engineers since they were added to the corresponding regex engines. The most user-friendly of these defenses is C#'s time-oriented defense, released in 2012 and off by default, and we mine software to determine the extent to which software engineers have adopted it. *Third*, based on the results of our first two studies, we consider what a time-oriented resource cap defense might look like if it were designed into a web framework from scratch.

Findings We report that the two extant algorithm-oriented defenses are ineffective, protecting only 11% of super-linear regex evaluations in Perl and at most 73% in PHP. The extant time-oriented defense, in C#, achieved perfect protection in our experiment. Unfortunately, we also found that this defense is rarely adopted by software engineers. One conclusion that can be drawn from these findings is that time-oriented defenses are effective but that software engineers will not adopt them retroactively. We therefore conducted a case study of incorporating a time-oriented resource cap defense into a web framework from scratch, and found that we could protect Node.js web applications against ReDoS and a family of related security vulnerabilities with overheads between 0% and 24% depending on the workload. We describe the design and implementation considerations involved.

Statement of Attribution The material presented in §9.6 is excerpted from papers published at EuroSec 2017 [138] and USENIX Security 2018 [140].

9.2 Related work — resource caps in mainstream regex engines

Many Spencer-style backtracking regex engines place responsibility for super-linear evaluations on application developers. Application developers are responsible either for ensuring that their regexes exhibit worst-case acceptable performance in their context, e.g., by composing unambiguous regexes or by sanitizing the input on which their regexes are evaluated. This approach is discussed in Chapter 6.

Some of these regex engines, however, apply resource caps to control the amount of computational resources that a given regex match can consume. Their schemes fall into *algorithm-oriented* (§9.2.1) and *time-oriented* (§9.2.2).

9.2.1 Algorithm-oriented solutions

The two regex engines that employ an algorithm-oriented solution are those of PHP and Perl. This solution is part of the reason that they fall into the “Medium” regex engine performance class. On many regex evaluations that entail super-linear match complexity in the “Slow” Spencer-style engines, the regex engines of PHP and Perl detect that the evaluation is consuming too many resources and short-circuit it with an exception.

Resource usage measures and configurability The PHP and Perl regex engines use different measures of the resources consumed by their respective algorithms, and vary in their configurability.

PHP: The PHP regex engine measures resource usage in terms of the number of PHP “backtracking frames”, which are equivalent to the BacktrackingPoints from Listing 3. The PHP regex engine tracks the number of backtracking frames created from a given search state, and throws an exception if this number exceeds the `recursion_limit`. The PHP regex engine also tracks the cumulative number of backtracking frames created over all search states, and throws an exception if this number exceeds the `backtracking_limit`. These parameters can be tuned from their defaults of 100,000 (`recursion_limit`) and 1,000,000 (`backtracking_limit`).[1]

Perl: Like the PHP regex engine, the Perl regex engine measures resource usage in terms of the degree of backtracking. The Perl regex engine's measure is analogous to PHP's per-search-state `recursion_limit`, but is only applied on search states corresponding to quantified sub-patterns that contain more than one character. Unlike PHP's engine, Perl's engine does not maintain a global cost counter. The Perl engine's threshold for exceptional resource usage is called `REG_INFTY`, defaults to 32,767 (i.e., $2^{15}-1$), and cannot be configured by an application.

9.2.2 Time-oriented solutions

Alone among the regex engines built into mainstream programming languages, the .NET regex engine used by C# offers a time-oriented solution instead. Each of the default .NET regex match APIs has a sibling API through which an engineer can set a time limit for the regex match [240].

Resource usage measure In a time-oriented solution, the resource usage measure is wall clock time. The .NET APIs accept an approximate time limit in milliseconds, and they throw a `Regex.MatchTimeout` exception if the time limit is exceeded. These APIs are somewhat imprecise, because the timer is tested at the top of the backtracking loop and the time limit may expire at any point during the loop. But because each loop does a small amount of work as a function of the automaton and the candidate string, they will deliver a `Regex.MatchTimeout` if the match time exceeds $TimeLimit + \varepsilon_{f(Q,\delta,w)}$ for a small ε.

Configurability To employ this solution, a software engineer must identify an appropriate time limit, and apply it across all regex matches on their application's critical path.

9.2.3 Solution analysis

Backwards (in-)compatibility Addressing ReDoS by capping resource utilization will change the API of the regex engine. Instead of always answering a recognition or parse query, the regex engine may instead *reject* the query, e.g., by means of an exception. Changing an API in this manner, whether through an algorithm-oriented approach or a time-oriented

[1]See `http://php.net/manual/en/pcre.configuration.php`.

approach, has two shortcomings: (1) the API may not actually answer the match query posed by the software engineer, and (2) the defense cannot be applied to existing applications without application-level refactoring.

First, introducing an exception into the regex engine's API may be acceptable in some contexts, but in others it is not. When a regex is being used to filter invalid input, it is reasonable to interpret an input that takes a super-linear amount of time as invalid, at least from the perspective of ensuring that the majority of clients receive acceptable responses. This was the case in the MediaWiki and Stack Overflow case studies (Chapter 3), as well as the ReDoS attacks conducted by Staicu and Pradel [307] — typical input was processed with acceptable latency, and any super-linear evaluations were the result of malicious or highly abnormal input. But there are ReDoS contexts where treating such inputs as invalid would be unacceptable. For example in the Cloudflare and Atom case studies, a broad range of *legitimate* input required super-linear evaluation times, and rejecting all such regex match queries would render the affected service unusable for many legitimate clients. An exception from the regex engine is useful under exceptional circumstances, but cannot be a typical case.

The second shortcoming is that retrofitting a resource-cap solution into a regex engine cannot be done in a backwards-compatible manner. In some cases, introducing resource caps may convert a minor performance problem into a major stability problem (i.e., a crash). The relative merits of one or the other can be debated, and in some circumstances an explicit crash may be preferable [196]. But the software engineering community has generally agreed that backwards compatibility is valuable. As a result, this approach can only be retrofitted into an existing regex engine in an optional or off-by-default manner. Applications can only benefit if their maintainers adopt the safe version of the APIs.

Configuration To apply either class of resource caps, a software engineer must determine an appropriate threshold for resource usage. In an *algorithm-oriented* approach, this threshold may be difficult for an application developer to identify, because the usage metric is not defined in terms that an application developer is familiar with. A regex engine developer could propose a default value, but this would be complicated by variation in the wall-clock cost of the algorithm based on the underlying hardware. In a *time-oriented* approach, it may be easier for an application developer to determine an appropriate threshold based on business requirements, e.g., their SLOs.

9.3 Study design and research questions

With the preceding discussion in mind, two questions can be asked of existing resource-cap approaches: are they effective, and will engineers adopt a retrofitted solution?

Theme 1: Analyzing existing resource-cap approaches

RQ1: How effective are existing resource-cap solutions?
RQ2: How commonly do software engineers adopt a retrofitted resource-cap solution once it becomes available?

The answers we find to these questions are discouraging — existing algorithm-oriented resource caps do not completely protect applications from ReDoS, and even if resource caps are retrofitted, software engineers do not adopt them. It appears that retrofitting resource caps is an ineffective solution to ReDoS. These findings raise a third question: how might a framework be designed from scratch to incorporate resource caps that eliminate ReDoS?

Theme 2: Implementing a resource-cap solution
RQ3: How might a web framework be designed from scratch to incorporate resource caps that eliminate ReDoS?

9.4 RQ1: How effective are existing resource-cap solutions?

In this section we determine how effective existing resource-cap solutions are. These solutions are algorithm-oriented (Perl, PHP) and time-oriented (.NET/C#). The general approach of our experiment is to evaluate super-linear regexes under long attack strings, and then determine whether or not the defense defeats the attack.

9.4.1 Methodology

Super-linear regexes For this experiment we ran a series of regex match queries on the regex engines of PHP, Perl, and C#. We used the full set of 51,224 super-linear regexes extracted from all programming languages in Chapter 5. We filtered these regexes to the 50,435 regexes on which the C# regex engine exhibited super-linear behavior, as the C# regex engine uses a less optimized backtracking implementation than Perl and PHP do.

Super-linear query structure Following the methodology used in prior chapters, we issued queries to each regex engine query uses a super-linear regex. We used an attack input pumped 200,000 times with a timeout of 10 seconds to accommodate the longer launch time of our C# regex test tool — our experimental setup is Linux-based, so we launched the tool under `wine` [206] which is slow to start. For regex engines in the "Slow" family, inputs of this size take approximately 30 seconds to evaluate on the hardware used in our experimental setup. We queried the C# regex engine with a timeout of 10 milliseconds, and used the default parameterization of the resource cap defenses of Perl and PHP.

Table 9.1: ***The effectiveness of algorithm-oriented resource caps.*** *This table shows our measurements of the efffectiveness of the resource caps available in Perl, PHP, and C#. We tested using the corpus of 51,224 super-linear regexes extracted in Chapter 5. *: The vulnerable behavior in C# is anomalous; see text.*

	# Unsupp.	**# Linear time**	**# Threw (defended)**	**# Timed out (vuln.)**
Perl	3	36,600	332	13,182
PHP	18	13,521	17,595	18,983
.NET (C#)	1	1	49,898	217 *

Measure of successful defense Each match query has four possible outcomes: (1) The regex was not supported by the regex engine's dialect; (2) The input matched or mismatched (e.g., avoiding super-linear behavior through regex engine optimizations); (3) Exceptional (defense mechanism worked); or (4) Timed out (defense mechanism failed).

Outcomes (1) and (2) are uninteresting for this experiment. Outcome (3) indicates that the regex engine's defense mechanism was successful. Outcome (4) indicates that the regex engine's defense mechanism failed.

9.4.2 Results and Analysis

Of the 51,224 regexes in the corpus, 50,435 exhibited super-linear behavior in C# in our experimental conditions. We were able to collect full measurements on 50,117 regexes, with a few measurements missing due to miscellaneous errors.

The results of this experiment are summarized in Table 9.1. Analyzing these results yields three findings:

1. *Perl has stronger optimizations than PHP:* Among the super-linear regexes they support, Perl is able to produce a linear-time response to the query far more frequently than PHP.
2. *PHP has stronger resource caps than Perl:* For the regexes that they cannot handle in linear time, PHP's resource caps are much more successful than Perl's. PHP delivers an exception for 48% of these regexes, while Perl delivers an exception only 2% of the time. PHP's success is unsurprising in light of our analysis in §9.2.1, as its regex engine uses a measure of cost that is cumulative across the entire regex match.
3. *Time-oriented resource caps are consistently stronger:* Although PHP's algorithm-oriented caps were superior to Perl's, it is clear that the .NET framework's time-oriented resource cap was consistently more effective than both. The .NET regex engine honored the 10-millisecond timeout we applied to **all** of the super-linear regexes we tested in C#-Mono. Although 217 of the super-linear regexes timed out in our experiments, this appears to have been an artifact of our measurement instruments. Inspection of the logs showed that these timeouts occurred on a single node and that the timeouts were associated with

launching `wine`, not the regex match.

Tuning We performed this experiment using the default parameters for the algorithm-oriented defenses. It is possible that by tuning these parameters we could achieve a level of defensive effectiveness comparable to that of the .NET regex engine.

9.5 RQ2: How commonly do software engineers adopt a retrofitted resource-cap solution once it becomes available?

The resource-cap solutions used in the PHP, Perl, and .NET regex engines were all added after their respective initial releases. In this study, we evaluate the extent to which software engineers have adopted these defenses after they were retrofitted. In our opinion, the documentation and configuration of the .NET's time-oriented approach lends itself far more to adoption than the algorithm-oriented parameters of PHP, and Perl does not permit users to tune its parameter. Therefore, in this experiment we considered the adoption of .NET's timeouts, which were added in 2012 [44].

9.5.1 Methodology

Specifying a timeout in .NET Within the .NET regex engine used by C#, the time limit associated with a regex match can be specified in three ways. First, it can be set globally for all matches via the `AppDomain` `REGEX_DEFAULT_MATCH_TIMEOUT` property. Second, it can be specified for all uses of a `Regex` via an optional `TimeSpan` parameter in the constructor. Lastly, it can be set in individual static regex method calls by supplying a `TimeSpan` parameter along with the regex pattern and candidate string.

Software to analyze For consistency with our established software selection methodology, we studied the use of regexes within software modules written in the .NET framework. We studied all clone-able projects in the C# package ecosystem, NuGet. At the time of our analysis NuGet had 135,125 modules, although many were derived from the same project repository in the monorepo style. We were able to clone 35,194 distinct project repositories for study.

Extracting regexes and usage For each project, we used static analysis to identify the use of the global Regex timeout as well as the various uses of regexes: `new Regex` and `Regex` method calls. We could not identify a convenient Linux-based AST generator for

C# so we wrote a custom parser focused on regexes. To identify regexes with super-linear behavior, we used our standard ReDoS detector ensemble and dynamically confirmed super-linear behavior using a C#-Mono program. We did not employ the regex variants that we introduced in our later research.

Specific questions Our analysis focused on two questions: (1) Do developers make frequent use of timeouts, and (2) Do developers tend to use timeouts on super-linear regexes? As the application of timeouts can be on a per-use basis, when a regex is used multiple times we consider each use.

9.5.2 Results and Analysis

Summary of regex extraction Here is a general snapshot of our findings for comparison with our earlier regex extractions (cf. Table 5.5). Among the 35,194 projects, 2,812 (8%) used regexes. In these modules we identified a total of 12,213 unique regexes. Of these, 826 regexes exhibited super-linear behavior in the .NET regex engine, 97 exponentially so.

Timeouts are rarely used We found that these projects rarely used timeouts. As shown in Table 9.2, 95% of calls to the Regex constructor do not request a timeout, meaning that subsequent uses of Regex methods for these patterns will not be protected by a timeout. Similarly, 98% of static calls to Regex methods — i.e., the calls that specify both regex pattern and candidate string — do not use a timeout. From another perspective, in the third row we can see that only 5% of the modules ever made use of a timeout feature.

The last row of Table 9.2 includes both local (per-regex or per-use) timeouts and global timeouts. Only 7 projects used the global `REGEX_DEFAULT_MATCH_TIMEOUT` property. These projects were not heavy users of regexes, so the global timeout protected a total of 22 regex occurrences (`new Regex` or static method calls).

Timeouts are not used on super-linear regexes We found that the use of timeouts appears to be unrelated to the worst-case behavior of the regex evaluations being capped. In Table 9.3 we compare the worst-case behavior of regexes compared to the use of timeouts,. This table includes only static calls to Regex methods, which encode the regex directly and are thus only protected by a global timeout or one provided in the same call. We can see that developers do not seem to use timeouts when it might be advisable to do so. Only 1.5% of the uses of timeouts protected vulnerable regexes (top row), and 0.5% of calls using vulnerable regexes were made with a timeout (right column).

Based on these findings, we conclude that software engineers do not typically retrofit regex resource limitations into their software.

Table 9.2: ***Adoption of regex match timeouts in C# projects.*** *The first two columns list the number of occurrences with and without timeouts. In the final column are the number of regex occurrences that our custom parser could not handle. The first two rows cover all uses of the Regex APIs, including occurrences using dynamically-defined patterns (which are omitted in Table 9.3). The third row indicates the number of C# projects that ever used a timeout, either globally or in any regex declaration or static method call.*

	Used timeout	**No timeout**	**Unparseable**
Per-call: Constructor	959	22,306	210
Per-call: Static methods	599	27,151	0
Per-project: Any usage	130	2,682	N/A

Table 9.3: ***Adoption of C# timeouts in super-linear regex method calls.*** *Use of timeouts in static regex method calls containing a parseable, statically-defined regex pattern. Regexes were deemed super-linear or linear-time using dynamic validation in a standalone C#-Mono program. Among the regexes with resource-limited evaluations, most would evaluate in linear-time anyway.*

	# Linear-time	**# Super-linear**
Timeout	203	3
No timeout	16,640	513

9.6 RQ3: How might a web framework be designed from scratch to incorporate resource caps?

In RQ1, we found that time-oriented resource caps were more effective and easier to tune than algorithm-oriented ones. In RQ2, however, we found that time-oriented resource caps are rarely adopted after they are retrofitted into a regex engine. These results suggest that the time-oriented approach is superior, but that retrofitting such an approach is unlikely to see widespread adoption.

In this section, we examine the from-scratch design and implementation of a time-oriented defense for ReDoS. We are exploring the solution space for a time-oriented defense, and thus can generalize from ReDoS to a larger family of denial of service vulnerabilities. We embody our work in the server-side event-driven architecture (EDA) used by the popular Node.js framework.

Web frameworks that use the EDA, as Node.js does, avoid threading overheads by having a small number of threads handle requests from many clients. This multiplexing is achieved using cooperative multi-tasking, partitioning client requests into events that are handled in atomic steps. Applications then switch between pending clients at the boundaries between event handlers. If the application developer does not ensure that each of these steps completes quickly, then they run the risk of permitting a single client to dominate one of a limited set of resources, i.e., "poisoning" one of these event-handling threads. When an attacker can exhaust a limited set of resources, a denial of service vulnerability exists. ReDoS is a notable example of such a vulnerability in this architecture: super-linear regexes are common, regex evaluations typically run in super-linear time with no resource caps, and a super-linear regex evaluation can cause an event handler to run far longer than normal [266].

This section proceeds as follows. We first introduce Node.js and its vulnerable software architecture (§9.6.1). Next we place ReDoS within a broader family of security vulnerabilities affecting EDA-based applications (§9.6.2). Then we describe a defense strategy, *First-Class Timeouts*, which applies a time-oriented defense to cap each client's resource utilization (§9.6.3). Finally we discuss first-class timeouts in terms of our implementation (§9.6.4) and evaluation (§9.6.5).

Statement of Attribution The material presented in this section is excerpted from a paper published at USENIX Security 2018 [140]. An early version of this work appeared at the EuroSec workshop in 2017 [138].

9.6.1 Solution context: The server-side EDA

Web service architectures Many web services are built using two popular software architectures: the One Thread Per Client Architecture (OTPCA) and the Event-Driven Architecture (EDA). The OTPCA and the EDA architectures differ in the resources they dedicate to each client. As Pariag et al. summarized [275], the OTPCA dedicates resources to each client, for strong isolation but higher memory and context-switching overheads. The EDA tries the opposite approach and reverses these tradeoffs. In the EDA, many client connections are multiplexed onto a small number of threads, leading to weak isolation but lower threading overheads.

Usage of the EDA in industry The EDA has only recently become a mainstream architecture for web services. Although EDA approaches have been discussed and implemented in the academic and professional communities for decades (e.g. [74, 150, 175, 289, 318, 337]), historically the EDA was only applied in user interface settings. The EDA has become widely used in web services thanks to the adoption of Node.js, an EDA-based server-side JavaScript framework [16, 17, 48, 51, 54, 185, 265, 272].[2]

The EDA paradigm and vocabulary To the best of our knowledge, all server-side EDA frameworks use the Asymmetric Multi-Process Event-Driven (AMPED) architecture [273].[3] This architecture (hereafter "the EDA") is illustrated in Figure 9.1. In the EDA the OS, or a framework, places events in a queue, and the *callbacks* of pending events are executed sequentially by the *Event Loop*. The Event Loop may offload expensive *tasks* such as file I/O to the queue of a small *Worker Pool*, whose workers execute tasks and generate "task done" events for the Event Loop when they finish [154]. We refer to the Event Loop and the Workers as *Event Handlers*.

Because the Event Handlers are shared by all clients, the EDA requires a particular development paradigm. Each callback and task is guaranteed atomicity: once scheduled, it runs to completion on its Event Handler. Because of the atomicity guarantee, if an Event Handler blocks, the time it spends being blocked is wasted rather than being preempted. Without preemptive multitasking, developers must implement cooperative multitasking to avoid starvation [298]. They do this by partitioning the handling of each client request into multiple stages, typically at I/O boundaries. For example, with reference to Figure 9.1, a callback might perform some string operations in CB_{A1}, then offload a file I/O to the Worker Pool in $Task_{A1}$ so that another client's request can be handled on the Event Loop. The result of

[2]The popularity of Node.js may be driven less by performance considerations and more by business needs. One of the premises of Node.js is "full stack JavaScript." In this model, client- and server-side engineers can use the same programming language and share the same libraries, potentially reducing overall engineering costs.

[3]For example, this is the architecture used by Node.js, libuv (C/C++), EventMachine (Ruby), Vert.x (Java), Zotonic (Erlang), and Twisted (Python).

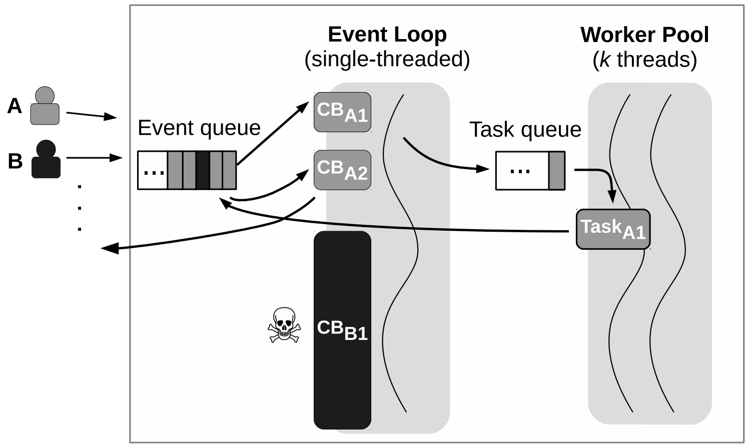


Figure 9.1: ***Illustration of the Event-Driven Architecture.*** *This figure illustrates the EDA as it is commonly implemented in server-side frameworks. In particular, the figure illustrates the AMPED EDA. Incoming events from clients A and B are stored in the event queue, and the associated callbacks (CBs) will be executed sequentially by the Event Loop. Client B has conducted a successful denial-of-service attack by poisoning one of the server's Event Handlers.*

this partitioning is a per-request lifeline [68], a DAG describing the partitioned steps needed to complete an operation. A lifeline can be seen by following the arrows in Figure 9.1.

9.6.2 Attack: Event Handler Poisoning

In this section we define Event Handler Poisoning (EHP) attacks and estimate their incidence in practice.

9.6.2.1 Attack definition

Summary The EDA offers scalability, but its use carries risks: multiplexing destroys isolation. The EDA moves the burden of time sharing out of the OS and into the application itself. Without the preemptive multitasking assumed by OTPCA-based services, EDA-based services must enforce their own *cooperative multitasking* [298]. If cooperative multitasking is not enforced, one request can unfairly dominate the time spent by an Event Handler, preventing the server from handling other clients and leading to denial of service. In the style of an algorithmic complexity attack [136], we assume that the service's average-case request latency and throughput is acceptable.

Threat model We assume a relatively strong attacker. The victim is an EDA-based server with an Event Handler Poisoning (EHP) vulnerability. The attacker knows how to exploit this vulnerability: they know the victim provides user input to a *vulnerable API*, and they know *evil input* that will cause the vulnerable API to block the Event Handler executing it. This may require knowledge of the web service's implementation, but Staicu and Pradel have demonstrated that educated guesses may suffice [307].

Vulnerable APIs Recall the EDA illustrated in Figure 9.1. As discussed in §9.6.1, a client request is handled by a lifeline, a sequence of operations partitioned into one or more callbacks and tasks. A lifeline is a DAG whose vertices are callbacks or tasks and whose edges are events or task submissions.

We define the *total complexity* of a lifeline as the cumulative complexity of all of its vertices as a function of their cumulative input. The *synchronous complexity* of a lifeline is the greatest individual complexity among its vertices. Two EDA-based services may have lifelines with the same total complexity if they offer the same functionality, but these lifelines may have different synchronous complexity due to different choices of partitions. While computational complexity is an appropriate measure for compute-bound vertices, time may be a more appropriate measure for vertices that perform I/O. Consequently, we define a lifeline's *total time* and *synchronous time* analogously.

If there is a difference between a lifeline's average and worst-case synchronous complexity (time), then we call this a *vulnerable lifeline.*[4] We attribute the root cause of the difference between average and worst-case performance to a *vulnerable API* invoked in the problematic vertex.

The notion of a "vulnerable API" is a convenient abstraction. The trouble may of course not be an API at all but the use of an unsafe language feature (e.g. ReDoS). If an API is asynchronous, it may itself be partitioned and have its own sub-Lifeline. In this case we are concerned about the costs of its vertices.

Event Handler Poisoning attacks An EHP attack exploits an EDA-based service with an incorrect implementation of cooperative multitasking. The attacker identifies a *vulnerable lifeline* (server API) and *poisons* the Event Handler that executes the corresponding large-complexity callback or task with *evil input.* This evil input causes the Event Handler executing it to block, starving pending requests.

An EHP attack can be carried out against either the Event Loop or the Workers in the Worker Pool. A poisoned Event Loop brings the server to a halt, while the throughput of the Worker Pool will degrade for each simultaneously poisoned Worker. Thus, an attacker's aim is to poison either the Event Loop or enough of the Worker Pool to harm the throughput of the server. Based on typical Worker Pool sizes, we assume the Worker Pool is small enough that poisoning it will not attract the attention of network-level defenses.

[4]Differences in complexity are well defined. For differences in I/O time we are referring to performance outliers.

Listing 11 File server with two Event Handler Poisoning vulnerabilities: ReDoS (Line 2) and ReadDoS (Line 3).

```
def serveFile(name):
  if name.match(/(\/.+)+$/): # ReDoS
    data = await readFile(name) # ReadDoS
    client.write(data)
```

Since the EDA relies on cooperative multitasking, a lifeline's synchronous complexity (time) provide theoretical and practical bounds on how vulnerable it is. Note that a lifeline with large total complexity (time) is not vulnerable so long as each vertex (callback/task) has small synchronous complexity (time). It is for this reason that not all AC attacks can be used for EHP attacks. If an AC attack triggers large total complexity (time) but not large synchronous complexity (time) then it is not an EHP attack. For example, an AC attack could result in a lifeline with $\mathcal{O}(n^2)$ callbacks each costing $\mathcal{O}(1)$. Although many concurrent AC attacks of this form would degrade the service's throughput, this would comprise a DDoS attack, which is outside our threat model.

Not all DoS attacks are EHP attacks. An EHP attack must cause an Event Handler to block. This blocking could be due to computation or I/O, provided it takes the Event Handler a long time to handle. Other ways to trigger DoS, such as crashing the server through unhandled exceptions or memory exhaustion, are not time oriented and are thus out of scope. Distributed denial of service (DDoS) attacks are also out of scope; they consume a server's resources with myriad light clients providing normal input, rather than one heavy client providing malicious input.

We conjecture that EHP attacks have not previously been explored because the EDA has only recently seen popular adoption by the server-side community. On the client side EHP attacks are not a concern, as misbehaving users will hurt only themselves. On the server side, the lack of isolation permits one malicious client to affect others.

Two EHP attacks To illustrate EHP attacks, we developed a minimal vulnerable file server with EHP vulnerabilities common in real *npm* modules as described in §9.6.2.2. Listing 11 shows pseudocode, with the EHP vulnerabilities indicated.

The regular expression on Line 2 is vulnerable to ReDoS. A string composed of /'s followed by a newline takes exponential time to evaluate in Node.js's regular expression engine, poisoning the Event Loop in a CPU-bound EHP attack.

The second EHP vulnerability is on Line 3. Our server has a directory traversal vulnerability, permitting clients to read arbitrary files. In the EDA, directory traversal vulnerabilities can be parlayed into I/O-bound EHP attacks, "ReadDoS", provided the attacker can identify a

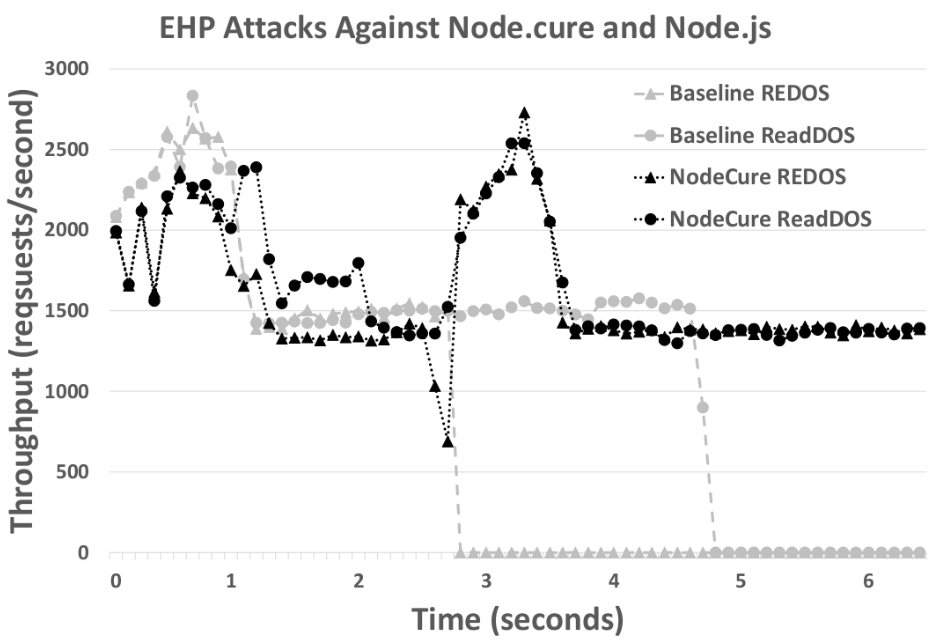


Figure 9.2: ***Performance impact of an Event Handler Poisoning attack.*** *This figure shows the effect of evil input on the throughput of a server based on Listing 11, with realistic vulnerabilities. Legitimate requests came from 80 clients using the Apache benchmarking tool* `ab` *from another machine. The attacks are against either baseline Node.js (grey) or our prototype, Node.cure (black). For ReDoS (triangles), evil input was injected after three seconds, poisoning the baseline Event Loop. For ReadDoS (circles), evil input was injected four times at one second intervals beginning after three seconds, eventually poisoning the baseline Worker Pool's four threads. The lines for Node.cure show its effectiveness against these EHP attacks. When attacked, Node.cure's throughput dips until a* `TimeoutError` *aborts the malicious request(s), after which its throughput temporarily rises as it bursts through the built-up queue of pending events or tasks.*

slow file from which to read.[5] Since Line 3 uses the asynchronous framework API `readFile`, each ReadDoS attack on this server will poison a Worker in an I/O-bound EHP attack.

Figure 9.2 shows the impact of EHP attacks on baseline Node.js, as well as the effectiveness of our *Node.cure* prototype. The methodology is described in the caption. On baseline Node.js these attacks result in complete DoS, with zero throughput. Without *Node.cure* the only remedy would be to restart the server, dropping all existing client connections. Unfortunately, restarting the server would not solve the problem, since the attacker could simply submit another malicious request. With *Node.cure* the server can return to its steady-state performance.

The architecture-level behavior of the ReDoS attack is illustrated in Figure 9.1. After client

[5]In addition to files exposed on network file systems, `/dev/random` is a good example of a slow file.

A's benign request is sanitized (CB_{A1}), the `readFile` task goes to the Worker Pool ($Task_{A1}$), and when the read completes the callback returns the file content to A (CB_{A2}). Then client B's malicious request arrives and triggers ReDoS (CB_{B1}), dropping the server throughput to zero. The ReadDoS attack has a similar effect on the Worker Pool, with the same unhappy result.

Comparison to OTPCA EHP attacks are only possible when clients share execution resources. In the OTPCA, a blocked client affects only its own thread, and frameworks such as Apache support thousands of "Event Handlers" (client threads) [159]. In the EDA, all clients share one Event Loop and a limited Worker Pool. For example, in Node.js the Worker Pool can contain at most 128 Workers. Exhausting the set of Event Handlers in the OTPCA requires a DDoS attack, while exhausting them in the EDA is trivial if an EHP vulnerability can be found.

9.6.2.2 Study of reported EHP vulnerabilities in *npm*

Modern software commonly relies on open-source libraries [286], and Node.js applications are no exception. Third-party libraries from the Node.js/JavaScript package registry, *npm*, are frequently used in production [59]. As a result, EHP vulnerabilities in *npm* may translate directly into EHP vulnerabilities in Node.js servers. For example, Staicu and Pradel have demonstrated that many ReDoS vulnerabilities in popular *npm* modules can be used for EHP attacks in hundreds of websites from the Alexa Top Million [307].

In this section we present an EHP-oriented analysis of the security vulnerabilities reported in *npm* modules. As shown in Figure 9.3, we found that 35% (403/1132) of the security vulnerabilities reported in an *npm* vulnerability database could be used as an EHP vector.

Methodology We examined the vulnerabilities in *npm* modules reported in the database of Snyk.io, a security company that monitors open-source library ecosystems for vulnerabilities. We also considered the vulnerabilities in the CVE database and the Node Security Platform database, but found that these databases were subsets of the Snyk.io database.

We obtained a dump of Snyk.io's *npm* database in June 2018. Each entry was somewhat unstructured, with inconsistent CWE IDs and descriptions of different classes of vulnerabilities. Based on its title and description, we assigned each vulnerability to one of 17 main categories based on those used by Snyk.io. We used regular expressions to ensure our classification was consistent. We iteratively improved our regular expressions until we could automatically classify 93% of the vulnerabilities, and marked the remaining 7% as "Other".

Some of the reported security vulnerabilities could be used to launch EHP attacks: Directory Traversal vulnerabilities that permit arbitrary file reads, Denial of Service vulnerabilities

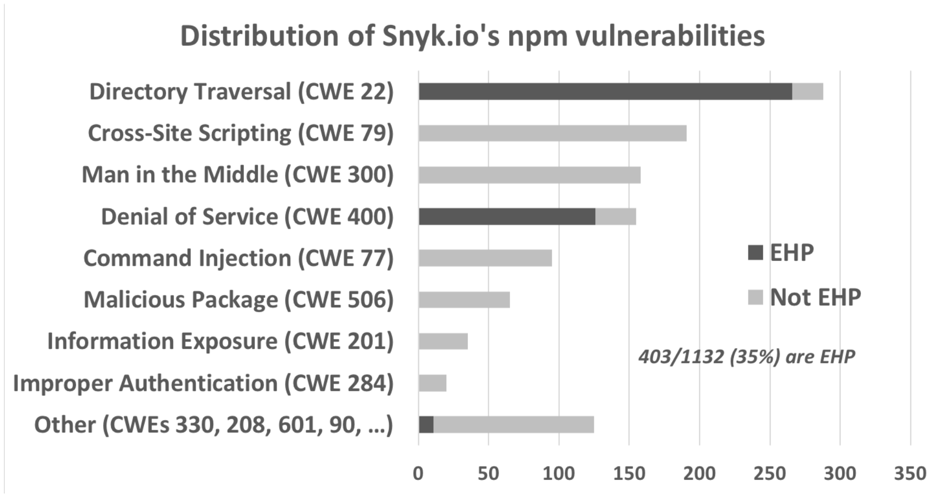


Figure 9.3: ***EHP vulnerabilities in npm modules.*** *Classification of the 1132 npm module vulnerabilities, by category and by usefulness in EHP attacks.*

(those that are CPU-bound, e.g. ReDoS), and Arbitrary File Write vulnerabilities. We identified such vulnerabilities using regular expressions on the descriptions of the vulnerabilities in the database, manually verifying the results. In the few cases where the database description was too terse, we manually categorized vulnerabilities based on the issue and patch description in the module's bug tracker and version control system.

Results Figure 9.3 shows the distribution of vulnerability types, absorbing categories with fewer than 20 vulnerabilities into the aforementioned "Other" category. A high-level CWE number is given next to each class.

The dark bars in Figure 9.3 show the 403 vulnerabilities (35%) that can be employed in an EHP attack under our threat model. The 266 EHP-relevant *Directory Traversal* vulnerabilities are exploitable because they allow arbitrary file reads, which can poison the Event Loop or the Worker Pool through ReadDoS. The 121 EHP-relevant *Denial of Service* vulnerabilities poison the Event Loop; 115 are ReDoS,[6] and the remaining 11 can trigger infinite loops or worst-case performance in inefficient algorithms. In *Other* are 11 Arbitrary File Write vulnerabilities that, similar to ReadDoS, can be used for EHP attacks by writing to slow files.

[6]The proportion of ReDoS vulnerabilities in the Snyk.io database may be skewed by a recent academic interest in ReDoS, from both Staicu and Pradel's work [307] and our own efforts described in Chapter 4.

9.6.3 Defense: First-Class Timeouts

In this section we analyze two paths to EHP-safety in the EDA. EHP vulnerabilities stem from vulnerable APIs that fail to provide fair cooperative multitasking. If a service cannot provide a (small) bound on the synchronous time of its APIs, then it is vulnerable to EHP attacks. Conversely, if an application can bound the synchronous time of its APIs, then it is EHP-safe. An EHP attack can be addressed in two places: (1) at the source, the vulnerable API, or (2) at the symptom, the poisoned Event Handler. Either the vulnerable API can be refactored, or a poisoned Event Handler can be identified and healed.

Ultimately we recommend *First-Class Timeouts* as a universal defense with strong security guarantees. Since time is a precious resource in the EDA, built-in `TimeoutErrors` are a natural mechanism to protect it. Just as `OutOfBoundsErrors` allow applications to detect and react to buffer overflow attacks, so `TimeoutErrors` allow EDA-based applications to detect and react to EHP attacks.

9.6.3.1 Prevent through partitioning

An API is *vulnerable* if there is a difference between its average-case and worst-case synchronous costs, provided of course that this worst-case cost is unbearable. A service can achieve EHP safety by statically bounding the cost of each of its APIs, both those that it invokes and those that it defines itself. For example, a developer could partition every API into a sequence of Constant Worst-Case Execution Time stages. Such a partitioning would render the service immune to EHP attacks since it would bound the synchronous complexity and time of each lifeline.

9.6.3.2 Detect and react through timeouts

The goal of the partitioning approach is to bound a lifeline's synchronous complexity as a way to bound its synchronous time. Instead of statically bounding an API's synchronous complexity through program refactoring, using timeouts we can dynamically bound its synchronous time. Then the worst-case complexity of each callback and task would be irrelevant, because they would be unable to take more than the quantum provided by the runtime. In this approach, the runtime detects and aborts long-running callbacks and tasks by emitting a `TimeoutError`, thrown from synchronous code (callbacks) and returned from asynchronous code (tasks).

We refer to this approach as *First-Class Timeouts*. First-class timeouts are distinct compared to the two existing timeout schemes. One timeout scheme is per-API, e.g. the timeout option in the .NET framework's regular expression API to combat ReDoS. Another timeout scheme is on a per-process or per-thread basis. For example, desktop and mobile operating systems commonly use a heartbeat mechanism to detect and restart unresponsive applications, and in the OTPCA a client thread can easily be killed and replaced if it exceeds a timeout. This approach fails in the EDA because clients are not isolated on separate execution resources. Detecting and restarting a blocked Event Loop will break all existing client connections, resulting in DoS. Because of this, timeouts must be a first-class member of an EDA framework, *non-destructively* guaranteeing that no Event Handler can block.

9.6.3.3 Analysis

Soundness The partitioning approach can prevent EHP attacks that exploit operations with high computational complexity. However, soundly preventing EHP attacks by this means is difficult since it requires case-by-case changes. In addition, it is not clear how to apply the partitioning approach to I/O. At the application level, I/O can be partitioned at the byte granularity, but an I/O may be just as slow for 1 byte as for 1 MB. If an OS offers truly asynchronous I/O interfaces then these provide an avenue to more fine-grained partitioning, but unfortunately Linux's asynchronous I/O mechanisms are incomplete for both file I/O and DNS resolution.

If timeouts are applied systematically across the software stack (application, framework, language), then they offer a strong guarantee against EHP attacks. When a timeout is detected, the application can respond appropriately to it. The difficulty with timeouts is choosing a threshold [278], since a too-generous threshold still permits an attacker to disrupt legitimate requests. As a result, if the timeout threshold cannot be tightly defined, then it ought to be used in combination with a blocklist; after observing a client request time out, the server should drop subsequent connections from that client.

Refactoring cost Both of these approaches incur a refactoring cost. For partitioning the cost is prohibitive. Any APIs invoked by an EHP-safe service must have (small) bounded synchronous time. To guarantee this bound, developers would need to re-implement any third-party APIs with undesirable performance. This task would be particularly problematic in a module-dominated ecosystem similar to Node.js. As the composition of safe APIs may be vulnerable,[7] application APIs might also need to be refactored. The partitioning approach must be done in a case-by-case manner, and future service development and maintenance would need to preserve the bounds required by the service.

For timeouts, we perceive a lower refactoring cost. The timeout must be handled by application developers, but they can do so using existing exception handling mechanisms. Adding a new `try-catch` block should be easier than re-implementing functionality in a partitioned manner.

[7]For example, consider `while(1){}`, which makes an infinite sequence of constant-time language "API calls".

Our recommendation We believe that relying on developers to implement fair cooperative multitasking via partitioning is unsafe. Just as modern languages offer null pointer exceptions and buffer overflow exceptions to protect against common security vulnerabilities, so too should modern EDA frameworks offer timeout exceptions to protect against EHP attacks.

9.6.4 Implementation of First-Class Timeouts in Node.js

Next we describe our implementation of *Node.cure*, a prototype implementation of first-class timeouts for Node.js. Supporting first-class timeouts requires changes across the entire Node.js stack [161], from the language runtime (V8), to the event-driven library (libuv), and to the Node.js core libraries.

Though first-class timeouts are conceptually simple, realizing them in a real-world framework such as Node.js is not trivial. For soundness, every aspect of the Node.js framework must be able to emit `TimeoutErrors` without compromising the system state, from the language to the libraries to the application logic, and in both synchronous and asynchronous aspects. For practicality, monitoring for timeouts must be lightweight, lest they cost more than they are worth.

Desired behavior of first-class timeouts We want to bound the synchronous time of every callback and task and deliver a `TimeoutError` if this bound is exceeded. A long-running callback poisons the Event Loop; with first-class timeouts a `TimeoutError` should be thrown within such a callback. A long-running task poisons its Worker; such a task should be aborted and fulfilled with a `TimeoutError`.

To ensure soundness, we begin with a taxonomy of the places where vulnerable APIs can be found in a Node.js application. The subsequent subsections describe how we provide `TimeoutErrors` across this taxonomy, and touch on some optimizations.

9.6.4.1 Taxonomy of vulnerable Node.js APIs

Table 9.4 classifies vulnerable APIs along three axes. Along the first two axes, a vulnerable API affects either the Event Loop or a Worker, and it might be CPU-bound or I/O-bound. Along the third axis, a vulnerable API can be found in the language, the framework, or the application. In our evaluation we provide an exhaustive list of vulnerable APIs for Node.js at the time of this study (§9.6.5.1). Although the examples in Table 9.4 are specific to Node.js, the same general classification can be applied to other EDA frameworks.

*Table 9.4: **Taxonomy of vulnerable APIs in Node.js**, with examples. An EHP attack through a vulnerable API poisons the Event Loop or a Worker, and its synchronous time is due to CPU-bound or I/O-bound activity. A vulnerable API might be part of the language, framework, or application, and might be synchronous (runs on the Event Loop) or asynchronous (runs on the Worker Pool).* zlib *is the Node.js compression library.* N/A*: At the time of this study, JavaScript has no native Worker Pool nor any I/O APIs. At present, the Web Workers proposal is increasingly relevant and support for Web Workers should be included in any future embodiment of First Class Timeouts. We do not consider memory access as I/O.*

Vuln. APIs	**Event Loop (§9.6.4.3)**		**Worker Pool (§9.6.4.2)**	
	CPU-bound	*I/O-bound*	*CPU-bound*	*I/O-bound*
Language	Regexp, JSON	*N/A*	*N/A*	*N/A*
Framework	Crypto, zlib	FS	Crypto, zlib	FS, DNS
Application	while(1)	DB query	Regexps via [18]	DB query

9.6.4.2 Timeout-aware tasks

EHP attacks targeting the Worker Pool use vulnerable APIs to submit long-running tasks that poison a Worker. *Node.cure* defends against such attacks by bounding the synchronous time of tasks. *Node.cure* short-circuits long-running tasks with a `TimeoutError`.

Timeout-aware Worker Pool Node.js's Worker Pool is implemented in libuv. As illustrated in Figure 9.1, the Workers pop tasks from a shared queue, handle them, and return the results to the Event Loop. Each Worker handles its tasks synchronously.

We modified the libuv Worker Pool to be timeout-aware, replacing libuv's *Workers* with *Executors* that combine a permanent *Manager* with a disposable Worker. Every time a Worker picks up a task, it notifies its Manager. If the task takes the Worker too long, the Manager kills it with a Hangman and creates a new Worker. The long-running task is returned to the Event Loop with a `TimeoutError` for processing, while the new Worker resumes handling tasks. These roles are illustrated in Figure 9.4.

This design required several changes to the libuv Worker Pool API. The libuv library exposes a task submission API `uv_queue_work`, which we extended as shown in Table 9.5. Workers invoke `work`, which is a function pointer describing the task. On completion the Event Loop invokes `done`. This is also the typical behavior of our timeout-aware Workers. When a task takes too long, however, the potentially-poisoned Worker's Manager invokes the new `timed_out` callback. If the submitter does not request an extension, the Manager creates a replacement Worker so that it can continue to process subsequent tasks, creates a Hangman

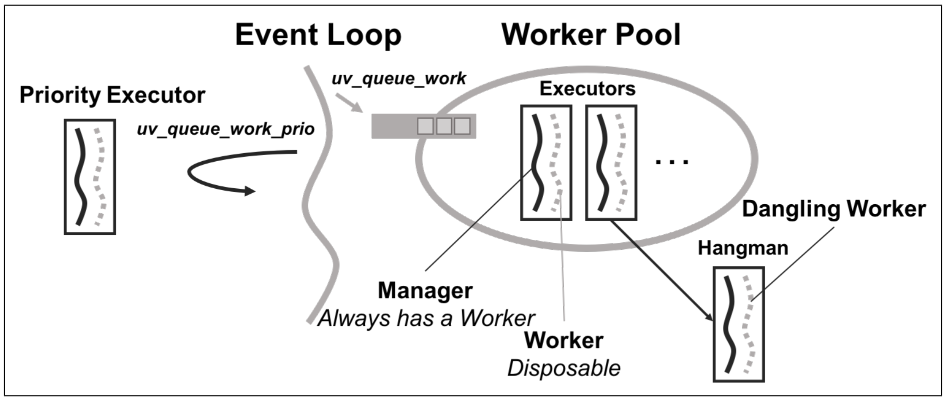


Figure 9.4: ***Node.cure's Event Handler Poisoning-proof architecture.*** *This figure illustrates Node.cure's timeout-aware Worker Pool, including the roles of Event Loop, executors (both worker pool and priority), and Hangman. Grey entities were present in the original Worker Pool, and black are new. The Event Loop can synchronously access the Priority Executor, or asynchronously offload tasks to the Worker Pool. If an Executor's manager sees its worker time out, it creates a replacement worker and passes the dangling worker to a Hangman.*

thread for the poisoned Worker, and notifies the Event Loop that the task timed out. The Event Loop then invokes its `done` callback with a `TimeoutError`, permitting a rapid response to evil input. Concurrently, once the Hangman successfully kills the Worker thread, it invokes the task's `killed` callback for resource cleanup, and returns. We used synchronization primitives to prevent races when a task completes just after it is declared timed out.

Differentiating between `timed_out` and `killed` permits more flexible error handling, but introduces technical challenges. If a rapid response to a timeout is unnecessary, then it is simple to defer `done` until `killed` finishes, since they run on separate threads. If a rapid response is necessary, then `done` must be able to run before `killed` finishes, resulting in a *dangling worker* problem: an API's `work` implementation may access externally-visible state after the Event Loop receives the associated `TimeoutError`. We addressed the dangling worker problem in Node.js's Worker Pool customers using a mix of `killed`-waiting, message passing, and blocklisting.

Affected APIs The Node.js APIs affected by this change (viz. those that create tasks) are in the encryption, compression, DNS, and file system modules. In all cases we allowed timeouts to proceed, killing the long-running Worker. Handling encryption and compression was straightforward, while the DNS and file system APIs were more complex.

Node.js's asynchronous encryption and compression APIs are implemented in Node.js C++ bindings by invoking APIs from `openssl` and `zlib`, respectively. If the Worker Pool notifies these APIs of a timeout, they wait for the Worker to be `killed` before returning, to ensure it

Table 9.5: ***Summary of our extended Worker Pool API.*** *This table summarizes our extended Worker Pool API to support first-class timeouts in Node.cure. The `work` function is invoked on the Worker. The `done` callback is invoked on the Event Loop. The new callbacks, `timed_out` and `killed`, are invoked on the Manager and the Hangman, respectively. On a timeout, `work`, `timed_out`, and `done` are invoked, in that order; there is no ordering between the `done` and `killed` callbacks, which sometimes requires reference counting for safe memory cleanup. *New callbacks.*

Callback	**Description**
void work	Perform task.
int timed_out*	When task has timed out. Can request extension.
void done	When task is done. Special error code for timeout.
void killed*	When a timed_out task's thread has been killed.

no longer modifies state in these libraries nor accesses memory that might be released after `done` is invoked. Since `openssl` and `zlib` are purely computational, the dangling worker is killed immediately.

Node.js implements its file system and DNS APIs by relying on libuv's file system and DNS support, which on Linux make the appropriate calls to libc. Because the libuv file system and DNS implementations share memory between the Worker and the submitter, we modified them to use message passing for memory safety of dangling workers — wherever the original implementation's `work` accessed memory owned by the submitter, e.g. for `read` and `write`, we introduced a private buffer for `work` and added copyin/copyout steps. In addition, we used `pthread_setcancelstate` to ensure that Workers will not be killed while in a non-cancelable libc API [27]. DNS queries are read-only so there is no risk of the dangling worker modifying external state. In the file system, `write` modifies external state, but we avoid any dangling worker state pollution via blocklisting. Our blocklisting-based Slow Resource policy is discussed in more detail in §9.6.4.6.

At the top of the Node.js stack, when the Event Loop sees that a task timed out, it invokes the application's callback with a `TimeoutError`.

9.6.4.3 Timeouts for callbacks

Node.cure defends against EHP attacks that target the Event Loop by bounding the synchronous time of callbacks. To make callbacks timeout-aware, we introduce a TimeoutWatchdog that monitors the start and end of each callback and ensures that no callback exceeds the timeout threshold. We time out JavaScript instructions using V8's interrupt mechanism, and we modify Node.js's C++ bindings to ensure that callbacks that enter these bindings will also be timed out.

Timeouts for JavaScript *TimeoutWatchdog.* Our TimeoutWatchdog instruments every callback using the experimental Node.js `async-hooks` module [19], which allows an application to register special callbacks before and after a callback is invoked.

Before a callback begins, our TimeoutWatchdog starts a timer. If the callback completes before the timer expires, we erase the timer. If the timer expires, the watchdog signals V8 to interrupt JavaScript execution by throwing a `TimeoutError`. The watchdog then starts another timer, ensuring that recursive timeouts while handling the previous `TimeoutError` are also detected. While an infinite sequence of `TimeoutErrors` is possible with this approach, this concern seems more academic than practical.[8]

V8 interrupts. To handle the TimeoutWatchdog's request for a `TimeoutError`, *Node.cure* extends the interrupt infrastructure of Node.js's V8 JavaScript engine to support timeouts. In V8, low priority interrupts such as a pending garbage collection are checked regularly (e.g. each loop iteration, function call, etc.), but no earlier than *after* the current JavaScript instruction finishes. In contrast, high priority interrupts take effect immediately, interrupting long-running JavaScript instructions. Timeouts require the use of a high priority interrupt because they must be able to interrupt long-running individual JavaScript instructions such as `str.match(regexp)` (possible ReDoS).

To support a `TimeoutError`, we modified V8 as follows: (1) We added the definition of a `TimeoutError` into the Error class hierarchy; (2) We added a `TimeoutInterrupt` into the list of high-priority interrupts; and (3) We added a V8 API to raise a `TimeoutInterrupt`. The TimeoutWatchdog calls this API, which interrupts the current JavaScript stack by throwing a `TimeoutError`.

The only JavaScript instructions that V8 instruments to be interruptible are regular expression matching and JSON parsing; these are the language-level vulnerable APIs. Other JavaScript instructions are viewed as effectively constant-time, so these interrupts may be slightly deferred, e.g. to the end of the nearest basic block. We agreed with the V8 developers in this,[9] and did not instrument other JavaScript instructions to poll for pending interrupts.

9.6.4.4 Timeouts for the Node.js C++ bindings

The TimeoutWatchdog described in the previous section will interrupt any vulnerable APIs implemented in JavaScript, including language-level APIs such as regular expressions and application-level APIs that contain blocking code such as `while(1){}`. It remains to give

[8]To obtain an infinite sequence of `TimeoutErrors` in a first-class timeouts system, place a `try-catch` block containing an infinite loop inside another infinite loop.

[9]For example, we found that string operations complete in milliseconds even when a string is hundreds of MBs long.

a sense of time to the Node.js C++ bindings that allow the JavaScript code in Node.js applications to interface with the broader world. A separate effort is required here because a pending `TimeoutError` triggered by the TimeoutWatchdog will not be delivered until control returns from a C++ binding to JavaScript.

Node.js has asynchronous and synchronous C++ bindings. The asynchronous bindings are safe in general because they do a fixed amount of synchronous work to submit a task and then return; the tasks are protected as discussed earlier. However, the synchronous C++ bindings complete the entire operation on the Event Loop before returning, and therefore must be given a sense of time. The relevant vulnerable synchronous APIs are those in the file system, cryptography, and compression modules. Both synchronous and asynchronous APIs in the `child_process` module are also vulnerable, but these are intended for scripting purposes rather than the server context with which we are concerned.

Because the Event Loop holds the state of all pending clients, we cannot `pthread_cancel` it as we do poisoned Workers, since this would result in the DoS the attacker desired. We could build off of our timeout-aware Worker Pool by offloading the request to the Worker Pool and awaiting its completion, but this would incur high request latencies when the Worker Pool's queue is not empty. We opted to combine these approaches by offloading the work in vulnerable synchronous framework APIs to a dedicated Worker, which can be safely killed and whose queue never has more than one item.

In our implementation, we extended the Worker Pool paradigm with a *Priority Executor* whose queue is exposed via a new API: `uv_queue_work_prio` (Figure 9.4). This Executor follows the same Manager-Worker-Hangman paradigm as the Executors in *Node.cure*'s Worker Pool. To make these vulnerable synchronous APIs timeout-aware, we offload them to the Priority Executor using the existing asynchronous implementation of the API, and had the Event Loop await the result. Because these synchronous APIs are performed on the Event Loop as part of a callback, we propagate the callback's remaining time to this Executor's Manager to ensure that the TimeoutWatchdog's timer is honored.

9.6.4.5 Timeouts for application-level vulnerable APIs

As described above, *Node.cure* makes tasks (§9.6.4.2) and callbacks (§9.6.4.3) timeout-aware to defeat EHP attacks against language and framework APIs. An application composed of calls to these APIs will be EHP-safe.

However, an application could still escape the reach of these timeouts by defining its own C++ bindings. These bindings would need to be made timeout-aware, following the example we set while making Node.js's vulnerable C++ bindings timeout-aware (file system, DNS, encryption, and compression). Without refactoring, applications with their own C++ bindings may not be EHP-safe. In our evaluation we found that application-defined C++ bindings are rare (§9.6.5.3).

9.6.4.6 Performance optimizations

Since first-class timeouts are an always-on mechanism, it is important that their performance impact be negligible. Here we describe two optimizations.

Lazy TimeoutWatchdog Promptly detecting `TimeoutErrors` with a *precise* TimeoutWatchdog can be expensive, because the Event Loop must synchronize with the TimeoutWatchdog every time a callback is entered and exited. If the application workload contains many small callbacks, whose cost is comparable to this synchronization cost, then the overhead of a precise TimeoutWatchdog may be considerable.

If the timeout threshold is soft, then the overhead from a TimeoutWatchdog can be reduced by making the Event Loop-TimeoutWatchdog communication asynchronous. When entering and exiting a callback the Event Loop can simply increment a shared counter. A *lazy* TimeoutWatchdog wakes up at intervals and checks whether the callback it last observed has been executing for more than the timeout threshold; if so, it emits a `TimeoutError`. A lazy TimeoutWatchdog reduces the overhead of making a callback, but decreases the precision of the `TimeoutError` threshold based on the frequency of its wake-up interval.

Slow resource policies Our *Node.cure* runtime detects and aborts long-running callbacks and tasks executing on Node.js's Event Handlers. For unique evil input this is the best we can do at runtime, because accurately predicting whether a not-yet-seen input will time out is difficult. If an attacker might re-use the same evil input multiple times, however, we can track whether or not an input led to a timeout and short-circuit subsequent requests that use this input with an early timeout.

While evil input memoization could in principle be applied to any API, the size of the input space to track is a limiting factor. The evil inputs that trigger CPU-bound EHP attacks such as ReDoS exploit properties of the vulnerable algorithm and are thus usually not unique. In contrast, the evil inputs that trigger I/O-bound EHP attacks such as ReadDoS must name a particularly slow *resource*, presenting an opportunity to short-circuit requests on this slow resource.

In *Node.cure* we implemented a slow resource management policy for libuv's file system APIs, targeting those that reference a single resource (e.g. `open`, `read`, `write`). When one of the APIs we manage times out, we mark the file descriptor and the associated inode number as slow. We took the simple approach of permanently blocklisting these aliases by rejecting subsequent accesses,[10] with the happy side effect of solving the dangling worker problem for `write`. This policy is appropriate for the file system, where access times are not likely to change.[11] We did not implement a policy for DNS queries. In the context of DNS, timeouts

[10] To avoid leaking file descriptors, we do not eagerly abort `close`.

[11] Of course, if the slow resource is in a networked file system such as NFS or GPFS, slowness might be

might be due to a network hiccup, and a temporary blocklist might be more appropriate.

9.6.4.7 Prototype details

Node.cure is built on top of Node.js LTS v8.8.1, a recent long-term support version of Node.js at the time of this study.[12] Our prototype is for Linux. We added 4,000 lines of C, C++, and JavaScript code across 50 files spanning V8, libuv, the Node.js C++ bindings, and the Node.js JavaScript libraries.

Node.cure passes the core Node.js test suite, with a handful of failures due to bad interactions with experimental or deprecated features. In addition, several cases fail when they invoke rarely-used file system APIs we did not make timeout-aware. Real applications run on *Node.cure* without difficulty (Table 9.6).

In *Node.cure*, timeouts for callbacks and tasks are controlled by environment variables. Our implementation would readily accommodate a fine-grained assignment of timeouts for individual callbacks and tasks.

9.6.5 Evaluation

Node.cure enables the detection of and response to EHP attacks with application performance overheads ranging from 0% to 24%. Our prototype secures real applications from all known EHP attacks with low overhead.

We evaluated *Node.cure* in terms of its effectiveness (§9.6.5.1), runtime overhead (§9.6.5.2), and security guarantees (§9.6.5.3). In summary: with a lazy TimeoutWatchdog, *Node.cure* detects all known EHP attacks with overhead ranging from 1.3x-7.9x on micro-benchmarks but manifesting at 1.0x-1.24x using real applications. *Node.cure* guarantees EHP-safety to all Node.js applications that do not define their own C++ bindings.

All measurements provided in this section were obtained on an otherwise-idle desktop running Ubuntu 16.04.1 (Linux 4.8.0-56-generic), 16 GB RAM, Intel i7 @3.60GHz, 4 physical cores with 2 threads per core. For a baseline we used Node.js LTS v8.8.1 from which *Node.cure* was derived, compiled with the same flags. We used a default Worker Pool (4 Workers).

9.6.5.1 Effectiveness

To evaluate the effectiveness of *Node.cure*, we developed an EHP test suite that makes every type of EHP attack, as enumerated in Table 9.4. Our suite is comprehensive and conducts EHP attacks using every vulnerable API we identified, including the language level

due to a network hiccup, and incorporating temporary device-level blocklisting might be more appropriate.

[12] Specifically, we built *Node.cure* on Node.js v8.8.1 commit dc6bbb44da from Oct. 25, 2017.

(regular expressions, JSON), framework level (all vulnerable APIs from the file system, DNS, cryptography, and compression modules), and application level (infinite loops, long string operations, array sorting, etc.). This test suite includes each type of real EHP attack from our study of EHP vulnerabilities in *npm* modules. *Node.cure* detects all 92 EHP attacks in this suite: each synchronous vulnerable API throws a `TimeoutError`, and each asynchronous vulnerable API returns a `TimeoutError`. Our suite could be used to evaluate alternative defenses against EHP attacks.

To evaluate any difficulties in porting real-world Node.js software to *Node.cure*, we ported the `node-oniguruma` [18] *npm* module. This module offloads worst-case exponential regular expression queries from the Event Loop to the Worker Pool using a C++ add-on. We ported it using the API described in Table 9.5 without difficulty, as we did for the core modules, and *Node.cure* then successfully detected ReDoS attacks against this module's vulnerable APIs.

9.6.5.2 Runtime overhead

We evaluated the runtime overhead using micro-benchmarks and macro-benchmarks. We address other costs in the Discussion.

Micro-benchmarks Whether or not they time out, *Node.cure* introduces several sources of overheads to monitor callbacks and tasks. We evaluated the most likely candidates for performance overheads using micro-benchmarks:

1. Every time V8 checks for interrupts, it now tests for a pending timeout as well.
2. Both the precise and lazy versions of the TimeoutWatchdog require instrumenting every asynchronous callback using async-hooks, with relative overhead dependent on the complexity of the callback.
3. To ensure memory safety for dangling workers, Workers operate on buffered data that must be allocated when the task is submitted. For example, Workers must copy the I/O buffers supplied to `read` and `write` twice.

New V8 interrupt. We found that the overhead of our V8 Timeout interrupt was negligible, simply a test for one more interrupt in V8's interrupt infrastructure.

TimeoutWatchdog's async hooks. We measured the additional cost of invoking a callback due to TimeoutWatchdog's async hooks. A precise TimeoutWatchdog increases the cost of invoking a callback by 7.9x due to the synchronous communication between Event Loop and TimeoutWatchdog, while a lazy TimeoutWatchdog increases the cost by 2.4x due to the reduced cost of asynchronous communication. While these overheads are large, note that they are for an empty callback. As the number of instructions in a callback increases, the cost of executing the callback will begin to dominate the cost of issuing the callback. For example, if the callback executes 500 empty loop iterations, the precise overhead drops to

2.7x and the lazy overhead drops to 1.3x. At 10,000 empty loop iterations, the precise and lazy overheads are 1.15x and 1.01x, respectively.

Worker buffering. Our timeout-aware Worker Pool requires buffering data to accommodate dangling workers, affecting DNS queries and file system I/O. Our micro-benchmark indicated a 1.3x overhead using `read` and `write` calls with a 64 KB buffer. This overhead will vary from API to API.

Macro-benchmarks Our micro-benchmarks suggested that the overhead introduced by *Node.cure* may vary widely depending on what an application is doing. Applications that make little use of the Worker Pool will pay the overhead of the additional V8 interrupt check (minimal) and the TimeoutWatchdog's async hooks, whose cost is strongly dependent on the number of instructions executed in the callbacks. Applications that use the Worker Pool will pay these as well as the overhead of Worker buffering (variable, perhaps 1.3x).

We chose macro-benchmarks using a GitHub potpourri technique: we searched GitHub for "language:JavaScript", sorted by "Most starred", and identified server-side projects from the first 50 results. To add additional complete servers, we also included LokiJS [8], a popular key-value store, and IBM's Acme-Air airline simulation [15], which is used in the Node.js benchmark suite.

Table 9.6 lists the macro-benchmarks we used and the performance overhead for each type of TimeoutWatchdog. These results show that *Node.cure* introduces minimal overhead on real server applications, and they confirm the value of a lazy TimeoutWatchdog. Matching our micro-benchmark assessment of the TimeoutWatchdog's overhead, the overhead from *Node.cure* increased as the complexity of the callbacks used in the macro-benchmarks decreased — the middleware benchmarks sometimes used empty callbacks to handle client requests. In non-empty callbacks similar to those of the real servers, this overhead is amortized.

9.6.5.3 Security guarantees

Our *Node.cure* prototype implements first-class timeouts for Node.js. *Node.cure* enforces timeouts for all vulnerable JavaScript and framework APIs identified by both us and the Node.js developers as long-running: regular expressions, JSON, file system, DNS, cryptography, and compression. Application-level APIs composed of these timeout-aware language and framework APIs are also timeout-aware.

However, Node.js also permits applications to add their own C++ bindings, and these may not be timeout-aware without refactoring. To evaluate the extent of this limitation, we measured the number of *npm* modules that define C++ bindings. These modules typically depend on the `node-gyp` and/or `nan` modules [52, 53]. We obtained the dependency list for each of the 628,863 *npm* modules from `skimdb.npmjs.com` and found that 4,384 modules

Table 9.6: ***Performance evaluation of Node.cure using macro-benchmarks.*** *Results of our macro-benchmark evaluation of Node.cure's overhead. Where available, we used the benchmarks defined by the project itself. Otherwise, we ran its test suite. Overheads are reported as "precise, lazy", and are the ratio of Node.cure's performance to that of the baseline Node.js, averaged over several steady-state runs. We report the average overhead because we observed no more than 3% standard deviation in all but LokiJS, which averaged 8% standard deviation across our samples of its sub-benchmarks. *: Median of sub-benchmark overheads.*

Benchmark	**Description**	**Overheads**
LokiJS [8]	Server, Key-value store	1.00, 1.00
Node Acme-Air [15]	Server, Airline simulation	1.03, 1.02
webtorrent [40]	Server, P2P torrenting	1.02, 1.02
ws [41]	Utility, websockets	1.00, 1.00*
Three.js [23]	Utility, graphics library	1.09, 1.08
Express [25]	Middleware	1.24, 1.06
Sails [21]	Middleware	1.23, 1.14*
Restify [38]	Middleware	1.63, 1.14*
Koa [7]	Middleware	1.60, 1.24

(0.7%) had these dependencies.[13]

As only 0.7% of *npm* modules define C++ bindings, we conclude that C++ bindings are not widely used and that they thus do not represent a serious limitation of our approach. In addition, we found the refactoring process for C++ bindings straightforward when we performed it on the Node.js framework and the `node-oniguruma` module as described earlier.

9.6.6 Discussion of EHP and First-Class Timeouts

Programming with first-class timeouts What would it be like to develop software for an EDA framework with first-class timeouts? First-class timeouts change the language and framework specifications. First, developers must choose a timeout threshold. Then, exception handling code will be required for both asynchronous APIs, which may be fulfilled with a `TimeoutError`, and synchronous APIs, which may throw a `TimeoutError`.

The choice of a timeout is a Goldilocks problem. Too short, and legitimate requests will result in an erroneous `TimeoutError` (false positive). Too long, and malicious requests will waste a lot of service time before being detected (false negative). Timeouts in other contexts have been shown to be selected without much apparent consideration [278], but for first-class timeouts we suggest that a good choice is relatively easy. Consider that a typical web server can handle hundreds or thousands of clients per second. Since each of these clients requires the invocation of at least one callback on the Event Loop, simple arithmetic tells us that in an EDA-based server, individual callbacks and tasks must take no longer than milliseconds to complete. Thus, a universal callback-task timeout on the order of 1 second should not result in erroneous timeouts during the normal execution of callbacks and tasks, but would permit relatively rapid detection of and response to an EHP attack.[14] By definition, first-class timeouts preclude the possibility of undetected EHP attacks (false negatives) with a reasonable choice of timeout, and our *Node.cure* prototype demonstrates that this guarantee can be provided in practice.

Developers can assign tighter timeout thresholds to reduce the impact of an EHP attack. If a tight timeout can be assigned, then a malicious request trying to trigger EHP will get about the same amount of server time as a legitimate request will, before the malicious request is detected and aborted with a `TimeoutError`. The lower the variance in callback and task times, the more tightly the timeout thresholds can be set without false positives. Though our implementation uses coarse-grained timeouts for callbacks and tasks, more fine-grained timeouts are possible. Such an API might be called `process.runWithTimeout(func)`. Appropriate coarse or fine-grained timeout thresholds could also be suggested automatically or tuned over the process lifetime of the server.

[13]We counted those that matched the regexp `"nan"|"node-gyp"` on 11 May 2018.

[14]If a service is unusually structured so as to run operations on behalf of many clients in a single callback, then when this service is overloaded such a callback might throw a `TimeoutError`. We recommend that such a callback be partitioned.

If a tight timeout cannot be assigned, perhaps because there is significant natural variation in the cost of handling legitimate requests, then we recommend that the `TimeoutError` exception handling logic incorporate a blocklist. With a blocklist, the total time wasted by EHP attacks is equal to the number of attacks multiplied by the timeout threshold. Since DDoS is outside of our threat model, this value should be small and EHP attacks should not prove overly disruptive.

After choosing a timeout, software engineers would need to modify their code to handle any `TimeoutErrors`. For asynchronous APIs that submit tasks to the Worker Pool, a `TimeoutError` will be delivered just like any other error, and error handling logic should already be present. This logic could be extended, for example to blocklist the client. For synchronous APIs or synchronous links in an asynchronous sequence of callbacks, we acknowledge that it is a bit strange that an unexceptional-looking sequence of code such as a loop can now throw an error, and wrapping every function with a `try-catch` block seems inelegant. Happily, recent trends in asynchronous programming techniques have made it easier for software engineers to handle these errors. The ECMAScript 6 specification made Promises a native JavaScript feature, simplifying data-flow programming (explicit encoding of a lifeline) [101]. Promise chains permit catch-all handling of exceptions thrown from any link in the chain, so existing catch-all handlers can be extended to handle a `TimeoutError`.

Other examples of EHP attacks Two other EHP attacks are worth mentioning. *First*, if the EDA framework uses a garbage collected language for the Event Loop (as do Node.js, Vert.x, Twisted, etc.), then triggering many memory allocations could lead to unpredictable blockage of the Event Loop. We are not aware of any reported attacks of this form, but such an attack would defeat first-class timeouts unless the GC were partitioned. *Second*, Linux lacks kernel support for asynchronous DNS requests, so they are typically implemented in EDA frameworks in the Worker Pool. If an attacker controls a DNS nameserver configured as a tarpit [226] and can convince an EDA-based victim to resolve name requests using this server, then each such request will poison one of the Workers in the Worker Pool. First-class timeouts will protect against this class of attacks as it does ReadDoS.

Detecting EHP attacks without first-class timeouts Without first-class timeouts, a service that is not perfectly partitioned may have EHP vulnerabilities. In existing EDA frameworks there is no way to elegantly detect and recover from an EHP attack. Introducing a heartbeat mechanism into the service would enable the detection of an EHP attack, but what then? If more than one client is connected, as is inevitable given the multiplexing philosophy of the EDA, it is not feasible to interrupt the hung request without disrupting the other clients, nor it does seem straightforward to identify which client was responsible. In contrast, first-class timeouts will produce a `TimeoutError` at some point during the handling of the malicious request, permitting exception handling logic to easily respond by dropping the client and, perhaps, adding them to a blocklist.

Other avenues toward EHP-safety In §9.6.3 we described two ways to achieve EHP-safety within the existing EDA paradigm. Other approaches are also viable but they depart from the EDA paradigm. Significantly increasing the size of the Worker Pool, performing speculative concurrent execution [111], or switching to preemptable callbacks and tasks could each prevent or reduce the impact of EHP attacks. However, each of these is a variation on the same theme: dedicating isolated execution resources to each client, a road that leads to the One Thread Per Client Architecture. The recent development of serverless architectures [214] is yet another form of the OTPCA, with the load balancing role played by a vendor rather than the service provider. If the server community wishes to use the EDA, which offers high responsiveness and scalability through the use of cooperative multitasking, we believe first-class timeouts are a good path to EHP-safety.

Generalizability Our first-class timeouts technique can be applied to any EDA framework. Callbacks must be made interruptible, and tasks must be made abortable. While these properties are more readily obtained in an interpreted language, they could in principle be enforced in compiled or VM-based languages as well.

9.6.7 Related Work

JavaScript and Node.js Ojamaa and Duuna assessed the security risks in Node.js applications [266]. Their analysis included ReDoS and other expensive computation as a means of blocking the event loop, though they overlooked the risks of I/O and the fact that the small Worker Pool makes its poisoning possible.

Other works have identified the use of untrusted third-party modules as a common liability in Node.js applications. DeGroef et al. proposed a reference monitor approach to securely integrate third-party modules from *npm* [143]. Vasilakis et al. went a step further in their BreakApp system, providing strong isolation guarantees at module boundaries with dynamic policy enforcement at runtime [327]. The BreakApp approach is complete enough that it can be used to defeat EHP attacks, through what might be called Second-Class Timeouts. Our work mistrusts particular *instructions* and permits the delivery of `TimeoutErrors` at arbitrary points in sequential code, while these reference monitor approaches mistrust *modules* and thus only permit the delivery of `TimeoutErrors` at module boundaries. In addition, moving modules to separate processes in order to handle EHP attacks incurs significant performance overheads at start-up and larger performance overheads than *Node.cure* at run-time, and places more responsibility on developers to understand implementation details in their dependencies.

Static analysis can be used to identify a number of vulnerabilities in JavaScript and Node.js applications. Guarnieri and Livshits demonstrated static analyses to eliminate the use of vulnerable language features or program behaviors in the client-side context [182]. Staicu et al. offered static analyses and dynamic policy enforcement to prevent command injection vulnerabilities in Node.js applications [308]. Static taint analysis for JavaScript, as proposed by Tripp et al. enables the detection of other injection attacks as well [322]. The techniques in these works can detect the possibility of EHP attacks that exploit known-vulnerable APIs (e.g. I/O such as `fs.readFile`), but not those exploiting arbitrary computation. Our first-class timeouts approach is instead a dynamic detect-and-respond defense against EHP attacks.

Embedded systems Time is precious in embedded systems as well. Lyons et al. proposed the use of `TimeoutErrors` in mixed-criticality systems to permit higher-priority tasks to interrupt lower-priority tasks [228]. Their approach incorporates timeouts as a notification mechanism for processes that have overrun their time slices, toying with preemption in a non-preemptive operating system. Our work is similar in principle but differs significantly in execution.

Denial of Service attacks Research on DoS can be broadly divided into network-level attacks (e.g. DDoS attacks) and application-level attacks [60]. Since EHP attacks exploit the semantics of the application, they are application-level attacks, not easily defeated by network-level defenses.

DoS attacks seek to exhaust the resources critical to the proper operation of a server, and various kinds of exhaustion have been considered. The brunt of the literature has focused on exhausting the CPU, e.g. via worst-case performance [135, 136, 231, 267, 302], infinite recursion [113], and infinite loops [106, 304]. We are not aware of prior research work that incurs DoS using the file system, as our ReadDoS attacks do, though we have found a handful of CVE reports to this effect through a semi-automated search.[15]

9.7 Discussion

Algorithm-oriented resource caps From our investigation of RQ1, we learned that existing algorithm-oriented resource caps are ineffective. We were particularly surprised to learn that Perl's approach cannot detect certain exponentially slow regex matches, which we suggest should be the *minimum* standard for such a resource cap. This finding may speak to the difficulty of identifying and monitoring algorithm-oriented measures of resource utilization in sufficiently complex software. Regardless, this finding certainly speaks to the value of a strong test suite for such defenses. Our corpus of super-linear regexes may be of value to other regex engine developers interested in incorporating algorithm-oriented resource

[15]For DoS by reading the slow file `/dev/random`, see CVE-2012-1987 and CVE-2016-6896. For a related DOS by reading large files, CVE-2001-0834, CVE-2008-1353, CVE-2011-1521, and CVE-2015-5295 mention DoS by memory exhaustion using `/dev/zero`.

caps, in validating that their implementations actually address a variety of super-linear regexes.

Time-oriented resource caps In RQ1 we found that the .NET framework's time-oriented solution was more effective at preventing super-linear regex evaluations than the algorithm-oriented solutions of PHP and Perl were. We therefore recommend that the maintainers of Spencer-style regex engines consider a time-oriented approach to resource caps.

However, our investigation of RQ2 showed that most of the C# software projects that we considered have not availed themselves of the time-oriented resource caps offered by the .NET framework. In order to nudge engineers towards the safer APIs, it might be advisable for the .NET framework maintainers to deprecate the unprotected versions of the regex APIs.

In RQ3 we explored a from-scratch implementation of a time-oriented resource cap. The First-Class Timeouts we proposed are capable of protected applications against ReDoS as well as the larger family of Event Handler Poisoning attacks. We believe this stronger approach to resource caps is feasible in new web frameworks, and represents a more systematic approach in the resource cap design space than introducing resource caps into expensive APIs one by one.

9.8 Threats to validity

In §9.5 we conducted an empirical study whose conclusions face certain threats to validity. Many of these threats are shared with other, similar experiments in this dissertation and have been discussed earlier. Two threats are new.

Internal validity Our methodology considered *all* C# modules that were hosted on GitHub. We did not filter out projects based on versions of the .NET framework that predate the support for regex timeouts. In particular, regex timeouts were introduced in .NET framework version 4.5. Although .NET framework version 4.5 is was dropped in 2016 in favor of version 4.5.2, Microsoft still supports the legacy .NET framework 3.5 SP1.[16] It strikes us as unlikely that the vast majority of C# modules would target a legacy version of the .NET framework, but we have not tested this assumption.

External validity C# is less well represented in the open-source community than many other programming languages [169]. The threat of non-generalization to (closed-source) applications may be exacerbated when studying open-source C# software.

[16]Microsoft's lifecycle policy for the .NET framework is described here: `https://support.microsoft.com/en-us/help/17455/lifecycle-faq-net-framework`.

Part IV

Conclusions and Recommendations

Chapter 10

Conclusions and recommendations

10.1 Summary

In §1.2 I stated my thesis:

> Because 10% of regexes exhibit super-linear worst-case behavior in typical regex engine implementations, ReDoS is a significant security threat to real-world software. Existing solutions are ineffective or impractical, while our new approaches appear promising.

In the preceding chapters, I provided evidence in support of this thesis. This evidence is summarized next.

10.1.1 ReDoS is a problem in practice

In Part II I provided evidence for the first part of my thesis: *ReDoS is a serious security threat for modern software applications.* These empirical studies are the only large-scale surveys of the risk of ReDoS in practice, both in depth (conducted at ecosystem scale) and breadth (covering eight programming languages). Although these studies have limitations, two findings from these efforts are clear:

Common in practice Regexes with the potential to be super-linear are commonly deployed in practice (Chapter 4). Up to 10% of real regexes are potentially super-linear, a result that holds across many programming languages (Chapter 5).

Risky regex engines In many programming languages, these regexes exhibit super-linear worst-case behavior in the real regex engines in which they are deployed (§5.7). In six of the eight regex engines we considered, up to 10% of regexes exhibit polynomial or exponential worst-case behavior.

These empirical studies of software products are corroborated by qualitative studies of engineering practices. In their surveys of practicing software engineers, Michael et al. found that a majority of those surveyed were unaware of the risk of ReDoS and the performance risks of copy-pasting regexes across programming language boundaries [237, 238]. Given the relative ease of creating a super-linear regex, and the widespread ignorance of the risks of ReDoS, it is unsurprising that ReDoS vulnerabilities appear to be widespread.

10.1.2 Recommended ReDoS solution

In Part III I provided evidence for the second part of my thesis: *Existing ReDoS solutions are ineffective or impractical, but our new approaches appear promising.* To summarize my findings:

Regex refactoring is difficult Although many super-linear regex behaviors can be identified using existing heuristics and research tools, software engineers struggle to repair a super-linear regex once they identify it (Chapter 6).

Regex engines are incompatible Real regex engines are insufficiently compatible to permit trivially substituting one regex engine for another (Chapter 7). Even accounting for syntactic differences, PCRE-style regex engines differ in their match semantics. Some of these semantic differences are subtle but apparently deliberate, while others are defects that had not been identified by engine maintainers. These regex engines were designed to be compatible with one another. Their inconsistencies highlight the risk of regressions that could result from a major refactoring (e.g., to a linear-time algorithm).

Selective memoization is promising Memoization has been proposed as a solution to address ReDoS, but its high space cost on typical regexes makes it unattractive. We have proposed and proved the theoretical guarantees afforded by two selective memoization schemes (Chapter 8). We showed that they offer similar time complexity as full memoization with far lower space complexity, and that with an appropriate encoding scheme, the space complexity can be reduced to a constant in most cases.

Time-based resource caps can protect web services Various classes of resource caps, both algorithmic and time-based, have been proposed and implemented in several regex engines. We report that time-based schemes are far more effective than algorithmic-oriented schemes, and that off-by-default schemes are rarely adopted (Chapter 9). We explored the design and implementation of First-Class Timeouts, always-on time-based resource cap, which can protect web services against ReDoS and a variety of other denial of service attacks. This approach combines strong security guarantees with a natural programming model.

Although first-class timeouts have strong security guarantees, they would entail a significant refactoring burden for existing applications. We therefore recommend that regex engine developers pursue selective memoization as a solution to ReDoS. Memoization is a transparent solution, improving the worst-case performance of a regex match without introducing any change in the outcome of the match. Selective memoization reduces the accompanying space cost, making memoization a practical approach. Even if it is not extended to E-regexes (§8.7), selective memoization is applicable to the truly regular K-regexes that comprise 95% of our large-scale many-language regex corpus.

10.2 Future work

In this dissertation, we believe that we have conducted the critical measurements to establish the risk of REDOS in modern software, and that we have considered all of the major means of addressing ReDoS. There are, however, opportunities for several lines of work in these directions.

10.2.1 Empirical studies

Our empirical studies have focused on the use of regexes in software modules, and the concomitant risk of ReDoS for applications that depend on these modules. We omitted, however, two aspects of regex usage: the ways in which these modules are used by software applications, and the regexes used in applications themselves.

An end-to-end understanding of ReDoS It would be helpful to understand the end-to-end implications of the super-linear regexes we have identified in software modules. Each of these super-linear regexes may affect many applications among its dependencies. We are not concerned about over-stating the risk of ReDoS— the analysis of Wüstholz et al. included reachability, and on a small set of high-quality Java applications they still identified 1.5% of regex usages as super-linear [341]. But a better understanding of the typical implications of using the modules we studied may guide software engineers as they select dependencies.

Regex engineering practices in specialized domains One aspect of our research has been to better understand software engineering practices surrounding regexes. Our studies have examined regexes as they are used in "standard" engineering practice, with no eye on particular application domains. One common and critical use of regexes is in processing network traffic, e.g., in routing (§3.3) and NIDS use cases ([127]). Another common case is in text processing, e.g., in natural language processing or as input to machine learning models. Understanding the variation of regexes by application domain may expose further opportunities for tool development. It may also motivate domain-specific regex engine optimizations, as Intel has recently demonstrated [334].

10.2.2 Solution approaches

We have demonstrated that ReDoS is a security vulnerability affecting many real-world applications. We have discussed several solution approaches at different levels of the application stack. Several of these approaches require further study. In the near term, refactoring will be necessary, but in the long term we hope to have given regex engine developers sufficient motivation to address this problem.

Regex engine semantics No matter the approach, any ReDoS solution must avoid unexpected changes in the behavior of the regex match observed by the application. Our study in Chapter 7 demonstrated that production regex engines exhibit subtle semantic differences. A richer understanding of practical regex engine semantics is necessary to ensure that ReDoS solutions do not introduce application errors. Work led by Câmpeanu [107, 108] and Berglund [84, 87, 88] has begun this effort, but we do not yet understand individual regex engines in enough detail to validate ReDoS solutions. Defining semantics would also permit the systematic testing of a regex engine in isolation, rather than relying on differential testing as we did in Chapter 7.

Automatic refactoring In Chapter 6 we described the difficulty that practitioners had in solving ReDoS problems. Automated refactoring support seems to be in order. These solutions should propose regexes that meet three conditions: they are linear-time, they match a similar-enough language of strings to be useful, and they are maintainable. If the refactoring can guarantee identical semantics (but against what semantics?), as does that of van der Merwe et al. [326], then it can be performed at compilation or run-time without regard to maintainability. If the refactoring does not preserve the regex's language [125], then understanding its usefulness (difficult without an adequate test suite [332]) and ensuring maintainability (an under-studied area [116]) are both concerns.

Run-time cost prediction In Chapter 9 we discussed resource-cap approaches to addressing ReDoS: *monitor* the cost of an evaluation, and limit it using an exception. An alternative approach is to *predict* the cost of a regex evaluation. Such an approach might take several forms, including:

- Super-linear regexes could be identified at regex compilation time. Accurate analysis has high time complexity [69], so this approach might be more preferable during software testing than deployment. This cost need be paid only once per regex, however, and regex engines could incorporate a directory of (the hashes of) commonly-used regexes and their complexity.
- The sound and complete identification of super-linear regexes is expensive, making their identification potentially undesirable in production using accurate methods. Perhaps the structural characteristics that lead to super-linear behavior could be learned offline by a machine learning model. If the necessary regex characteristics can be extracted quickly, inferences could be made at runtime instead.
- Although many regexes exhibit super-linear worst-case behavior, they only do so on specially crafted input. Problematic *input* could be soundly identified by testing for membership using Wüstholz's attack automata [341]; the regex engine could construct an accompanying attack automaton when a regex is compiled. As proposed for regexes, an unsound approach for identifying problematic input would be to search it for the *signature* of problematic input: a prefix, a sequence of pumps, and a suffix. This task is trivial

when repeated strings are unique (e.g., `/(a+)+$/`, which is only triggered by pumping "a"). But for some regexes, the problematic repeated strings must instead be generalized in terms of a regular language (e.g., `/(\w|\d)*$/`, for which any sequence of digits is problematic). The symbolic regex analysis techniques of Veanes et al. might be helpful to apply in this context [328].

The engineering costs of memoization in practice In Chapter 8 we showed that selective memoization is a promising approach to offer linear time complexity with lowered constant-to-linear space complexity. Two aspects of this work remain for future study. First, incorporating memoization into a prototype regex engine is straightforward, just a few lines of code on the matching algorithm (Listing 10). The engineering costs of incorporating memoization into one of the production-grade backtracking regex engines, with their many special cases and optimizations, remain to be seen. Second, a simple RLE scheme offered constant space costs for the majority of the super-linear regexes, under the worst-case inputs proposed by the super-linear regex detectors. It is unclear whether this property is guaranteed or merely an accident of the input generation regime, and whether the length of the runs can be tuned to the regex to guarantee constant space costs.

10.2.3 Additional future work

The human side of the problem This dissertation has focused on the technical side of the ReDoS problem. Qualitative work by Michael et al. [238], Michael [237], and Donohue [149] has shown that there is a human side as well. Unlike well-known security vulnerabilities like SQL injection [311] and cross-site scripting (XSS) [211], fewer than half of the hundreds of practitioners they surveyed understood the risks of ReDoS. Addressing ReDoS in practice will therefore require considering social aspects including how best to educate practitioners about this problem, and how to incentivize practitioners to adopt ReDoS defenses. Lessons learned from such studies may be applicable to other emerging security vulnerabilities, e.g., GraphQL-based DoS [186, 339].

Applications of our regex corpus In Chapter 8 we showed two applications of our polyglot regex corpus: in motivating regex engine optimizations, and in evaluating the effect of these optimizations. We believe that software engineers and researchers can benefit from other applications of this corpus. For example, it could be used to support the development of semantic code search techniques [207]. This should be an improvement over the current practice of browsing Stack Overflow and other code for relevant regexes (§7.5). As another example, in analysis of our corpus we found that many regexes are unique, suggesting that regexes could be used to fingerprint otherwise-obfuscated software. Regex-based fingerprinting may in turn may motivate the development of semantics-preserving regex obfuscation techniques, but we believe that syntactic obfuscation cannot obscure the equivalence of the underlying automata.

Regex tools and regex engines In Chapter 7 and [141] we showed that regexes are commonly (but unsafely) used across programming language boundaries. We envision a regex "universal translator" to help developers port regexes between languages, perhaps assisted by a regex-specific "diff" tool. This task is complicated by incomplete regex specifications, different feature support in different programming languages, and performance variations.

Rethinking Regular Expressions We have shown that super-linear regexes commonly occur in practice. This may indicate that regexes are a fundamentally hazardous tool — useful in theory, but dangerous in practice. It may instead indicate that regexes are an abused tool, used in inappropriate contexts or by ill-informed engineers. We invite a deeper qualitative investigation into regex usage, as well as usability studies of the various approaches for encoding string constraints (e.g., `indexof`, regular expressions, PEGs, CFGs, etc.). We expect such studies to have implications for other tools used by software engineers.

10.3 Broader implications for computing systems

Beyond regular expressions, my work has broader implications for the design of computing systems. Although the details will vary from one system to the next, I suggest that this dissertation provides a case study in the value of data-driven engineering. Most production-grade regex engines are legacy systems. They were developed before the Internet era and its concomitant security considerations. Changing a legacy system requires strong justification, which I provided in Part II: although these engines were suitable in their original deployment contexts, through large-scale empirical studies I and others have shown that super-linear regexes are widely used in security-sensitive contexts. In light of how regexes are used in practice, we can conclude that applications or regex engines should be re-designed. In Part III we examined a range of potential re-designs with a range of compatibility and security. Thanks to our empirical studies, we were able to evaluate these re-designs on representative use cases.

In brief, my dissertation completes the general engineering cycle as applied to regex engines — *build* and *deploy* (done in the 1980s, cf. Chapter 2), then *measure* (Part II) and *refine* (Part III) in light of practical usage, leaving *iterate* for future work. It is disconcerting that legacy regex engines have persisted so long without refinement. As a discipline, we should not take for granted that the assumptions underlying existing systems will continue to hold in the future. For example, similar questions should be (and are being) asked about our compilers and IDEs (e.g., for engineers from different linguistic and cultural contexts [75]), our software analysis and verification systems (e.g., considering factors beyond accuracy to

Table 10.1: ***Artifacts for reproducibility.*** *Summary of the research artifacts associated with this dissertation.*

Material	Relevant chapter(s)	Link to Artifact
Original regex corpus	Chapters 4 and 6	`https://doi.org/10.5281/zenodo.1294300`
Polyglot corpus	Chapters 5 and 7	`https://doi.org/10.5281/zenodo.3257777`
Regex metrics	Chapter 5	`https://doi.org/10.5281/zenodo.3424960`
Node.cure prototype	Chapter 9	`https://github.com/VTLeeLab/node-cure`
Memoized regex engine	Chapter 8	Not yet available

encourage adoption in practice [122]), our version control systems and continuous integration tools (e.g., as applied to engineering trends like mono-repos [72]), our web API designs (e.g., for typed, quickly-evolving data [186]), and our data serialization tools (e.g., accommodating developers' preferred serialization formats [224]). What other critical computing infrastructure requires refinement as its deployment context shifts?

10.4 Reproducibility and open science

Much of the data, analysis software, and experimental results from this dissertation are available in artifacts hosted by Zenodo and GitHub. See Table 10.1 for details.

10.5 Closing remarks

Regular expressions are an ancient technology, predating most aspects of modern programming. That regular expressions are still widely used in modern software is a testament to their utility. That regular expressions are a widespread security risk could be interpreted as a testament to futility. Engineering secure software is a difficult task, from the technical challenges of secure systems design to the social aspects of human operators. When even the fundamental building blocks of software are a security risk, it is hard to say with any confidence that software can truly be made secure. But we have also shown that at least one building block, regular expressions, can be secured in a backwards-compatible manner. Perhaps this is one security vulnerability that can be made a thing of the past.

Bibliography

[1] astor: Python ast read/write. https://github.com/berkerpeksag/astor.

[2] Atom: A hackable text editor for the 21st century. https://atom.io/.

[3] Babel. https://babeljs.io/.

[4] About cloudflare. https://www.cloudflare.com/about-overview/.

[5] Possessive quantifiers. https://www.regular-expressions.info/possessive.html.

[6] Javaparser. https://javaparser.org/.

[7] Koa. https://github.com/koajs/koa.

[8] Lokijs. https://github.com/techfort/LokiJS.

[9] Mediawiki manual: Extensions. https://www.mediawiki.org/wiki/Manual:Extensions, .

[10] Mediawiki. https://www.mediawiki.org/wiki/MediaWiki, .

[11] Mediawiki extension: Quiz. https://www.mediawiki.org/wiki/Extension:Quiz, .

[12] Sites using mediawiki. https://www.mediawiki.org/wiki/Sites_using_MediaWiki/Wikimedia, .

[13] MySQL Reference Manual: Regular Expressions. URL https://dev.mysql.com/doc/refman/8.0/en/regexp.html.

[14] networkx.algorithms.simple_paths.all_simple_paths. https://web.archive.org/save/https://networkx.github.io/documentation/networkx-2.3/reference/algorithms/generated/networkx.algorithms.simple_paths.all_simple_paths.html.

[15] acmeair-node. https://github.com/acmeair/acmeair-nodejs, .

[16] *2017 User Survey Executive Summary*. The Linux Foundation, .

[17] Node.js foundation members. https://foundation.nodejs.org/about/members, .

[18] Node-oniguruma regexp library. https://github.com/atom/node-oniguruma, .

[19] Nodejs async hooks. https://nodejs.org/api/async_hooks.html, .

[20] Perl regular expressions - perl. https://perldoc.perl.org/5.22.0/perlre.html.

[21] sails. https://github.com/balderdashy/sails.

[22] Welcome to stack overflow. https://stackoverflow.com/tour.

[23] three.js. https://github.com/mrdoob/three.js.

[24] distutils: Building and installing python modules. https://docs.python.org/3/library/distutils.html.

[25] express. https://github.com/expressjs/express.

[26] gradle: Adaptable, fast automation for all. https://github.com/gradle/gradle.

[27] Gnu libc – posix safety concepts. https://www.gnu.org/software/libc/manual/html_node/POSIX-Safety-Concepts.html.

[28] Maven repository. https://mvnrepository.com/, .

[29] mvn: Apache maven. https://github.com/apache/maven, .

[30] nosetests: Nose is nicer testing for python. https://github.com/nose-devs/nose.

[31] nox: Flexible test automation for python. https://github.com/theacodes/nox.

[32] npm - the heart of the modern development community. https://www.npmjs.com/, .

[33] npm: A package manager for javascript. https://github.com/npm/cli, .

[34] Pypi - the python package index. https://pypi.org/.

[35] pytest: The pytest framework. https://github.com/pytest-dev/pytest.

[36] Online regex tester and debugger: Php, pcre, python, golang and javascript. https://regex101.com, .

[37] Regexr: Learn, build, & test regex. https://regexr.com, .

[38] restify. https://github.com/restify/node-restify.

[39] tox: Command line driven ci frontend and development task automation tool. https://github.com/tox-dev/tox.

[40] webtorrent. https://github.com/webtorrent/webtorrent.

[41] ws: a node.js websocket library. https://github.com/websockets/ws.

[42] taskset – set or retrieve a process's cpu affinity. https://web.archive.org/web/20180801003855/https://linux.die.net/man/1/taskset, 2004.

[43] regex(7) - linux manual page - posix.2 regular expressions. http://man7.org/linux/man-pages/man7/regex.7.html, 2009.

[44] What's new in the .net framework 4.5. https://web.archive.org/web/20180801003332/https://docs.microsoft.com/en-us/dotnet/framework/whats-new/index, 2012.

[45] Cve-2015-6736. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-6736, 2015.

[46] Revision history for post 38484433: "join tiles in corona sdk into one word for a breakout game grid?". https://stackoverflow.com/posts/38484433/revisions, 2016.

[47] Atomic grouping. https://web.archive.org/web/20180801003637/https://www.regular-expressions.info/atomic.html, 2017.

[48] Microsoft's Node.js Guidelines. https://github.com/Microsoft/nodejs-guidelines, 2017.

[49] Babylon: Babylon is a javascript parser used in babel. http://web.archive.org/web/20171231170138/https://github.com/babel/babel/tree/master/packages/babylon, 2017.

[50] regexp-tree: Regular expressions processor in javascript. https://web.archive.org/web/20180801004201/https://github.com/DmitrySoshnikov/regexp-tree, 2017.

[51] Node.js at IBM. https://developer.ibm.com/node/, 2018.

[52] Node.js v10.1.0: C++ Addons. https://nodejs.org/api/addons.html, 2018.

[53] Node.js v10.1.0: N-API. https://nodejs.org/api/n-api.html, 2018.

[54] Digital Transformation with the Node.js DevOps Stack. https://pages.nodesource.com/digital-transformation-devops-stack-tw.html, 2018.

[55] cloc: Count lines of code. https://web.archive.org/web/20180801003246/https://github.com/AlDanial/cloc, 2018.

[56] npm. https://web.archive.org/web/20180801003712/https://www.npmjs.com, 2018.

[57] Pypi – the python package index. https://web.archive.org/web/20180801003833/https://pypi.org/, 2018.

[58] Stackexchange traffic. https://stackexchange.com/sites?view=list#traffic, 2020.

[59] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. Why Do Developers Use Trivial Packages? An Empirical Case Study on npm. In *Foundations of Software Engineering (FSE)*, 2017. ISBN 9781450351058. doi: 10.1145/3106237.3106267.

[60] Mehmud Abliz. Internet Denial of Service Attacks and Defense Mechanisms. Technical report, 2011.

[61] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Selective memoization. In *Principles of Programming Languages (POPL)*, 2003. ISBN 1581136285. doi: 10.1145/640128.604133.

[62] Alfred V Aho. Pattern matching in strings. In *Formal Language Theory*, pages 325–347. Elsevier, 1980.

[63] Alfred V Aho. *Algorithms for finding patterns in strings*, chapter 5, pages 255–300. Elsevier, 1990.

[64] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM (CACM)*, 18(6):333–340, 1975. ISSN 00010782. doi: 10.1145/360825.360855.

[65] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. Compilers: Principles, Technologies, and Tools, 2006.

[66] Fahad Aldebeyan. *Improving Software Quality for Regular Expression Matching Tools Using Automated Combinatorial Testing*. PhD thesis, Simon Fraser University, 2018.

[67] James Algina, HJ Keselman, and Randall D Penfield. An alternative to cohen's standardized mean difference effect size: a robust parameter and confidence interval in the two independent groups case. *Psychological methods*, 10(3):317, 2005.

[68] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. Understanding Asynchronous Interactions in Full-Stack JavaScript. In *International Conference on Software Engineering (ICSE)*, 2016. ISBN 9781450339001. doi: 10.1145/2884781.2884864.

[69] Cyril Allauzen, Mehryar Mohri, and Ashish Rastogi. General Algorithms for Testing the Ambiguity of Finite Automata. In *International Conference on Developments in Language Theory*, 2008.

[70] Rene Alquezar and A Sanfeliu. Incremental Grammatical Inference From Positive and Negative Data Using Unbiased Finite State Automata. 1999.

[71] Torben Amtoft and Jesper Larsson Träff. Partial memoization for obtaining linear time behavior of a 2DPDA. *Theoretical Computer Science*, 98(2):347–356, 1992. ISSN 03043975. doi: 10.1016/0304-3975(92)90008-4.

[72] Sundaram Ananthanarayanan, Masoud Saeida Ardekani, Denis Haenikel, Balaji Varadarajan, Simon Soriano, Dhaval Patel, and Ali Reza Adl-Tabatabai. Keeping master green at scale. In *European Conference on Computer Systems (EuroSys)*, 2019. ISBN 9781450362818. doi: 10.1145/3302424.3303970.

[73] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. MutRex: A Mutation-Based Generator of Fault Detecting Strings for Regular Expressions. In *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2017. ISBN 9781509066766. doi: 10.1109/ICSTW.2017.23.

[74] Bill Atkinson. *Hypercard*. Apple Computer, 1988.

[75] Titus Barik, Denae Ford, Emerson Murphy-Hill, and Chris Parnin. How should compilers explain problems to developers? In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 633–643, 2018. ISBN 9781450355735. doi: 10.1145/3236024.3236040.

[76] Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. Inference of Regular Expressions for Text Extraction from Examples. *IEEE Transactions on Knowledge and Data Engineering*, 28(5):1217–1230, 2016. ISSN 10414347. doi: 10.1109/TKDE.2016.2515587.

[77] Daniel Bates, Adam Barth, and Collin Jackson. Regular expressions considered harmful in client-side XSS filters. In *The Web Conference (WWW)*, 2010. ISBN 9781605587998. doi: 10.1145/1772690.1772701. URL http://portal.acm.org/citation.cfm?doid=1772690.1772701.

[78] Michela Becchi. *Data Structures, Algorithms, and Architectures for Efficient Regular Expression Evaluation*. PhD thesis, 2009.

[79] Michela Becchi and Patrick Crowley. Extending finite automata to efficiently match perl-compatible regular expressions. In *ACM International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*, 2008. ISBN 9781605582108. doi: 10.1145/1544012.1544037.

[80] Fabian Beck, Stefan Gulan, Benjamin Biegel, Sebastian Baltes, and Daniel Weiskopf. RegViz: Visual Debugging of Regular Expressions. In *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE)*, 2014. ISBN 9781450327688. doi: 10.1145/2591062.2591111. URL http://stackoverflow.com/questions/46155/.

[81] Ralph Becket and Zoltan Somogyi. DCGs + Memoing = Packrat parsing but is it worth it? In *International Symposium on Practical Aspects of Declarative Languages*, 2008. ISBN 3540774416. doi: 10.1007/978-3-540-77442-6{_}13.

[82] Peter Van Beek. Backtracking Search Algorithms. In *Handbook of Constraint Programming*, chapter 4, pages 85–134. 2006.

[83] Richard Bellman. Dynamic programming. *Science*, 153(3731):34–37, 1966. ISSN 23249757. doi: 10.1007/978-3-319-17933-9{_}5.

[84] Martin Berglund and Brink Van Der Merwe. Regular Expressions with Backreferences. In *Prague Stringology*, pages 30–41, 2017. ISBN 9788001061930.

[85] Martin Berglund and Brink van der Merwe. On the Semantics of Regular Expression parsing in the Wild. *Theoretical Computer Science*, 578:292–304, 2015. ISSN 03043975. doi: 10.1016/j.tcs.2015.03.032.

[86] Martin Berglund, Henrik Björklund, Frank Drewes, Brink Van Der Merwe, and Bruce Watson. Cuts in regular expressions. In *Conference on Developments in Language Theory*, volume 7907 LNCS, pages 70–81, 2013. ISBN 9783642387708. doi: 10.1007/978-3-642-38771-5{_}8.

[87] 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.

[88] Martin Berglund, Brink Van Der Merwe, Bruce Watson, and Nicolaas Weideman. On the Semantics of Atomic Subgroups in Practical Regular Expressions. *Springer CIAA*, 2017. ISSN 03043975. doi: 10.1016/j.tcs.2015.03.032.

[89] Martin Berglund, Willem Bester, and Brink van der Merwe. Formalising Boost POSIX Regular Expression Matching. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 11187 LNCS:99–115, 2018. ISSN 16113349. doi: 10.1007/978-3-030-02508-3{_}6.

[90] Alexander Birman and Jeffrey D Ullman. Parsing Algorithms With Backtrack. *Symposium on Switching and Automata Theory (SWAT)*, 1970. doi: 10.1109/swat.1970.18.

[91] A Blackwell. SWYN: A visual representation for regular expressions. *Your Wish is My Command: Programming by ...*, pages 1–18, 2001. URL http://books.google.com/books?hl=en&lr=&id=wM2JYafw11gC&oi=fnd&pg=PA245&dq=SWYN+:+A+Visual+Representation+for+Regular+Expressions&ots=xyFQTGRSo0&sig=6ee-HY_N5hknlI40TmMPys1dPgo.

[92] Ronald Book, Shimon Even, Sheila Greibach, and Gene Ott. Ambiguity in Graphs and Expressions. *IEEE Transactions on Computers*, C-20(2):149–153, 1971. ISSN 00189340. doi: 10.1109/T-C.1971.223204.

[93] Taylor L Booth. Sequential machines and automata theory. 1967.

[94] Mehra Nouroz Borazjany. *Applying Combinatorial Testing to Systems with a Complex Input Space*. PhD thesis, The University of Texas at Arlington, 2013.

[95] Hudson Borges and Marco Tulio Valente. What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform. *Journal of Systems and Software*, 146:112–129, 2018. ISSN 01641212. doi: 10.1016/j.jss.2018.09.016. URL https://linkinghub.elsevier.com/retrieve/pii/S0164121218301961.

[96] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Communications of the ACM (CACM)*, 20(10):762–772, 1977. ISSN 00010782. doi: 10.1145/359842.359859.

[97] Claus Brabrand and Jakob G. Thomsen. Typed and unambiguous pattern matching on strings using regular expressions. *Symposium on Principles and Practice of Declarative Programming (PPDP)*, pages 243–254, 2010. doi: 10.1145/1836089.1836120.

[98] Claus Brabrand, Robert Giegerich, and Anders Møller. Analyzing ambiguity of context-free grammars. *Science of Computer Programming*, 75(3):176–191, 2010. ISSN 01676423. doi: 10.1016/j.scico.2009.11.002. URL http://dx.doi.org/10.1016/j.scico.2009.11.002.

[99] James Britt and Neurogami Secret Laboratory. Regexp - ruby. https://ruby-doc.org/core-2.3.1/Regexp.html.

[100] Benjamin C. Brodle, Ron K. Cytron, and David E. Taylor. A scalable architecture for high-throughput regular-expression pattern matching. In *International Symposium on Computer Architecture (ISCA)*, volume 2006, pages 191–202, 2006. ISBN 076952608X. doi: 10.1109/ISCA.2006.7.

[101] Etienne Brodu, S Frénot, and F Oblé. Toward automatic update from callbacks to Promises. In *Workshop on All-Web Real-Time Systems (AWeS)*, 2015. ISBN 9781450334778. doi: 10.1145/2749215.2749216. URL https://hal.archives-ouvertes.fr/hal-01132776https://hal.archives-ouvertes.fr/hal-01132776/.

[102] Jr. Brooks, Frederick P. The Computer Scientist as Toolsmith II. *Communications of the ACM (CACM)*, 39(3):61–68, 1996. ISSN 0001-0782. doi: 10.1145/227234.227243.

[103] Janusz A. Brzozowski. Derivatives of Regular Expressions. *Journal of the Association for Computing Machinery*, 11(4):481–494, 1964.

[104] Ivan Budiselic, Sinisa Srbljic, and Miroslav Popovic. RegExpert: A tool for visualization of regular expressions. In *The International Conference on Computer as a Tool (EUROCON)*, pages 2387–2389, 2007. ISBN 142440813X. doi: 10.1109/EURCON.2007.4400374.

[105] Arthur W Burks and Hao Wang. The Logic of Automata - part 2. *Journal of the Association for Computing Machinery (JACM)*, 4(3):279–297, 1957.

[106] Jacob Burnim, Nicholas Jalbert, Christos Stergiou, and Koushik Sen. Looper: Lightweight detection of infinite loops at runtime. In *International Conference on Automated Software Engineering (ASE)*, 2009. ISBN 9780769538914. doi: 10.1109/ASE.2009.87.

[107] Cezar Câmpeanu and Nicolae Santean. On the intersection of regex languages with regular languages. *Theoretical Computer Science*, 410(24-25):2336–2344, 2009. ISSN 03043975. doi: 10.1016/j.tcs.2009.02.022. URL http://dx.doi.org/10.1016/j.tcs.2009.02.022.

[108] CEZAR CÂMPEANU, KAI SALOMAA, and SHENG YU. A Formal Study of Practical Regular Expressions. *International Journal of Foundations of Computer Science*, 14(06):1007–1018, 2003. ISSN 0129-0541. doi: 10.1142/s012905410300214x.

[109] Niccolo Cascarano, Pierluigi Rolando, Fulvio Risso, Riccardo Sisto, Niccolo Cascarano, Pierluigi Rolando, Fulvio Risso, Riccardo Sisto, and Politecnico Torino. Public Review for iNFAnt: NFA Pattern Matching on GPGPU Devices. *ACM SIGCOMM Computer Communication Review*, 40(5):20–26, 2010.

[110] Matthew Casias, Kevin Angstadt, Tommy Tracy II, Kevin Skadron, and Westley Weimer. Debugging Support for Pattern-Matching Languages and Accelerators. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019. ISBN 9781450362405.

[111] Gaurav Chadha, Scott Mahlke, and Satish Narayanasamy. Accelerating Asynchronous Programs Through Event Sneak Peek. In *International Symposium on Computer Architecture (ISCA)*, 2015. ISBN 978-1-4503-3402-0. doi: 10.1145/2749469.2750373.

[112] Chia-Hsiang Chang and Robert Paige. From regular expressions to DFA's using compressed NFA's. *Theoretical Computer Science*, 178(178):1–36, 1997. URL https://pdfs.semanticscholar.org/cb61/da2f10ed3521140f0c6ecdd1c524fc0aab59.pdf.

[113] Richard Chang, Guofei Jiang, Franjo Ivančić, Sriram Sankaranarayanan, and Vitaly Shmatikov. Inputs of coma: Static detection of denial-of-service vulnerabilities. In *IEEE Computer Security Foundations Symposium (CSF)*, 2009. ISBN 9780769537122. doi: 10.1109/CSF.2009.13.

[114] Yeim Kuan Chang and Ching Hsuan Shih. A Memory Efficient Pattern Matching Scheme for Regular Expressions. *Procedia Computer Science*, 110:250–257, 2017. ISSN 18770509. doi: 10.1016/j.procs.2017.06.092. URL http://dx.doi.org/10.1016/j.procs.2017.06.092.

[115] Carl Chapman and Kathryn T Stolee. Exploring regular expression usage and context in Python. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2016. ISBN 9781450343909. doi: 10.1145/2931037.2931073.

[116] Carl Chapman, Peipei Wang, and Kathryn T Stolee. Exploring Regular Expression Comprehension. In *Automated Software Engineering (ASE)*, 2017.

[117] Chen Chen, Baojiang Cui, Jinxin Ma, Runpu Wu, Jianchao Guo, and Wenqian Liu. A systematic review of fuzzing techniques. *Computers and Security*, 75:118–137, 2018. ISSN 01674048. doi: 10.1016/j.cose.2018.02.002. URL https://doi.org/10.1016/j.cose.2018.02.002.

[118] Zhifeng Chen, T. V. Lakshman, Randy H. Katz, Fang Yu, and Yanlei Diao. Fast and memory-efficient regular expression matching for deep packet inspection. *Symposium on Architecture For Networking And Communications Systems (ANCS)*, 2006. doi: 10.1145/1185347.1185360.

[119] Sang Cho and Dung T. Huynh. The parallel complexity of finite-state automata problems. *Information and Computation*, 97(1):1–22, 1992. ISSN 10902651. doi: 10.1016/0890-5401(92)90002-W.

[120] Seongmyun Cho. iptables string regex, 2016. URL https://web.archive.org/web/20191010134733/https://github.com/smcho-kr/kpcre/wiki/iptables-string-regex.

[121] Noam Chomsky. Three Models for the Description of Language. *IRE Transactions on information theory*, 2(3):113–124, 1956.

[122] Maria Christakis and Christian Bird. What developers want and need from program analysis: An empirical study. In *Automated Software Engineering (ASE)*, 2016. ISBN 9781450338455. doi: 10.1145/2970276.297.

[123] Alonzo Church. A Note on the Entscheidungsproblem. *The Journal of Symbolic Logic*, 1(1):40–41, 1936. ISSN 1098-6596. doi: 10.1017/CBO9781107415324.004.

[124] Charles L. A. Clarke and Gordon V. Cormack. On the use of regular expressions for searching text. *ACM Transactions on Programming Languages and Systems*, 19(3): 413–426, 2002. ISSN 01640925. doi: 10.1145/256167.256174.

[125] Brendan Cody-Kenny, Michael Fenton, Adrian Ronayne, Eoghan Considine, Thomas McGuire, and Michael O'Neill. A Search for Improved Performance in Regular Expressions. In *Genetic and Evolutionary Computation Conference*, 2017. doi: doi:10.1145/3071178.3071196. URL http://arxiv.org/abs/1704.04119.

[126] Jacob Cohen. A power primer. *Psychological bulletin*, 112(1):155, 1992.

[127] C. Jason Coit, Stuart Staniford, and Joseph McAlerney. Towards faster string matching for intrusion detection or exceeding the speed of Snort. In *Proceedings DARPA Information Survivability Conference and Exposition II (DISCEX)*, 2001. ISBN 0769512127. doi: 10.1109/DISCEX.2001.932231.

[128] Byron Cook and John Launchbury. Disposable memo functions. In *Haskell Workshop*, 1997. doi: 10.1145/258949.258979.

[129] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[130] Oracle Corp. Pattern - java. https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/regex/Pattern.html.

[131] Erik Corry, Christian Plesner Hansen, and Lasse Reichstein Holst Nielsen. Irregexp, Google Chrome's New Regexp Implementation, 2009. URL https://blog.chromium.org/2009/02/irregexp-google-chromes-new-regexp.html.

[132] Russ Cox. Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, PHP, Python, Ruby, ...), 2007.

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

[134] Russ Cox. RE2, 2010. URL https://github.com/google/re2https://github.com/google/re2/wiki/WhyRE2.

[135] Scott Crosby and T H E Usenix Magazine. Denial of service through regular expressions. In *USENIX Security work in progress report*, volume 28, 2003.

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

[137] James R Dabrowski and Ethan V Munson. Is 100 milliseconds too fast? In *CHI'01 extended abstracts on Human factors in computing systems*, pages 317–318, 2001.

[138] James Davis, Gregor Kildow, and Dongyoon Lee. The Case of the Poisoned Event Handler: Weaknesses in the Node.js Event-Driven Architecture. In *European Workshop on Systems Security (EuroSec)*, 2017. ISBN 9781450349352.

[139] 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.

[140] 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 *USENIX Security Symposium (USENIX Security)*, 2018.

[141] 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.

[142] 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.

[143] Willem De Groef, Fabio Massacci, and Frank Piessens. NodeSentry: Least-privilege library integration for server-side JavaScript. In *Annual Computer Security Applications Conference (ACSAC)*, 2014. ISBN 9781450330053. doi: 10.1145/2664243.2664276.

[144] Erik DeBill. Module counts. http://modulecounts-production.herokuapp.com/.

[145] François Denis. Learning regular languages from simple positive examples. *Machine Learning*, 44(1-2):37–66, 2001. ISSN 08856125. doi: 10.1023/A:1010826628977.

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

[147] MDN Web Docs. Regular expressions - javascript. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions, .

[148] MDN Web Docs. Regexp - javascript. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp, .

[149] James Donohue. *Industrial developers' perspectives and processes around regular expression use and ReDoS*. PhD thesis, University of Bradford, 2019.

[150] Stephen C. Drye and William C. Wake. *Java Swing reference*. Manning Publications Company, 1999.

[151] Jay Earley. An Efficient Context-Free Parsing Algorithm. *Communications of the ACM*, 13(2):94–102, 1970. ISSN 15577317. doi: 10.1145/357980.358005.

[152] Eclipse. Eclipse Find/Replace. URL https://help.eclipse.org/kepler/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Freference%2Fviews%2Fshared%2Fref-findreplace.htm.

[153] Stack Exchange. Outage postmortem. http://web.archive.org/web/20180801005940/http://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016, 2016.

[154] Steve Ferg. Event-driven programming: introduction, tutorial, history. 2006. URL http://eventdrivenpgm.sourceforge.net/.

[155] Stenio Fernandes, Géza Szabó, Judith Kelner, Rafael Antonello, and Djamel Sadok. Design and optimizations for efficient regular expression matching in DPI systems. *Computer Communications*, 61:103–120, 2015. ISSN 01403664. doi: 10.1016/j.comcom.2014.12.011. URL http://dx.doi.org/10.1016/j.comcom.2014.12.011.

[156] Michael Fitzgerald. *Introducing regular expressions.* O'Reilly Media, Inc., 2012.

[157] Bryan Ford. Packrat Parsing: Simple, Powerful, Lazy, Linear Time. In *The International Conference on Functional Programming (ICFP)*, volume 37, 2002. ISBN 1581134878. URL http://arxiv.org/abs/cs/0603077.

[158] Bryan Ford. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. In *Principles of Programming Languages (POPL)*, page 354. ACM, 2004. ISBN 158113729X.

[159] Apache Software Foundation. The Apache web server, . URL http://www.apache.org.

[160] Python Software Foundation. re – regular expression operations - python. https://docs.python.org/3.6/library/re.html, .

[161] Scott Frees. *C++ and Node.js Integration.* 2016. URL https://scottfrees.com/ebooks/nodecpp/.

[162] Dominik D. Freydenberger. Extended Regular Expressions: Succinctness and Decidability. *Theory of Computing Systems*, 53(2):159–193, 2013. ISSN 14324350. doi: 10.1007/s00224-012-9389-0.

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

[164] Ugo Galassi and Attilio Giordana. Learning regular expressions from noisy sequences. In *International Symposium on Abstraction, Reformulation, and Approximation*, 2005. ISBN 3540278729. doi: 10.1007/11527862{_}7.

[165] David Galbraith. How i fixed atom: When good regexes go bad. https://web.archive.org/web/20191226230445/http://davidvgalbraith.com/how-i-fixed-atom/, 2016.

[166] David Galbraith. decreasenextindentpattern: prevent catastrophic backtracking. https://github.com/atom/language-go/pull/79, 2016.

[167] Garun, Natt. Downdetector down as another cloudflare outage affects services across the web. https://www.theverge.com/2019/7/2/20678958/downdetector-down-cloudflare-502-gateway-error-discord-outage, 2019.

[168] Gershgorn, Dave. An internet backbone leaked data from millions of sites. here's how to check if you're affected. https://qz.com/918941/cloudflare-leaked-user-data-from-millions-of-websites-heres-how-to-check-if-you-we 2017.

[169] GitHub. The state of the octoverse. https://octoverse.github.com/, 2018.

[170] Herbert Glantz. On the recognition of information with a digital computer. In *ACM national meeting*, 1956.

[171] V M Glushkov. The Abstract Theory of Automata. *Russian Mathematical Surveys*, 16(5):1–53, 1961. ISSN 0036-0279. doi: 10.1070/rm1961v016n05abeh004112.

[172] Viktor M Glushkov. On a synthesis algorithm for abstract automata. *Ukr. Matem. Zhurnal*, 12(2):147–156, 1960.

[173] Kurt Godel. Uber formal unentscheidbare Satze der Principia Mathematica und verwandter Systeme I. *Monthly magazines for mathematics and physics*, 38(1), 1931.

[174] Eric Goebelbecker. Using grep. *Linux Journal*, 1995.

[175] Danny Goodman and Paula Ferguson. *Dynamic HTML: The Definitive Reference*. O'Reilly, 1 edition, 1998.

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

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

[178] Jan Goyvaerts. A list of popular tools, utilities and programming languages that provide support for regular expressions, and tips for using them, 2016. URL https://www.regular-expressions.info/tools.html.

[179] Jan Goyvaerts and Steven Levithan. *Regular expressions cookbook*. O'Reilly Media, Inc., 2012.

[180] 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/.

[181] The PHP Group. Regexp - php. http://php.net/manual/en/regexp.introduction.php.

[182] Salvatore Guarnieri and V Benjamin Livshits. GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. *USENIX Security*, 2009. URL https://css.csail.mit.edu/6.858/2014/readings/gatekeeper.pdf.

[183] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[184] Christian Hagenah and Anca Muscholl. Computing e-free NFA from regular expressions in O(n log^2(n)) time. *Informatique theorique et applications*, 34(4):257–277, 2000.

[185] Jeff Harrell. Node.js at PayPal. https://www.paypal-engineering.com/2013/11/22/node-js-at-paypal/, 2013.

[186] Olaf Hartig and Jorge Pérez. Semantics and Complexity of GraphQL. In *Conference on World Wide Web (WWW)*, 2018. ISBN 9781450356398. doi: 10.1145/3178876.3186014. URL http://dl.acm.org/citation.cfm?doid=3178876.3186014.

[187] Philip Hazel. PCRE - Perl Compatible Regular Expressions, 1997.

[188] Philip Hazel. PCRE2 - Perl Compatible Regular Expressions, 2ed, 2018. URL https://www.pcre.org/current/doc/html/pcre2pattern.html.

[189] Hazel, Philip. Pcre - perl compatible regular expressions. https://web.archive.org/web/20180919101106/https://www.pcre.org/, 2018.

[190] Marius Hoch. Change 209153: Quote params passed into regular expressions. https://gerrit.wikimedia.org/r/#/c/mediawiki/extensions/Quiz/+/209153/, 2015.

[191] Renáta Hodován, Zoltán Herczeg, and Ákos Kiss. Regular expressions on the web. In *International Symposium on Web Systems Evolution (WSE)*, pages 29–32, 2010. ISBN 9781424486366. doi: 10.1109/WSE.2010.5623572.

[192] Markus Holzer and Martin Kutrib. Descriptional and computational complexity of finite automata - A survey. *Information and Computation*, 209(3):456–470, 2011. ISSN 08905401. doi: 10.1016/j.ic.2010.11.013. URL http://dx.doi.org/10.1016/j.ic.2010.11.013.

[193] John Hopcroft. An n log n algorithm for minimizing states in a finite automaton. Technical Report 2, 1971.

[194] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. *Automata theory, languages, and computation*, volume 24. 2006.

[195] Juraj Hromkovič, Sebastian Seibert, and Thomas Wilke. Translating regular expressions into small ε-free nondeterministic finite automata. *Journal of Computer and System Sciences*, 62:565–588, 2001. ISSN 16113349.

[196] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray Failure: The Achilles' Heel of Cloud-Scale Systems. In *Hot Topics in Operating Systems (HotOS)*, 2017. doi: 10.1145/3102980.3103005. URL https://doi.org/10.1145/3102980.3103005.

[197] John Hughes. Lazy memo-functions. In *Conference on Functional Programming Languages and Computer Architecture*, 1985. ISBN 9783540159759. doi: 10.1007/3-540-15975-4{_}34.

[198] Andrew Hume. A Tale of Two Greps. *Software - Practice and Experience*, 18(11): 1063–1072, 1988.

[199] Andrew Hume and Daniel Sunday. Fast String Searching. *Software - Practice and Experience*, 21(11):1221–1248, 1991. doi: 10.2991/ifsa-eusflat-15.2015.60.

[200] IBM. IBM Db2. URL https://www.ibm.com/support/knowledgecenter/en/SSEPGG_10.5.0/com.ibm.db2.luw.xml.doc/doc/xqrregexp.html.

[201] IEEE and The Open Group. *The Open Group Base Specifications Issue 7, 2018 edition*. 2018.

[202] IEEE and The Open Group. The open group base specifications issue 7, 2018 edition, ieee std 1003.1-2017, 2018.

[203] Jamie Jennings. The Rosie Pattern Language. https://rosie-lang.org/, 2020.

[204] Tao Jiang and B Ravikumar. Minimal NFA problems are Hard. *SIAM Journal on Computing*, 22(6):1117–1141, 1993.

[205] Derek Jones. Patterns of regular expression usage: duplicate regexs. http://shape-of-code.coding-guidelines.com/2020/02/16/patterns-of-regular-expression-usage-duplicate-regexs/, 2020.

[206] Julliard, Alexandre. Wine is not an emulator (wine): a windows compatibility layer. https://www.winehq.org/.

[207] Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun. Repairing programs with semantic code search. In *Automated Software Engineering (ASE)*, pages 295–306, 2016. ISBN 9781509000241. doi: 10.1109/ASE.2015.60.

[208] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. HAMPI: A solver for string constraints. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 105–115, 2009. ISBN 9781605583389. doi: 10.1145/1572272.1572286.

[209] James Kirrage, Asiri Rathnayake, and Hayo Thielecke. Static Analysis for Regular Expression Denial-of-Service Attacks. In *International Conference on Network and System Security (NSS)*, pages 35–148, 2013. ISBN 9783642386305.

[210] S. C. Kleene. Representation of events in nerve nets and finite automata. *Automata Studies*, pages 3–41, 1951.

[211] Amit Klein. DOM Based Cross Site Scripting or XSS of the Third Kind. Technical Report July, 2005. URL http://www.webappsec.org/projects/articles/071105.shtml.

[212] Donald E Knuth, Jr. Morris, James H, and Vaughan R Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[213] Andrew Koenig. Patterns and antipatterns. *The patterns handbook: techniques, strategies, and applications*, 13:383, 1998.

[214] Ricardo Koller and Dan Williams. Will Serverless End the Dominance of Linux in the Cloud? In *Hot Topics in Operating Systems (HotOS)*, pages 169–173, 2017. ISBN 9781450350686. doi: 10.1145/3102980.3103008. URL https://pdfs.semanticscholar.org/7e4c/7ed7c772ef43eda726afe68c25cb2e2357f3.pdfhttp://dl.acm.org/citation.cfm?doid=3102980.3103008%0Ahttps://www.sigops.org/hotos/hotos17/papers/hotos17-final99.pdf.

[215] William H Kruskal and W Allen Wallis. Use of ranks in one-criterion variance analysis. *Journal of the American statistical Association*, 47(260):583–621, 1952.

[216] A.M. Kuchling. Regular expression howto - python. https://docs.python.org/3.6/howto/regex.html.

[217] Sailesh Kumar, Balakrishnan Chandrasekaran, Jonathan Turner, and George Varghese. Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. In *Symposium on Architecture For Networking And Communications Systems (ANCS)*, volume 25, page 155, 2007. ISBN 9781595939456. doi: 10.1145/1323548.1323574.

[218] Mark Kvale. Perl regular expressions tutorial - perl. https://perldoc.perl.org/5.22.0/perlretut.html.

[219] Eric Larson. Automatic Checking of Regular Expressions. In *Source Code Analysis and Manipulation (SCAM)*, 2018.

[220] Eric Larson and Anna Kirk. Generating Evil Test Strings for Regular Expressions. In *International Conference on Software Testing, Verification and Validation (ICST)*, 2016. ISBN 9781509018260. doi: 10.1109/ICST.2016.29. URL https://pdfs.semanticscholar.org/abbb/a5867872ab723b272a13607b649f6e7bf008.pdf.

[221] Ville Laurikari. NFAs with tagged transitions, their conversion to deterministic automata and application to regular expressions. *International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 181–187, 2000. doi: 10.1109/SPIRE.2000.878194.

[222] Johnson Ching-Hong Li. Effect size measures in a two-independent-samples case with nonnormal and nonhomogeneous data. *Behavior Research Methods*, 48(4):1560–1574, Dec 2016.

[223] Nuo Li, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. Reggae: Automated Test Generation for Programs Using Complex Regular Expressions. In *Automated Software Engineering (ASE)*, 2009. ISBN 978-1-4244-5259-0. doi: 10.1109/ASE.2009.67. URL https://www.researchgate.net/profile/Wolfram_Schulte/publication/220883583_Reggae_Automated_Test_Generation_for_Programs_Using_Complex_Regular_Expressions/links/00b7d5141dce1354d8000000.pdfhttp://ieeexplore.ieee.org/document/5431742/.

[224] Yinan Li, Nikos R Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. Mison: A Fast JSON Parser for Data Analytics. *Very Large DataBases (VLDB)*, 10(10):1118–1129, 2017. ISSN 21508097. URL http://www.vldb.org/pvldb/vol10/p1118-li.pdfhttps://www.microsoft.com/en-us/research/wp-content/uploads/2017/05/mison-vldb17.pdf.

[225] Yunyao Li, Rajasekar Krishnamurthy, Sriram Raghavan, Shivakumar Vaithyanathan, and H. V. Jagadish. Regular expression learning for information extraction. In *Conference on Empirical Methods in Natural Language Processing*, pages 21–30, 2008. doi: 10.3115/1613715.1613719.

[226] Tom Liston. Welcome To My Tarpit: The Tactical and Strategic Use of LaBrea. http://www.threenorth.com/LaBrea/LaBrea.txt, 2001.

[227] Blake Loring, Duncan Mitchell, and Johannes Kinder. Sound regular expression semantics for dynamic symbolic execution of JavaScript. In *Programming Language Design and Implementation (PLDI)*, pages 425–438. Association for Computing Machinery, 6 2019. ISBN 9781450367127. doi: 10.1145/3314221.3314645.

[228] Anna Lyons, Kent McLeod, Hesham Almatary, and Gernot Heiser. Scheduling-Context Capabilities: A Principled, Light-Weight Operating-System Mechanism for Managing

Time. In *European Conference on Computer Systems (EuroSys)*, 2018. doi: 10.1145/3190508.3190539. URL https://doi.org/10.1145/3190508.3190539.

[229] Konstantinos Manikas and Klaus Marius Hansen. Software ecosystems-A systematic literature review. *Journal of Systems and Software*, 86(5):1294–1306, 2013. ISSN 01641212. doi: 10.1016/j.jss.2012.12.026. URL https://pdfs.semanticscholar.org/75fb/1e203859e11afa5deaec07acfff215adeed0.pdf.

[230] Warren S. McCulloch and Walter Pitts. A Logical Calculus of the Ideas Immanent in Nervous Activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133, 1943. ISSN 00074985. doi: 10.1007/BF02478259.

[231] M D Mcilroy. Killer adversary for quicksort. *Software - Practice and Experience*, 29(4):341–344, 1999. ISSN 00380644. doi: 10.1002/(SICI)1097-024X(19990410)29:4<341::AID-SPE237>3.0.CO;2-R.

[232] William M Mckeeman. Differential Testing for Software. *Digital Technical Journal*, 10(1), 1998.

[233] Lee E. McMahon. sed, 1973. URL http://sed.sourceforge.net/sedfaq2.html.

[234] R McNaughton and H Yamada. Regular Expressions and State Graphs for Automata. *IRE Transactions on Electronic Computers*, 5:39–47, 1960.

[235] George H. Mealy. A Method for Synthesizing Sequential Circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955. ISSN 15387305. doi: 10.1002/j.1538-7305.1955.tb03788.x.

[236] Sérgio Medeiros, Fabio Mascarenhas, and Roberto Ierusalimschy. From regexes to parsing expression grammars. *Science of Computer Programming*, 93(PART A):3–18, 2014. ISSN 01676423. doi: 10.1016/j.scico.2012.11.006. URL http://dx.doi.org/10.1016/j.scico.2012.11.006.

[237] Louis G Michael IV. *Exploring the Process and Challenges of Programming with Regular Expressions*. PhD thesis, Virginia Tech, 2019.

[238] 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.

[239] Donald Michie. ”Memo” Functions and Machine Learning. *Nature*, 1968.

[240] Microsoft. Regex class - c#. https://docs.microsoft.com/en-us/dotnet/api/system.text.regularexpressions.regex, .

[241] Microsoft. Automata and transducer library for .net. https://github.com/AutomataDotNet/Automata, .

[242] Microsoft. Microsoft Word Help and Training: Find and replace text, 2019. URL https://support.office.com/en-us/article/find-and-replace-text-c6728c16-469e-43cd-afe4-7708c6c779b7.

[243] Microsoft. Use regular expressions in Visual Studio, 2019. URL https://docs.microsoft.com/en-us/visualstudio/ide/using-regular-expressions-in-visual-studio.

[244] Matthew Might, David Darais, and Daniel Spiewak. Parsing with derivatives: A functional pearl. In *The International Conference on Functional Programming (ICFP)*, 2011. ISBN 9781450308656. doi: 10.1145/2034574.2034801.

[245] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, 1990. ISSN 00010782. doi: 10.1145/96267.96279. URL http://ftp.cs.wisc.edu/paradyn/technical_papers/fuzz.pdf.

[246] Jelena Mirkovic and Peter Reiher. A taxonomy of DDoS attack and DDoS defense mechanisms. *ACM SIGCOMM Computer Communication Review*, 34(2):39, 2004. ISSN 01464833. doi: 10.1145/997150.997156.

[247] Jelena Mirkovic, Sven Dietrich, David Dittrich, and Peter Reiher. *Internet denial of service: attack and defense mechanisms*. Prentice Hall PTR, 2004.

[248] David G Mitchell. A SAT Solver Primer. Technical report, 2005.

[249] Kota Mizushima, Atusi Maeda, and Yoshinori Yamaguchi. Packrat parsers can handle practical grammars in mostly constant space. *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 29–36, 2010. doi: 10.1145/1806672.1806679.

[250] Abhijeet Mohapatra and Michael Genesereth. Incrementally maintaining run-length encoded attributes in column stores. *ACM International Conference Proceeding Series*, pages 146–154, 2012. doi: 10.1145/2351476.2351493.

[251] Mohd_PH. Regex Search (Firefox plugin). URL https://addons.mozilla.org/en-US/firefox/addon/regexsearch/.

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

[253] Edward F Moore. Gedanken-experiments on sequential machines. In *Automata Studies*. 1956.

[254] Robert Muth and Udi Manber. Approximate Multiple String Search. In *Annual Symposium on Combinatorial Pattern Matching*, 1996.

[255] Eugene W. Myers and Webb Miller. Approximate matching of regular expressions. *Bulletin of Mathematical Biology*, 51(1):5–37, 1989. ISSN 15229602. doi: 10.1007/BF02458834.

[256] Kedar Namjoshi and Girija Narlikar. Robust and fast pattern matching for intrusion detection. *IEEE INFOCOM*, 2010. ISSN 0743166X. doi: 10.1109/INFCOM.2010.5462149.

[257] Gonzalo Navarro. NR-grep: A fast and flexible pattern-matching tool. *Software - Practice and Experience*, 31(13):1265–1312, 2001. ISSN 00380644. doi: 10.1002/spe.411.

[258] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2002. ISSN 03600300. doi: 10.1145/375360.375365.

[259] Gonzalo Navarro and Mathieu Raffinot. New techniques for regular expression searching. *Algorithmica*, 41:89–116, 2005. ISSN 01784617. doi: 10.1007/s00453-004-1120-3.

[260] Gonzalo Navarro and Mathieu Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. pages 14–33, 2005. doi: 10.1007/bfb0030778.

[261] Nehman, Lily Hay. Cloudflare's plan to protect the whole internet comes into focus. https://www.wired.com/story/cloudflare-spectrum-iot-protection/, 2018.

[262] Cyril Nicaud. On the Average Size of Glushkov's Automata. In *International Conference on Language and Automata Theory and Applications (LATA)*, number Lata 2009, pages 626–637, 2009.

[263] Eric Norige and Alex Liu. A De-compositional Approach to Regular Expression Matching for Network Security Applications. *International Conference on Distributed Computing Systems (ICDCS)*, 2016-Augus:680–689, 2016. doi: 10.1109/ICDCS.2016.63.

[264] Peter Norvig. Techniques for Automatic Memoization with Applications to Context-Free Parsing. *Computational Linguistics*, 17(1):91–98, 1991. ISSN 0362-613X.

[265] J. O'Dell. Exclusive: How LinkedIn used Node.js and HTML5 to build a better, faster app. http://venturebeat.com/2011/08/16/linkedin-node/, 2011.

[266] A Ojamaa and K Duuna. Assessing the security of Node.js platform. In *7th International Conference for Internet Technology and Secured Transactions (ICITST)*, 2012. ISBN VO -.

[267] Oswaldo Olivo, Isil Dillig, and Calvin Lin. Detecting and Exploiting Second Order Denial-of-Service Vulnerabilities in Web Applications. *ACM Conference on Computer and Communications Security (CCS)*, 2015. ISSN 15437221. doi: 10.1145/2810103.2813680.

[268] John K. Ousterhout. Scripting: Higher-level programming for the 21st century. *Computer*, 31(3):23–30, 1998. ISSN 00189162. doi: 10.1109/2.660187.

[269] Overleaf. (Overleaf) How do I get to the next search match result, or perform a search and replace? URL https://www.overleaf.com/learn/how-to/How_do_I_get_to_the_next_search_match_result,_or_perform_a_search_and_replace%3F.

[270] OWASP. Web application firewall. https://owasp.org/www-community/Web_Application_Firewall.

[271] Scott Owens, John Reppy, and Aaron Turon. Regular-expression derivatives re-examined. *Journal of Functional Programming*, 19(2):173–190, 2009. ISSN 09567968. doi: 10.1017/S0956796808007090.

[272] Senthil Padmanabhan. How We Built eBay's First Node.js Application. https://www.ebayinc.com/stories/blogs/tech/how-we-built-ebays-first-node-js-application/, 2013.

[273] Vivek S Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An Efficient and Portable Web Server. In *USENIX Annual Technical Conference (ATC)*, 1999. doi: 10.1.1.119.6738.

[274] pam. Issue 287: Long, complex regexp pattern (in webkit layout test) hangs. https://bugs.chromium.org/p/v8/issues/detail?id=287.

[275] David Pariag, Tim Brecht, Ashif Harji, Peter Buhr, Amol Shukla, and David R Cheriton. Comparing the performance of web server architectures. In *European Conference on Computer Systems (EuroSys)*. ACM, 2007.

[276] Davide Pasetto, Fabrizio Petrini, and Virat Agarwal. Tools for very fast regular expression matching. *IEEE Computer*, pages 50–58, 2010.

[277] Perl monks. Perl regexp matching is slow?? https://perlmonks.org/?node_id=597262.

[278] Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, and Rebecca Isaacs. 30 seconds is not enough! In *European Conference on Computer Systems (EuroSys)*, 2008. ISBN 978-1-60558-013-5. doi: 10.1145/1357010.1352614. URL http://delivery.acm.org/10.1145/1360000/1352614/p205-peter.pdf?ip=128.173.237.147&id=1352614&acc=ACTIVESERVICE&key=B33240AC40EC9E30.80AE0C8B3B97B250.4D4702B0C3E38B35.4D4702B0C3E38B35&__acm__=1516978495_3a3c2334d5b881c3ca6d5d24400d34b4http://portal.acm.o.

[279] 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.

[280] Lutz Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, 2000.

[281] M. Rabin and D. Scott. Finite Automata and their Decision Problems. *IBM Journal of Research and Development*, 3:114–125, 1959. URL https://www.researchgate.net/profile/Dana_Scott3/publication/230876408_Finite_Automata_and_Their_Decision_Problems/links/582783f808ae950ace6cd752/Finite-Automata-and-Their-Decision-Problems.pdf.

[282] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, Scott A. Smolka, Terrance Swift, and David S. Warren. Efficient model checking using tabled resolution. In *International Conference on Computer Aided Verification (CAV)*, 1997. ISBN 3540631666. doi: 10.1007/3-540-63166-6{_}16.

[283] Ghulam Rasool and Nadim Asif. Software artifacts recovery using abstract regular expressions. In *IEEE International Multitopic Conference (INMIC)*, 2007. ISBN 1424415535. doi: 10.1109/INMIC.2007.4557710.

[284] Asiri Rathnayake and Hayo Thielecke. Static Analysis for Regular Expression Exponential Runtime via Substructural Logics. Technical report, 2014.

[285] Eric S. Raymod. The Jargon File: Entry for 'grep'. URL http://www.catb.org/~esr/jargon/html/G/grep.html.

[286] Eric S. Raymond. *The Cathedral and the Bazaar*. Number July 1997. 2000. ISBN 0596001088. doi: 10.1007/s12130-999-1026-0.

[287] A. H. Robinson and Colin Cherry. Results of a Prototype Television Bandwidth Compression Scheme. *Proceedings of the IEEE*, 55(3):356–364, 1967. ISSN 15582256. doi: 10.1109/PROC.1967.5493.

[288] Martin Roesch. Snort - Lightweight Intrusion Detection for Networks. In *Large Installation System Administration Conference (LISA)*, 1999. ISBN 1-880446-25-1. doi: http://portal.acm.org/citation.cfm?id=1039834.1039864.

[289] Rick Rogers, John Lombardo, Zigurd Mednieks, and Blake Meike. *Android application development: Programming with the Google SDK*. O'Reilly Media, Inc., 1 edition, 2009.

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

[291] Indranil Roy, Ankit Srivastava, Matt Grimm, Marziyeh Nourian, Michela Becchi, and Srinivas Aluru. Evaluating High Performance Pattern Matching on the Automata Processor. *IEEE Transactions on Computers*, 68(8):1201–1212, 2019. ISSN 15579956. doi: 10.1109/TC.2019.2901466.

[292] Rshen. Chrome Regex Search (Chrome plugin). URL https://chrome.google.com/webstore/detail/chrome-regex-search/bpelaihoicobbkgmhcbikncnpacdbknn?hl=en-US.

[293] Olli Saarikivi, Margus Veanes, Tiki Wan, and Eric Xu. Symbolic regex matcher. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 11427 LNCS, pages 372–378, 2019. ISBN 9783030174613. doi: 10.1007/978-3-030-17462-0{_}24.

[294] Markus L. Schmid. Characterising REGEX languages by regular languages equipped with factor-referencing. *Information and Computation*, 249:1–17, 2016. ISSN 10902651. doi: 10.1016/j.ic.2016.02.003.

[295] Niko Schwarz, Aaron Karper, and Oscar Nierstrasz. Efficiently extracting full parse trees using regular expressions with capture groups. *PeerJ Preprints*, 2015. doi: 10.7287/peerj.preprints.1248.

[296] Claude E Shannon and John McCarthy. *Automata studies*, volume 34. 1956.

[297] 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.

[298] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 9th edition, 2012. ISBN 0470128720.

[299] Janice Singer and Timothy Lethbridge. What's so great about 'grep'? Implications for program comprehension tools. Technical report, 2002.

[300] Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. An examination of software engineering work practices. In *Centre for Advanced Studies on Collaborative Research (CASCON)*, 1997. doi: 10.1145/1925805.1925815.

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

[302] Randy Smith, Cristian Estan, and Somesh Jha. Backtracking Algorithmic Complexity Attacks Against a NIDS. In *Annual Computer Security Applications Conference (ACSAC)*, pages 89–98, 2006.

[303] Randy Smith, Cristian Estan, Somesh Jha, and Shijin Kong. Deflating the Big Bang: Fast and Scalable Deep Packet Inspection with Extended Finite Automata. In *SIGCOMM*, 2008. ISBN 9781605581750. URL http://pages.cs.wisc.edu/~estan/publications/bigbang.pdf.

[304] Sooel Son and Vitaly Shmatikov. SAFERPHP Finding Semantic Vulnerabilities in PHP Applications. In *Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 1–13, 2011. ISBN 9781450308304. doi: 10.1145/2166956.2166964.

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

[306] Eric Spishak, Werner Dietl, and Michael D. Ernst. A type system for regular expressions. In *Workshop on Formal Techniques for Java-like Programs. (FTfJP)*, 2012. ISBN 9781450312721. doi: 10.1145/2318202.2318207.

[307] 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.

[308] Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. Synode: Understanding and Automatically Preventing Injection Attacks on Node.js. In *Network and Distributed System Security (NDSS)*, 2018. ISBN 1-891562-49-5. doi: 10.14722/ndss.2018.23071. URL http://software-lab.org/publications/ndss2018.pdfhttps://www.microsoft.com/en-us/research/publication/understanding-automatically-preventing-injection-attacks-node-js/.

[309] Richard M. Stallman. GNUs Flashes, February 1988, 1988. URL https://www.gnu.org/bulletins/bull4.html.

[310] R E Stearns and H 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.

[311] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. In *Principles of Programming Languages (POPL)*, 2006. doi: 10.1145/1111320.1111070.

[312] substack. safe-regex. https://web.archive.org/web/20180801003748/https://github.com/substack/safe-regex, 2013.

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

[314] Bryan Sullivan. Security Briefs - Regular Expression Denial of Service Attacks and Defenses. Technical report, 2010. URL https://msdn.microsoft.com/en-us/magazine/ff646973.aspx.

[315] Bryan Sullivan. New Tool: SDL Regex Fuzzer, 2010. URL https://blogs.microsoft.com/microsoftsecure/2010/10/12/new-tool-sdl-regex-fuzzer/.

[316] Martin Sulzmann and Kenny Zhuo Ming Lu. Regular expression sub-matching using partial derivatives. In *ACM SIGPLAN Principles and Practice of Declarative Programming (PPDP)*, pages 79–90, 2012. ISBN 9781450315227. doi: 10.1145/2370776.2370788.

[317] 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.

[318] Richard E Sweet. The Mesa programming environment. *ACM SIGPLAN Notices*, 20 (7):216–229, 1985. ISSN 0362-1340. doi: 10.1145/17919.806843.

[319] Hisao Tamaki and Taisuke Sato. OLD resolution with tabulation. In *International Conference on Logic Programming*, 1986. ISBN 9783540164920. doi: 10.1007/3-540-16492-8{_}66.

[320] Jorma Tarhio and Esko Ukkonen. Approximate Boyer-Moore String Matching. *SIAM*, 22(2):243–260, 1993.

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

[322] Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. Andromeda : Accurate and Scalable Security Analysis of Web Applications. In *International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 210–225, 2013. doi: 10.1007/978-3-642-37057-1{_}15. URL https://link.springer.com/content/pdf/10.1007/978-3-642-37057-1_15.pdfhttps://s3.amazonaws.com/academia.edu.documents/45250213/ANDROMEDA_accurate_and_scalable_security20160501-11331-1y0jt30.pdf?AWSAccessKeyId=AKIAIWOWYYGZ2Y53UL3A&Expires=1513738670&Signa.

[323] Iain Truskett. Perl regular expressions reference - perl. https://perldoc.perl.org/5.22.0/perlreref.html.

[324] Alan M Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Mathematica*, 38(1936):230–265, 1936. ISSN 1460244X. doi: 10.1112/plms/s2-42.1.230. URL http://draperg.cis.byuh.edu/archive/winter2014/cs320/Turing_Paper_1936.pdf.

[325] D. A. Turner. The Semantic Equivalence of Applicative Languges. In *Conference on Functional Programming Languages and Computer Architecture*, pages 85–92, 1981. ISBN 0897910605.

[326] 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.

[327] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhan, Andre DeHon, and Jonathan M Smith. BreakApp: Automated, Flexible Application Compartmentalization. In *Network and Distributed System Security (NDSS)*, 2018. doi: 10.14722/ndss.2018.23131. URL http://nathandautenhahn.com/downloads/publications/vasilakis-breakapp-2018.pdf.

[328] Margus Veanes, Peli De Halleux, and Nikolai Tillmann. Rex: Symbolic regular expression explorer. *International Conference on Software Testing, Verification and Validation (ICST)*, 2010. ISSN 2159-4848. doi: 10.1109/ICST.2010.15.

[329] Larry Wall. Perl, 1988.

[330] Wall, Larry. Pcre – perl compatible regular expressions. https://web.archive.org/web/20180919103344/http://perldoc.perl.org/perlre.html, 2018.

[331] Kai Wang, Zhe Fu, Xiaohe Hu, and Jun Li. Practical regular expression matching free of scalability and performance barriers. *Computer Communications*, 54:97–119, 2014. ISSN 01403664. doi: 10.1016/j.comcom.2014.08.005. URL http://dx.doi.org/10.1016/j.comcom.2014.08.005.

[332] Peipei Wang and Kathryn T Stolee. How well are regular expressions tested in the wild? In *Foundations of Software Engineering (FSE)*, 2018. ISBN 9781450355735.

[333] Peipei Wang, Gina R Bai, and Kathryn T Stolee. Exploring Regular Expression Evolution. In *Software Analysis, Evolution, and Reengineering (SANER)*, 2019.

[334] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. Hyperscan: A Fast Multi-pattern Regex Matcher for Modern CPUs. In *Networked Systems Design and Implementation (NSDI)*, pages 631–648, 2019. ISBN 9781931971492. URL https://www.usenix.org/conference/nsdi19/presentation/wang-xiang.

[335] 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.

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

[337] Matt Welsh, David Culler, and Eric Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Symposium on Operating Systems Principles (SOSP)*, 2001. ISBN 1581133898. doi: 10.1145/502059.502057.

[338] Wikipedia contributors. Regular expression — Wikipedia, the free encyclopedia. `https://web.archive.org/web/20180920152821/https://en.wikipedia.org/w/index.php?title=Regular_expression`, 2018.

[339] Erik Wittern, Alan Cha, James C. Davis, Guillaume Baudart, and Louis Mandel. An Empirical Study of GraphQL Schemas. In *International Conference on Service-Oriented Computing (ICSOC)*, pages 3–19, 2019. doi: 10.1007/978-3-030-33702-5{_}1.

[340] Sun Wu and Udi Manber. AGREP - A fast approximate pattern-matching tool. In *USENIX Annual Technical Conference (ATC)*, 1992.

[341] Valentin Wüstholz, 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.

[342] Chengcheng Xu, Jinshu Su, and Shuhui Chen. Exploring efficient grouping algorithms in regular expression matching. *PLoS ONE*, 13(10):1–15, 2018. ISSN 19326203. doi: 10.1371/journal.pone.0206068.

[343] Di Yang, Pedro Martins, Vaibhav Saini, and Cristina Lopes. Stack Overflow in Github: Any Snippets There? In *Mining Software Repositories (MSR)*, 2017. ISBN 9781538615447. doi: 10.1109/MSR.2017.13.

[344] Liu Yang, Vinod Ganapathy, Pratyusa Manadhata, and Ye Wu. A novel algorithm for pattern matching with back references. In *International Performance Computing and Communications Conference (IPCCC)*, 2015. ISBN 9781467385909.

[345] Xiaochun Yang, Tao Qiu, Bin Wang, Chen Li, Baihua Zheng, and Yaoshu Wang. Negative Factor: Improving Regular-Expression Matching in Strings. *ACM Transactions on Database Systems*, 40(25):1–46, 2016. doi: 10.1145/2847525. URL `http://dx.doi.org/10.1145/2847525`.

[346] Yi-Hua E. Yang and Viktor K. Prasanna. Optimizing Regular Expression Matching with SR-NFA on Multi-Core Systems. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2011. ISBN 9780769545660. doi: 10.1109/PACT.2011.73.

[347] Xiaodong Yu and Michela Becchi. Exploring different automata representations for efficient regular expression matching on GPUs. *ACM SIGPLAN Notices*, 48(8):287–288, 2013. ISSN 15232867. doi: 10.1145/2517327.2442548.

[348] Xiaodong Yu and Michela Becchi. GPU acceleration of regular expression matching for large datasets: Exploring the implementation space. In *Proceedings of the ACM International Conference on Computing Frontiers (CF)*, 2013. ISBN 9781450320535. doi: 10.1145/2482767.2482791.

[349] Lukasz Ziarek, K. C. Sivaramakrishnan, and Suresh Jagannathan. Partial memoization of concurrency and communication. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 161–172, 2009. ISBN 9781605583327. doi: 10.1145/1596550.1596575.

[350] Yuan Zu, Ming Yang, Zhonghu Xu, Lin Wang, Xin Tian, Kunyang Peng, and Qunfeng Dong. GPU-based NFA implementation for memory efficient high speed regular expression matching. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2012. ISBN 9781450311601. doi: 10.1145/2370036.2145833.

Appendices

Appendix A

Notes on the Perl regex engine

A.1 Introduction

When scientists have considers a "production" regex engine, they use the Java regex engine as a representative of the "production" regex engines [297, 335, 341]. The Perl regex engine is also of scientific interest, because as discussed in Chapter 8 it is the most sophisticated of the Spencer-style backtracking regex engines in terms of its optimizations. Here I give some notes on its structure to guide future researchers.

A.2 Origins

As mentioned in Chapter 2, many production regex engines follow the structure of Henry Spencer's backtracking-based regex engine. The Perl regex engine is in fact a literal descendant of Henry Spencer's library. The Perl file `regexec.c` begins as follows: "*NOTE: this is derived from Henry Spencer's regexp code, and should not confused with the original package. Thanks, Henry!*"

A.3 Conversion from pattern to automaton

A regular expression is converted to an automaton in `regcomp.c`. This conversion builds up a structure of vertices whose types are taken from `regcomp.sym`. The compilation also analyzes the sub-patterns of the regex, for example distinguishing between constant, fixed-width, and variable-width sub-patterns. These distinctions permit various optimizations, e.g., the backtracking optimization discussed in §8.2. These analyses are performed in the `study_chunk` function.

A.4 Automaton simulation

The automaton simulation is performed by the `S_regmatch` function. This function tracks the current engine search state, $\langle q, i\rangle$, and manipulates its backtracking stack based on the type and information associated with each vertex. The simulation is begun at various feasible starting points, as determined by `Perl_re_intuit_start`.

A.5 Avoiding super-linear behavior

The memoization discussed in §8.2 is applied during the processing of complex quantified sub-patterns (`CURLYX-WHILEM`). There is a lengthy comment block that begins “super-linear cache processing” and discusses the properties of the cache. The cache uses a bitmap representation to track visits to up to 16 automaton vertices. The cache is invalidated under the conditions explained in §8.2, by setting `reginfo->poscache_maxiter = 0`.

As discussed in Chapter 9, the Perl regex engine also measures resource usage and short-circuits evaluations that are too costly. Resource usage is measured in `regexec.c`. The measure used by Perl is the “recursion limit”; this terminology is presumably a holdover from when the regex engine used recursion rather than iteration during evaluations. The Perl regex engine’s measure limits the cost of a single path through the automaton simulation, but unlike PHP’s approach it does not limit the overall cost of the simulation. The relevant variable is the `curlyx.count` associated with the automaton’s loop vertices. The interested reader may search `regexec.c` for the error message, “Complex regular subexpression recursion limit exceeded”, to learn more.