Rethinking Regex Engines to Address ReDoS

James C. Davis
Virginia Tech, USA
davisjam@vt.edu

ABSTRACT

Regular expressions (regexes) are a powerful string manipulation tool. Unfortunately, in programming languages like Python, Java, and JavaScript, they are unnecessarily dangerous, implemented with worst-case exponential matching behavior. This high time complexity exposes software services to regular expression denial of service (ReDoS) attacks.

We propose a data-driven redesign of regex engines, to reflect how regexes are used and what they typically look like. We report that about 95% of regexes in popular programming languages can be evaluated in linear time. The regex engine is a fundamental component of a programming language, and any changes risk introducing compatibility problems. We believe a full redesign is therefore impractical, and so we describe how the vast majority of regex matches can be made linear-time with minor, not major, changes to existing algorithms. Our prototype shows that on a kernel of the regex language, we can trade space for time to make regex matches safe.

CCS CONCEPTS

• **Software and its engineering** → *Software libraries and repositories*; • **Security and privacy** → *Denial-of-service attacks.*

KEYWORDS

Regular expressions, empirical software engineering, ReDoS, catastrophic backtracking

ACM Reference Format:
James C. Davis. 2019. Rethinking Regex Engines to Address ReDoS. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3338906.3342509

1 INTRODUCTION

A regular expression (regex) is a way to describe strings that follow a certain pattern. Regexes are supported in most popular programming languages, and are commonly used to solve problems such as input validation and find/replace [20]. Regexes are widely used, appearing in 30-40% of the Python, Java, and JavaScript software analyzed in prior work [3, 7, 18].

Programming language designers and regex engine developers have several regex matching algorithms to choose from, including Thompson's [17] and Spencer's [14]. While the pros and cons of these algorithms can be debated, some are noticeably more suitable than others on some regexes, *e.g.,* the well-known advantage of Thompson over Spencer engines on regexes with high ambiguity [4].

To the best of our knowledge, the regex engines built into many popular programming languages were designed without considering the characteristics of real regexes. In this work we characterize regexes from a security perspective.

- We measure both their vulnerability to regular expression denial of service (ReDoS) attacks and their potential for linear-time evaluation.
- We report that although many regexes evaluate in super-linear time in practice, they could be evaluated in linear time.
- What's more, we show that a linear-time implementation need not require significant changes to a regex engine.

2 BACKGROUND AND RELATED WORK

Regular Expression Algorithms. Regex matching is commonly implemented through automaton simulation. A programming language's *regex engine* converts a regex pattern to a Non-deterministic Finite Automaton (NFA) or Deterministic Finite Automaton (DFA) representation. The regex engine tests for a pattern match by simulating the behavior of the automaton on a candidate string using static DFA simulation [13], Spencer's backtracking NFA simulation [14], or Thompson's NFA-to-DFA simulation [17].

The Spencer algorithm [14] is used in most programming languages, including JavaScript, Java, and Python [8] . The Spencer algorithm relies on a backtracking-based NFA simulation. Each time a Spencer-style matching 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.

ReDoS. The worst-case time complexity of a Spencer-style regex engine exposes applications to a denial of service vector. Spencer-style regex engines exhibit linear, polynomial, or exponential worst-case time complexity [4, 5] due to the "catastrophic backtracking" that can occur while simulating NFAs with high ambiguity [11, 19, 21]. High-complexity regex matches depend on three conditions: (1) the regex engine's algorithm; (2) a regex whose NFA has high ambiguity; and (3) an input that exploits this ambiguity.

This super-linear worst-case regex match behavior can be leveraged in an algorithmic complexity attack [6] known as Regular expression Denial of Service (ReDoS) [5, 12, 16]. In a ReDoS attack, an attacker exploits super-linear (polynomial or exponential) regex match behavior in server-side software to divert resources away from legitimate clients.

Regular expression Denial of Service (ReDoS) is a major problem facing Node.js applications due to their event-driven architecture [9, 10, 15]. Staicu and Pradel [15] identified many Node.js-based websites that have ReDoS vulnerabilities, and Davis *et al.* reported

ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia

ACM ISBN 978-1-4503-5572-8/19/08.
https://doi.org/10.1145/3338906.3342509

thousands of SL regexes occurring in over 10,000 JavaScript modules in the Node.js Package Manager, npm [7].

Language developers consider the Spencer algorithm easier to implement and maintain [14], *e.g.,* to support extended regex features like backreferences and lookaround assertions. Thus there is a tension between language designers' desires and the needs of software engineers who rely on "pathological" regexes in practice.

3 EMPIRICAL MOTIVATION

We analyzed a polyglot regex corpus to understand the worst-case complexity of real regexes in a Spencer-style regex engine. Davis *et al.* [8] recently published a dataset of 537,806 regexes statically extracted from 193,524 popular software modules written in 8 programming languages: JavaScript, Java, PHP, Python, Ruby, Go, Perl, and Rust. *We report that super-linear regex complexity is common, but most regexes can be analyzed in linear time.*

Super-linear complexity is common. We measure a regex's worst-case partial-match complexity in a Spencer-style engine using Weideman *et al.*'s analysis [19]. We report the proportion of regexes that this analysis reports to be super-linear among those it successfully analyzes. In Davis *et al.*'s corpus, the language with the highest proportion of super-linear regexes was Python (38.4%), and Ruby had the lowest with "only" 19.1% super-linear regexes.

Most super-linear complexity is avoidable. 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 [2], and lookaround assertions, which are typically implemented with super-linear complexity. All other commonly-used features can be implemented in linear time [4]. We measure the proportion of regexes that rely on these super-linear features using a Chapman feature vector [3]. Most regexes do not rely on these features; JavaScript had the highest usage (4.3% of JS regexes) and Perl the lowest (2.3%). Thus, *although 19.1-38.4% of regexes have super-linear complexity as implemented in major regex engines, over 95% can be implemented in linear time.*

4 REGEX ENGINE DESIGN

How can we make regexes safe without breaking existing software? On one hand, software developers frequently rely on regexes with super-linear worst case behavior. Clearly it would be nice if the regex engine in their programming language supported these regexes in linear time. On the other hand, programming language designers are naturally hesitant to make changes to fundamental language components like regex engines due to the risk of introducing backwards incompatibilities. It would thus be naive to overhaul an existing regex engine, *e.g.,* to replace a Spencer algorithm with a Thompson one. We are not aware of any research discussing a systematic way to prove that two regex engines are identical, and recent empirical evidence shows that engines in the same specification family have subtle differences in behavior [8].

We propose incorporating a state cache into existing Spencer-based regex engines. This approach has two desirable properties. First, it will offer the same worst-case linear-time complexity as Thompson's algorithm, albeit with larger space complexity. Second, it will not change the match/mismatch behavior of an engine, merely the time and space required to reach the conclusion.

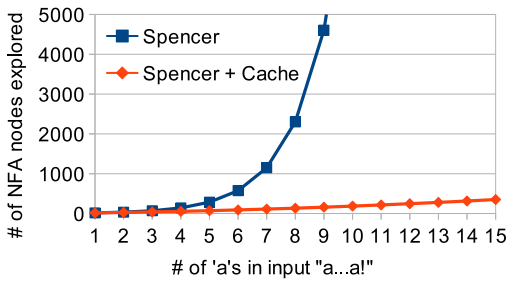

Figure 1: Cache prevents redundant state exploration.

To see why the time complexity becomes linear, consider a regex whose ϵ-free NFA has m states[1]. An NFA simulation consists of attempting to move from the NFA start state to the NFA accept state via the transitions in the NFA, consuming one of the n characters in the input at each transition and backtracking when multiple transitions are available. The number of NFA states visited during the simulation indicates the cost of determining the regex match — if the engine backtracks exponentially many times, then exponentially many states are visited. However, when a given <NFA state, input offset> is visited more than once, this visit is clearly redundant. If a regex engine maintains a cache of the <NFA state, input offset> pairs it visits (space complexity $O(m \times n)$), then upon filling the cache it can declare a mismatch. When considering a single <NFA state, input offset> pair, the regex engine must consider at most m transitions, for a time complexity of $O(m \times (m \times n)) = O(m^2 \times n)$. In the corpus discussed in §3, the NFA size is typically small, so the space and time costs are both approximately $O(n)$.

We have not yet considered how to support regexes with super-linear features like backreferences and lookaround assertions, but the use of these features is so rare that the common path of a regex engine need not be optimized for them.

5 RESULTS

We prototyped a cache in a simple backtracking regex engine implementation published by Cox [1]. Though the engine is simple, it is sufficient for us to demonstrate the potential of a cache. Given the regex `/(a+)*/`, Figure 1 plots the number of NFA states visited for increasing length of input "`a.....a!`" with and without a state cache. While the original Spencer algorithm blows up quickly, the engine with cache does not explore redundant states, achieving linear-time rather than exponential performance in the worst case.

6 CONCLUSIONS

We have showed that although an appreciable proportion of real regexes have worst-case super-linear behavior, it is possible to support them in linear time at a modest space cost without making significant changes to a regex engine. As future work, we will extend our analysis to handle more complex regex features and then incorporate our approach into a real regex engine.

[1] Existing regex engines construct this NFA already, so there is no need to consider the complexity of the construction itself.

REFERENCES

[1] [n. d.]. re1: A simple regular expression engine, easy to read and study. https://code.google.com/archive/p/re1.

[2] Alfred V Aho. 1990. *Algorithms for finding patterns in strings.* Elsevier, Chapter 5, 255–300.

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

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

[5] Scott Crosby. 2003. Denial of service through regular expressions. *USENIX Security work in progress report* (2003).

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

[7] James C Davis, Christy A Coghlan, Francisco Servant, and Dongyoon Lee. 2018. 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)*.

[8] James C Davis, Michael Michael, Christy A Coghlan, Francisco Servant, and Dongyoon Lee. 2019. Are 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)*.

[9] James C Davis, Eric R Williamson, and Dongyoon Lee. 2018. 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)*.

[10] A Ojamaa and K Duuna. 2012. Assessing the security of Node.js platform. In *7th International Conference for Internet Technology and Secured Transactions (ICITST)*.

[11] Asiri Rathnayake and Hayo Thielecke. 2014. *Static Analysis for Regular Expression Exponential Runtime via Substructural Logics.* Technical Report.

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

[13] Michael Sipser. 2006. *Introduction to the Theory of Computation.* Vol. 2. Thomson Course Technology Boston.

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

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

[16] Bryan Sullivan. 2010. New Tool: SDL Regex Fuzzer. https://blogs.microsoft.com/microsoftsecure/2010/10/12/new-tool-sdl-regex-fuzzer/

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

[18] Peipei Wang and Kathryn T Stolee. 2018. How well are regular expressions tested in the wild?. In *Foundations of Software Engineering (FSE)*.

[19] Nicolaas Hendrik Weideman. 2017. *Static Analysis of Regular Expressions.* Ph.D. Dissertation. Stellenbosch University.

[20] Wikipedia contributors. 2018. Regular expression — Wikipedia, The Free Encyclopedia. https://web.archive.org/web/20180920152821/https://en.wikipedia.org/w/index.php?title=Regular_expression.

[21] Valentin Wustholz, Oswaldo Olivo, Marijn J H Heule, and Isil Dillig. 2017. 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)*.