Using Selective Memoization to Defeat Regular Expression Denial of Service (ReDoS)

James C. Davis[*]
Electrical & Computer Engineering
Purdue University
davisjam@purdue.edu

Francisco Servant
Computer Science
Virginia Tech
fservant@vt.edu

Dongyoon Lee[*]
Computer Science
Stony Brook University
dongyoon@cs.stonybrook.edu

***Abstract*—Regular expressions (regexes) are a denial of service vector in most mainstream programming languages. Recent empirical work has demonstrated that up to 10% of regexes have super-linear worst-case behavior in typical regex engines. It is therefore not surprising that many web services are reportedly vulnerable to regex denial of service (ReDoS).**

If the time complexity of a regex engine can be reduced transparently, ReDoS vulnerabilities can be eliminated at no cost to application developers. Unfortunately, existing ReDoS defenses — replacing the regex engine, optimizing it, or replacing regexes piecemeal — struggle with soundness and compatibility. Full memoization is sound and compatible, but its space costs are too high. No effective ReDoS defense has been adopted in practice.

We present techniques to provably eliminate super-linear regex behavior with low space costs for typical regexes. We propose *selective memoization* schemes with varying space/time tradeoffs. We then describe an *encoding scheme* that leverages insights about regex engine semantics to further reduce the space cost of memoization. We also consider how to safely handle *extended regex features*. We implemented our proposals and evaluated them on a corpus of real-world regexes. We found that selective memoization lowers the space cost of memoization by an order of magnitude for the median regex, and that run-length encoding further lowers the space cost to *constant* for 90% of regexes.

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

***Index Terms*—Regular expressions, denial of service, ReDoS, algorithmic complexity attacks, memoization, legacy systems**

I. Introduction

Regular expressions (regexes) are a fundamental building block of computing systems [1]. It is unfortunate that such a widely used tool is a denial of service vector. For the sake of expressiveness and flexibility [2], most regex engines follow a backtracking framework with worst-case super-linear behavior in the length of the input string w (*e.g.,* $\mathcal{O}(|w|^2)$ or $\mathcal{O}(2^{|w|})$) [3]. Meanwhile, 30-40% of software projects use regexes to solve string matching problems [4], [5], and up to 10% of those regexes exhibit super-linear worst-case behavior [6], [7]. These trends expose regex-reliant services to an algorithmic complexity attack [8] known as Regex-based Denial of Service (ReDoS) [9]–[11].

The threat of ReDoS is well understood. Empiricists have demonstrated that software engineers commonly compose ReDoS-vulnerable regexes [5], [6] and that thousands of web services are exploitable [12]. For example, root cause analysis implicated super-linear regex evaluation in outages at Stack Overflow [13] and Cloudflare [14]. We know the risks — now we need a ReDoS defense.

ReDoS attacks require three ingredients: a slow (*i.e., backtracking*) regex engine, a slow regex, and reachability by user input. Assuming reachability, ReDoS defenses thus speed up the regex engine or change the regex. A proper ReDoS defense should be (1) *sound* (works for all regexes), (2) *backwards-compatible* (regex engines are "legacy systems" whose stability is critical), and (3) *low-cost*. Existing ReDoS defenses suffer from unsoundness or incompatibility. For example, moving regexes to a faster regex engine [3], [15] risks semantic differences [7], while refactoring slow regexes is an error-prone and piecemeal solution [6].

In light of these design goals, we propose memoization to speed up backtracking regex engines, addressing ReDoS *soundly* in a *backwards-compatible* manner with *small runtime costs* for typical regexes. For soundness, we prove theorems guaranteeing worst-case time complexity that is linear in $|w|$. For compatibility, our approach can be introduced within existing backtracking frameworks, just like other common regex optimizations (*e.g.,* [16]–[18]). We employ two techniques to make our approach low-cost. Selective memoization asymptotically reduces the size of the memo table. An efficient data representation lets us compress the memo table.

We measured the practicality of our approach on the largest available corpus of super-linear regexes [7]. Our prototype achieves linear-in-$|w|$ time costs for all of these regexes. Through selective memoization, the median storage cost (copies of $|w|$) for a super-linear regex decreases by an order of magnitude compared to a standard memo table. Adding an efficient representation, that space cost falls to *constant* for 90% of super-linear regexes.

Our contributions are:

- We propose a novel memoized NFA to enable the analysis of memoization schemes (§VI).
- We present two *selective memoization schemes* that reduce the space complexity of memoization (§VII).
- To further reduce space costs, we compress the memo table with a space-efficient *memo function representation* (§VIII).
- We extend these techniques from regular expressions to

[*]Some work performed while at Virginia Tech.

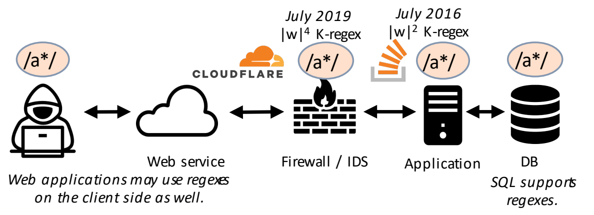


Fig. 1: Regexes across the system stack. ReDoS may occur when a slow regex meets unsanitized input on a slow regex engine. We discuss ReDoS outages at Cloudflare and Stack Overflow (§II).

two commonly-used extended regex features: zero-width assertions and backreferences (§IX).

- We evaluate our proposals on a large-scale corpus and report substantial benefits for a wide range of regexes (§X).

We have demonstrated techniques by which ReDoS can be defeated — soundly, for all commonly-used regex features, and with minimal changes to legacy regex engines.

Outline: We begin with background material on ReDoS and regexes. Next we show the limitations of existing ReDoS defenses. Then we present our approach, evaluate, and conclude.

II. Regex Denial of Service (ReDoS)

Regexes are used in latency-sensitive contexts on the critical path. They validate untrusted input throughout the system stack (Figure 1), *e.g.,* to process HTML headers [12], [19] or detect cross-site scripting (XSS) attacks [20], [21]. Ironically, regexes used as a defensive filter may themselves be exploited.

A. ReDoS attacks

Crosby and Wallach observed that super-linear (*i.e.,* polynomial or exponential in $|w|$) regex engine behavior could be exploited in an algorithmic complexity attack [8], [9]. ReDoS attacks require three *ReDoS Conditions*:

1) The victim uses a regex engine with super-linear matching behavior in $|w|$ (*i.e.,* a backtracking implementation).[1]
2) The victim uses a super-linear regex (§III-B).
3) The regex is reachable by untrusted input.

If these conditions are met, then an attacker can submit input designed to trigger the super-linear regex behavior. The costly regex evaluation will divert computational resources (*e.g.,* CPUs, threads) and reduce or deny service to legitimate clients. Such attacks are applicable to most web services; Davis *et al.* have demonstrated super-linear regex behavior in every major programming language but Rust and Go [7]. ReDoS exploits are particularly problematic for services that use multiplexed architectures like Node.js [12], [23], [24].

B. Threat model

We suppose a realistic threat model: the attacker can specify the string w to which the victim's regex is applied (ReDoS Condition 3). This is in keeping with a primary use of regexes, namely to sanitize and process untrusted input [4], [25], [26].

[1]Some authors restrict ReDoS to exponential in w [22]. However, real-world outages have involved polynomial worst-case behavior [13], [14].

C. ReDoS in the wild: Two case studies

Thousands of ReDoS vulnerabilities have recently been identified [6], [12]. We illustrate these vulnerabilities through two case studies. These studies show the diverse usage of regexes and the implications of super-linear behavior.

The Q&A forum **Stack Overflow** had a 34-minute ReDoS outage in July 2016 [13]. They used the quadratic regex `.*\s+` (simplified) to trim trailing whitespace from each post as part of response generation, to improve rendering and reduce network traffic. A post with 20,507 tab characters reached the front page, triggering the worst-case quadratic behavior of the regex on page load. Stack Overflow's load balancer interpreted slow page load times as instability and marked their servers as offline. The resulting capacity loss brought down their service.

ReDoS outages also occur due to dependencies on other services. The web infrastructure company **Cloudflare** had a 27-minute ReDoS outage in July 2019 [14], affecting the availability of thousands of their customers [27]. As part of an XSS detector, Cloudflare used the quartic regex `a*b?c?a*a*a*` (simplified) to detect JavaScript tokens within web traffic. Some typical traffic triggered the worst-case behavior of this regex, exhausting Cloudflare's computational resources and affecting their customers' web services.

III. Background on regular expressions

A. Regular expressions (K-regexes)

Kleene introduced regular expressions (*K-regexes*) to describe strings constructed using a finite number of concatenations, repetitions, and disjunctions [28], [29]. Rabin and Miller showed that they were equivalent in expressive power to non-deterministic finite automata (NFAs) [30]. We will denote NFAs using the standard 5-tuple $\langle$states Q, start q_0, accepting F, alphabet Σ, transitions $\delta\rangle$ [31].[2] The typical regular expression notation is given in Figure 2, with equivalent NFAs according to the Thompson-McNaughton-Yamada construction [32], [33].

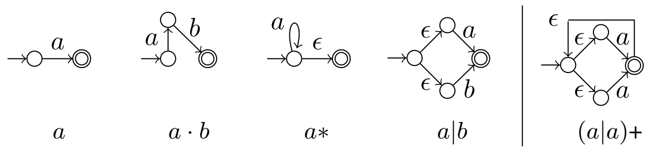


Fig. 2: K-regex operators and NFA equivalents for character, concatenation, repetition, and disjunction. NFAs for K-regexes have in-degree and out-degree ≤ 2, although this is higher under ϵ-closure. The final figure shows an exponentially ambiguous K-regex.

B. Regex languages and ambiguity

Regexes describe a *language*, a set of strings that meets the criteria embodied in the expression [31]. As with other pattern languages like context-free grammars [34], [35], regexes can be *ambiguous*: there may be multiple ways for a string to be in the language of the regex [36]. For example, the last expression

[2]Readers unfamiliar with this notation should refer to Table II.

TABLE I: Worst-case time and space complexities for typical regex recognition algorithms. Candidate string w is tested on K-regex R represented as an NFA with Q states. Although Thompson's algorithm has better theoretical guarantees, Spencer's algorithm is widely used (*e.g.,* JavaScript-V8, Java, PHP, Python, Ruby, Perl, .NET). The last row gives representative complexity of our approach.

Algorithm	$\lvert Q\rvert$	**Time cxty.**	**Space cxty.**
Spencer	$\mathcal{O}(\lvert R\rvert)$	$\mathcal{O}(\lvert Q\rvert^{2+\lvert w\rvert}\times\lvert w\rvert)$	$\mathcal{O}(\lvert Q\rvert\times\lvert w\rvert)$
Thompson	$\mathcal{O}(\lvert R\rvert)$	$\mathcal{O}(\lvert Q\rvert^{2}\times\lvert w\rvert)$	$\mathcal{O}(\lvert Q\rvert)$
Memo-Spencer	$\mathcal{O}(\lvert R\rvert)$	$\mathcal{O}(\lvert Q\rvert^{2}\times\lvert w\rvert)$	$\mathcal{O}(\lvert Q\rvert\times\lvert w\rvert)$

in Figure 2 is ambiguous, because it can parse the string 'a' in two different ways. If there is a maximum number of distinct parses for any string in the language, a regular expression or NFA is called *finitely ambiguous*, otherwise it is *infinitely ambiguous*. Finite ambiguity results from disjunctions, *e.g.,* the 2-ambiguous `a|a`. Infinite ambiguity requires a quantifier, *e.g.,* `(a|a)*`. Infinite ambiguity is one of the necessary conditions for typical super-linear regex behavior [25], [37].

C. String problems and regex engines

Software engineers use regular expressions to answer two string problems [31]. The *recognition problem* tests whether or not a candidate string w is in the language of a regular expression R (*i.e.,* the regex *matches* the string). This permits engineers to determine appropriate control flow in a software application. The *parse problem* returns the matching substring and any sub-captures ("capture groups"). This supports activities like scraping web pages or log files.

Programming languages use a *regex engine* to solve these problems. These engines follow two general algorithms that can be modeled as NFA simulation (with extensions for parsing [38] and irregular features [39], [40]). Under this model, the algorithms search for a path to an accept state, either depth-first (Spencer's) or breadth-first (Thompson's). The semantics of regular expression membership follow the NFA membership problem, with extensions and deviations discussed by Câmpeanu [41], [42] and Berglund [43]–[46].

Table I summarizes these algorithms and our approach.

1) The Spencer algorithm: This resolves non-determinism using a backtracking [47] NFA simulation (Listing 1). Given a choice of edges, it tries one and saves the others for later. Its state consists of its current *simulation position* $\pi = \langle q \in Q, i \in \mathbb{N}^{|w|}\rangle$ (an automaton state and an index i into the string w), and a *backtracking stack* that records the un-tested alternative branches, for $\mathcal{O}(|Q|\times|w|)$ space complexity.

Spencer's search policy can exhibit exponential time complexity [22], [25], [37]. To see this, examine the last illustration of Figure 2. Consider a backtracking simulation of this NFA on the string $w = a^k b$, and count the number of paths to the accept state for the prefix $a^k = a \overset{k}{\ldots} a$. There are two paths to the accept state on the first a; twice that many for a^2; and twice again for a^3. The geometric recurrence yields 2^k paths for the prefix a^k. When the simulation encounters the suffix b, each of these paths results in failure. Note that many of these paths

Listing 1 Spencer-style backtracking K-regex recognition. Regex engines often use an explicit stack, not recursion.

```
# Invoke as recognize(NFA, w, NFA.q0, 0)
# Returns True if the NFA recognizes w
def recognize(NFA, w, currQ, i):
  if i == len(w): # Max recursion depth
    return True if currQ in NFA.F else False
  for nextQ in NFA.delta(currQ, w[i], i): # O(Q)
    if recognize(NFA, w, nextQ, i+1): # DFS
      return True # Bubble up a success
  return False # Backtrack
```

are redundant. Nevertheless, the backtracking algorithm may attempt exponentially many paths before returning a mismatch, with time complexity $\mathcal{O}(|Q|^{2+|w|}\times|w|)$.

2) The Thompson algorithm: This resolves non-determinism using a lockstep NFA simulation. When there is a choice of edges, it simulates taking all of them together. Its state tracks the current offset i and the vertex-set of the currently activated NFA states: $\langle i, \Phi \subseteq Q\rangle$, for space complexity $\mathcal{O}(|Q|)$. As for time complexity, for each character in the input string the algorithm queries the transition function δ for each of the current NFA states Φ. Thus the time complexity is $\mathcal{O}(|Q|^2\times|w|)$.

D. The persistence of ReDoS: Regexes in practice

In principle, applying Thompson's algorithm is a straightforward remedy for ReDoS. There are obstacles in practice.

Dialects: Perl introduced regexes as a first-class programming language construct based on Spencer's library [48]. All subsequent programming languages support regexes and generally follow Perl-Compatible Regular Expressions (PCRE) notation and semantics [1], [49]. However, they also maintain independent (and inconsistent) regex engines, leading to multiple regex specifications [49], [50] and many dialects [1], [7], [51].

Extended features: PCRE's extended regexes (*E-regexes*) are more expressive and more powerful than Kleene's K-regexes. They offer "syntax sugar" regular operators (*K-compatible*) including character classes (`[a-z]`), cuts `(?>a)` [1], and limited repetition (`a{3,5}`), as well as irregular operators like zero-width assertions `(?=a)` and backreferences (`(a)\1`) [52].

Engine implementations: In practice, form may follow function — regex features dictate regex engine algorithms. Spencer's backtracking algorithm is used by all "PCRE" regex engines, including Java, JavaScript-V8, PHP, Python, Ruby, Perl, and .NET [1], [7]. This was an engineering decision; Spencer was aware of Thompson's approach but chose backtracking for its greater flexibility (E-regexes) and simplicity of implementation [2]. Thompson's algorithm is used only by Rust [53] and Go [54], as well as Google's standalone RE2 regex engine [15]. These engines do not support E-regexes.

Maintainer priorities: The maintainers of backtracking regex engines are aware of the threat of ReDoS. Several engines have longstanding bug reports describing problematic time costs [55], [56]. Historically, maintainers may have viewed super-linear regexes as aberrant usage to be addressed at the application level. New research has demonstrated the extent of the ReDoS problem [6], [12], perhaps motivating recent optimizations [57].

IV. Limitations of existing ReDoS defenses

Our threat model assumes that developers will use regexes to handle user input (ReDoS Condition 3). Defenders must address Condition 1 (slow engine) or 2 (slow regex). Let us consider existing ReDoS defenses.

A. Slow engines: Remove super-linear matching behavior

Addressing Condition 1 would be a fundamental solution. If a regex engine guaranteed linear-in-$|w|$ match times, then ReDoS would be addressed once-for-all.

1) Use another algorithmic framework: This comes via application-level substitution or engine-level overhaul.

Applications can adopt a third-party linear-time regex engine. For example, after their outage Cloudflare moved from Lua's backtracking regex engine to RE2 [14]. Unfortunately, substituting one regex engine for another is fraught. E-regexes cannot be ported, because the current generation of linear-time regex engines only support K-regexes. Most K- and K-compatible regexes can be ported, but this is complicated by the abundance of regex dialects. Regexes are under-tested [58] so finding portability problems may be difficult [7].

At the regex engine level, the engine maintainers could overhaul their regex engine to use a faster algorithm. Although regex engine maintainers know about ReDoS [55], [56], the engineering cost of leaving the backtracking framework may be unpalatable. Maintainers regularly improve common-case performance (*e.g.,* V8 [59], [60] and .NET [57]), but have not undertaken an algorithmic overhaul to address ReDoS. Some newer regex engines are designed around Thompson's algorithm (RE2, Rust, Go), but legacy engines may have too much technical inertia to follow suit.

2) Backtracking optimizations: Although no regex engine has changed its algorithmic framework to address ReDoS, maintainers have incorporated "inline" optimizations that fit into the backtracking framework. Some optimizations find good starting points for an NFA simulation [16]–[18], and are orthogonal to our approach. More apropos to our approach, other optimizations remove some of the redundant paths in the backtracking search. For example, .NET [57] and Perl use *prefix factoring* to unite overlapping paths, while *caching* is used to accelerate backtracking in Perl and a rare path in RE2. Our approach is also of this kind; we improve on previous approaches by guaranteeing soundness (see §X).

3) Capping super-linear behavior: Three backtracking regex engines defend against ReDoS by limiting the resources consumed by a regex match. The .NET regex engine offers a wall clock *time-based* cap on matches [61]. The PHP and Perl regex engines use *counter-based* caps [62], [63], throwing an exception if a match exceeds their cost measures. Resource caps certainly protect web services, but are a perennial "Goldilocks" problem [24], [64]: too tight and they reject valid input, too loose and they permit moderate ReDoS. Our approach does not require tuning. It guarantees that users always get a regex answer, not an exception.

B. Slow regexes: Remove super-linear regexes

If a super-linear regex engine must be used, application developers may instead refactor their use of super-linear regexes. This defense is problematic in two ways. *First*, it is *ad hoc*. The maintainers of individual computing systems must determine that they are vulnerable, choose a refactoring strategy, and apply it correctly. This process is error prone [6].[3] *Second*, it is indirect, putting the burden for a solution on application developers rather than addressing the root cause. Developers might prefer a regex with super-linear structure, *e.g.,* to facilitate maintainability or comprehension [67]. Our approach accommodates such preferences in linear time.

V. Our approach: Context and overview

We propose a memoization approach to guarantee linear-time matching within the backtracking algorithmic framework. Our approach improves on the state of practice (Spencer's algorithm). We emphasize that we do not attempt to improve on the theoretical state of the art for regex matching (Thompson's algorithm). But in the 60 years since Thompson proposed his algorithm, practitioners have shown no inclination to abandon the backtracking framework in their legacy regex engines (§IV). Therefore, we propose an approach that can be adopted within a backtracking framework with minimal changes.

Our approach builds on Michie's general function memoization technique [68], an optimization that spends space to save on time. Memoization records a function's known input-output pairs to avoid evaluating it more than once per input. Many algorithms benefit from the time savings [69]–[71]. Space costs depend on the input-output domain.

Memoization techniques have previously been applied to parsing problems, notably "Packrat Parsers" for context-free grammars (CFGs) and parsing expression grammars (PEGs) [47], [72]–[75]. Memoization can also be used for regex matching. If the input-output pairs in Spencer's algorithm were recorded, then redundancy can be eliminated. As an example, consider Listing 1 when applied to the final illustration from Figure 2. The many paths to the accepting NFA vertex correspond to many (redundant) recursive queries to `recognize` for the same simulation positions. Memoizing the results of `recognize` would eliminate this redundancy.

Full memoization, *i.e.,* recording every input-output pair for `recognize`, has never been applied for practical regex matching due to its space complexity [3], [43], [76], [77]. Large regexes are used on long input and so the $\mathcal{O}(|Q|\times|w|)$ space complexity of full memoization is too costly. To employ memoization for regex matching in practice, its space complexity must be reduced. Some regex engines have done so with unsound heuristics (§X). Our memoization techniques offer linear-time K-regex matching *soundly and with low space*

[3]The regex refactoring process might be automated. Existing techniques are unsound [57], [65] or entail exponential space complexity [66].

costs. For E-regexes, we obtain linear-time matching for zero-width assertions and parameterized costs for backreferences.

Here are the key ingredients of our approach. First, we prove that full memoization records more data than necessary for linear-in-$|w|$ K-regex time complexity. This goal can be achieved by selectively memoizing visits to only a subset of the automaton vertices (decreasing space *complexity*). Then, we consider the embodiment of the memo function M, and observe that for many regexes its state will be low-entropy and compressible via run-length encoding (decreasing space *cost*). Finally, we extend these techniques to two E-regex features.

We divide our presentation into four parts: a formalization of Memoized Non-Deterministic Finite Automata (M-NFA) to reason about our approach (§VI); analyses of the behavior of M-NFA for K-regex recognition under three memoization schemes (§VII); a space-efficient representation of the memo function (§VIII); and extensions for E-regexes (§IX).

VI. Memoized non-deterministic finite automata

Using memoization, a regex engine can record the areas of the search space that it has explored. If it revisits those regions, it can short-circuit a redundant exploration. Although a *full memoization* approach is a standard technique from Packrat Parsing, we introduce two novel *selective memoization* schemes in §VII whose properties are more subtle.

To provide a framework for the analysis of our selective memoization schemes, we introduce a novel extension of Rabin-Miller NFAs. We define this entity as a *Memoized Non-Deterministic Finite Automaton* (M-NFA), with components described in Table II.[4] This model enables us to reason about the behavior of an NFA simulation algorithm when applied to an M-NFA. The additional components are:

M The memo function M of an M-NFA is updated during the backtracking simulation. It initially returns 0 to all queries. After a simulation position π is marked, the memo function returns 1 for subsequent queries to that position.

δ_M An M-NFA's memoized transition function δ_M accepts the typical arguments to δ, plus a candidate string index i:

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

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

Simulation: An M-NFA can be simulated on a string w beginning from q_0, by repeated application of the memoized transition function δ_M (see Listing 2). If the simulation ends in a state $q \in F$, the M-NFA accepts the candidate string. Note that for K-regexes, the outcome of a match starting from a given position π is determined solely by the current position, and not on previous decisions. Thus, the memo function tracks at most the $|Q|\times|w|$ possible positions.

Memoization scheme: During M-NFA simulation, the choice of which simulation positions π to memoize is determined by

[4]Strictly speaking, an M-NFA is not finite because it uses memory based on w. The components M and δ_M can be viewed as maintained by a memoized simulation of the NFA. We feel the M-NFA conceit is simpler.

TABLE II: Components of a Memoized Non-Deterministic Finite Automaton (M-NFA) derived from an NFA $A = \langle Q, q_0 \in Q, F \subseteq Q, \Sigma, \delta \rangle$. The components of A are listed above the mid-rule (cf. §III-A). The components of the M-NFA for A include those, plus the additional components below the mid-rule: M and δ_M.

Component	Meaning
Q	Automaton states
$q_0 \in Q$	Initial state
$F \subseteq Q$	Accepting states
Σ	String alphabet: $w \in \Sigma*$
$\delta : Q \times \Sigma \to \mathbb{P}(Q)$	Transition function
$M : Q \times \mathbb{N}^{\|w\|} \to \{0,1\}$	Memo function ("memo table")
$\delta_M : Q \times \Sigma \times \mathbb{N}^{\|w\|} \to \mathbb{P}(Q)$	Memoized transition function

Listing 2 Memoized backtracking K-regex recognition. Differences from Listing 1 are highlighted.

```
# Invoke as memoRecognize(MNFA, w, MNFA.q0, 0)
def memoRecognize(MNFA, w, currQ, i):
  if i == len(w): # Max recursion depth
    return True if currQ in MNFA.F else False
  for nextQ in MNFA.deltaM(currQ, w[i], i) :
    if memoRecognize(MNFA, w, nextQ, i+1):
      return True # Bubble up a success
  MNFA.M.mark(currQ, i)  # This position failed
  return False # Backtrack
```

a *memoization scheme*. Our schemes memoize all simulation positions associated with a selected subset $Q_{sel.}$ of the automaton vertices, *i.e.*, all $\pi = \langle q \in Q_{sel.}, i \in \mathbb{N}^{|w|} \rangle$. When $Q_{sel.} = \emptyset$, the M-NFA is equivalent to an NFA.

Ambiguity: We define an ambiguous M-NFA analogous to an ambiguous NFA (§III-A). An M-NFA is *ambiguous* if there exists a string w such that when it is simulated from q_0, there are multiple paths to an accepting state. The degree of ambiguity of an M-NFA can be tuned by the memoization scheme. For example, with $Q_{sel.} = \emptyset$ the M-NFA has the same ambiguity as the NFA, while with $Q_{sel.} = Q$ (*i.e.*, Packrat Parsing [73]) the M-NFA is unambiguous.

Space complexity: Based on our M-NFA model, a memoization scheme incurs additional space complexity $\mathcal{O}(|Q_{sel.}|\times|w|)$ to store the memo function.

Time complexity: Based on our M-NFA model, the time cost of an M-NFA simulation can be calculated as:

$$(\textit{\# sim. pos.})\times(\textit{max visits per pos.})\times(\textit{cost per visit}). \quad (1)$$

For K-regexes there are $|Q|\times|w|$ simulation positions. We assume that visits cost $\mathcal{O}(|Q|)$ per the loop in Listing 2, with $\mathcal{O}(1)$ updates to M and queries to δ_M [78]. If each position is visited once (Table III), the time complexity is $\mathcal{O}(|Q|^2\times|w|)$.

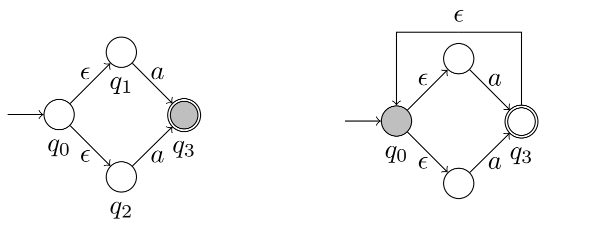
VII. Selective memoization for K-regex recognition

In this section we present three selective memoization schemes for backtracking M-NFA simulations that follow Listing 2. Table III summarizes their properties. For each scheme

TABLE III: Time and space complexity of K-regex matching using the selective memoization schemes. Each scheme adds space complexity $|Q_{sel.}| \times |w|$, and they are ordered from largest to smallest vertex set size. The time complexity for $Q_{ancestor}$ has a factor $f(Q)$ indicating the NFA's maximal bounded ambiguity (see appendix).

Memo scheme	Visits per pos.	Time cxty.	Add'l. space cxty.
None (Spencer)	$\mathcal{O}(\|Q\|^{\|w\|})$	$\mathcal{O}(\|Q\|^{2+\|w\|} \times \|w\|)$	—
$Q_{all} = Q$	≤ 1	$\mathcal{O}(\|Q\|^2 \times \|w\|)$	$\|Q\| \times \|w\|$
$Q_{in-deg>1}$	≤ 1	$\mathcal{O}(\|Q\|^2 \times \|w\|)$	$\|Q_{in-deg}\| \times \|w\|$
$Q_{ancestor}$	$\leq f(Q)$	$\mathcal{O}(\|Q\|^2 \times \|w\| \times f(Q))$	$\|Q_{anc.}\| \times \|w\|$

we bound the number of visits per simulation position. We sketch intuition here using Figure 3. See appendix for proofs.



(a) Q_{in-deg}: M-NFA for $(a|a)$ (b) $Q_{anc.}$: M-NFA for $(a|a)+$

Fig. 3: Memoized automata used to illustrate the selective memoization schemes. Shading indicates the vertices associated with memoized simulation positions for (a) $Q_{in-deg>1}$ and (b) $Q_{ancestor}$.

A. Select all vertices: Q_{all}

In this scheme, we record the outcome of the simulation positions associated with every vertex of the M-NFA, $Q_{all} = Q$. From the definition of δ_M, each simulation position can be reached at most once. Theorem 1 follows directly.

Theorem 1. *Let the memo function track simulation positions involving all M-NFA vertices Q. Then the M-NFA is unambiguous — every simulation position $\pi = \langle q, i \rangle$ is visited at most once.*

B. Select vertices with in-degree > 1: $Q_{in-deg>1}$

This scheme memoizes only the vertices $Q_{in-deg>1}$ with in-degree greater than one. Like Q_{all}, memoizing $Q_{in-deg>1}$ eliminates redundant visits; the M-NFA is unambiguous.

Theorem 2. *Let the memo function track simulation positions involving the M-NFA vertices with in-degree greater than one, $Q_{in-deg>1}$. Then the M-NFA is unambiguous — every simulation position π is visited at most once.*

Proof sketch: Figure 3 (a) illustrates this scheme. The shaded vertex q_3 has in-degree 2. By memoizing it, on the string a it can only be visited from q_1 and not q_2. Intuitively, in order for there to be redundancy (ambiguity) in the M-NFA simulation, we must reach the same simulation position π along two different paths. To reach this position twice: (1) There must have been a choice; and (2) Both branches must have led to π along edges with the same labels. At the fork, the two paths diverged; later, they converged. For two paths to converge, they must share some vertex with in-degree > 1. *Memoizing visits to this vertex prevents multiple visits to π.*

Observations: This scheme eliminates non-determinism (outgoing edges with the same label) that results in ambiguity (branches converge). Per Figure 2, these conditions may arise in the Thompson-Glushkov construction from the use of disjunctions or Kleene stars (*e.g.,* `a|a` or `a*a*`). The $Q_{in-deg>1}$ scheme memoizes visits to the convergence points.

C. Select cycle ancestors: $Q_{ancestor}$

This scheme memoizes the vertices $Q_{ancestor}$ that are cycle ancestors according to a topological sort from q_0. This memoization scheme prevents the compounding of ambiguity; it eliminates infinite ambiguity, but still permits $f(Q)$ redundant visits based on a notion of *bounded ambiguity*.[5] The M-NFA is thus finitely ambiguous.

Theorem 3. *Let the memo function track simulation positions involving the M-NFA vertices that are "cycle ancestors", $Q_{ancestor}$, i.e., automaton vertices to which back-edges are directed in a topological sort from q_0. Then the M-NFA is* f*-ambiguous. Every simulation position will be visited at most a finite number of times* f(Q).

Proof sketch: Figure 3 (b) illustrates this scheme. The shaded vertex q_0 is an ancestral node that can compound the two ambiguous paths to q_3. By memoizing q_0, only one of the two simple paths to q_3 can form a cycle with q_0, and the M-NFA is 2-ambiguous just as `a|a` is. Intuitively, infinite ambiguity arises when finite ambiguity is compounded by the presence of cycles in an NFA. For example, `a?a?` and `a|a` are finitely ambiguous, while `a*a*` and `(a|a)*` are infinitely so. These infinitely ambiguous variations add cycles to the NFA. *Memoizing the cycle ancestors permits ambiguity, but this ambiguity is bounded because all but one of the possible cycles are eliminated by δ_M.* This yields the result in Theorem 3.

Observations: This memoization scheme permits redundant visits that result from finite ambiguity. Although Cox observes that a regex can be constructed with arbitrarily large finite ambiguity [3], we lack analysis tools to determine whether such regexes are practical or pathological. Regardless, the degree of redundancy permitted by this memoization scheme is not super-linear — it is limited as a function of the automaton, not of the length of w. Beyond a certain point, increasing $|w|$ will not increase the worst-case behavior.

In terms of K-regex features, Figure 2 shows that ancestral back-edges only occur from the use of Kleene stars. The $Q_{ancestor}$ memoization scheme memoizes the vertices to which these back-edges are directed. If there is finite ambiguity in the sub-pattern to which the Kleene star is applied, that ambiguity remains after $Q_{ancestor}$-memoization.

[5]See Appendix for the refined theorem with a precise definition of $f(Q)$.

This memoization scheme has lower space complexity than the preceding one because $Q_{ancestor} \subseteq Q_{in-deg>1}$. Under standard regex-to-NFA constructions, all vertices in an NFA graph are reachable from q_0 and thus have in-degree ≥ 1. The vertices in $Q_{ancestor}$ have an additional in-degree due to the back-edge, and hence are also in $Q_{in-deg>1}$.

D. Time and space complexity guarantees

Time: In §VI we stated that the time complexity of an M-NFA simulation depends on the maximum number of visits to each simulation position. Our theorems provide upper bounds on visits, resulting in the time complexities given in Table III.

Space: Memoization schemes incur additional space complexity $\mathcal{O}(|Q_{sel.}|\times|w|)$. This space complexity decreases monotonically as: $Q = Q_{all} \supseteq Q_{in-deg>1} \supseteq Q_{ancestor} \supseteq \emptyset$.

E. Discussion

1) Semantics: The use of memoization eliminates redundant path exploration, but does not otherwise affect the behavior of the regex engine. Recall that paths are pruned when *re*-encountered while backtracking; their original exploration was unsuccessful. The regex engine's semantics remain unchanged.

2) Parsing with memoization: These selective memoization schemes improve the worst-case performance of the K-regex *recognition* problem (regex match) for backtracking regex engines. They similarly improve the worst-case performance of the other regular string problem, *parsing* (capture groups). For K-regexes, capture group contents do not affect the match, so memoization is orthogonal. Matters change for E-regexes (cf. backreferences in §IX).

3) Processing costs: The automaton vertices $Q_{sel.}$ selected by our selective memoization schemes can be identified during a parsing pass. These vertices are associated with disjunction and Kleene star operators in a 1-to-1 manner.

4) Unnecessary memoization: Our formal selection approaches will eliminate all super-linear behavior. It is instructive to consider unsound variations, cf. §X-A.

Our selective memoization schemes may involve unnecessary memoization. Our schemes select vertices according to analysis on an "unlabeled skeleton" of the NFA. Only vertices that meet a stronger condition actually need be memoized, namely that they be reachable along multiple ambiguous paths. For example, the regex `a|b` is unambiguous, so the vertex with in-degree 2 need not be memoized. However, our approximations have the advantage that the vertices involved can be identified in $\mathcal{O}(|Q|)$ steps during the NFA construction. For example, the vertices in $Q_{ancestor}$ are a superset of the "pivot nodes" necessary for infinitely ambiguous behavior; identifying those pivot nodes requires $\mathcal{O}(|Q|^6)$ to $\mathcal{O}(2^{|Q|})$ time complexity [25], [37], [79]. In our evaluation we find that $Q_{in-deg>1}$ and $Q_{ancestor}$ are typically small, so further refinement may not be worthwhile. Faster ambiguity analyses for typical regexes would enable further space reduction.

TABLE IV: Properties of the memo function representations, ordered by best-case space complexity. The first two are standard techniques. Based on regex engine semantics, we propose the application of run-length encoding to the memo function. k denotes the number of runs.

Representation	**Access time**	**Space complexity**
Memo table	$\mathcal{O}(1)$	$\Theta(\lvert Q_{sel.}\rvert\times\lvert w\rvert)$
Positive entries	$\mathcal{O}(1)$	$\Omega(\lvert w\rvert)$; $\mathcal{O}(\lvert Q_{sel.}\rvert\times\lvert w\rvert)$
Run-length encoding	$\mathcal{O}(\log k)$	$\Omega(\lvert Q_{sel.}\rvert)$; $\mathcal{O}(\lvert Q_{sel.}\rvert\times\lvert w\rvert)$

VIII. Representations of the memo function

Implementing an M-NFA requires an embodiment of the memo function indicated in Table II (the implementation of `mark` in Listing 2). Our selective memoization schemes will decrease the amount of state tracked by this memo function. If this state can be efficiently represented, the total space cost of memoization can be made lower still. We discuss three implementations of the memo function, two conventional and one novel. Their properties are summarized in Table IV.

A. Memo table

One implementation of the memo function is a *memo table*, an array with a cell indicating each input-output pair. For K-regex memoization, this memo table would be a two-dimensional array whose cells are 0-1 valued. This array offers optimal access times and requires $|Q_{sel.}|\times|w|$ space.

B. Positive entries

Because the entries in the memo table can contain only two values, only the cells with one of the values need be tracked. In our context, we might track only the visited positions. A missing entry means no visit.

A data structure with efficient random access and update times (*e.g.,* a hash table) can be used to store only the visited cells, as is common for memoization in functional programming [70], [71], [80], This approach offers the same asymptotic access times as an array, although with larger constants. However, its space complexity is input-dependent and may be superior to an array. On some pathological inputs, the space cost is $\Omega(|w|)$, when only one of the memoized vertices is visited along w. For example, this would occur if the input exploits only some of the potential ambiguity in the regular expression. In the worst case, all of the memoized vertices are visited repeatedly, for cost $\mathcal{O}(|Q_{sel.}|\times|w|)$.

C. Run-length encoding

We propose to further decrease the space cost by compressing the information in the memo table. We interpret the memo table as an array of $|Q_{sel.}|$ *visit vectors*, one per memoized NFA vertex, each of length $|w|$. In the context of regular expressions these visit vectors may be compressible because the engine's search regime is ordered.

The ordered search regimes necessitated by regex engine match semantics (*e.g.,* PCRE's leftmost-greedy behavior) cause the memo function to be updated in an orderly manner. Updates will tend to accrete to adjacent indices as characters

TABLE V: Time complexity for E-regexes in a backtracking (BT) framework. Analysis is based on our engine model.

E-regex feature	Time: BT w/o memo	Time: BT w/ memo
REWZWA	$\mathcal{O}(\lvert Q\rvert^{3+2\lvert w\rvert}\times\lvert w\rvert^2)$	$\mathcal{O}(\lvert Q\rvert^2\times\lvert w\rvert)$
REWBR	$\mathcal{O}(\lvert Q\rvert^{2+\lvert w\rvert}\times\lvert w\rvert^2)$	$\mathcal{O}(\lvert Q\rvert^2\times\lvert w\rvert^{2+2\lvert \mathbf{CG}_{BR}\rvert})$

are processed. Because the alphabet of the memo table is small, this property results in visit vectors with consistently low entropy, and hence a high compression ratio. This observation has been described as the locality of NFA traversals [81].

As a result, it is reasonable to expect that the memo tables in the K-regex 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, *e.g.,* quantifier destinations, may be visited at many indices. Some of these visits will be especially compressible. Consider, for example, the behavior of a regex engine on a catch-all expression like `/.*/`. The NFA vertex corresponding to the quantifier would be memoized under all of our selective memoization schemes. Once an NFA simulation reaches this structure, it will recursively check for regex matches after consuming each of the *adjacent* characters in an ordered manner, accreting a compressible sequence of memo table entries.[6] Such quantifiers are common. In Davis *et al.*'s regex corpus [7], we found that among the 253,216 regexes that use an unbounded quantifier, fully 103,664 (41%) include the catch-all expression `/.*/` or `/.+/`.

As a compression strategy, we propose to use run-length encoding (RLE) [82] because it supports in-place updates. When a visit vector is implemented as a binary tree, with elements the runs keyed by their offsets [83], this scheme offers $\mathcal{O}(\log k)$ access time for a vertex with k runs. If the encoding scheme is effective, k will be small and the time cost will be competitive with the two other memo function representations we have discussed. If it is ineffective, k may be as large as $|w|$, substantially increasing the time cost of this scheme relative to the others. By the same token, if the encoding scheme is effective, the space cost of this scheme is *constant* for each visit vector. It remains $\mathcal{O}(|w|)$ per vector in the worst case.

IX. Memoization for E-regexes

A. E-regex background: REWZWA and REWBR

Most regex engines support extensions beyond K-regexes. Up to 5% of general-purpose regexes use E-regex operators, most commonly involving zero-width assertions and backreferences [4], [5]. We denote regexes with zero-width assertions as REWZWA, and use REWBR for backreferences.

1) REWZWA: A *zero-width assertion* tests a condition on w without changing the simulation position. Fixed-width assertions examine a constant number of characters from the current position, *e.g.,* the word boundary `\b`. More general assertions, called lookaround assertions, may examine the entire string w. For example, the regex `(?=a+)\w+(?<=z+)` uses assertions to find words that begin with a's and end with z's.

Programming languages vary in assertion expressiveness. Rust and Go support fixed-width but not lookaround assertions. For lookahead, *i.e.,* `(?=...)`, JavaScript, Java, Python, PHP, and Ruby support K-regexes and some E-regex features (*e.g.,* backreferences). For lookbehind, *i.e.,* `(?<=...)`, JavaScript and Java support K-regexes, while Perl, PHP, Ruby, and Python support only fixed-width assertions. In our analysis we suppose a semantic descriptive of typical REWZWA usage: that REWZWA can assert K-regexes.

2) REWBR: A *backreference* permits an intra-pattern reference to the substring of w matched by an earlier capture group (labeled sub-pattern). For example, the regex `(a|b)\1` matches "aa" and "bb" but not "ab" or "ba". There are dialectal variations on REWBR semantics [41], [44], [52], [84], but the details are unimportant for our purposes.

Contribution: We study the worst-case performance of REWZWA and REWBR in typical backtracking implementations. We describe current implementations and show time and space bounds with memoization. Table V gives time complexity improvements.

B. Regexes with zero-width assertions (REWZWA)

In typical implementations, REWZWA have exponential time complexity. Memoization lowers the cost to linear in $|w|$.

1) Implementations: REWZWA implementations have not previously been described in the literature. We examined the approaches of Perl, PHP, Python, and JavaScript-V8. They are recursive: they compile the asserted sub-pattern, save the simulation position when they reach it, and evaluate the sub-pattern using their K-regex machinery. If the sub-pattern matches, the simulation position is reset (zero-width), else they backtrack. In terms of standard NFA representations, this type of implementation can be modeled as introducing α-edges that describe a regular string pattern via sub-automata. Like ϵ-edges, these α-edges consume no characters.

2) Complexity analysis: The space complexity of REWZWA in these implementations is the same as for K-regexes. The time complexity is worse. For REWZWA with K-regex assertions, because the assertions do not consume characters, *each step* of the simulation may traverse α-edges at an exponential cost. Following Eqt. (1), we again have $|Q|\times|w|$ simulation positions, up to $|Q|^{|w|}$ visits per position, and any α-edges may bear the full cost of a backtracking sub-simulation over w. The expressiveness of the assertions bounds the cost of these α-edges. In a backtracking framework, fixed-width assertions cost constant time, K-regexes cost exponential, and backreferences or nested assertions still further. Table V assumes K-regex α-edges, hence a cost of $\mathcal{O}(|Q|^{|w|})$ per visit.

3) Memoization: Memoization reduces the time complexity for REWZWA to linear in $|w|$. This reduction in time complexity follows from a simple observation: for each α-edge, the simulations of the corresponding sub-automaton will all

[6]Non-greedy quantifiers do not change this locality, but merely reverse the order in which the indices are explored. The exploration entropy remains low.

operate on the *same* sub-automaton and on *some* substring of w. Rather than treating these simulations independently, we retain what we learn in one for use in the next.

Extension of M-NFA: We can operationalize this observation with two changes to the REWZWA model. First, convert the top-level automaton and each sub-automaton (α-edges) to M-NFAs. Second, preserve the memo functions of the sub-automata throughout the simulation of the top-level M-NFA, remembering the results from sub-simulations that begin at different indices i of w.

Time complexity: Because REWZWA implementations use a single large tagged automaton (flat, with no levels), in practice this modeling requires minimal modification to an existing memoized implementation. The time complexity analysis is easy to follow when expressed in terms of this single automaton. By counting the sub-automaton vertices in Q and memoizing Q, we obtain the familiar:

$$\mathcal{O}\big(\ (|Q|\times|w|)\ \times\ 1\ \times\ |Q|\ \big) = \mathcal{O}\big(|Q|^2\times|w|\big)$$

This result can also be obtained in our hierarchical α model by dividing the cost across the hierarchy of automata. The top-level automaton simulation has time complexity $\mathcal{O}(|Q|^2\times|w|)$ ignoring the α-edges. The cost of evaluating an α-edge varies based on the simulation position. However, because the memo function is saved from one sub-simulation to the next, redundancy is eliminated and the cost amortizes to $\mathcal{O}(|Q_\alpha|^2\times|w|)$.

Space complexity: Counting the sub-automaton vertices in Q,[7] the space complexity is the standard $\mathcal{O}(|Q|\times|w|)$.

Selective memoization: Our proposal for REWZWA is essentially recursive. Selective memoization remains applicable.

C. Regexes with backreferences (REWBR)

The general REWBR matching problem is NP-complete [52]. REWBR algorithms are exponential in some combination of $|Q|$ and $|w|$.

1) Implementations: Backtracking regex engines support parsing by tracking the substring of w matched by each capture group using a *capture group vector* $\mathbf{CG}$ of index pairs $\langle j,k\rangle$. To evaluate a backreference with index pair $\langle j,k\rangle$ at offset i, they compare *w[i:i+(k-j)]* to *w[j:k]*, *i.e.,* the current contents of the appropriate group. This can be modeled in a tagged NFA by introducing β-edges that perform string comparison.

2) Complexity analysis: REWBR are context-sensitive: the path through the automaton determines the capture group contents, and so evaluating a REWBR may require exploring the exponentially many automaton paths [44], [52], [85]. Of course, in the worst case, existing backtracking engines *already* explore all these paths (hence ReDoS). But note that it is more expensive to test a β-edge than a regular one, because it entails an $\mathcal{O}(|w|)$ string comparison. The worst-case time complexity for a REWBR match within an un-memoized backtracking framework is thus:

$$\begin{aligned}\mathcal{O}\big(\ (|Q|\times|w|)\ \times\ |Q|^{|w|}\ \times\ (|Q| + \underbrace{|w|}_{\beta\text{-edge cost is } |w|,\text{ not } 1}\ )\ \big)\\ = \mathcal{O}\big(|Q|^{2+|w|}\times|w|^2\big)\end{aligned}$$

The simplified form uses the property $|Q| + |w| \leq |Q|\times|w|$.

3) Memoization: Compared to typical backtracking implementations, memoization reduces the time complexity from exponential in $|w|$ to exponential in $|Q|$ — roughly speaking, from $\mathcal{O}(|Q|^{|w|})$ to $\mathcal{O}(|w|^{|Q|})$. Since we typically expect $|w| \gg |Q|$, the reduction is substantial.

Extension of M-NFA: Because the contents of a capture group depend on the path taken through the automaton, REWBR disrupts our path-independent memoization scheme. To identify redundancy, we need to track the capture groups as well. If we reach the same simulation position $\langle q,i\rangle$ but with a different capture group vector, the simulation outcome may differ.

For example, the regex `<([a-z]+)>(a|a)+</\1>` uses a backreference to match HTML tags. It contains the exponential K-sub-regex `(a|a)+`. This sub-pattern may result in exponentially many visits to the β-edge for `\1`, all of which share the same capture group vector $\mathbf{CG}$ and vary only in their simulation positions. Incorporating $\mathbf{CG}$ into the memo functions can safely eliminate these redundant paths. To accomplish this, we extend the M-NFA described in Table II:

Capture groups The simulation must track the capture group vector $\mathbf{CG}$, where CG_i denotes the ith capture group.

Memo functions The domains of the REWBR memo function M' and transition function δ'_M must consider the simulation position π as well as the capture group vector $\mathbf{CG}$.

Time complexity: Using Eqt. (1), the change in memo functions inflates the first term but is offset by the decrease in the second term. In the first term, each CG_i can take on $|w|^2$ distinct values. If we select Q_{all}, the time complexity is:

$$\begin{aligned}\mathcal{O}\big(\ (|Q|\times|w|\times(|w|^2)^{|\mathbf{CG}|})\ \times\ 1\ \times\ (|Q| + |w|)\ \big)\\ = \mathcal{O}\big(|Q|^2\times|w|^{2+2|\mathbf{CG}|}\big)\end{aligned}$$

Note that the exponents have changed places when compared to the un-memoized version: from $\mathcal{O}(|Q|^{|w|})$ by counting (potentially-overlapping) paths, to $\mathcal{O}(|w|^{2|Q|})$ by counting distinct path configurations. This bound remains problematic if more than a few backreferences are used.

Space complexity: There are $|Q|\times|w|$ simulation positions, and $|\mathbf{CG}|$ capture groups. The memo function's domain is thus

$$\mathcal{O}\big(|Q|\times|w|\times(|w|^2)^{|\mathbf{CG}|}\big) = \mathcal{O}\big(|Q|\times|w|^{1+2|\mathbf{CG}|}\big).$$

Selective memoization: Selective memoization can be applied to reduce the cost of REWBR. First, only the indices of the *backreferenced* capture groups affect the simulation result, so we need only memoize the sub-vector $\mathbf{CG}_{BR} \subseteq \mathbf{CG}$. This reduces the corresponding exponent for both time and

[7]This is similar to replacing the zero-width assertions with the corresponding sub-patterns, *e.g.,* `a(?=b*)` $\rightarrow$ `ab*`.

space complexity. Second, as with REWZWA, the $Q_{in-deg>1}$ and $Q_{ancestor}$ can be applied to REWBR to reduce the space complexity's vertex factor $|Q|$.

D. Other extended features

Our approach can be applied to other extended features. To memoize features that depend on additional state (*e.g.,* general conditional expressions), that state must be tracked by the memo function. Features with side effects should be treated more carefully. For example, some regex engines support semantic actions through callouts (embedded code), which can change the candidate string w. If so, memoized state would need to be updated.

X. Evaluation

The memoization schemes described in §VII provably eliminate ReDoS by guaranteeing K-regex matching times that are linear in $|w|$. We extended this guarantee to REWZWA within the backtracking framework, and parameterized the super-linearity for REWBR. Our selective memoization schemes incur varying additional space complexity, which may be offset by efficient representations of the memo function (§VIII).

Here we analyze comparable defenses, experimentally confirm our security guarantees (time complexity), and assess the practicality for typical regexes (space complexity and costs).

A. Security analysis: Existing memoization-like defenses

We surveyed regex engine implementations and behavior described in the literature [3], [7], [15], and identified memoization-like ReDoS defenses in Perl and RE2. These defenses unsoundly target only exponential worst-case behavior. To avoid the $|Q|\times|w|$ cost of full memoization, Perl unsoundly and incompletely memoizes certain vertices. while RE2 unsoundly caches a constant number of simulation results. Both schemes may exhibit exponential and polynomial behavior.

Perl: Since 1999, the Perl regex engine has employed an unsound semi-selective memoization scheme called the "super-linear cache". It is undocumented and its workings have not previously been described in the scientific literature.[8] Perl memoizes visits to the first k=16 NFA vertices associated with repetitions `A*` if the language of `A` has strings of varying lengths, *e.g.,* `(a|aa)*`. Perl omits other repetitious states, *e.g.,* bounded repetition (`a{3}`) or repetition with a fixed-length pattern (`.*.*`). The memo table is erased when a backreference β-edge is tested.

Perl's approach is similar in spirit to our $Q_{ancestor}$ scheme, but it is restricted to k=16 cycle ancestors and only considers those whose cycles can be of varying lengths. This scheme is not sound — *e.g.,* it protects `(a*)*` (otherwise exponential), but not `(a|a)*` (exponential) nor `a*a*` (quadratic). We conjecture that this scheme was designed to eliminate exponential worst-case K-regex behavior due to nested quantifiers [6], but did not consider other forms of exponential behavior nor

[8]We describe it as of February 2020. See https://github.com/Perl/perl5 commit 34667d08d.

weaker polynomial behavior. Davis *et al.* described many regexes unprotected by this defense [7].

RE2: Although the RE2 regex engine generally uses Thompson's lockstep algorithm, in rare cases it uses backtracking [15]. An unsound *memoization cache* records visits to C=32 KB's worth of simulation positions (full memoization). This scheme is effective if $|Q|\times|w| \leq C$, *e.g.,* for small regexes and small inputs. It is not useful for large regexes or large inputs, the common case for polynomial worst-case behavior (*e.g.,* Stack Overflow's outage). Our space-efficient memoization proposal is more suited for those cases.

B. Dataset and measurement instruments for our approach

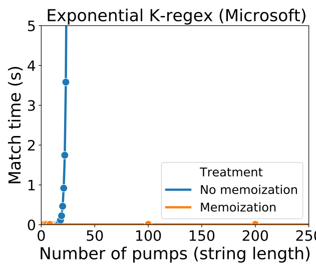
Regex corpus: Researchers have collected several regex corpuses [4], [6], [7], [58]. We use Davis *et al.*'s regex corpus [7], which is the largest and most representative of typical regex practices [5]. It contains 537,806 regexes from 193,524 software projects. This corpus includes 51,224 super-linear K-regexes and inputs to trigger worst-case behavior.

Prototype: We implemented the selective memoization and encoding schemes within a backtracking regex engine published by Cox [86]. This regex engine supports K-regexes, plus capture groups, the optional (`?`) and non-greedy (`*?`) quantifiers, and the catch-all character class (`.`). Unmodified, it could evaluate approximately 17% of the regex corpus and 15% of the super-linear K-regexes.

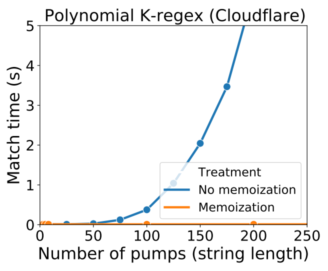
To enable measurement on a wider variety of regexes, we added feature support following popularity measurements by Chapman & Stolee and Davis *et al.* [4], [5]. Our prototype supports K-regexes as well as K-compatible features: anchors, escape sequences, character classes, and bounded repetition. We also implemented two E-regex features: REWZWA (lookahead) and REWBR ($\leq$ 9 groups) using the algorithms described in §IX. Our prototype supports 454,060 (84%) of the corpus regexes. We will refer to these as the *supported regexes*. Among these are 48,505 (95%) of the super-linear K-regexes.

For bounded repetition, we follow the standard expansion strategy: `A{m,n}` $\rightarrow$ `A...A(A(A(...)?)?)?` [15]. What is the impact of this approach on memoization? Flat repetitions like `a{m,n}` increase $|Q|$ by $|m-n|$, while rare nested repetitions like `(a{m,n}){m',n'}` have a geometric effect. However, the "tail recursive" nature of the expansion means that the optional groups for a given repetition all point to the *same* final vertex. Thus $|Q_{in-deg>1}|$ grows more slowly than $|Q|$: by 1 for flat repetitions, by $|m-n|$ for one level of nesting, etc. These structures are loop-free, so $|Q_{ancestor}|$ does not change — redundancy is possible but bounded (Theorem 3).

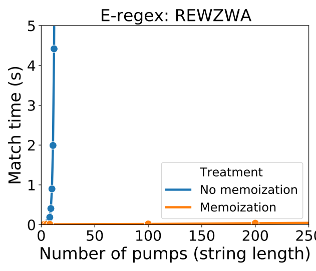
Discounting vendored files, our non-whitespace changes comprised: 2,000 lines for extended feature support; 900 lines for memoization; 850 lines for tests. We implemented the memo table representation using a bitmap; the positive entry representation using `uthash` [87]; and the RLE-based representation using Biggers's `avl_tree` balanced binary tree [88]. We parameterized RLE with runs of length 1 for the common case of catch-all expressions. Our prototype is available at http://github.com/PurdueDualityLab/memoized-regex-engine/.



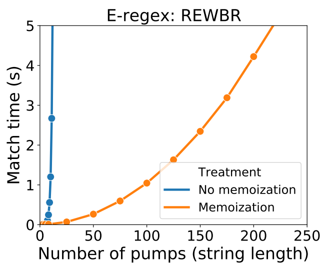
(a) Exp. K-regex (Microsoft)



(b) Poly. K-regex (Cloudflare)



(c) Sample REWZWA



(d) Sample REWBR

Fig. 4: Case studies of applying our techniques to K-regexes and E-regexes. The regexes are: (a) Exponential Microsoft username regex (responsibly disclosed); (b) Cloudflare regex (§II); (c,d) Hand-crafted examples. All K-regexes and REWZWA can be handled in linear-in-$|w|$. For REWBR, memoization reduces the degree of the exponent.

C. Security analysis: Time complexity

This experiment is to confirm our theoretical guarantees.

1) Methodology: Our theorems predict the maximum number of visits to each simulation position, with the effect that the worst-case behavior grows linearly with $|w|$. As shown in §VII (Table III), the selection schemes vary in the bound they offer on the number of visits to each simulation position. For each selection scheme we measured *the total number of simulation position visits* as we increased $|w|$. For a fixed regex, this quantity should grow linearly proportional to $|w|$.

We tested each supported super-linear (K-)regex with problematic inputs of varying lengths. Each regex has an *input signature* consisting of a constant prefix, a list of consecutive "pump strings", and a constant suffix. The more times each pump string is repeated, the more times certain simulation positions will be visited by an un-memoized backtracking search. We varied the length of each input, from 10,000 pumps to 100,000 pumps at intervals of 10,000.

2) Results: Our prototype achieved the linear-in-$|w|$ match complexity we predicted. The total number of simulation position visits grew linearly with $|w|$ in every case. Reductions in matching time are illustrated in Figure 4.

D. Practicality analysis: Space complexity and costs

This experiment assesses the practicality of our approach. We measure the *typical space complexities* of the supported regexes (*i.e.,* values for $|Q_{sel.}|$) and the *actual space costs* of memoization under different configurations.

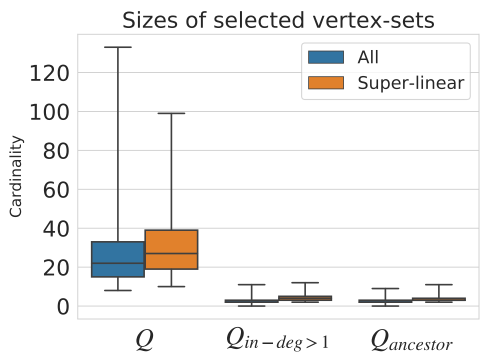


Fig. 5: Sizes of the vertex sets for the selective memoization schemes. Whiskers indicate the (1, 99)th percentiles. Outliers are not shown.

1) Methodology: The **space complexity** of a selective memoization scheme depends on the size of the selected vertex set Q_{sel}. We measured the sizes of Q_{all}, $Q_{in-deg>1}$, and $Q_{ancestor}$ for the supported regexes (Thompson construction).

The **space costs** of a scheme depend on the regex, the input, and the memo function representation. For each supported super-linear regex, we used the "most evil" (highest growth rate) of the inputs from the detectors. We measured the space used under each combination of selection scheme and memo function representation. For this experiment we used strings of length 20 KB to simulate the Stack Overflow scenario (§II-C).

2) Results: **Space complexity:** The distribution of $Q_{sel.}$ sizes under various selection schemes is shown in Figure 5. We note two aspects of the data. First, our data show that full memoization would be costly. The 75th percentile of $|Q| = |Q_{all}|$ is 33. On long inputs, the cost of full memoization would be significant for many regexes. Second, our data show that $|Q| \gg |Q_{in-deg>1}| \approx |Q_{ancestor}|$. Our selective memoization schemes may exhibit space complexities an order of magnitude lower than the full memoization scheme.

Space costs: The observed space costs for each regex-input pair over the nine configurations are shown in in Figure 6. The *memo table representation* cost (blue boxes) can be predicted from Figure 5, at one bit per vertex/offset pair. Surprisingly, the *positive representation* (orange) exhibits higher costs. We found that 30-50% of the possible simulation positions are explored, and the overheads of the hash table outweigh the savings in unfilled entries. Lastly, the *RLE representation* (green) achieves *constant* space costs for most regexes. It has a 95th percentile of 16 runs for $Q_{in-deg>1}$ and 10 runs for $Q_{ancestor}$. This scheme breaks down when run length and visit sequence are mis-aligned. Determining the limitations of RLE is a task we leave for future work.

E. Extensions: Complexity of E-regexes

§IX presented novel algorithms for evaluating two E-regex features (REWZWA and REWBR) within a backtracking framework. In Figure 4 we illustrated the effect of memoization on these features. Unfortunately, we cannot systematically evaluate our approach. The state-of-the-art super-linear regex analyses [22], [25], [37], [89] cannot identify super-linear

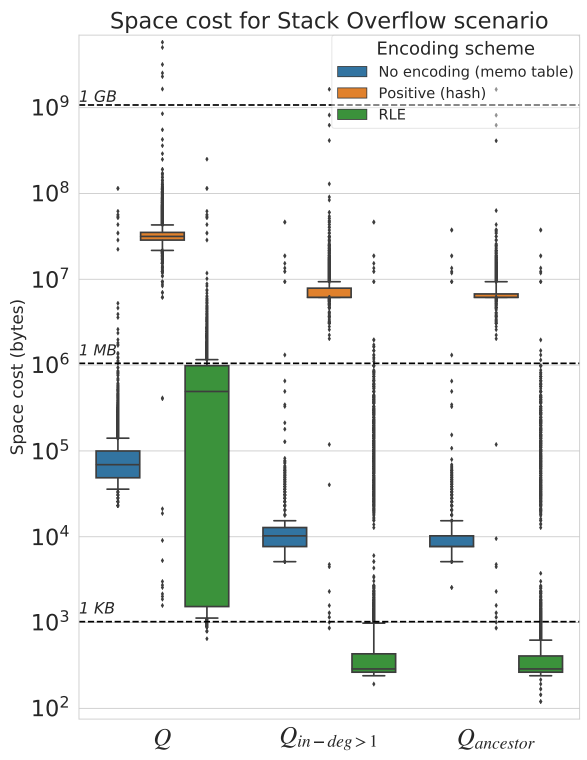


Fig. 6: Space costs to evaluate super-linear K-regexes with "Stack Overflow"-sized input strings (20 KB). Whiskers indicate the (5, 90)th percentiles. Some low outliers are omitted for space.

behavior arising from extended features (§IX).[9] A new regex analysis for E-regexes lies beyond the scope of this work.

However, we can provide measurements to indicate typical parameter values for REWBR. Although several authors have parameterized the worst-case time complexity of REWBR in general [40], [90], no regex corpus has been analyzed to understand typical REWBR use. Per our analysis in §IX-C, we are interested in $|\mathbf{CG}_{BR}|$ and $|Q_{BR}|$. To characterize these costs, we present novel measurements of the use of REWBR.

In the regex corpus, most regexes contain few distinct back-referenced capture groups and few backreference uses. Among the $\approx 3,500$ REWBR in the corpus, 98% contain at most three backreferences ($|BR| \leq 3$), and 98% contain backreferences to at most two distinct capture groups ($|\mathbf{CG}_{BR}| \leq 2$). For typical regexes the worst-case time and space complexity of REWBR is a (relatively) small polynomial of $|w|$.

Following Namjoshi *et al.* [40], we may further characterize the costs of typical REWBR. If a backreferenced capture group has unbounded width (*i.e.,* contains a *), it contributes a factor of $|w|^2$ to time and space complexities. With limited width, it can take on only $|w|$ distinct indices, reducing the exponent. In the corpus, 78% of the REWBR regexes use only fixed-width capture groups, most commonly to find a matching quote, *e.g.,* `('|")\w+\1`. Thus it is possible to address ReDoS due to REWBR in practice, as typical usage entails low-polynomial time and space complexity.

[9]ReScue supports E-regexes [89], but does not find E-regex problems [6].

F. Limitations

The corpus of Davis *et al.* covers regexes as used in general-purpose programming [7]. It may not be representative of domain-specific regexes, *e.g.,* intrusion detection systems.

Our prototype does not support all K-compatible nor E-regex features. Within the full corpus, the causes for lack of support are: 8% features (*e.g.,* lookbehind assertions, named capture groups); plus an additional 3% that use non-ASCII characters (*e.g.,* `\x5z`); plus an additional 5% that our parser mis-handled (*e.g.,* an unmatched `{` is treated as a literal in some regex engines). If the supported and unsupported regexes have different characteristics, then our findings are not completely representative.

XI. Discussion

A. How much space overhead is too much?

Super-linear regex behavior anywhere in the software stack can result in ReDoS. Memoization can eliminate ReDoS, but entails space overhead. If too much space is required, this overhead can convert ReDoS from a time-based attack to a space-based one. Regexes are a programming primitive used in myriad ways, so there is no definitive answer to the question "How much space overhead is too much?" As one data point, Cox proposed 32 KB as a reasonable memory footprint for memoization overheads [15]. In virtualized contexts, embedded devices, or web servers with many clients, even this memory footprint may be problematic. Selective memoization and efficient encoding are key to reducing the cost.

A study of the memory constraints across all ReDoS-vulnerable deployments is beyond the scope of this paper. Our theoretical bounds on space complexity can be used by practitioners to analyze and choose appropriate memory configurations. Our techniques accommodate a wide range of usage in a general-purpose regex engine, with small constant costs for a large fraction of regexes.

B. Incorporating into real-world backtracking regex engines

We believe our techniques can be applied to real-world backtracking regex engines with minimal disruption to the codebase. Conceptually, memoization is an "inline" change for these regex engines. For example, the memoization modifications for our prototype involve a single `if`-condition within the backtracking framework (Listing 2). In real-world regex engines the scope of changes could be slightly more involved. For example, Perl's unsound memoization scheme is updated at three points instead of one to accommodate local optimizations. However, in the many backtracking regex engines we have examined there is a search loop comparable to Listing 1 with clear points where memoization could be introduced. The difficulty has always lain not in introducing memoization, but rather in determining how little can be memoized while ensuring soundness, and in measuring the cost of memoization for typical regexes.

Based on our evaluation, we recommend that the maintainers of backtracking regex engines incorporate a $|Q_{in-deg>1}|$-based memoization scheme for K-regexes. Compared to

$Q_{ancestor}$, the $Q_{in-deg>1}$ selection scheme has a stronger time complexity guarantee (Table III) and similar space costs for typical regexes. However, the outlier values of $Q_{in-degree>1}$ are an order of magnitude larger than those of $Q_{ancestor}$, so $Q_{ancestor}$ might be used as a back-up option in some cases. In either case, an RLE-based memo function representation is quite effective for the common case of the visit vectors associated with catch-all expressions. A memo table may be more useful for other visit vectors.

To understand feasibility, we corresponded with maintainers of four backtracking regex engines. All indicated interest in our technique. None had explored sound selective memoization schemes nor efficient representations. As barriers to adoption, they cited many of the obstacles from §III-D: variations in dialects; the value of backtracking for maintainability and extended feature support; and whether ReDoS defenses should be prioritized. Interestingly, some maintainers also stated that an optimized backtracking engine outperforms Thompson implementations for *common-case* queries. The maintainers' primary questions surrounded the common-case impact, the worst-case guarantees, and the memory footprint. However, most lacked confidence in their benchmark suites to assess performance impacts. This gap shows the value of our corpus approach and suggests avenues for further research.

XII. Other Related Work

We treated many works earlier, but some remain.

A. Other applications of memoization

Beyond Packrat Parsers, memoization has been widely applied, *e.g.,* in functional programming [70], logic programming [71], [91], [92], and dynamic programming [69], [93].

Any algorithm that relies on memoization must address the accompanying space cost. Researchers have explored strategies including caching [15], [80], [94], partial memoization to avoid part (but not all) of the repeated computation [95], [96], and selective memoization to record some (but not all) of the input-output pairs [97]. Selective memoization has been applied heuristically to string parsing [98], but context matters — when CFGs are used to parse software projects (not a latency-critical task), the input is trusted, and the code being parsed is typically short [73]. When researchers have considered security-sensitive parsing contexts, *e.g.,* XML [99]–[101] and JSON [102], they have focused on time rather than space. For regexes both time and space must be considered. Large regexes are deployed on latency-critical paths on long untrusted input. Time costs must be lowered to avoid ReDoS, but this should be done while minimizing space overheads.

B. Domain-specific regex engines

We have described optimization techniques for the general-purpose regex engines provided with programming languages, and evaluated them on measurements of regex usage in a large sample of software modules. Other researchers have tailored regex engine optimizations for specific application domains. Most notably, when regexes are used in intrusion detection systems, they should run at line speed [103]. Researchers have focused on how to make these evaluations efficient and predictable for K-regexes [39], [104]–[107]. Smith *et al.* examined the use of non-selective memoization to suppress redundancy in a more general backtracking predicate search in Snort [108]. Our techniques support general-purpose K-regexes and E-regexes, without requiring substantial changes to existing algorithmic frameworks.

Lastly, while promising, algebraic regex engines *à la* Brzozowski [75], [109]–[111] remain uncommon in practice.

C. Other treatments of E-regexes

§IX presents the first analysis of the problematic current behavior of REWZWA, and the potential REWZWA performance within the backtracking framework. The time complexity we have achieved for REWZWA using memoization is unsurprising from an automata-theoretic perspective. REWZWA semantics are a form of intersection [112], [113], under which regular languages are closed [30]. Our approach obtains linear time complexity without pre-computing the intersection.

Other researchers have shown that REWBR problems are NP-hard. The best known algorithms have exponential worst-case time complexity [44], [52], [85], parameterized in terms of the number of backreferences [39], [40], [90]. Our approach shows how to obtain the same time complexity in legacy regex engines, without leaving the backtracking framework.

XIII. Concluding remarks

Regular expressions are used for string processing in every layer of the software stack. Most programming languages use a backtracking algorithm in their regex engines. Backtracking simplifies engine implementations and gives users expressiveness. But it also introduces exponential worst-case behavior that can lead to regex-based denial of service (ReDoS).

The ReDoS problem has thus far defied a practical solution. There are alternative algorithms with better time complexity guarantees, but they complicate implementations and limit extensibility. Meanwhile, a memoization-based optimization would fit nicely within a backtracking framework, but these have been rejected for their high space complexity.

In this paper we revisited this problem from a data-driven perspective. Inspired by a new large-scale regex corpus, we proposed two selective memoization schemes that offer comparable time complexity guarantees with lower space complexity. For typical regexes these schemes offer an order of magnitude reduction in memoization space costs. Then, leveraging a memo function representation based on regex semantics, memoization space costs can be further reduced to constant for typical regexes. In experiments, our ReDoS defense offers linear-time behavior with constant space costs for 90% of super-linear regexes.

Acknowledgments

We are indebted to Dr. Jamie Jennings for a fruitful conversation, her deep domain expertise, and her exemplary criticism. We also appreciate input from Drs. A. Kazerouni and P. Kumar, and Mssrs. S.A. Hassan and A. Shreepathi.

APPENDIX
PROOFS OF THE SELECTIVE MEMOIZATION THEOREMS

Here we present proofs of Theorems 1 to 3.

A. Definitions

These definitions are used in Theorems 2 and 3.

Definition 1 (Simulation position). *For a regex engine following the backtracking algorithm given in Listing 1, we define a* simulation position $\pi = \langle q \in Q, i \in \mathbb{N}^{|w|} \rangle$ *as one of the possible simulation positions on which the* `recurse` *function is called. Two simulation positions are* different *if they differ in the 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 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 positions 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.*

We introduce the following concept to refine the statement of Theorem 3 from that given in §VII.

Definition 3 (Bounded ambiguity). *Let A be an ϵ-free NFA. We define its* bounded ambiguity *as:*

$$\textit{boundedAmbiguity}(A) = \max_{0 \le i \le |Q|} \Big(\max_{s,t \in Q} \Big(\\ \textit{\# distinct simulation paths } s \rightsquigarrow t \textit{ of length } i \\ \Big)\Big)$$

Note that *boundedAmbiguity*(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 *boundedAmbiguity*(A) captures the maximum possible ambiguity for strings of length no more than $|Q|$.

B. 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 [31], [114].

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.

C. Theorems and proofs

Theorem 1. *Let the memo function track simulation positions involving all M-NFA vertices, $Q_{all} = Q$. Then the M-NFA is unambiguous — every simulation position $\pi = \langle q, i\rangle$ will be visited at most once.*

Proof. This result follows trivially from the definition of δ_M, the memoized transition function. See §VII-A. □

Theorem 2. *Let the memo function track simulation positions involving the M-NFA vertices with in-degree greater than one, $Q_{in-deg>1}$. Then the M-NFA is unambiguous — every simulation position $\pi = \langle q, i\rangle$ will be visited at most once.*

Put simply, this theorem states that all ambiguity is the result of joining paths. Without splits, only one path is possible; without joins, different paths cannot lead to ambiguity because of the q_F assumption.

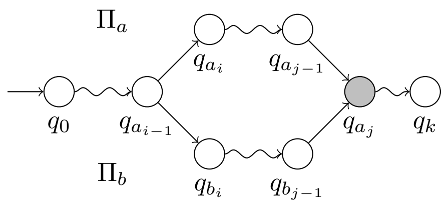


Fig. 7: Illustration for the proof of Theorem 2. Vertex $q_{a_j} = q_{b_j}$ must have in-degree > 1 because it is the first point of convergence after the split at vertex $q_{a_{i-1}} = q_{b_{i-1}}$.

Proof. The proof proceeds by contradiction. Suppose the $Q_{in-deg>1}$ memoization scheme is employed but some simulation position π_k is visited twice. This means an M-NFA simulation traverses different simulation paths Π_a and Π_b,

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

such that (Figure 7):

1) The paths diverge. At some $i < k$, $\pi_{a_i} \ne \pi_{b_i}$, *e.g.,* at a point of non-determinism.
2) The paths converge. There is some smallest j, $i < j \le k$, 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 $\pi_{a_k} \ne \pi_{b_k}$. 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 $\pi_{a_{j-1}} \ne \pi_{b_{j-1}}$, we have $q_{a_{j-1}} \ne q_{b_{j-1}}$, and so the in-degree of $q_{a_j} > 1$.

But if $q_{a_j} \in Q_{in-deg>1}$, 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 simulation position $\pi_{a_j} = \pi_{b_j}$ more than once along these paths, because Π_b would terminate at $\pi_{b_{j-1}}$.

□

Next we re-state Theorem 3, refined using the definition of $boundedAmbiguity(A)$.

Theorem 3 (Refined). *Let the memo function track simulation positions involving the M-NFA vertices that are "cycle ancestors", $Q_{ancestor}$, i.e., the automaton vertices to which any back-edges in a topological sort from q_0 are directed. Then the M-NFA is finitely ambiguous. A simulation position involving a vertex t will be visited at most $N = f(Q)$ times, where:*

1) $N = 1$ if $t \in Q_{ancestor}$;
2) $N = boundedAmbiguity(A)$ if t is not reachable from a cycle ancestor;
3) $N = |Q_{ancestor}| \times |Q| \times boundedAmbiguity(A)$ 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. Recall the illustration in Figure 3 (b): ambiguity remains possible but is limited. An example of the theorem calculations is given in Figure 8.

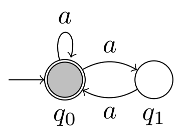


Fig. 8: Illustration of the calculations for Theorem 3. This is an ϵ-free M-NFA for the regex $(a|aa)*$. Here, $|Q| = 2$ and $|Q_{ancestor}| = 1$. There are two distinct $q_0 \rightsquigarrow q_0$ paths of length two, so $boundedAmbiguity(A) = 2$. The theorem gives a bound of $1 * 2 * 2 = 4$ visits to any simulation position, although this bound is not realized in this automaton because the two paths share a memoized vertex q_0.

Proof. Choose a target simulation position $\pi = \langle t \in Q, i \rangle$. We will show the visit bound for each case.

Case $t \in Q_{ancestor}$**:** If $t \in Q_{ancestor}$, then the memo function ensures that π is visited at most once.

Case $q \in Q_{ancestor} \not\rightsquigarrow t$**:** 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 $boundedAmbiguity(A)$. This result also covers the case when $Q_{ancestor} = \emptyset$.

Case $q \in Q_{ancestor} \rightsquigarrow t$**:** We partition the space on i.

Clearly, if $i \leq |Q|$ then π can be visited at most $boundedAmbiguity(A)$ 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 $q \in Q_{ancestor}$, that simulation path must terminate within $|Q|$ steps. This is because the back-edges to $Q_{ancestor}$ are the only means of introducing a cycle, and without further cycles a simulation path must come to an end.

Now then, what if $i > |Q|$? Then the distinct simulation paths to π must all include some "cycle" simulation position $\pi' = \langle q \in Q_{ancestor}, j \rangle$ 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 $|Q_{ancestor}| \times |Q|$ possible cycle positions π', so all of the distinct simulation paths to π must share at most $|Q_{ancestor}| \times |Q|$ distinct simulation path prefixes. From these π', each simulation prefix may diverge up to $boundedAmbiguity(A)$ times to reach π. Multiplying, we obtain an upper bound of $|Q_{ancestor}| \times |Q| \times boundedAmbiguity(A)$ distinct simulation paths that can reach π.

□

References

[1] J. E. Friedl, *Mastering regular expressions.* O'Reilly Media, Inc., 2002.
[2] H. Spencer, "A regular-expression matcher," in *Software solutions in C*, 1994, pp. 35–71.
[3] R. Cox, "Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, PHP, Python, Ruby, ...)," 2007.
[4] C. Chapman and K. T. Stolee, "Exploring regular expression usage and context in Python," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2016.
[5] J. C. Davis, D. Moyer, A. M. Kazerouni, and D. 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.
[6] J. C. Davis, C. A. Coghlan, F. Servant, and D. 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.
[7] J. C. Davis, L. G. Michael IV, C. A. Coghlan, F. Servant, and D. 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.
[8] S. A. Crosby and D. S. Wallach, "Denial of Service via Algorithmic Complexity Attacks," in *USENIX Security*, 2003.
[9] S. Crosby and T. H. E. U. Magazine, "Denial of service through regular expressions," in *USENIX Security work in progress report*, vol. 28, no. 6, 2003.
[10] J. Goyvaerts, "Runaway Regular Expressions: Catastrophic Backtracking," 2003. [Online]. Available: http://www.regular-expressions.info/catastrophic.html
[11] A. Roichman and A. Weidman, "VAC - ReDoS: Regular Expression Denial Of Service," *Open Web Application Security Project (OWASP)*, 2009.
[12] C.-A. Staicu and M. Pradel, "Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers," in *USENIX Security Symposium (USENIX Security)*, 2018.
[13] S. Exchange, "Outage postmortem," http://web.archive.org/web/20180801005940/http://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016, 2016.

[14] 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/.

[15] R. Cox, "Regular Expression Matching in the Wild," 2010. [Online]. Available: https://swtch.com/~rsc/regexp/regexp3.html

[16] D. E. Knuth, J. Morris, James H, and V. R. Pratt, "Fast pattern matching in strings," *SIAM Journal on Computing*, vol. 6, no. 2, pp. 323–350, 1977.

[17] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Communications of the ACM (CACM)*, vol. 18, no. 6, pp. 333–340, 1975.

[18] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM (CACM)*, vol. 20, no. 10, pp. 762–772, 1977.

[19] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Saner: Composing static and dynamic analysis to validate sanitization in web applications," in *IEEE Symposium on Security and Privacy (IEEE S&P)*, 2008, pp. 387–401.

[20] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *International Conference on Software Engineering (ICSE)*, 2008.

[21] D. Bates, A. Barth, and C. Jackson, "Regular expressions considered harmful in client-side XSS filters," in *The Web Conference (WWW)*, 2010.

[22] A. Rathnayake and H. Thielecke, "Static Analysis for Regular Expression Exponential Runtime via Substructural Logics," Tech. Rep., 2014.

[23] A. Ojamaa and K. Duuna, "Assessing the security of Node.js platform," in *International Conference for Internet Technology and Secured Transactions (ICITST)*, 2012.

[24] J. C. Davis, E. R. Williamson, and D. 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.

[25] V. Wüstholz, O. Olivo, M. J. H. Heule, and I. 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.

[26] L. G. Michael IV, J. Donohue, J. C. Davis, D. Lee, and F. Servant, "Regexes are Hard : Decision-making, Difficulties, and Risks in Programming Regular Expressions," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019.

[27] 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.

[28] S. C. Kleene, "Representation of events in nerve nets and finite automata," *Automata Studies*, pp. 3–41, 1951.

[29] W. S. McCulloch and W. Pitts, "A Logical Calculus of the Ideas Immanent in Nervous Activity," *The Bulletin of Mathematical Biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[30] M. Rabin and D. Scott, "Finite Automata and their Decision Problems," *IBM Journal of Research and Development*, vol. 3, pp. 114–125, 1959.

[31] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Automata theory, languages, and computation*, 2006, vol. 24.

[32] K. Thompson, "Regular Expression Search Algorithm," *Communications of the ACM (CACM)*, 1968.

[33] R. McNaughton and H. Yamada, "Regular Expressions and State Graphs for Automata," *IRE Transactions on Electronic Computers*, vol. 5, pp. 39–47, 1960.

[34] J. Earley, "An Efficient Context-Free Parsing Algorithm," *Communications of the ACM*, vol. 13, no. 2, pp. 94–102, 1970.

[35] C. Brabrand, R. Giegerich, and A. Møller, "Analyzing ambiguity of context-free grammars," *Science of Computer Programming*, vol. 75, no. 3, pp. 176–191, 2010.

[36] R. Book, S. Even, S. Greibach, and G. Ott, "Ambiguity in Graphs and Expressions," *IEEE Transactions on Computers*, vol. C-20, no. 2, pp. 149–153, 1971.

[37] N. Weideman, B. van der Merwe, M. Berglund, and B. Watson, "Analyzing matching time behavior of backtracking regular expression matchers by using ambiguity of nfa," in *International Conference on Implementation and Application of Automata*. Springer, 2016, pp. 322–334.

[38] V. Laurikari, "NFAs with tagged transitions, their conversion to deterministic automata and application to regular expressions," *International Symposium on String Processing and Information Retrieval (SPIRE)*, pp. 181–187, 2000.

[39] M. Becchi and P. Crowley, "Extending finite automata to efficiently match perl-compatible regular expressions," in *ACM International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*, 2008.

[40] K. Namjoshi and G. Narlikar, "Robust and fast pattern matching for intrusion detection," *IEEE INFOCOM*, 2010.

[41] C. Câmpeanu, K. Salomaa, and S. Yu, "A formal study of practical regular expressions," *International Journal of Foundations of Computer Science*, vol. 14, no. 6, pp. 1007–1018, 2003.

[42] C. Câmpeanu and N. Santean, "On the intersection of regex languages with regular languages," *Theoretical Computer Science*, vol. 410, no. 24-25, pp. 2336–2344, 2009.

[43] M. Berglund and B. van der Merwe, "On the Semantics of Regular Expression parsing in the Wild," *Theoretical Computer Science*, vol. 578, pp. 292–304, 2015.

[44] M. Berglund and B. V. D. Merwe, "Regular Expressions with Backreferences," in *Prague Stringology*, 2017, pp. 30–41.

[45] M. Berglund and B. van der Merwe, "Regular Expressions with Backreferences Re-examined," in *Prague Stringology*, 2017, pp. 30–41.

[46] M. Berglund, W. Bester, and B. van der Merwe, "Formalising boost posix regular expression matching," in *International Colloquium on Theoretical Aspects of Computing*. Springer, 2018, pp. 99–115.

[47] A. Birman and J. D. Ullman, "Parsing Algorithms With Backtrack," *Symposium on Switching and Automata Theory (SWAT)*, 1970.

[48] L. Wall, "Perl," 1988.

[49] P. Hazel, "PCRE - Perl Compatible Regular Expressions," 1997.

[50] IEEE and T. O. Group, *The Open Group Base Specifications Issue 7, 2018 edition*, 2018.

[51] J. Goyvaerts, "A list of popular tools, utilities and programming languages that provide support for regular expressions, and tips for using them," 2016. [Online]. Available: https://www.regular-expressions.info/tools.html

[52] A. V. Aho, *Algorithms for finding patterns in strings*. Elsevier, 1990, ch. 5, pp. 255–300.

[53] T. R. P. Developers, "regex - rust," https://docs.rs/regex/1.1.0/regex/.

[54] Google, "regexp - go," https://golang.org/pkg/regexp/.

[55] PerlMonks, "Perl regexp matching is slow??" https://perlmonks.org/?node_id=597262.

[56] pam, "Issue 287: Long, complex regexp pattern (in webkit layout test) hangs," https://bugs.chromium.org/p/v8/issues/detail?id=287.

[57] S. Toub, "Regex performance improvements in .net 5," https://devblogs.microsoft.com/dotnet/regex-performance-improvements-in-net-5/, 2020.

[58] P. Wang and K. T. Stolee, "How well are regular expressions tested in the wild?" in *Foundations of Software Engineering (FSE)*, 2018.

[59] Gruber, Jakob, "Speeding up v8 regular expressions," https://v8.dev/blog/speeding-up-regular-expressions, 2017.

[60] Thier, Patrick and Peško, Ana, "Improving v8 regular expressions," https://v8.dev/blog/regexp-tier-up, 2019.

[61] Microsoft, "Regex.matchtimeout property," https://docs.microsoft.com/en-us/dotnet/api/system.text.regularexpressions.regex.matchtimeout, 2012.

[62] "Php: Runtime configuration," https://www.php.net/manual/en/pcre.configuration.php.

[63] P. Monks, "Perl's complex regular subexpression recursion limit," https://www.perlmonks.org/?node_id=810857, 2009.

[64] S. Peter, A. Baumann, T. Roscoe, P. Barham, and R. Isaacs, "30 seconds is not enough!" in *European Conference on Computer Systems (EuroSys)*, 2008.

[65] B. Cody-Kenny, M. Fenton, A. Ronayne, E. Considine, T. McGuire, and M. O'Neill, "A search for improved performance in regular expressions," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2017, pp. 1280–1287.

[66] B. Van Der Merwe, N. Weideman, and M. Berglund, "Turning evil regexes harmless," in *Proceedings of the South African Institute of Computer Scientists and Information Technologists (SAICSIT)*, 2017.

[67] C. Chapman, P. Wang, and K. T. Stolee, "Exploring Regular Expression Comprehension," in *Automated Software Engineering (ASE)*, 2017.

[68] D. Michie, ""Memo" Functions and Machine Learning," *Nature*, 1968.

[69] R. Bellman, “Dynamic programming,” *Science*, vol. 153, no. 3731, pp. 34–37, 1966.

[70] D. A. Turner, “The Semantic Equivalence of Applicative Languges,” in *Conference on Functional Programming Languages and Computer Architecture*, 1981, pp. 85–92.

[71] H. Tamaki and T. Sato, “OLD resolution with tabulation,” in *International Conference on Logic Programming*, 1986.

[72] P. Norvig, “Techniques for Automatic Memoization with Applications to Context-Free Parsing,” *Computational Linguistics*, vol. 17, no. 1, pp. 91–98, 1991.

[73] B. Ford, “Packrat Parsing: Simple, Powerful, Lazy, Linear Time,” in *The International Conference on Functional Programming (ICFP)*, vol. 37, no. 9, 2002. [Online]. Available: http://arxiv.org/abs/cs/0603077

[74] ——, “Parsing Expression Grammars: A Recognition-Based Syntactic Foundation,” in *Principles of Programming Languages (POPL)*. ACM, 2004, p. 354.

[75] M. Might, D. Darais, and D. Spiewak, “Parsing with derivatives: A functional pearl,” in *The International Conference on Functional Programming (ICFP)*, 2011.

[76] M. Berglund, F. Drewes, and B. Van Der Merwe, “Analyzing Catastrophic Backtracking Behavior in Practical Regular Expression Matching,” *EPTCS: Automata and Formal Languages 2014*, vol. 151, pp. 109–123, 2014.

[77] N. Schwarz, A. Karper, and O. Nierstrasz, “Efficiently extracting full parse trees using regular expressions with capture groups,” *PeerJ Preprints*, 2015.

[78] A. W. Burks and H. Wang, “The Logic of Automata - part 2,” *Journal of the Association for Computing Machinery (JACM)*, vol. 4, no. 3, pp. 279–297, 1957.

[79] C. Allauzen, M. Mohri, and A. Rastogi, “General Algorithms for Testing the Ambiguity of Finite Automata,” in *International Conference on Developments in Language Theory*, 2008.

[80] J. Hughes, “Lazy memo-functions,” in *Conference on Functional Programming Languages and Computer Architecture*, 1985.

[81] M. Becchi, “Data Structures, Algorithms, and Architectures for Efficient Regular Expression Evaluation,” Ph.D. dissertation, 2009.

[82] A. H. Robinson and C. Cherry, “Results of a Prototype Television Bandwidth Compression Scheme,” *Proceedings of the IEEE*, vol. 55, no. 3, pp. 356–364, 1967.

[83] A. Mohapatra and M. Genesereth, “Incrementally maintaining run-length encoded attributes in column stores,” *ACM International Conference Proceeding Series*, pp. 146–154, 2012.

[84] D. D. Freydenberger and M. L. Schmid, “Deterministic regular expressions with back-references,” *Journal of Computer and System Sciences*, vol. 105, pp. 1–39, 2019. [Online]. Available: https://doi.org/10.1016/j.jcss.2019.04.001

[85] C. CÂmpeanu, K. Salomaa, and S. Yu, “A Formal Study of Practical Regular Expressions,” *International Journal of Foundations of Computer Science*, vol. 14, no. 06, pp. 1007–1018, 2003.

[86] “re1: A simple regular expression engine, easy to read and study.” https://code.google.com/archive/p/re1.

[87] Hanson, Troy D, “uthash: A hash table for c structures,” http://troydhanson.github.com/uthash/, 2018.

[88] Biggers, Eric, “avl_tree: High performance c implementation of avl trees,” https://github.com/ebiggers/avl_tree, 2016.

[89] Y. Shen, Y. Jiang, C. Xu, P. Yu, X. Ma, and J. Lu, “ReScue: Crafting Regular Expression DoS Attacks,” in *Automated Software Engineering (ASE)*, 2018.

[90] M. L. Schmid, “Regular Expressions with Backreferences: Polynomial-Time Matching Techniques,” 2019. [Online]. Available: http://arxiv.org/abs/1903.05896

[91] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift, and D. S. Warren, “Efficient model checking using tabled resolution,” in *International Conference on Computer Aided Verification (CAV)*, 1997.

[92] P. V. Beek, “Backtracking Search Algorithms,” in *Handbook of Constraint Programming*, 2006, ch. 4, pp. 85–134.

[93] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.

[94] B. Cook and J. Launchbury, “Disposable memo functions,” in *Haskell Workshop*, 1997.

[95] T. Amtoft and J. L. Träff, “Partial memoization for obtaining linear time behavior of a 2DPDA,” *Theoretical Computer Science*, vol. 98, no. 2, pp. 347–356, 1992.

[96] L. Ziarek, K. C. Sivaramakrishnan, and S. Jagannathan, “Partial memoization of concurrency and communication,” in *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2009, pp. 161–172.

[97] U. A. Acar, G. E. Blelloch, and R. Harper, “Selective memoization,” in *Principles of Programming Languages (POPL)*, 2003.

[98] R. Becket and Z. Somogyi, “DCGs + Memoing = Packrat parsing but is it worth it?” in *International Symposium on Practical Aspects of Declarative Languages*, 2008.

[99] B. Sullivan, “XML Denial of Service Attacks and Defenses,” *MSDN Magazine*, 2009.

[100] C. Späth, C. Mainka, V. Mladenov, and J. Schwenk, “SoK: XML Parser Vulnerabilities,” in *USENIX Workshop on Offensive Technologies (WOOT)*, 2016. [Online]. Available: http://www.w3.org/2001/XInclude

[101] N. Chida, Y. Kawakoya, D. Ikarashi, K. Takahashi, and K. Sen, “Is stateful packrat parsing really linear in practice? A counter-example, an improved grammar, and its parsing algorithms,” in *International Conference on Compiler Construction (CC)*, 2020.

[102] Y. Li, N. R. Katsipoulakis, B. Chandramouli, J. Goldstein, and D. Kossmann, “Mison: A Fast JSON Parser for Data Analytics,” *Very Large DataBases (VLDB)*, vol. 10, no. 10, pp. 1118–1129, 2017.

[103] C. Xu, S. Chen, J. Su, S. M. Yiu, and L. C. Hui, “A Survey on Regular Expression Matching for Deep Packet Inspection: Applications, Algorithms, and Hardware Platforms,” *IEEE Communications Surveys and Tutorials (CSUR)*, vol. 18, no. 4, pp. 2991–3029, 2016.

[104] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese, “Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia,” in *Symposium on Architecture For Networking And Communications Systems (ANCS)*, vol. 25, 2007, p. 155.

[105] R. Smith, C. Estan, S. Jha, and S. Kong, “Deflating the Big Bang: Fast and Scalable Deep Packet Inspection with Extended Finite Automata,” in *SIGCOMM*, 2008.

[106] R. Smith, C. Estan, and S. Jha, “XFA: Faster signature matching with extended automata,” *Proceedings - IEEE Symposium on Security and Privacy*, pp. 187–201, 2008.

[107] X. Wang, Y. Hong, H. Chang, K. Park, G. Langdale, J. Hu, and H. Zhu, “Hyperscan: A Fast Multi-pattern Regex Matcher for Modern CPUs,” in *Networked Systems Design and Implementation (NSDI)*, 2019, pp. 631–648.

[108] R. Smith, C. Estan, and S. Jha, “Backtracking Algorithmic Complexity Attacks Against a NIDS,” in *Annual Computer Security Applications Conference (ACSAC)*, 2006, pp. 89–98.

[109] J. A. Brzozowski, “Derivatives of Regular Expressions,” *Journal of the Association for Computing Machinery*, vol. 11, no. 4, pp. 481–494, 1964.

[110] S. Owens, J. Reppy, and A. Turon, “Regular-expression derivatives re-examined,” *Journal of Functional Programming*, vol. 19, no. 2, pp. 173–190, 2009.

[111] O. Saarikivi, M. Veanes, T. Wan, and E. Xu, “Symbolic regex matcher,” in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2019.

[112] A. Morihata, “Translation of regular expression with lookahead into finite state automaton,” *Computer Software*, vol. 29, no. 1, 2012.

[113] T. Miyazaki and Y. Minamide, “Derivatives of regular expressions with lookahead,” *Journal of Information Processing*, vol. 27, pp. 422–430, 2019.

[114] M. Sipser, *Introduction to the Theory of Computation*. Thomson Course Technology Boston, 2006, vol. 2.