USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Regular Expression Denial of Service Induced by Backreferences

Yichen Liu, *Stony Brook University;* Berk Çakar, *Purdue University;* Aman Agrawal and Minseok Seo, *Stony Brook University;* James C. Davis, *Purdue University;* Dongyoon Lee, *Stony Brook University*

https://www.usenix.org/conference/usenixsecurity26/presentation/liu-yichen

This paper is included in the Proceedings of the 35th USENIX Security Symposium.

August 12–14, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-58-8

Open access to the Proceedings of the 35th USENIX Security Symposium is sponsored by

جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

Regular Expression Denial of Service Induced by Backreferences

Yichen Liu[1], Berk Çakar[2], Aman Agrawal[1,3], Minseok Seo[1,4], James C. Davis[2], and Dongyoon Lee[1]

[1]Stony Brook University [2]Purdue University

Abstract

This paper presents the a systematic and theoretical study of denial-of-service vulnerabilities in Regular Expressions with Backreferences (REwB). We introduce the Two-Phase Memory Automaton (2PMFA), an automaton model that precisely captures REwB semantics. Using this model, we derive necessary conditions under which backreferences induce super-linear backtracking runtime, even when sink ambiguity is linear, a regime where existing detectors report no vulnerability. Based on these conditions, we identify three vulnerability patterns and validate them in practice. Using the Snort intrusion detection ruleset, our evaluation identifies 48 previously unknown REwB vulnerabilities with quadratic or worse runtime. We further demonstrate practical exploits against Snort, including slowing rule evaluation by 0.6-1.2 seconds and bypassing alerts by triggering PCRE's matching limit.

1 Introduction

Regular expressions (regexes) are a foundational mechanism for pattern matching and input validation across software systems. They are widely used to validate and filter untrusted input, including network intrusion detection systems [18, 37], web application firewalls [17, 21, 22], and server-side input validators [3, 10]. However, many modern regex engines rely on backtracking-based matching that can exhibit super-linear time complexity on certain regex pattern and input pairs [7].

This algorithmic complexity vulnerability, known as Regular Expression Denial of Service (ReDoS) [7, 12, 13], has caused significant real-world outages, including a 34-minute downtime of Stack Overflow in 2016 [20] and a 27-minute global outage of Cloudflare services in 2019 [23]. The prevalence of ReDoS vulnerabilities across software ecosystems is well-established through prior empirical studies (*e.g.,* [15, 28–30, 39, 41, 47]), which have collectively identified hundreds of vulnerable regex patterns in production code.

While prior work provides evidence that ReDoS vulnerabilities are widespread, the existing theoretical basis for ReDoS focuses on Kleene regexes (K-regexes)—regexes constructed using only concatenation, alternation, and repetition operators—and their corresponding Non-deterministic Finite Automata (NFAs) [2,45–47]. Yet, modern regex engines, such as those used in Python, Perl, PHP (uses PCRE or PCRE2), Ruby (uses Onigmo), and Java, commonly support backreferences and other extended constructs [4, 7], which cannot be represented by NFAs and therefore fall outside the scope of existing NFA-based complexity analyses. Prior work has examined the expressive power of regexes with backreferences (REwB) [6,8,33,34], and it is known that regex matching with backreferences is NP-complete [1]. However, this worst-case complexity result does not characterize which specific REwB patterns lead to super-linear backtracking behavior, nor does it provide practical algorithms for detecting such patterns or constructing attack inputs. The prevalence of such patterns in real-world deployments also remains unknown.

To address this gap, we extend ReDoS theory and detection to support REwB, providing a systematic investigation of ReDoS vulnerabilities caused by backreferences. We introduce Two-Phase Memory Finite Automaton (2PMFA), a new model that faithfully captures real-world REwB semantics, including self-references. Using 2PMFA, we formally show that for certain REwB patterns, a single backreference evaluation can incur non-$O(1)$ cost. When combined with a non-$O(1)$ number of backreference evaluations, this leads to super-linear runtime behavior that fundamentally differs from that of K-regexes.

Building on these insights, we formally derive necessary conditions under which REwB induce super-linear backtracking runtime due to non-$O(1)$ per-backreference cost and a non-$O(1)$ number of backreference evaluations. By combining these conditions, we introduce three ReDoS-vulnerable REwB patterns for the first time. Based on the patterns, we develop a ReDoS detector for REwB as well as attack-automaton generators. Collectively, these contributions enable the identification of REwB-related ReDoS vulnerabilities that were

3. Affiliated with Google at the time of publication.
4. Affiliated with Nexien at the time of publication.

previously invisible to existing detectors.

We evaluate our detection framework on 11K+ Snort [37] intrusion detection rules, and uncover 48 previously undocumented REwB-induced ReDoS vulnerabilities. Through dynamic analysis, we validate our detector's findings and demonstrate that exploiting backreferences in combination with infinite degree of ambiguity (IDA) patterns produces substantially larger slowdowns and enables ReDoS attacks with shorter inputs compared to exploiting IDA alone. Finally, we present four concrete exploits against Snort, together with malicious input strings and realistic attack scenarios, that either slow rule evaluation by 0.6–1.2 seconds or bypass alerts by triggering PCRE's matching limit.

In summary, this paper makes the following contributions:

- We introduce the Two-Phase Memory Finite Automaton (2PMFA), a new model that captures REwB semantics, and enables formal complexity analysis of backreference runtime under backtracking (§4).
- We prove necessary conditions under which REwB incur super-linear time complexity in a manner that fundamentally differs from K-regexes (§5).
- We proposed one of the first theories to characterize backreference-induced ReDoS patterns, each of which is sufficient to induce super-linear time complexity. We develop a ReDoS-vulnerable REwB detector and an attack-automaton generator (§6).
- Our evaluation on Snort's intrusion detection rules uncovers previously unknown 48 REwB-induced ReDoS vulnerabilities and demonstrates realistic exploit scenarios against Snort, highlighting our findings' real-world impact (§7).

Significance: Our study systematically analyzes ReDoS vulnerabilities induced by backreferences. We show that backreferences introduce a fundamentally distinct source of super-linear behavior that existing NFA-based detectors cannot capture. We recommend that developers and operators of security-critical systems audit REwB using our patterns. More broadly, we advocate for automaton models that incorporate non-$O(1)$ transition costs, such as 2PMFA, as a foundation for analyzing irregular regex features.

2 Background

This section introduces the regex constructs central to our work (§2.1), then reviews the algorithmic and complexity foundations of ReDoS that we later extend (§2.2).

2.1 Regular Expressions and Backreferences

A regular expression (regex) r formally describes a language—a set of strings over an alphabet Σ—through concatenation rr, alternation $r|r$, and repetition r^*, with parentheses (r) for grouping [26]. For example, the regex `/ab*c/` matches `'abbbc'` but rejects `'aabbc'`. Regexes constructed solely from these operations are called *Kleene regexes (K-regexes)*.

Definition 1: K-regexes and REwB

The syntax of K-regexes over an alphabet Σ is given by the six constructs below. REwB are obtained by extending this syntax with the two highlighted constructs: capturing groups and backreferences.

$r ::=$	rr	*concatenation*	$\mid r^*$	*repetition*
$\mid$	$r \mid r$	*alternation*	$\mid (r)$	*grouping*
$\mid$	σ	*symbol* ($\sigma \in \Sigma$)	$\mid \varepsilon$	*empty string*
$\mid$	$(\!\lvert_i r \rvert\!)_i$	*capturing group*	$\mid \backslash i$	*backreference*

Irregularity and Backreferences Modern engines extend K-regexes to *Extended regexes (E-regexes)* with (1) syntactic sugar (*e.g.,* one-or-more-repetition r^+, bounded quantifier $r^{\{m,m\}}$, and character classes $[\sigma\text{-}\sigma]$), and (2) new constructions like backreferences and lookarounds. While features in the first category preserve regularity, those in the second increase expressive power beyond regular languages, making E-regexes potentially *irregular*. [4, 8].

This paper focuses on *Regexes with Backreferences (REwB)*, which refer to K-regexes plus backreferences. In REwB, a capturing group $(\!\lvert_i r \rvert\!)_i$ records the substring matched by r, and a subsequent backreference $\backslash i$ matches that substring. For example, the regex $/(\!\lvert_1 \texttt{a}^* \rvert\!)_1 \texttt{b} \backslash 1/$ captures a sequence of `'a'`s in group 1 via $(\!\lvert_1 \texttt{a}^* \rvert\!)_1$, then requires the backreference $\backslash 1$ to match the identical sequence. This regex accepts `'aaabaaa'` (where `'aaa'` is captured and repeated) but rejects `'aaabaa'` (captured `'aaa'` $\neq$ trailing `'aa'`).

A special case is *self-backreferences* [5], in which $\backslash i$ occurs within its own capturing group $(\!\lvert_i \cdots \rvert\!)_i$; at each iteration the backreference matches the substring captured in the *preceding* iteration. Figure 1 illustrates the matching behavior of self-backreferences for the regex $(\!\lvert_1 \backslash 1 \texttt{b} \mid \texttt{a} \rvert\!)_1^*$ on input `'aababb'`. The capture table (right column) stores the most recently committed substring for each group. Initially, group 1 is empty ($\varnothing$), so $\backslash 1$ fails and `'a'` is matched via the right branch ①. In subsequent iterations, $\backslash 1$ matches the previously captured substring: `'a'` at step ②, then `'ab'` at step ③, and so on, enabling the pattern to match progressively longer substrings.

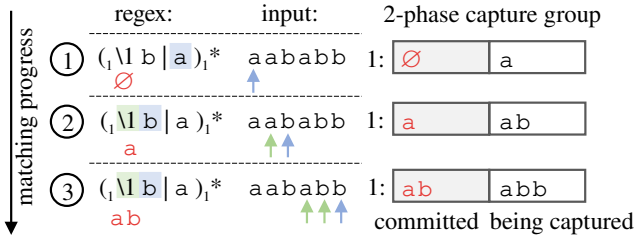

Figure 1: Matching $/(\!\lvert_1 \backslash 1 \texttt{b} \mid \texttt{a} \rvert\!)_1^*/$ against `'aababb'`. The capture table stores the *committed* value from the prior iteration alongside the substring *being captured* in the current iteration.

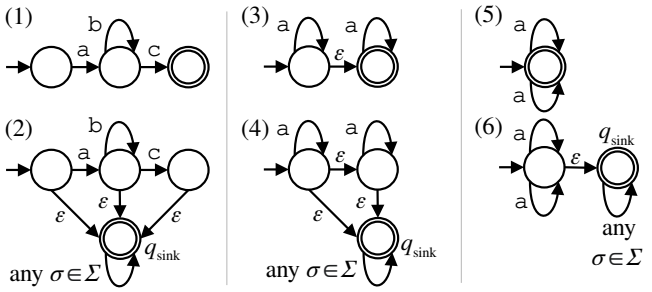

Figure 2: (1) NFA and (2) Sink-NFA of regex /ab*c/. (3) NFA and (4) Sink-NFA of regex /a*b*/. (5) NFA and (6) Sink-NFA of regex /(a|a)*/.

Automata Equivalence. K-regexes are equivalent in expressive power to regular languages and can be converted to non-deterministic finite automata (NFAs) via the Thompson-McNaughton-Yamada construction [31, 43], and vice versa. An NFA is a 5-tuple $A = (Q, \Sigma, \Delta, q_0, F)$, where Q is a finite set of states, Σ is the input alphabet, $\Delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$ is the transition function, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of accepting states. Figure 2(1) shows the NFA for regex /ab*c/: starting from q_0, the automaton consumes 'a', loops on 'b', then accepts after 'c'.

REwBs cannot be converted to NFAs because they are irregular. More specifically, matching a backreference requires comparing the current input against a previously captured substring of arbitrary length. Such unbounded dependency cannot be represented by memoryless finite-state automatons.

2.2 ReDoS and Regex Complexity

Regular Expression Denial of Service (ReDoS) [7, 12, 13] is an algorithmic complexity attack in which a crafted input triggers high runtime complexity—polynomial or exponential in input length—in a backtracking-based regex engine. Such inputs can cause service degradation or outage, as seen in notable incidents at Stack Overflow [20] and Cloudflare [23].

2.2.1 Matching Algorithms

A *regex engine* compiles a pattern into an intermediate representation and simulates it against input strings. Two principal algorithms underlie most implementations [7]. *Thompson's algorithm* [43] performs a breadth-first, lockstep simulation that tracks all active NFA states simultaneously, guaranteeing $O(|Q|^2 \cdot |s|)$ time—linear in input length—but cannot support features requiring memory of previously matched content. *Spencer's algorithm* [40] performs a depth-first search, exploring one path at a time and backtracking on failure; it accommodates the full E-regex feature set (*e.g.,* backreferences, lookarounds) but exhibits worst-case exponential time complexity $|Q|^{O(|s|)}$ on pathological inputs.

Most regex engines—including Perl, Python `re`, Java `java.util.regex`, PCRE, and Onigmo—adopt Spencer-style backtracking because it supports backreferences. Engines prioritizing worst-case performance (*e.g.,* Go, Rust) use Thompson-style matching and cannot handle backreferences.

2.2.2 Ambiguity

For an NFA A, the *degree of ambiguity* [45] with respect to a string s, denoted $\mathrm{AbgS}(A, s)$, is the number of distinct accepting paths for s. The degree of ambiguity for strings of length n is $\mathrm{AbgN}(A, n) = \max_{s \in \Sigma^n} \mathrm{AbgS}(A, s)$, and the overall degree of ambiguity is $\mathrm{Abg}(A) = \max_{n \in \mathbb{N}} \mathrm{AbgN}(A, n)$.

The NFA for /ab*c/ in Figure 2(1) has *finite* ambiguity: for any input 'ab^{n}c', exactly one accepting path traverses the 'b'-loop n times. In contrast, NFAs can exhibit *infinite degree of ambiguity* (IDA), where $\mathrm{Abg}(A) = \infty$. The NFA for /a*a*/ in Figure 2(3) has n accepting paths for input 'a^n'—any partition between the two loops yields a valid match. The NFA for /(a|a)*/ in Figure 2(5) has 2^n accepting paths, since each 'a' can match either branch of the alternation.

2.2.3 Sink Automaton

To analyze worst-case behavior of a backtracking-based matching algorithm, Weideman *et al.* [46] introduced the *sink automaton* $\mathrm{Sink}(A)$, constructed by adding a new accepting state q_{sink} with ε-transitions from every original state and a universal self-loop. The *sink ambiguity* $\mathrm{SinkAbg}(A) = \mathrm{Abg}(\mathrm{Sink}(A))$ captures all partial matching attempts, not just complete matches. Figure 2(2), (4), and (6) show the sink automata for the three example regexes.

2.2.4 Complexity Characterization

Two theorems connect sink ambiguity to backtracking runtime:

Theorem A (Backtracking Runtime Bound [46])**.** *For any ε-loop-free NFA A, its backtracking runtime satisfies* $\mathrm{BtRtN}(A, n) \in O(\mathrm{SinkAbgN}(A, n))$.

Theorem B (Two-Overlap-Loop Characterization [2, 45])**.** *For any ε-loop-free NFA A,* $\mathrm{SinkAbgN}(A, n) \in \Omega(n^2)$ *if and only if A contains a two-overlap-loop structure—two loops sharing a common path segment with overlapping accepted symbol sets.*

For example, the NFA in Figure 2(3) contains two 'a'-loops connected by an ε-edge; its sink automaton in Figure 2(4) exhibits $\Theta(n^2)$ ambiguity. The NFA in Figure 2(5) has two a-loops on the same state, yielding $2^{\Theta(n)}$ sink ambiguity in Figure 2(6). When an NFA is trim (*i.e.,* all states lie on some accepting path), IDA is equivalent to the presence of a two-overlap-loop structure [45]. Combined with Theorem A and Theorem B, this establishes that two-overlap-loop

structures are both necessary and sufficient for super-linear backtracking runtime in K-regexes. This forms the basis for existing static ReDoS detectors [24, 46, 47]. However, these theorems assume each NFA transition executes in $O(1)$ time. This assumption breaks for backreferences, where a single transition may compare substrings of length $O(n)$, invalidating the runtime bound of Theorem A.

3 Motivation and Problem Statement

Here we motivate the need for ReDoS analysis beyond K-regexes, present our threat model, and pose research questions.

3.1 Motivating Example

Backreferences are actively used in security-critical regex deployments. As of November 2025, the Snort intrusion detection system's registered ruleset contains 11,266 unique regexes, of which 282 (2.5%) use backreferences to describe malicious packet signatures. In Snort and similar intrusion detection systems, these regexes can be evaluated on every inspected packet, making their worst-case performance a security concern: a slow regex evaluation can degrade throughput or cause the system to skip rules entirely.

However, as discussed in §2.2.4, exisiting ReDoS theory provides no guidance on whether REwBs are vulnerable. Existing detectors may discover slow REwB inputs empirically (*e.g.,* via fuzzing [39]), but do not provide structural guarantees nor characterize backreference-induced patterns.

To illustrate the gap, consider the regex $/(\!(_1\texttt{a}^*)\!)_1\backslash 1\texttt{b}/$ This regex contains no two-overlap-loop structure, and its sink ambiguity is $O(n)$—existing detectors would report it as safe. Yet its backtracking runtime is $\Theta(n^2)$. Intuitively, on a non-matching input $\texttt{`a}^{2n}\texttt{'}$, the engine tries matching $(\!(_1\texttt{a}^*)\!)_1\backslash 1$ against each prefix $\texttt{`a}^{2k}\texttt{'}$ of input for $k \in \{n, n-1, \ldots, 1, 0\}$.

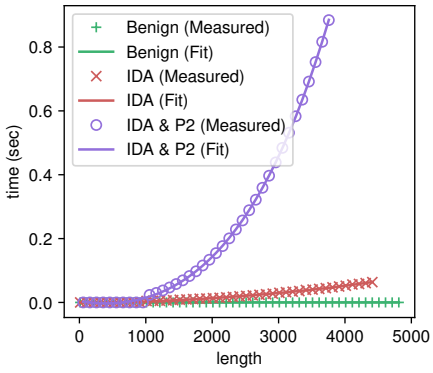

Figure 3: Matching time for a regex from the Snort ruleset, evaluated on a benign input and two adversarial inputs exploiting infinite degree of ambiguity (IDA) and a combination of IDA with backreferences.

When the group captures k symbols, the backreference performs an $O(k)$ string comparison before failing, yielding total cost $\sum_{k=0}^{n} k = \Theta(n^2)$. We formalize this analysis in §5.

Such gap has practical consequences, including missed vulnerabilities and underestimation of attack severity. Figure 3 illustrates this effect using a regex from Snort 2 rule with `sid:26544`. It compares matching times on three classes of strings: benign inputs, inputs that exploit only IDA, and inputs that exploit both IDA and a backreference.

When only IDA is exploited, runtime grows quadratically. When both IDA and the backreference are exploited together, runtime grows cubically and super-linear behavior is triggered with shorter inputs. This demonstrates that backreferences represent an additional and previously uncharacterized attack surface for ReDoS.

3.2 Threat Model

We adopt a variation of the standard ReDoS threat model from prior work [7, 16], with assumptions tailored to REwB.

Attacker capabilities. The attacker controls the input string evaluated by the victim's regex. This reflects the common use of regexes to process untrusted input in web applications [3, 10] and network intrusion detection [47]. The attacker can also analyze the target regex—for instance, Snort's rulesets are publicly available—to identify exploitable patterns and craft adversarial inputs.

Victim environment. The victim uses a backtracking-based regex engine that supports backreferences. This covers regex engines such as Python `re`, Perl, Java `java.util.regex`, PCRE, PCRE2, Onigmo and so on [7]. Notable exceptions are those used by Rust and Go, which use Thompson-style NFA simulation and do not support backreferences in their default engines. We note that some engines (*e.g.,* PCRE) employ mitigations such as matching limits; as we show in §7, an attacker can deliberately trigger these limits to cause the engine to abort matching, which itself can be exploited to bypass detection rules.

Attack goal. The attacker seeks to cause one of two outcomes: (1) *resource exhaustion*, where a crafted input forces the regex engine into super-linear evaluation, degrading service availability; or (2) *detection bypass*, where the input triggers the engine's matching limit, causing it to skip the remainder of the regex and fail to flag malicious content.

3.3 Research Questions

Our motivating example reveals that backreferences can induce super-linear backtracking runtime even when existing ReDoS detectors based on NFA-based analyses predict linear behavior. This leads to three research questions. We address RQ1 in §5, RQ2 in §6, and RQ3 in §7.

RQ1 **Theory:** *Under what conditions do REwB cause super-linear backtracking runtime, and can we characterize the structural patterns responsible?*

RQ2 **Detection:** *Can we develop algorithms that automatically identify vulnerable REwB, and generate corresponding adversarial inputs?*

RQ3 **Prevalence and Impact:** *How prevalent are REwB-induced ReDoS vulnerabilities in real-world regex deployments, and what is their practical impact?*

- **RQ3a** *How many REwB in a real-world deployment are vulnerable to backreference-induced ReDoS?*
- **RQ3b** *How does matching runtime scale on adversarial inputs, and does combining backreference patterns with IDA worsen the impact?*
- **RQ3c** *Can REwB vulnerabilities be exploited in a deployed intrusion detection system?*

4 Two-Phase Memory Finite Automaton

To analyze the backtracking behavior of REwB, we need an automaton model that captures both backreference semantics and self-referencing behavior. Schmid introduced the Memory Automaton (MFA) [38], which extends NFAs with a memory table that stores captured substrings and replays them on backreference transitions. However, MFA does not support self-references (§2.1).

We propose the *Two-Phase Memory Finite Automaton (2PMFA)*, which extends MFA with a two-phase memory design that cleanly separates the *committed* capture (from the previous iteration of a repeated group) from the *in-progress* capture (being recorded in the current iteration). This separation enables self-references: when $\backslash i$ is encountered inside group i, the engine matches against the committed phase while the in-progress phase continues recording (Figure 1).

4.1 Model Definition

Definition 2: Two-Phase Memory Automaton

A 2PMFA is a 6-tuple $A = (Q, \Sigma, I, \Delta, q_0, F)$ where:

- Q is a finite set of states,
- Σ is a finite input alphabet,
- I is a finite set of capture group identifiers,
- $\Delta \subseteq Q \times T(A) \times Q$ is the transition relation, with

$$T(A) = \{\sigma \in \Sigma\} \cup \{\varepsilon\} \cup \{(\!|_i, |\!)_i, \backslash i \mid i \in I\},$$

- $q_0 \in Q$ is the initial state, and
- $F \subseteq Q$ is the set of accepting states.

A transition $\Delta(q,t) \ni q'$ moves from state q to q' on label t, where labels fall into five categories: a *symbol* $\sigma \in \Sigma$ consumes one input character; ε consumes nothing; $(\!|_i$ opens capture group i (begins recording); $|\!)_i$ closes capture group i (commits the recording); and $\backslash i$ replays the string most recently committed by group i.

4.2 Matching Semantics

A 2PMFA is matched against an input string s via a backtracking algorithm that maintains a memory function M mapping each capture group to start and end indices into s. The algorithm explores transitions depth-first, recursively backtracking on failure—mirroring Spencer-style regex engines (§2.2.1). Symbol and ε transitions behave as in MFAs. Group i opening records the input index in $M((\!|'_i)$. Group i closing commits $M((\!|'_i)$ and the input index to $M((\!|_i)$ and $M(|\!)_i)$. A backreference transition $\backslash i$ compares $s[j..{<}j{+}l]$ against the committed capture $s[M((\!|_i)..{<}M(|\!)_i)]$, where $l = M(|\!)_i) - M((\!|_i)$, and advances the input index by l on success. The full pseudocode and the self-reference semantics are given in Appendix A.

Two properties of this algorithm are critical for the complexity analysis in §5:

1. **Non-constant transition cost.** A backreference transition costs $O(l)$ time for a captured substring of length l, which can be as large as $O(n)$. All other transitions cost $O(1)$.
2. **Repeated evaluation via backtracking.** Backtracking can cause the same transition to be evaluated multiple times across different search branches.

Both properties are absent in standard NFA simulation and are the root cause of backreference-induced ReDoS.

4.3 Path Notation

We establish path notation used throughout the complexity analysis (§5) and vulnerability pattern classification (§6).

Definition 3: 2PMFA Path

Given a 2PMFA $A = (Q, \Sigma, I, \Delta, q_0, F)$, a *path* π from q'_0 to q'_m is a sequence

$$q'_0 \xrightarrow[s'_0]{t'_0} q'_1 \xrightarrow[s'_1]{t'_1} \cdots q'_{m-1} \xrightarrow[s'_{m-1}]{t'_{m-1}} q'_m$$

here each step satisfies $q'_{k+1} \in \Delta(q'_k, t'_k)$ and t'_k matches s'_k.

- **Accepting path:** π is accepting if $q'_0 = q_0$ and $q'_m \in F$.
- **String of a path:** $\mathcal{S}(\pi) = s'_0 s'_1 \cdots s'_{m-1}$.
- **Loop path:** A path from q back to q is denoted π^* (emphasizing its role as a repeatable loop).
- **Backreference cost:** When a step has label $t' = \backslash i$, it matches s'_k of length up to $O(n)$ and costs $O(|s'_k|)$ time (contrasting with σ and ε transitions, which cost $O(1)$).
- **Path overlap:** We say $\pi_1, \cdots, \pi_m$ overlap when their strings are formed by the same repeated substring. Formally, $\mathrm{Ovlp}(\pi_1, \cdots, \pi_m)$ iff $\exists s_{\mathrm{ovlp}} \in \Sigma^*, u_1, \cdots, u_m \in \mathbb{N}$, s.t. for $k \in \{1, \cdots, m\}$, $\mathcal{S}(\pi_k) = s_{\mathrm{ovlp}}^{u_k}$.

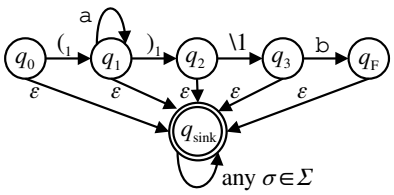

Figure 4: Sink automaton $\text{Sink}(A)$ for the regex $(\!|_1 \texttt{a}^*)\!|_1 \backslash 1\ \texttt{b}$. The original automaton A has a single a-loop at q_1; no two overlapping loops exist.

5 Theoretical Analysis of REwB

This section lays out the theoretical foundations for ReDoS vulnerabilities caused by REwB. We begin with a concrete example showing that existing runtime bounds fail for REwB (§5.1). We then identify two independent conditions that enable non-$O(1)$ per-backreference matching cost (§5.2). From these conditions, we derive sufficient conditions under which the existing runtime bound still holds (§5.3), and necessary conditions under which it is violated (§5.4).

5.1 Why Existing Bounds Fail

Recall from §2.2 that for K-regexes, Theorem A bounds backtracking runtime by sink ambiguity, and Theorem B shows that super-linear sink ambiguity requires two overlapping loops (the IDA condition). The following example demonstrates that *neither* conditions is necessary for super-linear runtime when backreferences are present.

Example 1

Let A be the 2PMFA for $/(\!|_1 \texttt{a}^*)\!|_1 \backslash 1\texttt{b}/$. Then, $\text{AbgN}(A,n) \in O(1)$, $\text{SinkAbgN}(A,n) \in O(n)$, yet $\text{BtRtN}(A,n) \in \Theta(n^2)$.

Proof. Figure 4 shows $\text{Sink}(A)$. The original automaton A contains a single loop (the `'a'`-loop at q_1); no two overlapping loops exist.

Ambiguity is $O(1)$. A accepts only strings of the form `'a`^{n}`b'`. For such an input, there exists exactly one accepting path:

$$q_0 \xrightarrow{(\!|_1} \pi^*_{\texttt{a}} \xrightarrow{)\!|_1} q_2 \xrightarrow[\texttt{a}^{n/2}]{\backslash 1} q_3 \xrightarrow{\texttt{b}} q_F, \quad \text{where } \mathcal{S}(\pi^*_{\texttt{a}}) = \texttt{a}^{n/2}.$$

Thus, $\text{AbgN}(A,n) \in O(1)$.

Sink ambiguity is $O(n)$. On input `'a`^{n}`b'`, the sink automaton admits the following families of accepting paths:

(i) Entering q_{sink} directly from q_0 via ε (1 path):

$$q_0 \xrightarrow{\varepsilon} \pi^*_{\text{sink}_1}, \quad \text{where } \mathcal{S}(\pi^*_{\text{sink}_1}) = \texttt{a}^n\texttt{b}.$$

(ii) Looping k times at q_1 and then entering q_{sink} from q_1 or q_2 ($2(n+1)$ paths):

$$q_0 \xrightarrow{(\!|_1} \pi^*_{\texttt{a}_2} \xrightarrow{\varepsilon} \pi^*_{\text{sink}_2} \quad \text{and} \quad q_0 \xrightarrow{(\!|_1} \pi^*_{\texttt{a}_2} \xrightarrow{)\!|_1} q_2 \xrightarrow{\varepsilon} \pi^*_{\text{sink}_2},$$

where $\mathcal{S}(\pi^*_{\texttt{a}_2}) = \texttt{a}^k$, $\mathcal{S}(\pi^*_{\text{sink}_2}) = \texttt{a}^{n-k}\texttt{b}$, $k \in \mathbb{N}_{0..n}$.

(iii) Capturing $\texttt{a}^k$, matching the backreference, and entering q_{sink} from q_3 ($\lfloor n/2 \rfloor + 1$ paths):

$$q_0 \xrightarrow{(\!|_1} \pi^*_{\texttt{a}_3} \xrightarrow{)\!|_1} q_2 \xrightarrow[s_{\backslash 1}]{\backslash 1} q_3 \xrightarrow{\varepsilon} \pi^*_{\text{sink}_3},$$

where $\mathcal{S}(\pi^*_{\texttt{a}_3}) = s_{\backslash 1} = \texttt{a}^k$, $\mathcal{S}(\pi^*_{\text{sink}_3}) = \texttt{a}^{n-2k}\texttt{b}$, $k \in \mathbb{N}_{0..n/2}$.

(iv) The unique full accepting path through q_F (1 path):

$$q_0 \xrightarrow{(\!|_1} \pi^*_{\texttt{a}_4} \xrightarrow{)\!|_1} q_2 \xrightarrow[\texttt{a}^{n/2}]{\backslash 1} q_3 \xrightarrow{\texttt{b}} q_F \xrightarrow{\varepsilon} q_{\text{sink}}.$$

In total there are $(3n/2) + 2$ accepting paths, so $\text{SinkAbgN}(A,n) \in O(n)$.

Runtime is $\Theta(n^2)$. Consider the input `'a`^{n}`'` (no trailing `'b'`; the match will ultimately fail). With a greedy loop, the engine first tries capturing all n symbols, then backtracks one symbol at a time. Table 1 summarizes the cost of each attempt. When the loop captures `'a`^{k}`'` ($k > n/2$), the backreference fails in $O(1)$ time because fewer than k symbols remain. When $k \leq n/2$, the backreference performs a full $O(k)$ string comparison before the suffix b mismatches. The total cost is:

$$n + \left(\frac{n}{2} - 1\right) + \sum_{k=1}^{n/2} k + 1 \;=\; \frac{n^2}{8} + \frac{7n}{4} \;\in\; \Theta(n^2).$$

The two root causes of this quadratic blowup are:

1. **Non-$O(1)$ per-backreference cost.** The backreference \1 matches a captured substring of length up to $n/2$, so a single evaluation costs $O(n)$.
2. **Non-$O(1)$ evaluation count.** Backtracking causes \1 to be evaluated $\Theta(n)$ times (once per loop iteration that is retried).

Together these yield $O(n) \cdot O(n) = O(n^2)$ runtime, violating Theorem A despite $O(n)$ sink ambiguity. We formalize each condition in §5.2. □

5.2 Conditions for Super-Linear REwB

Example 1 revealed two independent factors that cause Theorems A and B to fail: a single backreference evaluation may cost non-$O(1)$ time, and a backreference may be evaluated a non-$O(1)$ number of times. We now formalize each condition as a lemma.

Per-evaluation cost. In a standard NFA, every transition consumes exactly one symbol or ε, so each step costs $O(1)$. A backreference transition $\backslash i$, however, performs a string comparison against the captured content of group i, which may have length up to $O(n)$. Lemma 1 identifies the structural condition that permits this.

Table 1: Runtime analysis for $(_1\mathtt{a}^*)_1\backslash 1\ \mathtt{b}$ on input $\mathtt{a}^n$. A dash indicates that the backreference fails in $O(1)$ time (remaining input shorter than capture).

Attempt	Loop captures	\1 matches	Cost
0	$\mathtt{a}^n$	—	n
1	$\mathtt{a}^{n-1}$	—	1
⋮			
$n/2-1$	$\mathtt{a}^{n/2+1}$	—	1
$n/2$	$\mathtt{a}^{n/2}$	$\mathtt{a}^{n/2}$	$n/2$
$n/2+1$	$\mathtt{a}^{n/2-1}$	$\mathtt{a}^{n/2-1}$	$n/2-1$
⋮			
$n-1$	$\mathtt{a}^1$	$\mathtt{a}^1$	1
n	ε	ε	1

Lemma 1: Non-$O(1)$ Per-Backreference Cost

For an ε-loop-free 2PMFA A, if a backreference transition $\backslash i$ can match a string of non-$O(1)$ length, then capture group i must contain either: a loop, or a backreference that itself matches a string of non-$O(1)$ length.

Proof. Among the five transition types in a 2PMFA , only a backreference can match strings of unbounded length (*i.e.,* non-$O(1)$); all others match at most one symbol. Given this, there are two cases in which a backreference $\backslash i$ matches a string of non-$O(1)$ length.

Case 1: *Capture group i contains no backreference transitions (i.e., every transition inside the group matches $O(1)$-length strings).* We show by contradiction that some transition must appear more than once on a path through the group. If each transition appeared at most once, then because the total number of transitions is $O(1)$, any captured string would have length $O(1)$—a contradiction. If a transition appears more than once along a path, the path must take the form:

$$\pi_{\text{left}}\ q \xrightarrow[s]{t} q'\ \pi_{\text{pump}}\ q \xrightarrow[s]{t} q'\ \pi_{\text{right}}$$

which exhibits a subpath π_{pump} from q back to q (*i.e.,* a loop). In Figure 5(a), the backreference $\backslash 1$ incurs non-$O(1)$ cost when matching capture group 1 that contains such a loop.

Case 2: *Capture group i contains a backreference that matches strings of non-$O(1)$ length.* For this to occur, the inner backreference must reference another capture group that itself contains a loop (reducing to Case 1 or a further backreference capable of matching non-$O(1)$ strings, or the capture groups form a chain of references that ultimately terminates at a loop, or a cyclic referenced backreference. In Figure 5(a), the backreference $\backslash 2$ incurs non-$O(1)$ cost when matching a capture group that contains $\backslash 1$. □

Evaluation count. Even when each backreference evaluation is cheap, the *number* of evaluations may be super-linear due to backtracking. Lemma 2 formalizes this.

Lemma 2: Non-$O(1)$ Backreference Evaluations

For an ε-loop-free 2PMFA $A = (Q, \Sigma, I, \Delta, q_0, F)$, if a transition $\delta = ((q,t) \mapsto Q') \in \Delta$ is evaluated a non-$O(1)$ number of times, then there exists a path in A in which δ appears after or inside a loop.

Proof. There are two cases where δ is evaluated a non-$O(1)$ number of times.

Case 1: *δ is evaluated across non-$O(1)$ backtracking branches.* This implies that there exist a non-$O(1)$ number of distinct paths from q_0 to q. We prove by contradiction that within such paths, there must exist a transition $\delta_1 = ((q_1,t_1) \mapsto Q'_1)$ that appears more than once before δ. Assume instead that each transition appears at most once along any such path. Then the maximum number of paths from q_0 to q would be $\sum_{k=0}^{|\Delta|-1} \mathrm{P}^k_{|\Delta|-1}$, which is $O(1)$ with respect to the input length—contradicting the assumption. Therefore, δ_1 must occur multiple times on some path, implying the existence of a subpath from q_1 to q_1, *i.e.,* a loop before δ. In Figure 5(b), the backreference $\backslash 1$ may be evaluated a non-$O(1)$ number of times after such a loop.

Case 2: *δ is evaluated non-$O(1)$ times within a single backtracking path.* This means that along a single path starting from q_0, the transition δ appears non-$O(1)$ times. Consequently, the path must contain a subpath from q back to q (*i.e.,* a loop) in which δ is contained. In Figure 5(c), the backreference $\backslash 1$ may be evaluated a non-$O(1)$ number of times within a loop. □

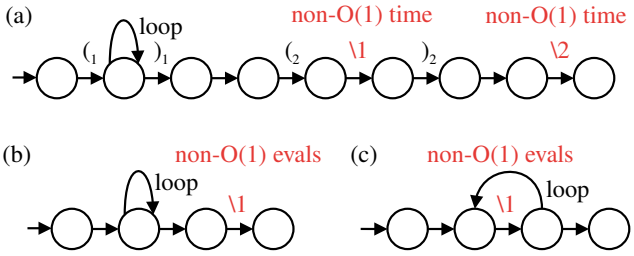

Figure 5: Structural conditions for super-linear backreference behavior. (a) A backreference incurs non-$O(1)$ cost when its capture group contains a loop (left) or another non-$O(1)$ backreference (right). (b)–(c) A backreference is evaluated non-$O(1)$ times when it appears after a loop (b) or inside a loop (c).

5.3 Conditions for Bounded Runtime

The Lemmata 1 and 2 identify *what can go wrong*. We now show that if neither condition is fully triggered on each transition, the classical runtime bound (Theorem A) continues to hold for REwB.

Theorem 1: Safe Backreferences

For an ε-loop-free 2PMFA A running on strings of length n: if every backreference in A either (i) captures a string of length $O(1)$, or (ii) is evaluated a total of $O(1)$ times, then $\mathrm{BtRtN}(A,n) \in O(\mathrm{SinkAbgN}(A,n))$.

Proof sketch. (Full proof in Appendix B.) We define the functions for computing sink ambiguity (SinkAbgS, Algorithm 2) and backtracking runtime (BtRtS, Algorithm 3). When all backreferences match substrings of $O(1)$ length, BtRtS can be scaled by a constant factor to obtain a function that is asymptotically equivalent to SinkAbgS. If additional backreferences match substrings of non-$O(1)$ length but are evaluated only $O(1)$ times, we further augment BtRtS with an additional term. We prove that the asymptotic growth of this term never exceeds that of the original function. Consequently, the augmented BtRtS remains bounded above by a constant multiple of SinkAbgS. Therefore, under conditions (i) and (ii), the backtracking runtime is asymptotically bounded by sink ambiguity. □

5.4 Necessary Conditions for Vulnerability

Taking the contrapositive of our own Theorem 1, we obtain the structural conditions that *must* hold whenever backreferences cause the runtime to exceed the sink-ambiguity bound.

Theorem 2: Necessities for REwB Vulnerability

For an ε-loop-free 2PMFA A running on strings of length n: if $\mathrm{BtRtN}(A,n) \notin O(\mathrm{SinkAbgN}(A,n))$, then there exists a backreference transition $\backslash i$ satisfying **both**:

C1. *Non-$O(1)$ cost.* Capture group i contains a loop or a backreference that matches a string of non-$O(1)$ length. (Lemma 1; Figure 5(a))

C2. *Non-$O(1)$ evaluations.* $\backslash i$ appears after or inside a loop. (Lemma 2; Figure 5(b–c))

Proof. The contrapositive of Theorem 1 is: if $\mathrm{BtRtN}(A,n) \notin O(\mathrm{SinkAbgN}(A,n))$, then some backreference simultaneously matches non-$O(1)$-length strings *and* is evaluated non-$O(1)$ times. Applying Lemma 1 to the first conjunct yields **C1**. Applying Lemma 2 to the second yields **C2**. □

Theorem 2 reduces vulnerability detection to a structural search problem over 2PMFA paths. Any REwB whose backtracking runtime exceeds its sink ambiguity must contain a backreference satisfying both **C1** and **C2**. In particular, when the sink ambiguity is $O(n)$ (*i.e.,* no two-overlap loops exist and existing detectors report *no vulnerability*), under certain conditions, **C1** and **C2** together can still produce non-$O(n)$ runtime—as demonstrated by Example 1. We exploit this characterization in §6 to derive three concrete vulnerability patterns and prove that each is sufficient to induce super-linear runtime.

Role of two-overlap loops. When Theorem 1 *does* hold (*i.e.,* backreferences are safe), super-linear runtime still requires super-linear sink ambiguity. Under the assumption that Theorem B is also true for 2PMFAs, this is equivalent to the presence of two overlapping loops—the classical IDA condition. In other words, safe backreferences do not introduce new vulnerability patterns beyond double-overlap loops.

Scope and completeness. The conditions in Theorem 2 are necessary but not sufficient: not each backreference satisfying **C1** and **C2** induces super-linear runtime. The three patterns we derive in §6 are each proven sufficient, but may be incomplete. They are necessary only when the sink ambiguity is $O(n)$ and no non-$O(1)$ backreferences in capture groups. Additionally, extending the structural equivalence between sink ambiguity and overlap loops (Theorem B) from NFAs to 2PMFAs remains open; we conjecture that backreferences do not introduce additional sink ambiguity, but leave formal proof to future work.

Answer to RQ1 (Theory)

REwB cause super-linear backtracking runtime when a backreference satisfies two conditions simultaneously: **(C1)** its capture group contains a loop, enabling non-$O(1)$ match length per evaluation; and **(C2)** it appears after or inside a loop, enabling non-$O(1)$ total evaluations during backtracking. When both conditions hold, the product of per-evaluation cost and evaluation count yields super-linear runtime, even when sink ambiguity remains $O(n)$.

6 Vulnerable REwB Patterns

In §5, we set the necessary conditions under which backreferences cause the backtracking runtime to exceed the sink-ambiguity bound (Theorem 2). In this section, we derive three concrete vulnerability patterns from those conditions and prove that each is *sufficient* to induce super-linear runtime, even when the sink ambiguity is $O(n)$—*i.e.,* when no double-overlap-loop (IDA) pattern exists. We begin by classifying the patterns (§6.1), then prove their sufficiency (§6.2), and finally show that the three patterns exhaustively cover the cases implied by Theorem 2 (§6.3).

6.1 Pattern Classification

Theorem 2 requires two conditions to hold simultaneously for a backreference to cause unbounded runtime:

C1 **Non-$O(1)$ per-evaluation cost** (Lemma 1): the referenced capture group must contain a loop π_{pump} (or another non-$O(1)$-length backreference), enabling the captured string to grow with input length. Figure 6(1) shows the generalized sub-pattern: a capture group delimited by

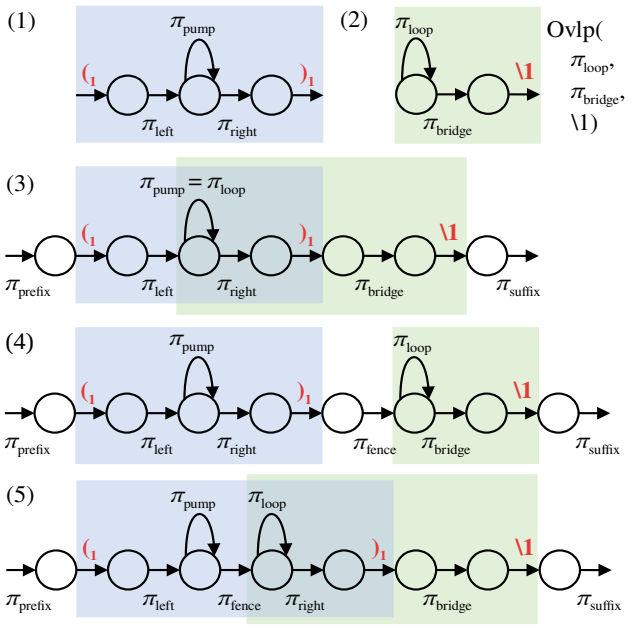

Figure 6: Sub-patterns for **C1** (1) and **C2** (2), and the three vulnerability patterns (3–5) derived by composing them.

$(\!|_i$ and $|\!)_i$, with a left path π_{left}, a loop π_{pump}, and a right path π_{right}.

C2 **Non-$O(1)$ evaluation count** (Lemma 2): the backreference must appear after or inside a loop π_{loop}, so that it is evaluated a non-$O(1)$ number of times during backtracking. Figure 6(2) shows this sub-pattern: a loop π_{loop} connected to the backreference via a bridge path π_{bridge}.

Additionally, because we restrict attention to the $O(n)$ sink-ambiguity regime (no IDA), the loop π_{loop}, bridge π_{bridge}, and the backreference must all accept a common overlap string s_{ovlp}. Without this overlap, the loop cannot produce a non-$O(1)$ number of *distinct* path decompositions prior to the backreference while staying non-IDA. Intuitively, the overlap allows the input to be partitioned in multiple ways between π_{loop} and the backreference—*e.g.,* for an input s^n_{ovlp}, the loop may consume between 0 and n copies, while the backreference matches the corresponding captured substring.

The two conditions can be composed in exactly three structurally distinct ways, depending on (i) whether π_{pump} and π_{loop} are the same loop or distinct, and (ii) whether π_{loop} lies inside or outside the capture group. We define each pattern:

Pattern 1: Backref-to-Overlap-Loop

A 2PMFA contains Pattern 1 if it has a path of the form

$$\pi_{\text{prefix}} \xrightarrow{(\!|_i} \pi_{\text{left}}\, \pi^*_{\text{pump}}\, \pi_{\text{right}} \xrightarrow{|\!)_i} \pi_{\text{bridge}} \xrightarrow{\backslash i} \pi_{\text{suffix}}$$

where π_{pump} serves as *both* the pump loop (**C1**) and the evaluation loop (**C2**), and the overlap condition is: $\text{Ovlp}(\pi_{\text{left}}, \pi_{\text{pump}}, \pi_{\text{right}}\pi_{\text{bridge}})$.

Figure 6(3) illustrates Pattern 1. Here, a single loop plays both roles: it inflates the captured string (satisfying **C1**) and, because the backreference appears after the same loop, generates multiple backtracking paths (satisfying **C2**). Because only one loop is involved, no double-overlap-loop structure exists, and existing ReDoS detectors may not flag this pattern.

Pattern 2: Loop-Before-Backref-to-Loop

A 2PMFA contains Pattern 2 if it has a path of the form

$$\pi_{\text{prefix}} \xrightarrow{(\!|_i} \pi_{\text{left}}\, \pi^*_{\text{pump}}\, \pi_{\text{right}} \xrightarrow{|\!)_i} \pi_{\text{fence}}\, \pi^*_{\text{loop}}\, \pi_{\text{bridge}} \xrightarrow{\backslash i} \pi_{\text{suffix}}$$

where π_{pump} and π_{loop} are *distinct* loops, π_{pump} is inside the capture group (**C1**), π_{loop} is *outside* the capture group (**C2**), and they are separated by a non-overlapping fence path π_{fence}. The overlap condition is: $\text{Ovlp}(\pi_{\text{left}}, \pi_{\text{pump}}, \pi_{\text{loop}}, \pi_{\text{bridge}})$.

Figure 6(4) illustrates Pattern 2. The fence path π_{fence} is critical: it separates the two loops so that they do not form a double-overlap-loop (classical IDA pattern). This is precisely what makes this pattern invisible to IDA detectors and unique to REwB.

Pattern 3: Backref-to-Loop-and-Loop

A 2PMFA contains Pattern 3 if it has a path of the form

$$\pi_{\text{prefix}} \xrightarrow{(\!|_i} \pi_{\text{left}}\, \pi^*_{\text{pump}}\, \pi_{\text{fence}}\, \pi^*_{\text{loop}}\, \pi_{\text{right}} \xrightarrow{|\!)_i} \pi_{\text{bridge}} \xrightarrow{\backslash i} \pi_{\text{suffix}}$$

where π_{pump} and π_{loop} are *distinct* loops that *both* reside inside the capture group, separated by a non-overlapping fence π_{fence}. π_{pump} provides the non-$O(1)$ captured length (**C1**), and π_{loop} provides the non-$O(1)$ evaluation count (**C2**). The overlap condition is: $\text{Ovlp}(\pi_{\text{left}}, \pi_{\text{pump}}, \pi_{\text{loop}}, \pi_{\text{right}}\pi_{\text{bridge}})$.

Figure 6(5) illustrates Pattern 3, which lies structurally between the other two. As in Pattern 2, two distinct loops are separated by a non-overlapping fence, preventing IDA. As in Pattern 1, all relevant loops reside inside the capture group.

Overall, three patterns evade existing IDA detectors. Patterns 2 and 3 contain two loops but separate them with a non-overlapping fence, breaking the double-overlap-loop structure. Pattern 1 contains only a single loop altogether. In each case, the vulnerability arises specifically from the interaction between the loop(s) and the backreference—a mechanism that previously established runtime analyses, which assume $O(1)$ per-transition cost, cannot capture.

6.2 Super-linear Runtime Proofs

We now prove that each pattern is sufficient to cause super-linear runtime.

Theorem 3: ReWB Super-Linear Runtime

For an ε-loop-free 2PMFA A, if A contains Patterns 1 to 3, then $\mathrm{BtRtN}(A,n) \notin O(n)$.

Proof sketch. (Full proofs in Appendix C). We sketch the proof for a simplified instance of Pattern 2 (two separated loops, π_{loop} outside capture group; Figure 6(4)). Consider the simplified path $(\!|_i \pi^*_{\text{pump}} |\!)_i \; \pi_{\text{fence}} \; \pi^*_{\text{loop}} \; \backslash i \; \pi_{\text{suffix}}$ and the input $s^{n'}_{\text{ovlp}} \; s_{\text{fence}} \; s^{2n'}_{\text{ovlp}}$, where $\mathcal{S}(\pi_{\text{pump}}) = \mathcal{S}(\pi_{\text{loop}}) = s_{\text{ovlp}} \neq \mathcal{S}(\pi_{\text{suffix}})$, $\mathcal{S}(\pi_{\text{fence}}) = s_{\text{fence}}$. The $(\!|_i \pi^*_{\text{pump}} |\!)_i$ matches $s^{n'}_{\text{ovlp}}$. In each backtracking, π^*_{loop} matches s^k_{ovlp} for $k \in \{2n', 2n'-1, \ldots, 1, 0\}$. In each backtracking where $k \in \{n', \ldots, 0\}$, $\backslash i$ matches a substring of length $n' \cdot |s_{\text{ovlp}}|$. Therefore the total backreference cost is $n' \cdot n' \cdot |s_{\text{ovlp}}| \in \Omega(n'^2)$. □

6.3 Exhaustiveness of the Classification

We now argue that Patterns 1–3 exhaustively cover the structural configurations implied by Theorem 2, under the restriction that the non-$O(1)$ captured length in **C1** arises from a loop (rather than from recursive backreferences within the capture group, which we leave to future work).

Theorem 2 requires two loops to co-exist: a pump loop π_{pump} inside the capture group (**C1**) and an evaluation loop π_{loop} before or around the backreference (**C2**). Figure 6 summarizes the case analysis. Three structural decisions determine the pattern:

D1. **Are π_{pump} and π_{loop} the same loop?** If yes, a single loop satisfies both conditions, yielding **Pattern 1**. If no, we proceed to **D2**.

D2. **Does π_{loop} reside inside or outside the capture group?** Since π_{pump} is inside the capture group (by **C1**), π_{loop} can be either inside or outside. If outside, we proceed to **D3(a)**; if inside, to **D3(b)**.

D3. **Do π_{pump} and π_{loop} overlap?**

(a) π_{loop} is outside the capture group. If the two loops overlap, they form a classical IDA (double-overlap-loop) pattern, which is already detectable by existing tools and falls outside our scope ($O(n)$ sink ambiguity). If they are separated by a non-overlapping fence π_{fence}, we obtain **Pattern 2**.

(b) π_{loop} is inside the capture group. By the same argument, overlapping loops yield IDA. Non-overlapping loops separated by π_{fence} yield **Pattern 3**.

One remaining case is when **C1** is satisfied not by a loop but by a backreference nested within the capture group (*i.e.,* cycle-referencing between capture groups). Such recursive patterns are complex, rarely encountered in practice (none appeared in our evaluation; §7), and their analysis involves undecidable intersection problems for 2PMFAs [9, 11]. We therefore leave their characterization to future work and note that our classification is exhaustive for the loop-based case, which covers all vulnerabilities found in our evaluation.

Answer to RQ2 (Detection)

We identify three structural vulnerability patterns (Patterns 1 to 3) derived from the necessary conditions in Theorem 2. Pattern 1 uses a single loop inside the capture group; Patterns 2 and 3 use two distinct loops separated by a non-overlapping fence (outside and inside the capture group, respectively). Each pattern evades existing IDA-based detectors yet induces $\Omega(n^2)$ runtime. For each detected pattern, we construct an attack automaton from which adversarial inputs can be systematically extracted.

7 Evaluation

We evaluate by answering the three parts of RQ3 (§3.3).

7.1 Methodology

Implementation. We implemented our detector REwBGuard by extending the Java library `dk.brics.automaton` [32], which provides NFA construction, compilation of K-regexes into NFAs, and standard NFA operations (union, intersection, minimization, emptiness checking). Our extensions add: (1) construction of 2PMFAs from practical REwB syntax, including two-phase memory capture and backreference evaluation; (2) detection algorithms for Patterns 1–3 (§6); and (3) attack-automaton generators that produce adversarial inputs for each detected pattern. We also implemented a traditional IDA detector following Wüstholz *et al.* [47] to serve as a baseline.

Dataset. We evaluate on regexes extracted from the Snort 2 registered ruleset (versions 2983–29200) [19], a widely used network intrusion detection system. The dataset contains 11,266 unique regexes, of which 282 (2.5%) contain backreferences. We excluded 8 regexes that failed to compile due to unsupported features (primarily lookaround assertions and flag modifiers) or that triggered detection errors when the tool could not compute intersections in the presence of backreferences. This yields 274 testable REwB regexes. Table 2 summarizes the dataset statistics and detection results.

Regex engines. We measure matching runtime on four production engines: PCRE 8.39 (used by Snort), Python 3.8.10's `re` module, OpenJDK 11.0.27's `java.util.regex` module, and Onigmo 6.2.0. All are Spencer-style backtracking engines that support backreferences.

Environment. All experiments ran on a server with an Intel Xeon Gold 5218R (2.10 GHz), 196 GB RAM, and Ubuntu 20.04.6 LTS (kernel 5.4.0-216).

7.2 Prevalence of REwB Vulnerabilities

Table 2 summarizes the detection results. Among the 274 testable REwB regexes, REwBGuard identifies 48 previously unknown backreference-induced ReDoS vulnerabilities—none of which are flagged by the IDA-only baseline. All 48 match one of our three patterns; we confirmed each by manual inspection (no false positives observed).

Table 2: Dataset statistics and detection results. *Pattern k only*: Pattern *k* without co-occurring IDA. *Pattern k + IDA*: Pattern *k* co-occurring with IDA. *IDA-only*: IDA-flagged regexes by the baseline [47] that do *not* match any of Patterns 1–3.

DATASET			
Total regexes			11,266
Containing backrefs			282 (2.5%)
Excluded (unsupported features)			2,942
Tested REwB			**274**
DETECTION RESULTS (AMONG 274 TESTED REWB)			
	Only	+ IDA	Total
Pattern 1	1	8	9
Pattern 2	11	28	47
Pattern 3	0	0	0
Patterns 1–3 (ours)	**12**	**36**	**48**
IDA-only (baseline) [47]			1,314
All vulnerable			**1,362**

Pattern distribution. Pattern 2 (Figure 6(4)) accounts for the majority of findings (47 of 48). This is unsurprising: many Snort regexes place an "any-character" quantifier such as `.*` before a backreference, which naturally forms the external loop π_{loop} required by Pattern 2. The overlap constraint is easily satisfied because such loops accept any symbol, and a non-overlapping fence π_{fence} frequently separates the two loops. Pattern 1 accounts for the 9 cases. Pattern 3, which requires two distinct loops within a single capture group, does not appear—consistent with the observation that capture groups in Snort regexes tend to be syntactically simple, typically containing at most one quantifier.

Co-occurrence with IDA. Of the 48 REwB vulnerabilities, 36 co-occur with an IDA pattern. The remaining are exclusively backreference-induced: their sink ambiguity is $O(n)$, so they are invisible to any IDA-based detector. As we show in §7.4, the co-occurring cases are particularly dangerous because the two vulnerability sources compound.

Detection time. Figure 7 reports detection time consumed by REwBGuard across all 274 regexes. Pattern 1 is the cheapest to detect (mean < 0.001 s), as it requires locating only a single loop. Pattern 3 is faster than Pattern 2 (mean 0.003 s vs. 0.008 s) because loop pairs are searched within the restricted scope of a capture group. Pattern 2 incurs the highest overhead, with a worst case of approximately 2.0 s for the most complex regexes. These times are acceptable for offline auditing and CI/CD integration but may be too high for online, per-packet analysis—a tradeoff consistent with other static ReDoS detectors [24, 47].

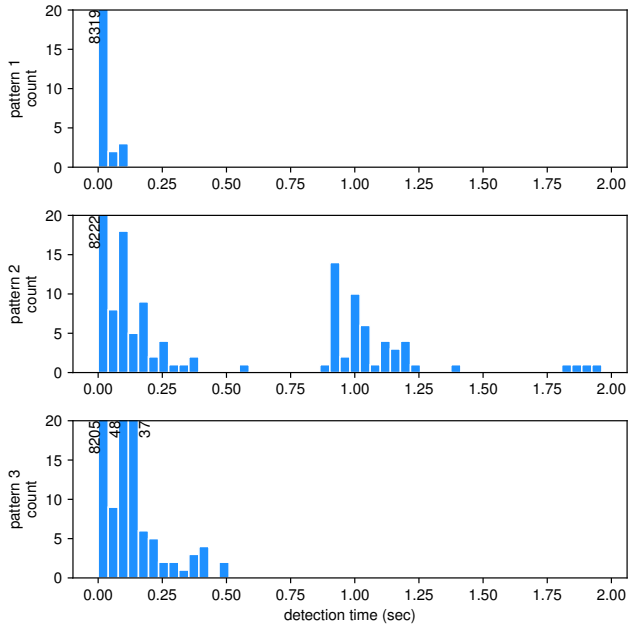

Figure 7: Static analysis time for detecting Patterns 1–3 across all 274 REwB. Most regexes are analyzed in under 0.1 s.

7.3 Comparison with Prior ReDoS Detectors

We compare REwBGuard with prior state-of-the-art "dynamic" or "hybrid" detectors: ReDoSHunter [28], Regulator [30], and Rengar [44]. Unlike static NFA-based analysis tools, these methods employ fuzzing-style techniques to probe runtime behavior and are primarily designed to detect classical IDA vulnerabilities. Nevertheless, by approximating backreferences as capture-group contents, they can sometimes reveal backreference-induced issues via double-overlap-loop detection, albeit at the cost of introducing false positives.

Methodology. We run all detectors on REwB patterns from the Snort 2 registered dataset, with a one-minute timeout per regex (the default for ReDoSHunter and Rengar), during which they generate as many attack inputs as possible. We filter out duplicate or irrelevant attacks using lightweight scripts, then sample 224 of 282 regexes for manual inspection. For each attack, we determine (1) whether it targets a backreference, (2) whether it is a true or false positive, and (3) whether the vulnerability is also detected by our REwBGuard.

Results. The results are summarized by the Venn diagrams in Figure 8, where counts are reported at the regex level (a single regex may contain multiple vulnerabilities). Existing dynamic tools detect a subset of REwB vulnerabilities, but their lack

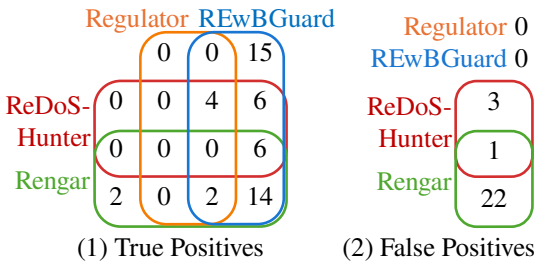

Figure 8: The number of regexes identified as containing backreference-induced vulnerabilities by each detector.

of backreference-specific analysis results in 15 false negatives. Conversely, Rengar identifies 2 vulnerable regexes that ours initially misses. Both cases use named backreferences, which are not yet supported by REwBGuard. After rewriting them as equivalent numbered backreferences, REwBGuard successfully detects the vulnerabilities.

The results also demonstrate the trade-off introduced by approximating backreferences as capture-group contents. This abstraction leads to false positives: Rengar and ReDoSHunter report 23 and 4 false positives, respectively, whereas our approach produces none. For example, consider the REwB $/(\!(_1(\!(_2\texttt{a}^*)\!)_2)\!)_1\texttt{ba}^*\backslash 1\texttt{c}\backslash 2\texttt{a}^*\texttt{d}/$. Replacing backreferences with the corresponding capture contents and removing capture transitions yields the K-regex $/\texttt{a}^*\texttt{ba}^*\texttt{a}^*\texttt{ca}^*\texttt{a}^*\texttt{d}/$. While this transformation enables detection of the true vulnerability in $\texttt{a}^*\backslash 1$, it also introduces spurious double-overlap loops $\backslash 2\texttt{a}^*$, leading to false positives.

7.4 Runtime Impact

We now measure how the detected vulnerabilities manifest as runtime degradation on real engines.

Methodology. For each of the 48 vulnerable regexes, we generate adversarial inputs from the corresponding attack automata. For regexes with both IDA and Pattern 1-3 vulnerabilities, we construct three families of inputs if possible: (1) inputs exploiting only the REwB pattern (Pattern k-only), (2) inputs exploiting only the co-occurring IDA pattern (IDA-only), and (3) inputs exploiting both simultaneously (Pattern k+IDA). For each family, we vary the pump length to produce inputs of increasing size. Each regex–input pair is executed 10 times per engine; we report the mean wall-clock matching time. To characterize the growth rate, we fit the length-runtime data points to a degree-4 polynomial via least-squares regression and identify the dominant term.

Results. Figure 9 shows representative results for a regex exhibiting both Pattern 2 and IDA. On all four engines, both the Pattern 2-only and IDA-only attacks yield *quadratic* runtime ($\Theta(n^2)$), consistent with our theoretical prediction. When both vulnerabilities are triggered simultaneously, runtime increases to *cubic* growth ($\Theta(n^3)$), demonstrating that their effects com-

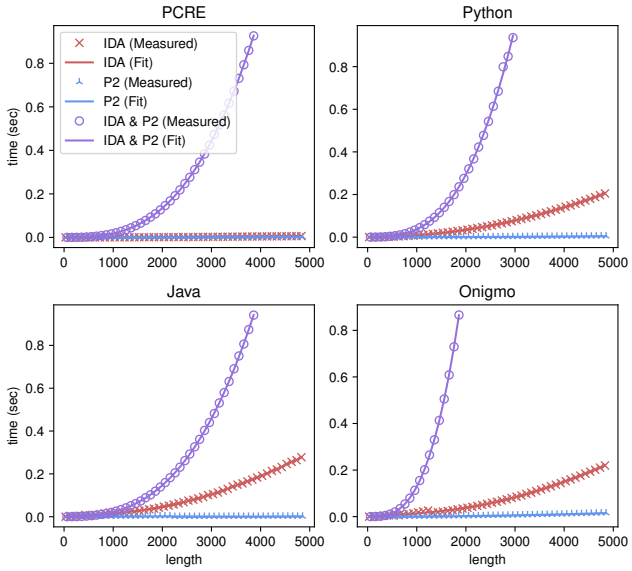

Figure 9: Matching time on PCRE, Python, Java, and Onigmo for a representative Snort regex, under three attack strategies: Pattern 2 only, IDA only, and their combination.

pose multiplicatively. The corresponding log–log fits (not shown) further support this observation, with slopes of approximately 2 for Patterns 1–3 and approximately 3 for the combined IDA+P1–P3 attacks. On PCRE, the IDA-only and Pattern 2-only attacks produce modest super-linear growth. The combined Pattern 2+IDA attack, however, triggers pronounced non-linear behavior, with matching times exceeding 1 s for inputs of length ∼3,000. This discrepancy arises because PCRE mitigates IDA by extracting required or starting characters at compile time, allowing it to skip many failing positions. Backreferences, however, disable this optimization. The other three engines do not have such mechanism. The remaining regexes exhibit qualitatively similar trends.

7.5 Exploitability in Snort

Finally, we assess whether the detected vulnerabilities are exploitable in a deployed system. We target Snort 2.9.20, which uses PCRE for regex-based packet inspection.

Setup. We implemented a TCP client–server pair running on two separate virtual machines on the same physical host (Snort does not inspect localhost traffic). Depending on the rule being triggered, either the request or the response contains the crafted payload, which Snort inspects against its loaded rules. To obtain precise timing, we instrumented the PCRE library to record per-match wall-clock time.

Exploit strategies. We identified four concrete exploits (Two exploits 1 and 2 are detailed in Appendix D) that demonstrate two distinct attack strategies. All exploits have been disclosed to the Snort development team.

Strategy 1: Performance degradation. Exploit 1 targets regexes whose combined Pattern 2 and non-anchored matching (implicit IDA) yields $\Omega(n^3)$ matching time. With attack strings of approximately 3,000 characters, PCRE matching takes 0.7–1.2 seconds per packet, which is orders of magnitude slower than the microsecond-scale budget of a packet inspection system. At network line rates, this is sufficient to degrade Snort's throughput or force it to drop packets.

Strategy 2: Alert bypass. Exploit 2 craft two-part attack packets. The first part triggers extensive backtracking, exhausting PCRE's configurable matching limit. Once the limit is reached, PCRE aborts the match and Snort skips the rule. The second part carries the actual malicious payload (*e.g.,* an ActiveX instantiation or an XSLT entity injection), which Snort no longer inspects. This enables complete evasion of the targeted detection rule.

Answer to RQ3 (Prevalence and Impact)

(a) Among 278 testable REwB in Snort, we detect 48 backreference-induced vulnerabilities, 15 of which are invisible to IDA-based detectors. **(b)** Backreference patterns alone induce $\Theta(n^2)$ runtime; when combined with IDA, runtime compounds to $\Theta(n^3)$ or worse. **(c)** We demonstrate two exploits: one causes 0.7–1.2 s matching delays per packet, and another bypasses detection entirely by exhausting PCRE's matching limit.

8 Related Work

ReDoS Detection. Static detectors [24, 25, 36, 46, 47] model regexes as NFAs and search for structural patterns (*e.g.,* two-overlap loops) that imply super-linear backtracking. Our work falls within this category. Dynamic tools [30, 35, 39] fuzz regex engines to find slow inputs, while hybrid approaches [28, 29, 44] combine both paradigms. To the best of our knowledge, all existing methods assume $O(1)$-cost transitions and operate on K-regex semantics, making them blind to the backreference-induced vulnerabilities we identify. For a comprehensive survey, see Bhuiyan et al. [7].

Complexity Foundations. Weber and Seidl [45] connect NFA ambiguity to two-overlap-loop structures, and Weideman et al. [46] show that sink ambiguity upper-bounds backtracking runtime. These form the basis for existing detectors but assume $O(1)$-cost transitions. Our 2PMFA extends this framework to non-$O(1)$ cost transitions, and our Theorems 1 and 2 generalize the runtime–ambiguity relationship to REwB.

Backreferences. Aho [1] proved that matching with backreferences is NP-complete. Kumabe et. al. [27] proved that the problem cannot be solved in $O(|s|^{2|I|-\zeta})$ for string length $|s|$, number of capture identifiers $|I|$, and a constant $\zeta > 0$. Subsequent work studied REwB expressiveness [6, 8, 33, 34]. However, none addresses *which* REwB patterns cause super-linear backtracking or *how* to detect them—the questions this paper answers. Recently, Terauchi [42] proposed a technique to transform a subset of REwB into ReDoS-safe regexes, extending classic DFA-based "regex fixing" approaches to those REwB representable by Deterministic Memory Finite Automata (DMFA). However, the approach does not characterize when a REwB admits a DMFA; many practical patterns fall outside this class (*e.g.,* $/(\!|_1(\mathtt{a}|\mathtt{b})^*|\!)_1\backslash 1/$ in Example II.4 [42]), thus cannot be transformed. Moreover, it provides only a sufficient condition for safety (constant-degree ambiguity) without identifying the underlying sources of vulnerability. Our theorems instead derive necessary 2PMFA characteristics for super-linear behavior, enabling systematic detection and pattern extraction when no safe transformation exists.

9 Discussion and Conclusion

This paper presents a systematic study of ReDoS vulnerabilities caused by backreferences. We introduced the Two-Phase Memory Finite Automaton (2PMFA) to formally analyze backreference-induced complexity, and derived necessary conditions under which REwB sees super-linear runtime despite appearing safe to prior tooling. From these conditions we identified three novel vulnerability patterns, developed detection and attack-generation algorithms, and uncovered 48 previously unknown vulnerabilities in the Snort intrusion detection ruleset. 15 of which are invisible to existing IDA-based detectors. We demonstrated practical exploits that degrade Snort's packet inspection by 0.7–1.2 s or bypass detection by exhausting PCRE's matching limit.

Limitations. Our pattern classification targets cases with linear sink ambiguity where runtime exceeds this bound: it neither covers nor rules out super-linear sink ambiguity from backreferences. False negatives remain for vulnerabilities arising from cyclic backreference chains (§6.3)—this scenario does not occur in the Snort data. Our detection algorithm cannot soundly decide overlap among all 2PMFA paths, which is an undecidable problem [9]. We evaluate on a single corpus (Snort); while Pattern 2's $.^*\backslash i$ idiom is common across regex-heavy applications, the prevalence of backreference patterns in other domains remains to be confirmed.

Implications. Operators of systems that evaluate regexes on untrusted input should audit REwBs with our patterns rather than relying solely on IDA-based tools. Engine developers should consider complexity guards that account for non-constant transition costs, as tightening matching limit alone can itself become an attack vector.

Future work. Extending 2PMFA to other irregular regex features, characterizing cyclic backreference vulnerabilities, and developing semantics-preserving repair strategies for vulnerable REwB are natural next steps.

Acknowledgements

We thank the anonymous reviewers for their valuable feedback and the Cisco PSIRT team for their time and effort in reviewing our ReDoS vulnerability report. This work was supported in part by the National Science Foundation (NSF) under grants #2135156, #2135157, and #2414504.

Ethical Considerations

This paper studies ReDoS risks in the ruleset of the Snort 2.9.20 intrusion detection system. The primary ethical consideration is the dual-use nature of vulnerability-discovery techniques, which may be used both to identify vulnerabilities and to facilitate their exploitation. We conducted a stakeholder-based ethics analysis [14] and disclosed our findings to the relevant maintainers before publication.

Stakeholders. Direct stakeholders include software developers and maintainers, regex engine developers, system operators, adversaries, and the research team. Indirect stakeholders include end users of affected software, the broader software ecosystem, and the security research community.

Potential Harms and Mitigations. Our techniques could reduce the effort required to identify ReDoS vulnerabilities and construct resource-exhaustion attacks. Mitigating identified vulnerabilities may also impose engineering and operational costs. To reduce these risks, we focus on vulnerability characterization and detection techniques rather than attack automation, explicitly discuss the scope and limitations of our methods, and avoid testing against production systems.

Responsible Disclosure. Prior to publication, we disclosed our findings to the Snort maintainers and Cisco PSIRT, providing details of the affected regexes, supporting artifacts, and representative examples. Cisco PSIRT assigned a tracking identifier to the report and engaged the Snort developers to review our findings. We participated in technical discussions with the maintainers and conducted additional validation on Snort 3, which utilizes PCRE2, based on their feedback.

Following their review, Cisco PSIRT acknowledged the observed slowdown and potential evasion behaviors, but indicated that these reflect intentional performance safeguards and accepted performance–security trade-offs in Snort. Cisco PSIRT stated that such trade-offs should be managed through user-selected deployment policies and configurations rather than modifications to the regex rules themselves. While we continue to believe that the demonstrated slowdowns and potential detection-evasion effects warrant attention, Cisco PSIRT did not request an embargo or delay of publication.

Benefits and Judgment. Our techniques enable more systematic identification of ReDoS-prone regexes and can inform both application-level remediation and regex-engine-level defenses. We believe the benefits of improving the robustness of intrusion detection system rulesets and regex-based software outweigh the associated risks. All experiments were conducted in a controlled laboratory environment using locally deployed software, without collecting user data or involving human subjects.

Open Science

The artifacts are available at https://zenodo.org/records/20762298. The artifact package includes:

1. **Detector:** The `usenixsec26_rebasil` contains the complete implementation of our detector REwBGuard (used to be named as REBASIL or SLMAD). It takes regexes as input and analyzes whether a given regex follows the IDA pattern or one of the three non-IDA patterns. If so, it generates corresponding attack strings.
2. **Dynamic Validation:** The `usenixsec26_atkre` focuses on dynamic runtime measurement. It invokes different regex engines to match the regexes against the attack strings generated by the detector, measures the execution time, and fits the relationship between input length and runtime using a polynomial curve.
3. **Plotting Scripts:** The scripts in the `usenixsec26_atkre/plot` are used to generate the plots in this paper.
4. **Data:** We provide sample input, output, and intermediate datasets. It contains regexes extracted from Snort 2 Registered rule set, detected vulnerabilities and generated attacking strings by `usenixsec26_rebasil`, as well as measured runtimes and the fitted polynomial by `usenixsec26_atkre`

Taken together, the artifact provides all materials necessary to inspect, understand, and reproduce the theoretical and empirical results presented in this paper. Additional details regarding artifact organization, setup, and usage are provided in the README files accompanying each repository.

References

[1] Alfred V. Aho. Pattern Matching in Strings. In RONALD V. Book, editor, *Formal Language Theory*, pages 325–347. Academic Press, January 1980. https://doi.org/10.1016/B978-0-12-115350-2.50016-6.

[2] Cyril Allauzen, Mehryar Mohri, and Ashish Rastogi. General algorithms for testing the ambiguity of finite automata. In Masami Ito and Masafumi Toyama, editors, *International Conference Developments in Language Theory (DLT)*, pages 108–120. Springer Berlin Heidelberg, 2008. https://doi.org/10.1007/978-3-540-85780-8_8.

[3] Efe Barlas, Xin Du, and James C. Davis. Exploiting input sanitization for regex denial of service. In *Pro-*

ceedings of the 44th International Conference on Software Engineering (ICSE), pages 883–895, May 2022. https://doi.org/10.1145/3510003.3510047.

[4] Michela Becchi and Patrick Crowley. Extending finite automata to efficiently match perl-compatible regular expressions. In *ACM International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*, 2008. https://doi.org/10.1145/1544012.1544037.

[5] Martin Berglund and Brink van der Merwe. Regular expressions with backreferences re-examined. In Jan Holub and Jan Žďárek, editors, *Proceedings of the Prague Stringology Conference 2017*, pages 30–41, 2017. https://www.stringology.org/event/2017/p04.html. Accessed 8 August 2025.

[6] Martin Berglund and Brink Van Der Merwe. Re-examining regular expressions with backreferences. *Theoretical Computer Science*, 940:66–80, January 2023. https://doi.org/10.1016/j.tcs.2022.10.041.

[7] Masudul Hasan Masud Bhuiyan, Berk Çakar, Ethan H. Burmane, James C. Davis, and Cristian-Alexandru Staicu. Sok: A literature and engineering review of regular expression denial of service (redos). In *Proceedings of the 20th ACM Asia Conference on Computer and Communications Security (ASIA CCS '25)*, page 1659–1675, 2025. https://doi.org/10.1145/3708821.3733912.

[8] Cezar Câmpeanu, Kai Salomaa, and Sheng Yu. A formal study of practical regular expressions. *International Journal of Foundations of Computer Science*, 14(06):1007–1018, 2003. https://doi.org/10.1142/S012905410300214X.

[9] Benjamin Carle and Paliath Narendran. On extended regular expressions. In Adrian Horia Dediu, Armand Mihai Ionescu, and Carlos Martín-Vide, editors, *Language and Automata Theory and Applications*, pages 279–289. Springer Berlin Heidelberg, 2009. https://doi.org/10.1007/978-3-642-00982-2_24.

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

[11] Nariyoshi Chida and Tachio Terauchi. On lookaheads in regular expressions with backreferences. *IEICE Transactions on Information and Systems*, E106.D(5):959–975, 2023. https://doi.org/10.1587/transinf.2022EDP7098.

[12] Scott Crosby. Denial of service through regular expressions. Technical Report 6, USENIX Security work in progress report, August 2003.

[13] Scott A Crosby and Dan S Wallach. Denial of Service via Algorithmic Complexity Attacks. In *12th USENIX Security Symposium (USENIX Security)*, August 2003. https://www.usenix.org/conference/12th-usenix-security-symposium/denial-service-algorithmic-complexity-attacks. Accessed: 03 June 2026.

[14] James C Davis, Sophie Chen, Huiyun Peng, Paschal C Amusuo, and Kelechi G Kalu. A guide to stakeholder analysis for cybersecurity researchers. https://doi.org/10.48550/arXiv.2508.14796, August 2025.

[15] James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. The impact of regular expression denial of service (redos) in practice: an empirical study at the ecosystem scale. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 246—256, 2018. https://doi.org/10.1145/3236024.3236027.

[16] James C. Davis, Francisco Servant, and Dongyoon Lee. Using selective memoization to defeat regular expression denial of service (redos). In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1–17, 2021. http://doi.org/10.1109/SP40001.2021.00032.

[17] AWS WAF Developers. Regex pattern set match rule statement - AWS WAF, AWS Firewall Manager, and AWS Shield Advanced. https://docs.aws.amazon.com/waf/latest/developerguide/waf-rule-statement-type-regex-pattern-set-match.html. Accessed: 06 Febrary 2026.

[18] Snort Developers. Regex - Snort 3 Rule Writing Guide. https://docs.snort.org/rules/options/payload/regex. Accessed: 06 Febrary 2026.

[19] Snort Developers. Snort rules download. https://www.snort.org/downloads/#rule-downloads. Accessed: 02 May 2025.

[20] Stack Exchange. Outage postmortem. http://web.archive.org/web/20180801005940/http://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016, 2016.

[21] OWASP Foundation. Modsecurity. https://github.com/owasp-modsecurity/ModSecurity, September 2024. Version: 3.0.13.

[22] OWASP Foundation. Owasp crs. https://github.com/coreruleset/coreruleset, March 2024. Version: 4.12.0.

[23] 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/.

[24] Sk Adnan Hassan, Zainab Aamir, Dongyoon Lee, James C. Davis, and Francisco Servant. Improving Developers' Understanding of Regex Denial of Service Tools through Anti-Patterns and Fix Strategies. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1238–1255, May 2023. https://doi.org/10.1109/SP46215.2023.10179442.

[25] James Kirrage, Asiri Rathnayake, and Hayo Thielecke. Static Analysis for Regular Expression Denial-of-Service Attacks. In *International Conference on Network and System Security (NSS)*, pages 35–148. Springer, 2013. https://doi.org/10.1007/978-3-642-38631-2_11. arXiv version: https://doi.org/10.48550/arXiv.1301.0849.

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

[27] Soh Kumabe and Yuya Uezato. On the complexity of the matching problem of regular expressions with backreferences, May 2026. https://doi.org/10.48550/arXiv.2605.07289.

[28] Yeting Li, Zixuan Chen, Jialun Cao, Zhiwu Xu, Qiancheng Peng, Haiming Chen, Liyuan Chen, and Shing-Chi Cheung. ReDoSHunter: A Combined Static and Dynamic Approach for Regular Expression DoS Detection. In *Proceedings of the 30th USENIX Conference on Security Symposium (USENIX Security)*, pages 3847–3864, August 2021. https://www.usenix.org/conference/usenixsecurity21/presentation/li-yeting.

[29] Yinxi Liu, Mingxue Zhang, and Wei Meng. Revealer: Detecting and Exploiting Regular Expression Denial-of-Service Vulnerabilities. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1468–1484, May 2021. https://doi.org/10.1109/SP40001.2021.00062.

[30] Robert McLaughlin, Fabio Pagani, Noah Spahn, Christopher Kruegel, and Giovanni Vigna. Regulator: Dynamic Analysis to Detect ReDoS. In *Proceedings of the 31st USENIX Security Symposium (USENIX Security)*, pages 4219–4235, August 2022. https://www.usenix.org/conference/usenixsecurity22/presentation/mclaughlin.

[31] Robert. McNaughton and Hisao Yamada. Regular Expressions and State Graphs for Automata. *IRE Transactions on Electronic Computers*, EC-9(1):39–47, 1960. https://doi.org/10.1109/TEC.1960.5221603.

[32] Anders Møller. `dk.brics.automaton` – finite-state automata and regular expressions for java. https://www.brics.dk/automaton/, https://github.com/cs-au-dk/dk.brics.automaton, January 2022. Version: 1.12-4.

[33] Taisei Nogami and Tachio Terauchi. On the Expressive Power of Regular Expressions with Backreferences. *LIPIcs, Volume 272, MFCS 2023*, 272:71:1–71:15, 2023. https://doi.org/10.4230/LIPICS.MFCS.2023.71.

[34] Taisei Nogami and Tachio Terauchi. Regular Expressions with Backreferences on Multiple Context-Free Languages, and the Closed-Star Condition, June 2024. https://doi.org/10.48550/arXiv.2406.18918.

[35] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, page 2155–2168, 2017. https://doi.org/10.1145/3133956.3134073.

[36] Asiri Rathnayake and Hayo Thielecke. Static Analysis for Regular Expression Exponential Runtime via Substructural Logics. Computer Research Repository (CoRR), May 2014. https://doi.org/10.48550/arXiv.1405.7058.

[37] Martin Roesch. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the 13th USENIX Conference on System Administration (LISA)*, pages 229–238, November 1999. https://doi.org/10.5555/1039834.1039864.

[38] Markus L. Schmid. Characterising regex languages by regular languages equipped with factor-referencing. *Information and Computation*, 249:1–17, 2016. https://doi.org/10.1016/j.ic.2016.02.003.

[39] Yuju Shen, Yanyan Jiang, Chang Xu, Ping Yu, Xiaoxing Ma, and Jian Lu. ReScue: Crafting Regular Expression DoS Attacks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, page 225–235, 2018. https://doi.org/10.1145/3238147.3238159.

[40] Henry Spencer. *A regular-expression matcher*, page 35–71. Academic Press Professional, Inc., 1994.

[41] Cristian-Alexandru Staicu and Michael Pradel. Freezing the web: A study of ReDoS vulnerabilities in JavaScript-based web servers. In *27th USENIX Security Symposium (USENIX Security)*, pages 361–376, August 2018. https://www.usenix.org/conference/usenixsecurity18/presentation/staicu.

[42] Tachi Terauchi. On dos vulnerability of regular expres-

sions, with and without backreferences. In *2025 IEEE 38th Computer Security Foundations Symposium (CSF)*, pages 190–204, 2025. https://doi.org/10.1109/CSF64896.2025.00011.

[43] Ken Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM (CACM)*, 11(6):419–422, June 1968. https://doi.org/10.1145/363347.363387.

[44] Xinyi Wang, Cen Zhang, Yeting Li, Zhiwu Xu, Shuailin Huang, Yi Liu, Yican Yao, Yang Xiao, Yanyan Zou, Yang Liu, and Wei Huo. Effective ReDoS Detection by Principled Vulnerability Modeling and Exploit Generation. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2427–2443, May 2023. https://doi.org/10.1109/SP46215.2023.10179328.

[45] Andreas Weber and Helmut Seidl. On the degree of ambiguity of finite automata. *Theoretical Computer Science*, 88(2):325–349, 1991. https://doi.org/10.1016/0304-3975(91)90381-B.

[46] Nicolaas Weideman, Brink van der Merwe, Martin Berglund, and Bruce Watson. Analyzing matching time behavior of backtracking regular expression matchers by using ambiguity of nfa. In Yo-Sub Han and Kai Salomaa, editors, *Implementation and Application of Automata*, pages 322–334. Springer International Publishing, 2016. https://doi.org/10.1007/978-3-319-40946-7_27.

[47] Valentin Wüstholz, Oswaldo Olivo, Marijn J. H. Heule, and Isil Dillig. Static detection of dos vulnerabilities in programs that use regular expressions (extended version). https://doi.org/10.48550/arXiv.1701.04045, Jan 2017. TACAS 2017 version: https://doi.org/10.1007/978-3-662-54580-5_1.

Appendices

A 2PMFA Matching Algorithm

Algorithm 1 defines the backtracking-based matching algorithm $\mathrm{BtRun}(A,s)$ for a 2PMFA A on input string s. The algorithm maintains a memory function $M : \{(\!|'_i, (\!|_i, |\!)_i \mid i \in I\} \to \mathbb{N}_{0..|s|} \cup \{\bot\}$ that implements the two-phase capture group table using start and end indices into s. All entries are initially $\bot$ (unset). We write $f_1 \lhd f_2$ for the function that agrees with f_2 on its domain and falls back to f_1 elsewhere (*i.e.,* $f_1 \lhd f_2 = f_2 \cup \{x \mapsto y \mid f_1(x) = y \wedge f_2(x) = \bot\}$).

Self-reference semantics. When $\backslash i$ is encountered before group i has ever been closed (*i.e.,* $M(|\!)_i) = \bot$), the behavior depends on the engine's semantics. Under $\varnothing$*-semantics* (the default in PCRE, Python, and Java), the match fails . Under ε*-semantics*, the uninitialized capture is treated as the empty string, so the backreference trivially succeeds. In Algorithm 1, the Boolean flag b_{MtBrE} selects between these two behaviors.

Algorithm 1 Backtracking matching for 2PMFA.

Require: 2PMFA $A = (Q, \Sigma, I, \Delta, q_0, F)$, input $s \in \Sigma^*$

1: $\mathrm{BtRun}(A,s) = \mathrm{BtRun}'(A,s,q_0,0,M_\bot)$
2: **function** $\mathrm{BtRun}'(A,s,q,j,M)$
3: **if** $q \in F \ \wedge \ j = |s|$ **then**
4: **return true**
5: **end if**
6: **for** each $(q,t,q') \in \Delta$ **do**
7: $\alpha' \leftarrow$ **false**
8: **switch** t **do**
9: **case** $\sigma \in \Sigma$ **and** $j < |s|$ **and** $s[j] = \sigma$
10: $\alpha' \leftarrow \mathrm{BtRun}'(A,s,q',j{+}1,M)$
11: **case** ε
12: $\alpha' \leftarrow \mathrm{BtRun}'(A,s,q',j,M)$
13: **case** $(\!|_i$
14: $\alpha' \leftarrow \mathrm{BtRun}'(A,s,q',j,\ M \lhd \{(\!|'_i \mapsto j\})$
15: **case** $|\!)_i$
16: $\alpha' \leftarrow \mathrm{BtRun}'(A,s,q',j,\ M \lhd \{(\!|_i \mapsto M((\!|'_i),\ |\!)_i \mapsto j\})$
17: **case** $\backslash i$ **and** $\mathrm{MtBr}(s,j,M,i)$
18: $l \leftarrow M(|\!)_i) - M(|\!)_i)$
19: $\alpha' \leftarrow \mathrm{BtRun}'(A,s,q',j{+}l,M)$
20: **end switch**
21: **if** α' **then**
22: **return true**
23: **end if**
24: **end for**
25: **return false**
26: **end function**
27: **function** $\mathrm{MtBr}(s,j,M,i)$
28: **if** $M(|\!)_i) = \bot$ **then**
29: **return** b_{MtBrE} {self-ref: $\varnothing$- vs. ε-semantics}
30: **end if**
31: $l \leftarrow M(|\!)_i) - M(|\!)_i)$
32: **return** $l \le |s| - j \ \wedge \ s[j..{<}j{+}l] = s[M(|\!)_i)..{<}M(|\!)_i)]$
33: **end function**

B Proof of Theorem 1

Proof. Algorithm 2 presents the algorithm for computing sink ambiguity (SinkAbgS). Recall that the degree of ambiguity counts the number of accepting paths, and that a sink automaton adds an ε-transition from every state in Q to a new accepting state q_{sink} (§2.2). Algorithm 2 aggregates all possible ways of reaching q_{sink} from each state.

Algorithm 3 computes backtracking runtime (BtRtS). For all transitions except backreferences, the runtime variable τ is incremented by a constant (Lines 10, 13, 16, 19). For a backreference transition, τ is incremented by the length of the captured substring (Line 23).

We now aim to scale up $\mathrm{BtRtS}(A,s)$ to $\mathrm{BtRtS}\uparrow$ so that it becomes a constant multiple of $\mathrm{SinkAbgS}(A,s)$. In other words, we seek to construct $\mathrm{BtRtS}\uparrow$ such that , $\exists$ constant ξ:

$$\mathrm{BtRtS}(A,s) \le \mathrm{BtRtS}\uparrow(A,s) \le \xi \cdot \mathrm{SinkAbgS}(A,s)$$

First, BtRtS currently returns a boolean indicating whether A accepts s, and it stops early upon acceptance. We can re-

move this early-stopping behavior to scale it up. The **if** statements at Lines 3 and 25 can be deleted, and the boolean return can be omitted.

Second, we move the constant additions out of the loops. To begin, we consider only backreferences that can match strings of maximum length $O(1)$. This corresponds to the first constraint of Theorem 1. Let $\mathrm{MaxFBrL}(A)$ denote the maximum finite backreference length among all backreferences. Formally,

$$\mathrm{MaxFBrL}(A) = 1 + \max_{\text{backref } \delta \in \Delta \ \wedge\ \delta \text{ match } O(1) \text{ length}} \text{length that } \delta \text{ can match}$$

Algorithm 2 Sink Ambiguity w.r.t. string (SinkAbgS)

Require: An 2PMFA $A = (Q, \Sigma, I, \Delta, q_0, F)$
Require: A string $s \in \Sigma^*$
Require: A current state $q \in Q$
Require: An index $j \in \mathbb{N}_{0..|s|}$ of s
Require: A memory function $M : \{(\!|'_i, (\!|_i, |\!)_i \mid i \in I\} \to \mathbb{N}_{0..|s|}$

$$\mathrm{SinkAbgS}(A,s) = \mathrm{SinkAbgS}'(A,s,q_0,0,\varnothing)$$

$$\mathrm{SinkAbgS}'(A,s,q,j,M) = 1 + \sum_{((q,t)\mapsto Q')\in\Delta} \sum_{q'\in Q'}$$

$$\begin{cases} \begin{cases} \mathrm{SinkAbgS}'(A,s,q',j+1,M) & j<|s| \wedge s[j]=t \\ 0 & \text{otherwise} \end{cases} & t \in \Sigma \\ \mathrm{SinkAbgS}'(A,s,q',j,M) & t = \varepsilon \\ \mathrm{SinkAbgS}'(A,s,q',j,M \lhd \{(\!|_i \mapsto j\}) & t \text{ is } (\!|_i \\ \mathrm{SinkAbgS}'(A,s,q',j,M \lhd \{(\!|_i \mapsto M((\!|'_i) + 1, |\!)_i \mapsto j\}) & t \text{ is } |\!)_i \\ \begin{cases} \mathrm{SinkAbgS}'(A,s,q',j+M(|\!)_i)-M((\!|_i),M) & \mathrm{MtBr}(s,j,M,i) \\ 0 & \text{otherwise} \end{cases} & t \text{ is } \backslash i \end{cases}$$

We can scale BtRtS' by replacing each $1+l$ with $\mathrm{MaxFBrL}(A)$. We can further scale up by removing all "$\tau := \tau + \tau' + \cdots$" statements at Line 10, 13, 16, 19, 23, and putting a statement "$\tau := \tau + \tau' + \mathrm{MaxFBrL}(A)$" called E at the end of "**for** $q \in Q$" loop (just above the Line 28).

After adding E, in each call to BtRtS', E may be evaluated up to the maximum number of ways to transition from a given current state to other states. Let $\mathrm{MaxOut}(A)$ denote such maximum number, formally:

$$\mathrm{MaxOut}(A) = \max_{q\in Q} \sum_{((q,t)\mapsto Q')\in\Delta} |Q'|$$

We can further scale up by replacing E_1 with "$\tau := \tau + \tau'$", and replacing the "$\tau := 0$" at Line 1 with "$\tau := \mathrm{MaxOut}(A) \cdot \mathrm{MaxFBrL}(A)$". Note that $\mathrm{MaxFBrL}(A), \mathrm{MaxOut}(A) \in O(1)$ with respect to $|s|$.

Next, we consider backreferences that can match strings of non-$O(1)$ length. The second constraint of Theorem 1 requires that "these backreferences are evaluated a total of $O(1)$ times." Let $\Delta_{\mathrm{IBr}} = \{\delta \mid \text{backref } \delta \in \Delta \wedge \delta \text{ matches string of non-}O(1)\text{ length}\}$. Define $\mathrm{IBrRCt}(A)$ as

Algorithm 3 Backtracking Runtime w.r.t. string (BtRtS)

Algorithm $\mathrm{BtRtS}(A,s)$
Require: An 2PMFA $A = (Q, \Sigma, I, \Delta, q_0, F)$
Require: A string $s \in \Sigma^*$
1: $(\tau, \alpha) := \mathrm{BtRtS}'(A, s, q_0, 0, \varnothing)$
2: **return** τ

Algorithm $\mathrm{BtRtS}'(A,s,q,j,M)$
Require: An 2PMFA $A = (Q, \Sigma, I, \Delta, q_0, F)$
Require: A string $s \in \Sigma^*$
Require: A current state $q \in Q$
Require: An index $j \in \mathbb{N}_{0..|s|}$ of s
Require: A memory function $M : \{(\!|'_i, (\!|_i, |\!)_i \mid i \in I\} \to \mathbb{N}_{0..|s|}$
1: $\tau := 0$
2: **for** $((q,t) \mapsto Q') \in \Delta$ **do**
3: **if** $q \in F \wedge j = |s|$ **then**
4: **return** (τ, true)
5: **end if**
6: **for** $q' \in Q'$ **do**
7: **switch** t **do**
8: **case** $t \in \Sigma$
9: $(\tau', \alpha') := \begin{cases} \mathrm{BtRtS}'(A,s,q',j+1,M) & j<|s| \wedge s[j]=t \\ (0, \mathrm{false}) & \text{otherwise} \end{cases}$
10: $\tau := \tau + \tau' + 1$
11: **case** ε
12: $(\tau', \alpha') := \mathrm{BtRtS}'(A,s,q',j,M)$
13: $\tau := \tau + \tau' + 1$
14: **case** $(\!|_i$
15: $(\tau', \alpha') := \mathrm{BtRtS}'(A,s,q',j,M \lhd \{(\!|'_i \mapsto j\})$
16: $\tau := \tau + \tau' + 1$
17: **case** $|\!)_i$
18: $(\tau', \alpha') := \mathrm{BtRtS}'(A,s,q',j,M \lhd \{(\!|_i \mapsto M((\!|'_i), |\!)_i \mapsto j\})$
19: $\tau := \tau + \tau' + 1$
20: **case** $\backslash i$
21: $l := \begin{cases} M(|\!)_i) - M((\!|_i) & M(|\!)_i) \neq \bot \\ 0 & \text{otherwise} \end{cases}$
22: $(\tau', \alpha') := \begin{cases} \mathrm{BtRtS}'(A,s,q',j+l,M) & \mathrm{MtBr}(s,j,M,i) \\ (0, \mathrm{false}) & \text{otherwise} \end{cases}$
23: $\tau := \tau + \tau' + 1 + l$
24: **end switch**
25: **if** α' **then**
26: **return** (τ, true)
27: **end if**
28: **end for**
29: **end for**
30: **return** (τ, false)

the maximum total evaluation count of infinite backreferences. Formally,

$$\mathrm{IBrRCt}(A) = |\Delta_{\mathrm{IBr}}| \cdot \max_{\delta\in\Delta_{\mathrm{IBr}}} \text{total number of evaluation of } \delta \text{ during one matching}$$

Note that $\mathrm{IBrRCt}(A) \in O(1)$ with respect to $|s|$. In each evaluation, a backreference transition can match a string of length at most $O(|s|)$ (i.e., up to the entire input string). Instead of computing the time consumed by backreferences in Δ_{IBr} recursively, we directly add the scaled-up time, $\mathrm{IBrRCt}(A) \cdot |s|$, to the result in BtRtS.

Algorithm 4 presents the scaled-up version of BtRtS, denoted $\mathrm{BtRtS}\uparrow$. By comparing $\mathrm{SinkAbgS}'$ and $\mathrm{BtRtS}\uparrow'$, we observe that they are structurally identical, differing only by

Algorithm 4 Backtracking Runtime w.r.t. string, Scaled Up (BtRtS↑)

Require: An 2PMFA $A = (Q, \Sigma, I, \Delta, q_0, F)$
Require: A string $s \in \Sigma^*$
Require: A current state $q \in Q$
Require: An index $j \in \mathbb{N}_{0..|s|}$ of s
Require: A memory function $M : \{(\!|_i', (\!|_i, |\!)_i \mid i \in I\} \to \mathbb{N}_{0..|s|}$

$$\mathrm{BtRtS}\uparrow(A,s) = \mathrm{BtRtS}\uparrow'(A,s,q_0,0,\varnothing) + \mathrm{IBrRCt}(A)\cdot|s|$$

$$\mathrm{BtRtS}\uparrow'(A,s,q,j,M) =$$

$$\mathrm{MaxOut}(A)\cdot\mathrm{MaxFRefL}(A) + \sum_{((q,t)\mapsto Q')\in\Delta}\ \sum_{q'\in Q'}$$

$$\begin{cases} \begin{cases} \mathrm{BtRtS}\uparrow'(A,s,q',j+1,M) & j<|s| \wedge s[j]=t \\ 0 & \text{otherwise} \end{cases} & t\in\Sigma \\ \mathrm{BtRtS}\uparrow'(A,s,q',j,M) & t=\varepsilon \\ \mathrm{BtRtS}\uparrow'(A,s,q',j,M \lhd \{(\!|_i \mapsto j\}) & t \text{ is } (\!|_i \\ \mathrm{BtRtS}\uparrow'(A,s,q',j,M \lhd \{(\!|_i \mapsto M((\!|_i'),\ |\!)_i \mapsto j\}) & t \text{ is } |\!)_i \\ \begin{cases} \mathrm{BtRtS}\uparrow'(A,s,q',j+M(|\!)_i)-M((\!|_i),M) & \mathrm{MtBr}(s,j,M,i) \\ 0 & \text{otherise} \end{cases} & t \text{ is} \backslash i \end{cases}$$

constant factors. Therefore,

$$\mathrm{BtRtS}\uparrow'(A,s,q,j,M) = \mathrm{MaxOut}(A)\cdot\mathrm{MaxFBrL}(A)\cdot\mathrm{SinkAbgS}'(A,s,q,j,M)$$

Plugging this into BtRtS↑ in Algorithm 4, and subsituting SinkAbgS from Algorithm 2, we obtain

$$\mathrm{BtRtS}\uparrow(A,s) = \mathrm{MaxOut}(A)\cdot\mathrm{MaxFBrL}(A)\cdot\mathrm{SinkAbgS}(A,s) + \mathrm{IBrRCt}(A)\cdot|s|$$

Since $\mathrm{BtRtS}(A,s) \le \mathrm{BtRtS}\uparrow(A,s)$, we have

$$\mathrm{BtRtS}(A,s) \le \mathrm{MaxOut}(A)\cdot\mathrm{MaxFBrL}(A)\cdot\mathrm{SinkAbgS}(A,s) + \mathrm{IBrRCt}(A)\cdot|s|$$

We now consider three possible cases for SinkAbgS and show that there exists a constant ξ such that

$$\mathrm{BtRtS}(A,s) \le \xi\cdot\mathrm{SinkAbgS}(A,s)$$

Case 1 $\mathrm{SinkAbgS}(A,s) \in \Omega(|s|)$. Since $\mathrm{IBrRCt}(A)\cdot|s| \in O(|s|)$, adding it to an $\Omega(|s|)$ term does not change the asymptotic scale. Therefore, $\exists\xi : \mathrm{BtRtS}(A,s) \le \xi\cdot\mathrm{SinkAbgS}(A,s)$.

Case 2 $\mathrm{SinkAbgS}(A,s) \in O(1)$ (constant). We argue by contradiction that, in this case, A cannot contain any reachable loop. If a reachable loop existed, it would combine with the sink loop to create a two-overlap-loop structure, causing the sink automaton to be IDA, *i.e.,* $\mathrm{SinkAbgN}(A,n) \notin O(1)$. This contradicts the assumption.

Without reachable loops, A cannot contain any loop in capture group or cyclic reference in loop. Consequently, no capture group can match a string of non-$O(1)$ length, and the same holds for backreferences. Thus $|\Delta_{\mathrm{IBr}}| = 0$, then $\mathrm{IBrRCt}(A) = 0$. Hence, $\exists\xi : \mathrm{BtRtS}(A,s) \le \xi\cdot\mathrm{SinkAbgS}(A,s)$.

Case 3 $\mathrm{SinkAbgS}(A,s) \in \mathrm{Complement}(\Omega(|s|)) \setminus O(1)$. In other word, $\mathrm{SinkAbgS}(A,s)$ is less than $O(n)$ but greater than $\Omega(1)$.

In this case, we first show by contradiction that A must contain a reachable loop. If A had no reachable loops, then along any path, each transition could appear at most once. The total number of possible paths from q_0 to any state $q \in Q$, denoted as $|\Pi|$, would then be bounded by $\sum_{k=0}^{|\Delta|} \mathrm{P}^k_{|\Delta|}$, which is a constant with respect to $|s|$. In the $\mathrm{Sink}(A)$, every state has an ε-transition to the sink state, so $|\Pi|$ equals the number of paths from q_0 to the sink state, i.e., the degree of sink ambiguity. This would imply that $\mathrm{SinkAbgN}(A,|s|) \in O(1)$, contradicting the assumption of this case.

Next, we show by contradiction that every reachable loop in A must contain a backreference that matches a non-$O(1)$-length string. Suppose there exists a reachable loop consisting only of the following types of transitions: symbol, ε, capture-open, capture-close, or backreferences that match only $O(1)$-length strings. Then, the double-overlap structure created by such a loop together with the sink loop would yield at least $\Omega(|s|)$ sink ambiguity, contradicting the assumption.

Because no reachable loop can be formed solely from $O(1)$-transitions, at least one backreference in A must match strings of non-$O(1)$ length via cyclic referencing. Such a backreference is therefore evaluated non-$O(1)$ number of times and matches non-$O(1)$-length substrings. This violates the theorem's precondition, so this entire case does not need to be considered.

Considering the two valid cases above, we obtain

$$\begin{aligned} \exists\xi : \forall n \in \mathbb{N} : \exists\, s \in \Sigma^n : \mathrm{BtRtN}(A,n) &= \mathrm{BtRtS}(A,s) \\ &\le \xi\cdot\mathrm{SinkAbgS}(A,s) \\ &\le \xi\cdot\mathrm{SinkAbgN}(A,n) \end{aligned}$$

Therefore, $\mathrm{BtRtN}(A,n) \in O(\mathrm{SinkAbgN}(A,n))$. □

C Proof of Theorem 3

C.1 Proof for Pattern 1

Proof. Pattern 1 contains a path of the form

$$\pi_{\mathrm{prefix}} \xrightarrow{(\!|_i} \pi_{\mathrm{left}}\pi^*_{\mathrm{pump}}\pi_{\mathrm{right}} \xrightarrow{|\!)_i} \pi_{\mathrm{bridge}} \xrightarrow[s_{\mathrm{ref}}]{\backslash i} \pi_{\mathrm{suffix}}$$

Let $s_{\mathrm{prefix}} = \mathcal{S}(\pi_{\mathrm{prefix}})$. Assume that there exists a string s_{ovlp} such that $\mathcal{S}(\pi_{\mathrm{left}}) = s_{\mathrm{ovlp}}^{u_l}$, $\mathcal{S}(\pi_{\mathrm{pump}}) = s_{\mathrm{ovlp}}^{u_p}$, $\mathcal{S}(\pi_{\mathrm{right}}) = s_{\mathrm{ovlp}}^{u_r}$, and $\mathcal{S}(\pi_{\mathrm{bridge}}) = s_{\mathrm{ovlp}}^{u_b}$. In addition, assume there exists a string s_{nsuffix} such that $\mathcal{S}(\pi_{\mathrm{suffix}}) \neq s_{\mathrm{nsuffix}}$.

For any $n' \in \mathbb{N}$, construct the input string

$$s = s_{\mathrm{prefix}}\, s_{\mathrm{ovlp}}^{2(u_l+n'u_p+u_r)+u_b}\, s_{\mathrm{nsuffix}}$$

During execution, the prefix path π_{prefix} first matches s_{prefix}. The loop π^*_{pump} then greedily matches as many copies of s_{ovlp} as possible. During backtracking, the number of iterations of π_{pump} is reduced by u_p at each step until it reaches zero. When π^*_{pump} matches between $n'u_p$ and 0 copies of s_{ovlp}, the bridge path π_{bridge} matches $s_{\text{ovlp}}^{u_b}$, after which the backreference $\backslash i$ is evaluated against

$$u_l + n'u_p + u_r,\ u_l + (n'-1)u_p + u_r,\ \cdots,\ u_l + u_p + u_r,\ u_l + u_r$$

copies of s_{ovlp}. In all cases, the suffix path π_{suffix} rejects on s_{nsuffix} or s_{ovlp}, forcing continued backtracking.

As a result, the backreference is evaluated n' times. The total time spent evaluating the backreference is

$$|s_{\text{ovlp}}| \sum_{k=0}^{n'} (u_l + ku_p + u_r) = |s_{\text{ovlp}}|(n'+1)(n'u_p/2 + u_l + u_r),$$

which is $\Omega(n'^2)$. Since the total input length is

$$|s| = |s_{\text{prefix}}| + (2 \cdot (u_l + n'u_p + u_r) + u_b)|s_{\text{ovlp}}| + |s_{\text{nsuffix}}|,$$

it follows that $n' \in \Theta(|s|)$. Therefore, the backreference time is $\Omega(|s|^2)$, and the overall matching time is not $O(|s|)$. □

C.2 Proof for Pattern 2

Proof. Pattern 2 consists of a path of the form

$$\pi_{\text{prefix}} \xrightarrow{(_i} \pi_{\text{left}}\pi^*_{\text{pump}}\pi_{\text{right}} \xrightarrow{)_i} \pi_{\text{fence}}\pi^*_{\text{loop}}\pi_{\text{bridge}} \xrightarrow[s_{\text{ref}}]{\backslash i} \pi_{\text{suffix}}$$

Let $s_{\text{prefix}} = \mathcal{S}(\pi_{\text{prefix}})$. Assume that there exists a string s_{ovlp} such that $\mathcal{S}(\pi_{\text{left}}) = s_{\text{ovlp}}^{u_l}$, $\mathcal{S}(\pi_{\text{pump}}) = s_{\text{ovlp}}^{n'_1}$, $\mathcal{S}(\pi_{\text{loop}}) = s_{\text{ovlp}}^{u_o}$, and $\mathcal{S}(\pi_{\text{bridge}}) = s_{\text{ovlp}}^{u_b}$. Let $s_{\text{right}} = \mathcal{S}(\pi_{\text{right}})$ and $s_{\text{fence}} = \mathcal{S}(\pi_{\text{fence}})$. In addition, assume that there exists a string s_{nsuffix} such that $s_{\text{nsuffix}} \neq \mathcal{S}(\pi_{\text{right}}\pi_{\text{suffix}})$.

Construct the input string

$$s = s_{\text{prefix}}\ s_{\text{ovlp}}^{u_l+n'_1}\ s_{\text{right}}\ s_{\text{fence}}\ s_{\text{ovlp}}^{n'_2u_o+u_b+u_l+n'_1}\ s_{\text{nsuffix}},$$

where $n'_1 \cdot n'_2 \notin O(|s|)$ (for example, $n'_1, n'_2 \in \Theta(|s|)$).

During execution, the prefix path π_{prefix} matches s_{prefix}, after which the capture group matches the substring $s_{\text{ovlp}}^{u_l+n'_1}\ s_{\text{right}}$. The fence path π_{fence} then matches s_{fence}. Subsequently, the loop π^*_{loop} greedily matches as many copies of s_{ovlp} as possible. During backtracking, the number of matched copies is reduced by u_o at each step until it reaches zero.

When π^*_{loop} matches between n'_2u_o and 0 copies of s_{ovlp}, the bridge path π_{bridge} matches $s_{\text{ovlp}}^{u_b}$. The backreference $\backslash i$ then attempts to match the prefix of the previously captured string, namely $s_{\text{ovlp}}^{u_l+n'_1}$. In all cases, the remaining suffix and the path π_{suffix} reject on s_{nsuffix} or s_{ovlp}, forcing further backtracking.

As a result, the backreference $\backslash i$ is evaluated $\Theta(n'_2)$ times, and each evaluation incurs a cost of $\Omega(u_l + n'_1)$. Consequently, the total time spent evaluating the backreference is $\Omega(n'_1n'_2)$. Since $n'_1n'_2 \notin O(|s|)$, the overall matching runtime is not in $O(|s|)$. □

C.3 Proof for Pattern 3

The proof for Pattern 3 is analogous to that for Pattern 2.

D Snort REwB ReDoS Exploits

Exploit 1

Rules SID 20156 Review 11, SID 20494 Review 19

Files `snapshot-29200/rules/file-pdf.rules`, `snapshot-29200/rules/file-identify.rules`

PCRE Regex

```
([A-Z\d_]+)\.write\x28.*?\1\.getCosObj\x28
```

Attack String

```
.write(.getCosObj(%PDF-Z.write(Z
```

(Repeat the 1st 'Z' 1000 times, the 2nd one 2000 times.)

Effect Slowing down 0.7-1.2 seconds.

Explanation The substring `'.write(.getCosObj('` satisfies the `content` constraint of the rule with SID 20156. The substring `'%PDF-'` triggers the rule with SID 20494, causing the `file.pdf` flowbit to be set. When the regex attempts to match the remaining portion of the input, it incurs $O(n^2)$ time per match due to Pattern 2. Because the regex is not anchored, the PCRE engine attempts to start matching at each `'Z'` character. Each attempt results in an $O(n^2)$ match. Consequently, the overall time complexity becomes $O(n^3)$.

Exploit 2

Rules SID 10417 Review 10

Files `snapshot-29200/rules/browser-plugins.rules`

PCRE Regex

```
(\w+)\s*=\s*(\x22JNILOADER\.JNILoaderCtrl\x22|\
x27JNILOADER\.JNILoaderCtrl\x27)\s*\x3b.*(\w+)\
s*=\s*new\s*ActiveXObject\s*\(\s*\1\s*\)(\s*\.\
s*(LoadLibrary)\s*\(|.*\3\s*\.\s*(LoadLibrary)\
s*\()|(\w+)\s*=\s*new\s*ActiveXObject\s*\(\s*(\
x22JNILOADER\.JNILoaderCtrl\x22|\x27JNILOADER\.
JNILoaderCtrl\x27)\s*\)(\s*\.\s*(LoadLibrary)\s
*\(|.*\7\s*\.\s*(LoadLibrary)\s*\()
```

Attack String

```
A='JNILOADER.JNILoaderCtrl';Z=new ActiveXObject
(A);Z.LoadLibrary('org.evil.Malicious');
```

(Repeat both 'Z's 2000 times.)

Effect Exceeds the backtracking limit, thereby bypassing alert.

Explanation Capture group 3 and its corresponding backreference place this regex in Pattern 2. In addition, the combination of capture group 3 with the preceding `/.*(\w+)/` introduces an IDA pattern. When the regex is matched against the attack string, the greedy `/.*/` initially consumes all occurrences of `Z` because it appears before `/(\w+)/`. The engine must then repeatedly backtrack until `(\w+)` can match the entire sequence of `Z` characters, allowing the backreference `/\3/` to match. In practice, PCRE reaches its backtracking limit before completing this search and aborts the match. Additionally, the input string is a snippet of malicious JavaScript code that can load arbitrary Java classes via ActiveX, illustrating a realistic exploitation scenario.