ESEC/FSE: G: On the Impact and Defeat of Regex DoS

James C. Davis (Advisor: Dongyoon Lee)
Virginia Tech, USA
davisjam@vt.edu

1 Problem and Motivation

From their obscure origins in neuron modeling [26], regular expressions (regexes) have emerged as a widely used string manipulation tool. Regexes are commonly used to bring order to unstructured text, *e.g.,* by web services to sanitize untrusted input. Unfortunately, regexes are risky in most mainstream programming languages: most regex engines have worst-case exponential matching behavior. This worst-case property has been known for decades [2, 39], and has been proposed as a vector for an algorithmic complexity attack [13]. Indeed, several major services have had outages caused by this behavior, including Stack Overflow [19] and Cloudflare [24]. But it is not clear whether these slow regexes are common enough to comprise a serious security concern. Any change to fundamental aspects of our programming languages requires a firm foundation, not anecdotal evidence. Furthermore, if Regex Denial of Service (ReDoS) is a common threat in practice, how ought it be defeated?

We have conducted empirical studies to understand the potential impact of ReDoS in practice. ReDoS is a real-world problem; up to 10% of regexes may comprise ReDoS vectors. We have performed the first evaluations of the effectiveness of existing ReDoS defenses; we found them wanting in completeness, utility, or performance. We propose a novel defense based on selective memoization. Our approach offers provable security guarantees for 95% of regexes, and constant space costs for typical regexes.

2 Background and Related Work

This section explains why regexes have super-linear worst-case behavior in many programming languages, how this property can be exploited for Regex Denial of Service (ReDoS) attacks, the limitations of empirical regex research to date, and existing ReDoS defenses.

Regexes and regex engines Regexes are a widely-used technique for solving string matching problems [21]. Although regexes may be used in low-risk, ephemeral ways, *e.g.,* during program comprehension [38], regexes are also used in business-critical contexts. Specifically, they are commonly used in web servers to validate untrusted input. They act as a defensive filter to prevent illegitimate data from poisoning a program. But are regexes themselves a potential vector for abuse? *Quis custodiet ipsos custodes?*

A regex is a notion and a notation for concisely describing a set of strings that share a property. We assume readers are generally familiar with the regex notation popularized by Perl, with core operators of concatenation (·), disjunction (|), and repetition (*), and syntax sugar for other operations (*e.g.,* character classes, one-or-more repetition, etc.). For example, the set of valid email addresses might be described as `/\S+@\S+\.\S+/`.[1]

Most programming languages support regexes. Under the hood, programming languages implement a *regex engine* to compile and evaluate regexes. These regex engines convert a regex to a nondeterministic finite automaton (NFA) [33] using an algorithm similar to those of Thompson [41] and Glushkov [23]. These constructions are easiest to understand through a botttom-up approach, building a more sophisticated automaton from the constituent parts depicted in the first four figures in Figure 1.

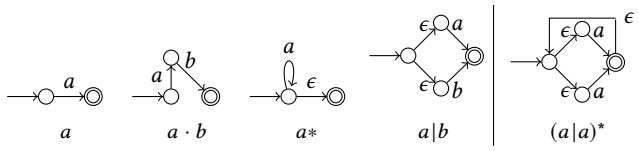

Figure 1: NFAs for the fundamental regex operations.

After constructing the NFA for a regex, a regex engine resolves a match query by simulating the automaton. Most regex engines use *backtracking* [6] to resolve any non-determinism. Unfortunately, a regex match using these backtracking-based algorithms has exponential worst-case time complexity [34, 45, 46]. The final example in Figure 1 illustrates the NFA for a regex with such behavior. Each time the regex engine encounters an 'a', it advances along one path and saves the other in a backtracking stack for later consideration if the first path does not end in a match. On the input $a^n b$, *i.e., n* a's and a b, there are 2^n paths through this automaton. The regex engine will explore all of these paths before concluding a mismatch.

The worst-case time complexity of a backtracking NFA simulation depends on the *degree of ambiguity* [4] in the regex's structure. Some regexes have exponential behavior, others polynomial, and others linear in the input length. Several researchers have proposed algorithms to identify *super-linear* (SL) regexes by finding inputs for which there are many distinct accepting paths through an NFA [45, 46] (*i.e.,* the NFA is highly ambiguous).

ReDoS Crosby and Wallach were the first to observe that SL regex match behavior could be exploited by attackers as a special case of an algorithmic complexity attack [12, 13]. An attacker can submit malicious input and trigger ReDoS if three *ReDoS conditions* are met: (1) A web server performs a regex match on unsanitized client input; (2) The regex has super-linear worst-case time complexity; and (3) The web server enforces no safeguards to cap a client's regex resource usage.

Condition (1) is commonly met, since one of the primary use cases of regexes is to perform sanitization. Condition (3) is commonly met as well, as most regex engines do not offer resource caps and modern web frameworks do not offer higher-level resource cap primitives [17]. However, we know little about Condition (2). Anecdotally, high-profile ReDoS incidents include outages at Stack Overflow [19] and Cloudflare [24]. Staicu & Pradel further showed that a few SL regexes in popular JavaScript modules could be leveraged to achieve ReDoS against thousands of popular websites [40]. But are SL regexes common enough to rethink regex engines?

[1] This regex is not RFC-compliant, but it is used in major software projects.

Empirical studies of regexes Regexes have received little empirical attention. Stolee's research team has found that regexes are used in up to 40% of Python projects [9], that synonymous regexes have differing comprehensibility [10], that regexes are under-tested [44], and that regexes tend to evolve towards matching a wider set of strings [43]. But we lack large-scale empirical studies to characterize software engineering practices *surrounding ReDoS*. Until then, we cannot tell whether ReDoS is a serious threat.

ReDoS defenses ReDoS defenses can be erected at the application and regex engine levels. At the application level, software engineers might replace an SL regex with a linear-time equivalent. At the regex engine level, several defenses against ReDoS have been deployed in production regex engines. The regex engines of PHP, Perl, and the .NET framework deliver exceptions if a regex match uses too many resources.[2] Perl and RE2 use memoization to decrease the worst-case time complexity of a match, although Cox showed that the Perl scheme is incomplete [11]. Rust and Go both employ Thompson's lockstep NFA simulation [41], which lowers the match complexity to linear in the length of the input string [11, 41].[3]

3 Approach and Uniqueness

Unknowns The literature is silent on two ReDoS questions: (1) How widespread is the use of SL regexes in practice (§3.1)?, and (2) How shall we defend against ReDoS (§3.2)?

3.1 Super-linear regexes in practice

The incidence of SL regexes (ReDoS Condition 2) In our first study, we considered SL regex use in two software ecosystems [14]. We considered major programming languages, measuring both JavaScript and Python for comparitive purposes. We studied the software modules published in each language's primary module registry, for two reasons. First, it permits a relatively fair cross-language comparison, as modules fill similar ecological niches, *e.g.,* logging or schema validation. Second, modules are published, maintained, and used by a mix of open-source and commercial software developers; their security vulnerabilities have a ripple effect.

We mapped software modules from their registry to GitHub, and cloned as many as possible. We used static analysis to identify regex-creating callsites and extract regexes from them.

To identify SL regexes, we used an ensemble of three SL regex detectors [34, 45, 46]. These detectors predict the worst-case behavior of a regex by modeling the underlying regex engine. Since production regex engines may deviate from this model [3, 8], we estimated each SL regex's worst-case time complexity by fitting curves to its match time when sampled across several input lengths.

Testing generalizability The previous study had two principal threats to generalizability. First, it is unclear whether the regexes obtained through that regex extraction methodology — statically finding regexes hard-coded into applications — are similar to those defined dynamically. Second, it is unclear whether findings from two programming languages will generalize to others (*e.g.,* from "scripting" languages to "systems" languages).

In our second study [16], we attempted to replicate our results across alternative regex extraction methodologies and programming languages. To compare extraction methodologies, we extracted regexes from software modules both statically and dynamically. For static regex extraction, we used the same extraction methodology as before. For dynamic regex extraction, we instrumented regex-creating callsites and then ran the project's test suite.

To determine whether regex characteristics have different properties by programming language, we compared regexes extracted from the 25,000 "most important" software modules written in many "major" programming languages. We measured importance using GitHub stars, shown to be a reasonable proxy [7]. We operationalized language prominence using two conditions: (1) The language has a large module ecosystem; and (2) The language is widely used by the open-source community. We consulted the ModuleCounts website [18] and the GitHub language popularity report [22] to identify the top five programming languages under these rules. We also included Perl, Go, and Rust for scientific interest: Perl popularized the idea of regexes as a first-class language member, and Go and Rust are relatively new mainstream languages.

We used a subset of these modules — those from JavaScript, Python, and Java — to test the effect of regex extraction methodology. We found that the extraction method had no effect. To simplify our experimental methodology, we then tested the effect of programming language using only statically extracted regexes from software written in these eight programming languages.

For all of these tests, we computed eight regex metrics across three dimensions. We used both existing metrics from the literature [9, 27, 44, 46], and new metrics designed to inform regex engine development. We built our measurement instruments on Microsoft's Automata library [31]. We compared each metric's distributions across the relevant subsets of the resulting regex corpora.

3.2 Existing ReDoS defenses

We considered the effectiveness of existing ReDoS defenses at the application and regex engine levels.

Refactoring applications (ReDoS Condition 2) Some regex engines leave the responsibility of avoiding ReDoS in the hands of application developers. To avoid ReDoS, application developers can refactor SL regexes into a linear-time alternative. No studies have examined ReDoS refactoring strategies, nor whether application developers can reasonably be expected to bear this responsibility.

As part of [14], we first identified typical ReDoS repair strategies, and then observed software engineers as they effected new repairs. To identify typical strategies, we characterized the repair strategies followed by engineers in previously-reported ReDoS vulnerabilities (CVE database). We then disclosed ReDoS vulnerabilities to the maintainers of 284 modules. In our disclosures we described the vulnerability in their software and examples of each repair strategy we identified. We then observed the maintainers' repair strategies.

Replacing regex engines (ReDoS Condition 2) Another path to avoid ReDoS is to replace a slow matching algorithm with a faster one, either via significant engine refactoring or by substituting one regex engine with another. The risk of this defense is the *portability problems* that may result. Any regression in regex behavior would entail substantial application-level refactoring.

[2] This addresses ReDoS Condition 1.
[3] This addresses ReDoS Condition 3.

In our third study [15], we explored the viability of this approach. To operationalize the concept, we measured the extent of the decrease in worst-case match times in eight regex engines, and the extent to which regexes are syntactically and semantically equivalent in these engines. These regex engines use a variety of algorithms and optimizations, but all adhere to the Perl-Compatible Regular Expressions (PCRE) standard [25]. Thus, this experiment simultaneously identifies the risks and benefits of changing regex engines (an application level defense) and estimates the risks of regressions after a major refactoring (a regex engine level defense).

We used a differential testing approach [28] in this experiment. We compared the behavior of a set of regexes on all pairs of regex engines. We used the polyglot regex corpus described in the *Generalizability* study [16], and generated inputs using an ensemble of regex input generators [5, 27, 32, 37, 42].

Resource caps (ReDoS Condition 3) Three production-grade regex engines enforce some notion of "maximum allowed resource usage" for each query. Perl and PHP measure resource usage by progress through the search algorithm. The .NET regex engine can be configured to measure resource usage by wall-clock time. These engines deliver exceptions if the usage threshold is exceeded.

In this experiment, we measured the effectiveness and utility of these solutions.[4] For effectiveness, we determined whether each of these schemes delivers an exception when problematically long-running regex match queries are issued. For utility, we determined the extent to which practitioners adopt the optional .NET defense.

3.3 A new ReDoS defense

Memoization [30][5] has been proposed as a ReDoS defense [20, 36]. Memoization reduces match time complexity from exponential to linear in the input string, but comes at the cost of inflating the typical space cost from constant to linear in the input string. The *time savings* occurs because many paths in an SL NFA simulation are redundant. For example, in the exponential simulation of the final NFA from Figure 1, each pair of non-deterministic choices has the same effect, and so only one need be explored. The *space cost* occurs because recording an NFA simulation using typical memoization data structures (bitmap, hash table) requires $O(|Q| \times |w|)$ space for $|Q|$ automaton states and an input string of length $|w|$. Two production-grade regex engines incorporate some memoization to reduce the time complexity of a regex match, but both implementations truncate the memoization table to the detriment of their time complexity.

In this part of the work, we propose space-saving memoization schemes that permit linear-time matches with lower, constant-to-linear, space complexity.[6] We prove the properties of two novel *selective memoization* [1] schemes. We also propose a novel use of run-length encoding (RLE) [35] as a memoization data structure. Intuitively, RLE takes advantage of regex engines' ordered search regime — PCRE-compatible regex engines guarantee a leftmost-greedy search semantic, which yields a low-entropy memo table at all points of the simulation. We prototype these schemes and evaluate the performance of all nine conditions (3 memoization schemes × 3 data structures) on our corpus of SL regexes.

[4]This work is not yet published.
[5]The idea of memoization is to record the result of prior calculations.
[6]This work is not yet published.

Table 1: Results from first study [14].

Registry	Scanned Modules	Unique Regexes	SL Regexes
npm (JS)	375,652 (66%)	349,852	3,589 (1%)
pypi (Python)	72,750 (58%)	63,352	704 (1%)

Table 2: Polyglot regex corpus.

Lang. (Registry)	# mod. anal.	Unique regexes
JavaScript (npm)	24,997	150,922
Java (Maven)	24,986	19,332
PHP (Packagist)	24,995	44,237
Python (pypi)	24,997	43,896
Ruby (RubyGems)	24,999	153,334
Go (Gopm)	24,997	22,105
Perl (CPAN)	31,827 (all)	142,777
Rust (Crates.io)	11,724 (all)	2,025

4 Results and Contributions

4.1 Super-linear regexes in practice

The incidence of SL regexes In this experiment, we analyzed around 450,000 software modules, extracted around 400,000 unique regexes from them, and found that 1% of these regexes exhibited SL worst-case behavior (Table 1). Applying curve fitting to the SL regex match times, we found that 74% of the regexes had worst-case quadratic behavior in their respective regex engines, with the remainder either exponential or a higher polynomial. These results were consistent between JavaScript and Python modules.

Testing generalizability To test whether regexes are similar when declared statically or obtained dynamically, we first examined 75,000 modules from JavaScript, Python, and Java. Following the two extraction techniques in the literature, we obtained regexes both statically and dynamically. We measured these regexes along our eight metrics. We found *no significant difference* between the distributions for any metric.

To test whether engineers write similar regexes across programming languages, we then analyzed around 200,000 software modules and statically extracted around 500,000 unique regexes from them (Table 2). The first five languages in the table are the "major" languages according to our methodology, and the others were included for scientific interest. After measuring these regexes, 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. Figure 2 shows the results for regex length — the distributions of Java, PHP, Python, and Ruby were indistinguishable using our statistical tests, while Perl regexes tend to be shorter than those of Go and Rust with moderate effect sizes.

4.2 ReDoS defenses

Refactoring applications We identified 37 historical ReDoS reports. Three fix strategies were typical in these reports: (1) *Trim*, *i.e.,* limit the size of input to the regex; (2) *Revise*, *i.e.,* change the regex; and (3) *Replace*, *e.g.,* use custom parsing logic. We then sent reports to 284 software module maintainers, yielding 48 fixes. The fix strategies are summarized in Table 3.

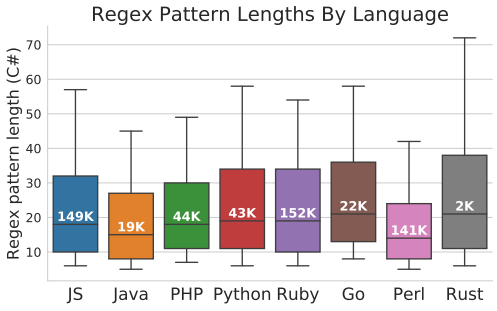

Figure 2: Regex lengths by programming language. Whiskers indicate the (10,90)th percentiles. Reported in [16].

Table 3: Fix approaches for SL regexes (from [14]).

		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

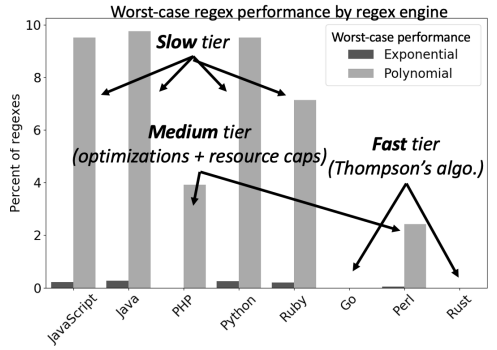

Figure 3: Proportion of SL regexes in eight prog. langs [15].

We note two findings from this experiment. First, Table 3 suggests that developers prefer to *Revise* when they consider all alternatives. Second, there were unsafe fixes in the historical data, and in the new data many initial fixes were unsafe and needed correction. ReDoS repair appears to be difficult; researchers should explore automatic refactoring techniques.

Replacing regex engines We found that strong safety benefits can be obtained by substituting one regex engine for another. The proportion of the polyglot regex corpus (Table 2) that exhibits SL behavior in each programming language is shown in Figure 3.[7] Clearly there are safety benefits to be gained by moving from one regex engine to another. As indicated in the annotation of Figure 3, engines can be divided into three performance tiers based on their optimizations, resource caps (discussed later), and algorithms. The performance improvement is monotonic; regexes never run more slowly when moved from a slower tier to a faster one.

However, we also found that software engineers should be careful when substituting one regex engine for another (or performing major regex engine refactoring). Most regexes were syntactically valid in all regex engines, *e.g.,* 88% were valid in all but Rust. To compare semantics, we generated a median of 2,410 distinct inputs for each regex, and found that 15% of regexes exhibited some semantic difference between some pair of regex engines. Figure 4 shows the extent of pairwise differences.

[7] In this experiment we improved the SL regex detector ensemble. Our new techniques identified an *order of magnitude* more SL regexes than were found in Table 1, suggesting that ReDoS may be a more serious problem than is implied by Table 1.

Table 4: The effectiveness of resource cap approaches.

	Lin.-time match	Capped (defended)	Timed out (vuln.)
Perl	73%	1%	26%
PHP	27%	35%	38%
.NET (C#)	0%	100%	0%

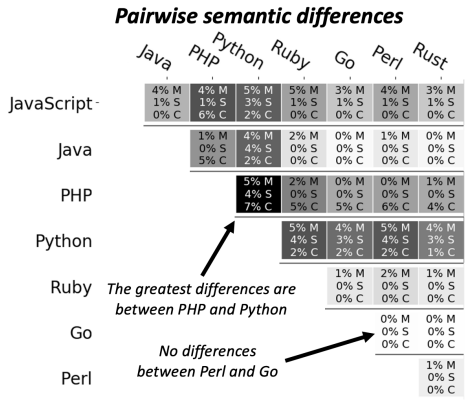

Figure 4: Potential semantic portability problems (from [15]). Darker cells indicate greater difference, with 1% ≈ 5,000 regexes.

Resource caps We found that existing time-based resource caps are more effective than existing progress-based caps. We tested the behavior of the three capped regex engines on the 51,224 SL regexes identified in Figure 3. Table 4 shows the relative effectiveness of each. For the progress-based caps, Perl's optimizations are stronger than PHP's, but PHP's measure of progress is more effective than Perl's. While both Perl and PHP permit a large proportion of SL regex behavior, the time-based cap of .NET was perfectly effective.

We also found that the utility (adoption rate) of *optional* resource caps is low. The Perl and PHP caps are on by default, and thus have near-perfect adoption. The .NET cap is off by default. Although the .NET documentation recommends its use, their pleas fall on deaf ears. Following our earlier module mining methodology, we cloned the 35,194 C# modules hosted on GitHub and found that 2,812 modules used regexes. Among these, only 5% of the modules used a timeout, and only 1% of SL regex use was protected by a timeout. We hope these findings guide the further improvement and adoption of resource cap-based defenses.

4.3 A new ReDoS defense

Although some argue that memoization in regex engines costs too much space [11], our results suggest that this conclusion is premature. To guide the use of memoization, we proved theorems describing the time complexity for two novel selective memoization schemes. These schemes involve tracking the visits to a subset Φ of the NFA states Q. Their time complexity depends on $|Q|$, $|\Phi|$, and the length of w, the input string.

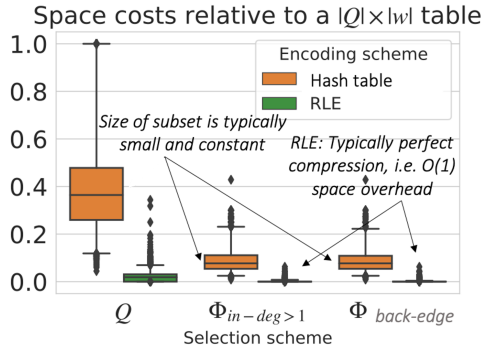

Figure 5: Space costs of memoization schemes, relative to a $|Q|\times|w|$ bitmap. Whiskers are the (1,99)$^{\text{th}}$ percentiles.

Theorem 4.1 *Memoizing only visits to vertices with in-degree* > 1 *yields time complexity of* $O(|\Phi_{in-deg>1}| * |w|)$.

Theorem 4.2 *Memoizing vertices that are the destinations of back-edges yields time complexity of* $O(f(Q) * |\Phi_{back-edge}| * |Q| * |w|)$.

(Space constraints prevent us from including the proofs).

Although the time complexity of the second scheme is somewhat larger, more critical is that both complexities are linear in $|w|$. Their space complexity is $O(|\Phi| \times |w|)$ for input string w.

We prototyped these selection schemes and the three data structures for the memoization table on a simple regex engine provided by Cox. With several extensions, this engine can perform regex matches for about 30% of the polyglot regex corpus. On the subset of these regexes that are SL in a backtracking regex engine, the memoization schemes exhibited (the predicted) linear *time* complexity. For this subset, the *space* costs for the experimental conditions are shown in Figure 5. We note first that, as the space cost of a hash table is a substantial fraction of $|\Phi| \times |w|$, so the orange bars show that *most NFA vertices are not in either of the selected vertex subsets* Φ. Second, the RLE representation (green bars) offers significant space benefits for the vast majority of regexes. Our prototype achieved linear-time matches at *effectively constant space cost.*

Our theorems only apply to "truly regular" regexes, limiting the supported regex features. However, only 5% of regexes use irregular features like backreferences, so our theorems apply to 95% of regexes. Thus, the combination of selective memoization and RLE appears promising as a transparent ReDoS solution.

4.4 Research impact

My research on regex engineering practices and their security implications has led to **four first-author publications** [14–17] and **one co-authored publication** [29] in top-tier software engineering and security venues. Two of these works won **ACM SIGSOFT Distinguished Paper** awards [14, 29]. Among other accomplishments, these works identified security flaws in major software projects (*e.g.,* core libraries of Python and Node.js) [14]; identified errors in the regex engines of Python, Google's JavaScript-V8, Ruby, and Rust [15]; and replicated, validated, and improved upon regex research methodology [16]. My contributions to regex safety have been acknowledged by Microsoft's security team.

References

[1] Umut A. Acar, Guy E. Blelloch, and Robert Harper. 2003. Selective memoization. In *Principles of Programming Languages (POPL).*

[2] Alfred V Aho. 1980. Pattern matching in strings. In *Formal Language Theory.*

[3] Alfred V. Aho and Margaret J. Corasick. 1975. Efficient string matching: an aid to bibliographic search. *CACM* 18, 6 (1975), 333–340.

[4] Cyril Allauzen, Mehryar Mohri, and Ashish Rastogi. 2008. General Algorithms for Testing the Ambiguity of Finite Automata. In *ICDLT.*

[5] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. 2017. MutRex: A Mutation-Based Generator of Fault Detecting Strings for Regexes. In *ICSTW.*

[6] Alex Birman and Jeff Ullman. 1970. Parsing Algos. With Backtrack. In *SWAT.*

[7] Hudson Borges and Marco Tulio Valente. 2018. What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform. In *JSS.*

[8] R. Boyer and J. Moore. 1977. A fast string searching algorithm. In *CACM.*

[9] Carl Chapman and Kathryn T Stolee. 2016. Exploring regular expression usage and context in Python. In *ISSTA.*

[10] C. Chapman, P. Wang, and K. Stolee. 2017. Exploring Regular Expression Comprehension. In *Automated Software Engineering (ASE).*

[11] Russ Cox. 2007. Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, PHP, Python, Ruby, ...).

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

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

[14] J. Davis, C. Coghlan, F. Servant, and D. Lee. 2018. The Impact of Regular Expression Denial of Service (ReDoS) in Practice. In *ESEC/FSE.*

[15] J. Davis, L. Michael IV, C. Coghlan, F. Servant, and D. Lee. 2019. Why aren't regular expressions a lingua franca?. In *ESEC/FSE.*

[16] J. Davis, D. Moyer, A. Kazerouni, and D. Lee. 2019. Testing Regex Generalizability And Its Implications: A Large-Scale Many-Language Measurement Study. In *ASE.*

[17] J. Davis, E. Williamson, and D. Lee. 2018. A Sense of Time for JavaScript and Node.js: First-Class Timeouts as a Cure for EHP. In *USENIX Security.*

[18] Erik DeBill. [n. d.]. Module Counts. http://www.modulecounts.com/.

[19] Stack Exchange. 2016. Outage Postmortem. http://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016.

[20] Bryan Ford. 2002. Packrat Parsing: Simple, Powerful, Lazy, Linear Time. In *ICFP.*

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

[22] GitHub. 2018. The State of the Octoverse. https://octoverse.github.com/.

[23] V M Glushkov. 1961. The Abstract Theory of Automata. *Russian Mathematical Surveys* 16, 5 (1961), 1–53.

[24] Graham-Cumming, John. 2019. Details of the Cloudflare outage on July 2, 2019. https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/.

[25] Philip Hazel. 2018. PCRE2 - Perl Compatible Regular Expressions, 2ed.

[26] S. C. Kleene. 1951. Representation of events in nerve nets and finite automata. *Automata Studies* (1951), 3–41.

[27] Eric Larson and Anna Kirk. 2016. Generating Evil Test Strings for Regular Expressions. In *ICST.*

[28] William M Mckeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998).

[29] L. Michael IV, J. Donohue, J. Davis, D. Lee, and F. Servant. 2019. Regexes are Hard: Decision-making, Difficulties, and Risks in Programming Regexes. In *ASE.*

[30] Donald Michie. 1968. "Memo" Functions and Machine Learning. *Nature* (1968).

[31] Microsoft. [n. d.]. Automata and transducer library for .NET.

[32] Anders Møller. 2010. dk. brics. automaton.

[33] M. Rabin and D. Scott. 1959. Finite Automata and their Decision Problems. *IBM Journal of Research and Development* 3 (1959), 114–125.

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

[35] A. H. Robinson and Colin Cherry. 1967. Results of a Prototype Television Bandwidth Compression Scheme. *Proc. IEEE* 55, 3 (1967), 356–364.

[36] Niko Schwarz, Aaron Karper, and Oscar Nierstrasz. 2015. Efficiently extracting full parse trees using regexes with capture groups. *PeerJ Preprints* (2015).

[37] Yuju Shen, Yanyan Jiang, Chang Xu, Ping Yu, Xiaoxing Ma, and Jian Lu. 2018. ReScue: Crafting Regular Expression DoS Attacks. In *ASE.*

[38] Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. 1997. An examination of software engineering work practices. In *CASCON.*

[39] Henry Spencer. 1994. A regular-expression matcher. In *Software solutions in C.*

[40] C. Staicu and M. Pradel. 2018. Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers. In *USENIX Security.*

[41] Ken Thompson. 1968. Regular Expression Search Algorithm. *CACM.*

[42] Margus Veanes, Peli De Halleux, and Nikolai Tillmann. 2010. Rex: Symbolic regular expression explorer. *ICST* (2010).

[43] P. Wang, G. Bai, and K. Stolee. 2019. Exploring Regular Expression Evolution. In *SANER.*

[44] Peipei Wang and Kathryn T Stolee. 2018. How well are regular expressions tested in the wild?. In *ESEC/FSE.*

[45] N. Weideman, B. van der Merwe, M. Berglund, and B. Watson. 2016. Analyzing matching time behavior of backtracking regular expression matchers by using ambiguity of NFA. In *LNCS.*

[46] Valentin Wüstholz, Oswaldo Olivo, Marijn J H Heule, and Isil Dillig. 2017. Static Detection of DoS Vulnerabilities in Programs that use Regexes. In *TACAS.*