How RegEx Catastrophic Backtracking Becomes a Denial-of-Service Attack (ReDoS)

Vincent Rionarlie (13524031)
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung
Bandung, Indonesia
13524031@std.stei.itb.ac.id, vinctlie06@gmail.com

***Abstract*—Regular expressions (regex) are widely used for string matching, input validation, and parsing, yet a naively written regex can turn a backtracking matching engine into an unbounded loop. This phenomenon, known as catastrophic backtracking, lets an attacker craft a short, malicious input string that forces an exponential number of matching attempts, exhausting CPU resources and producing a Regular Expression Denial of Service (ReDoS). This paper traces the algorithmic root cause of catastrophic backtracking back to the structure of backtracking-based regex matching itself, points out the structural patterns of ambiguity (nested and overlapping quantifiers) that trigger it, and demonstrates the effect with a real, reproducible experiment. Using Python's built-in backtracking regex engine on the classic vulnerable pattern `(a+)+$`, matching time was measured for increasing input lengths and shown to grow exponentially, doubling for every extra character and closely matching the theoretical $O(2^n)$ bound, while a structurally equivalent but unambiguous pattern `a+$` matches in constant time. The paper also surveys documented real-world ReDoS incidents and discusses mitigation strategies, ranging from pattern rewriting to switching to automaton-based, linear-time engines, along with their respective trade-offs.**

***Index Terms*—regular expression, catastrophic backtracking, ReDoS, denial of service, nondeterministic finite automaton, string matching**

I. Introduction

While filling out an online form, one may have noticed that there are some input columns that restrict the user into typing a specific type of string, e.g. email strings (<name>@<domain>.<TLD>), phone numbers, time formats, etc. The secret behind the restriction is the clever design of regular expression, or more commonly known as regex, which is now used in almost every aspects of modern programming. It is one of the main reasons the codes we write can be run.

Regular expression started out as the models developed by McCulloch and Pitts in 1943, used to describe how the human nervous system works. Later, the term 'regular expression' was created by Stephen Kleene in 1956 who represented the McCulloch-Pitts models with algebra notation. In 1968, Ken Thompson, a Unix pioneer, implemented Kleene's idea into the text editor 'ed', with an aim to enable pattern match finding inside a text file. This was the first time regex started entering the computation world.

Regex works by matching a text input to a wanted pattern. A lot of built-in regex library in famous languages like Python, Ruby, Javascript, and PHP implement backtracking-based regex matching. This method supports additional features like backreference and lookaround assertions as backtracking provides the ability to shift back every time there is a mismatch between the pattern and the input character. However, this exact ability is the reason for the Denial-of-Service attack it allowed.

Regex DoS (ReDoS) attack may happen when the queried pattern allows for repeat of certain group or character. When an input repeats those certain group of characters such that it becomes the worst-case searching scenario, the CPU running the program will be used up to its full capacity. If an attacker supposedly sends such input to a remote victim, the same thing will happen to the their CPU, causing the infamous Denial-of-Service attack. The worse part of this attack is the fact that the attacker typically only needs to send a short length of payload, making it very easy in the crafting.

ReDoS has happened multiple times throughout cyber security history. In July 2019, Cloudflare— a globally known company that handles a significant portion of the world web traffic—suffered a worldwide outage caused by a single regular expression deployed in its Web Application Firewall rule set [1]. Studies of open-source package ecosystems have found that ReDoS-vulnerable patterns are common: Davis *et al.* found thousands of vulnerable regexes across the npm, PyPI, and RubyGems ecosystems [2], and Staicu and Pradel specifically measured the impact of ReDoS on JavaScript-based web servers [3].

This paper is aimed to uncover three main points. First, it explains the algorithmic mechanism of backtracking nature, while also relating it to the exponential-time worst-case behaviour, as well as explaining how the exact behaviour and regex structure altogether is able to cause the catastrophic DoS attack. Second, it provides an experiment that measures the worst-case behaviour using a real wall-clock, confirming the $O(2^n)$ growth with a specific Dos-causing regex structure, while also constrasting it with an equivalent, but linear-time structure. Third, it notes down mitigation strategies ranging from pattern rewriting to switch of regex engine (pure finite automata), and also discusses their respective trade-offs.

The rest of this paper goes as follows. Section II lays out the relevant background and the literature this paper builds on. Section III digs into the actual mechanism that makes

catastrophic backtracking happen. Section IV shows how a vulnerable pattern can be spotted and turned into a working attack string. Section V presents the wall-clock experiment and what it tells us. Section VI goes through mitigation strategies one by one. Section VII closes the paper.

II. Background and Literature Review

A. From Nerve Nets to Regular Expressions

As mentioned briefly above, regular expression did not start out as a programming tool at all. The original idea came from Warren McCulloch and Walter Pitts, who in 1943 published a model of how neurons in the human nervous system might fire and interact with one another [4]. Their model was purely biological in intention, but it turned out to carry a clean mathematical structure underneath it. It took until 1956 for Stephen Kleene to notice that structure and give it an algebraic notation, which he called ”regular events”—the direct ancestor of what is now simply called a regular expression [5]. So in a sense, every time one writes a regex to validate an email field, one is reusing a seventy-year-old idea about how nerve cells switch on and off.

The jump from algebra to an actual running program happened a decade later, when Ken Thompson, who would go on to co-create Unix, built Kleene’s notation into the line editor `ed` so that users could search for a pattern inside a text file rather than only an exact string [6]. This is usually considered the moment regex left pure mathematics and turned into a tool that programmers actually touch every day. Thompson’s implementation already used the automaton-based idea explained next, which is part of why his name keeps coming up whenever the ”correct”, linear-time way of doing regex matching is discussed.

B. Backtracking as a General Algorithm Strategy

Before connecting backtracking to regex matching specifically, it is worth stepping back to backtracking as it is taught as a general problem-solving strategy in algorithm courses, independent of regex altogether [7]. In this general formulation, a solution is built incrementally as a path down a conceptual *solution tree*: at every step the algorithm holds a partial solution and a set of *simpul hidup* (live nodes)—nodes that have been generated but not yet fully explored—one of which is singled out as the *simpul-E* (E-node), the node currently being expanded. Expanding the E-node means extending the partial solution by one more component and generating its children; each child is checked against a feasibility condition before being accepted as a new live node. The moment a partial solution is found to violate that condition, or otherwise cannot possibly be extended into a valid complete solution, the node is declared a *simpul mati* (dead node) and the algorithm *backtracks*: it abandons that branch and returns to the nearest live node still left unexplored, picking a different child to expand instead. The whole process is, in other words, a depth-first search over the solution tree that prunes a branch as soon as it is known to be hopeless, rather than generating every leaf of the tree first and checking each one only at the end.

This general scheme already explains, almost word for word, why a backtracking regex matcher behaves the way Section III describes. A position in the pattern paired with a position in the input string is exactly the matcher’s ”partial solution”; the matcher’s E-node is whichever choice point—typically, how many repetitions a quantified group should consume—is currently being tried; and a path becomes a *simpul mati* the instant the chosen repetition count leads to a mismatch later on, at which point the matcher backtracks to the same choice point and tries the next untried repetition count instead of having committed to one count from the start. Catastrophic backtracking, then, is simply what happens when the solution tree that this general strategy walks happens to have exponentially many live nodes at every level, which is precisely the structural ambiguity identified in Section IV.

C. Two Competing Matching Strategies

There turn out to be two very different ways to build a regex engine, and the choice between them is exactly why ReDoS is able to exist at all.

The first way, the one Thompson actually used, is to convert the regex into a nondeterministic finite automaton (NFA) and simulate it by keeping track of the entire *set* of states the automaton could be in after reading each character, instead of committing to one state at a time [6] [8]. Since that set can never grow larger than the number of states in the automaton—which is proportional to the length of the pattern—the whole matching process runs in time proportional to (pattern length) $\times$ (input length), no matter how ”ambiguous” the pattern looks on paper. Russ Cox wrote a well-known pair of essays explaining and reviving this approach for modern engines, work that eventually fed into Google’s RE2 library [9] [10], a library built specifically so that it can never blow up regardless of what pattern it is given.

The second way is backtracking, and it happens to be what most of the languages people actually use day to day—Python, JavaScript, Java, PHP, Ruby—ship as their default engine. A backtracking engine tries one path through the pattern at a time, and only goes back to try a different path once the current one leads to a dead end. This is, structurally, the same recursive-descent walk over the same kind of NFA mentioned above [8], except that now only one state is tracked at a time instead of the whole set. That single design choice is what opens the door to exponential running time: if a path only turns out to be a dead end after many characters have already been consumed, the engine has to retry every other path before it can be sure that no match exists at all.

Table I lines the two strategies up side by side. None of these two strategies is strictly ”better”; they are simply optimized for different things, which is exactly why both are still in widespread use today rather than one having fully replaced the other.

D. When Convenience Costs Speed

If backtracking is strictly worse in the worst case, why does almost every popular language still build on it? The answer

TABLE I
AUTOMATON-BASED VS. BACKTRACKING MATCHING

	Automaton-based	Backtracking
Worst-case time	$O(nm)$, always	$O(2^n)$, possible
Backreferences	not supported	supported
Lookaround	not supported	supported
Example engines	RE2, Go `regexp`, Rust `regex`	PCRE, Python `re`, Java, Perl
ReDoS-prone	no	yes, if pattern is ambiguous

is convenience. Backtracking engines can support features that a pure automaton has no way of representing, most notably backreferences (matching the exact same substring twice using something like `\1`) and lookaround assertions. These features became mainstream largely because of Perl, whose regex implementation in the late 1980s extended the "regular" expression syntax with precisely these non-regular features—a history that is documented in detail by Friedl in what is still treated as the standard reference book on the subject [11]. Once backreferences entered the picture, a regex engine could no longer be reduced to a finite automaton in the formal sense, so library authors simply kept the backtracking design from then on, even for the overwhelming majority of patterns that never use a backreference at all. This is, essentially, the root cause this whole paper is about: a feature most patterns never end up using is the same feature that leaves those patterns open to ReDoS.

E. Why Detecting This Automatically Is Hard

A reasonable question at this point is: if the ambiguity that causes ReDoS has a fairly clean definition (a state in the NFA reachable from itself through more than one path on the same input), why is it not simply caught by the regex compiler itself before the pattern is ever used? The honest answer is that fully solving this in general is harder than it looks. Weideman *et al.* approach the problem by computing the *ambiguity* of the underlying NFA and showing that this measure correlates with how badly a backtracking engine can blow up on it, but their own tool, `rxxr2`, has to fall back to approximations once the pattern grows large or uses certain combinations of features, since exhaustively reasoning about every possible input string an attacker could submit is itself an expensive search problem [12]. In other words, detecting a ReDoS-vulnerable pattern is, ironically, its own smaller instance of "search a huge space for a bad case", which is precisely the kind of computational difficulty that motivated practitioners to write lighter, heuristic-based tools such as `safe-regex` and `regexploit` instead of waiting for a fully precise analysis (see Section VI). This is part of why, even today, ReDoS-vulnerable patterns keep slipping into production code despite the underlying cause being well understood since at least the mid-2010s.

F. How the Industry Eventually Responded

These incidents did not go unnoticed by the people who maintain the languages and runtimes everyone relies on, and over time a few concrete safeguards started showing up. Microsoft's .NET runtime, for instance, now lets a developer pass a `matchTimeout` parameter directly into `Regex.Match`, which simply kills the match attempt once it has run past a chosen wall-clock budget—essentially turning the "defensive timeout" idea mentioned later in Section VI into a first-class, built-in option rather than something a developer has to wire up by hand. Go took a more drastic route from day one: its standard `regexp` package is implemented directly on top of RE2, so a Go programmer never gets the backtracking footgun in the first place, at the cost of not being able to write a backreference even if they wanted to [9]. Rust's `regex` crate makes a similar promise and advertises its linear-time guarantee as a selling point rather than a footnote, which says something about how seriously the ecosystem now takes this class of bug compared to, say, fifteen years ago when Perl-style backtracking was simply the unquestioned default everywhere. None of these safeguards make catastrophic backtracking disappear as a concept—a developer using Python or Java today still has to actively choose to avoid it—but they do show that the industry's response to ReDoS has shifted from "patch the one regex that broke" toward "change the default so the whole class of bug becomes opt-in instead of opt-out."

G. Documented Incidents and Empirical Studies

ReDoS has caused real damage more than once, and not just at Cloudflare. Perhaps the earliest widely reported case happened to Stack Overflow itself: in July 2016, a regular expression used to trim whitespace inside a CSS-like string pinned the site's web servers' CPUs to 100% and took the site down for over half an hour, an incident the engineering team later wrote up in detail on their own blog [13]. A similar story repeated itself at a much larger scale in July 2019, when Cloudflare—whose infrastructure handles a significant slice of the world's web traffic—suffered a global outage because of a single overly permissive regular expression deployed in its own firewall rule set [1].

These are not isolated incidents either. Davis *et al.* scanned the npm, PyPI, and RubyGems package ecosystems and found thousands of regexes exhibiting exactly the kind of ambiguity this paper describes, many of them sitting inside packages that other projects depend on without anyone noticing [2]. Staicu and Pradel went further and specifically measured how many real, already-deployed Node.js web servers could be frozen with a single crafted HTTP request, confirming that the problem is not just theoretical but is sitting in production code right now [3].

III. HOW CATASTROPHIC BACKTRACKING HAPPENS

Now that the two competing strategies have been laid out, this section goes one level deeper into why the backtracking strategy specifically is the one that can blow up, using almost the same reasoning one would use to explain why an unpruned brute-force search blows up.

A. Backtracking as Brute-Force Search

A backtracking regex matcher is, structurally, nothing more than a brute-force search over a tree of candidate paths. Algorithm 1 sketches a minimal version of it for a pattern made of literals, concatenation, alternation, and the plus quantifier. Every time the matcher reaches a quantified group, it has no way of knowing in advance how many repetitions it should consume before moving on to the rest of the pattern, so it simply tries them one by one—the exact same "guess, then undo the guess if it does not work out" logic used to brute-force a constraint satisfaction problem by depth-first search.

Algorithm 1 Backtracking regex matcher (simplified)

```
function MATCH(pattern p, string s, position i)
if p is empty then
    return i = |s|
end if
if p starts with literal c then
    if i < |s| and s[i] = c then
        return MATCH(rest(p), s, i + 1)
    else
        return FALSE
    end if
end if
if p starts with (sub)+ then
    {try consuming 1, 2, 3, ... repetitions of sub}
    for each way to consume k ≥ 1 repetitions of sub
    ending at position j do
        if MATCH(rest(p), s, j) then
            return TRUE
        end if
    end for
    return FALSE
end if
```

B. Where the Exponent Comes From

The trouble starts the moment a quantified group can match the very same piece of input in more than one way. Take `(a+)+`: the inner group `(a+)` on its own can already consume any run of one or more a's, so a run of k consecutive a's can be split into exactly 2^{k-1} different ordered sequences of one-or-more-length pieces. The matcher has no way of knowing in advance which split, if any, eventually leads to a full match, so in the worst case—typically when the match is destined to fail anyway, because the string is one character too long or ends with one disqualifying character—it has to try every single one of those 2^{k-1} splits before it can finally give up. That is the entire mechanism behind catastrophic backtracking: an exponential number of ways to explain the same input, paired with a backtracking matcher that insists on trying every single one of them before failing.

IV. Structural Analysis and Attack Construction

A. Spotting a Vulnerable Pattern

The literature on ReDoS [2], [12] calls a regex *ambiguous*, and therefore potentially exponential, whenever its corresponding NFA has a state that can be reached from itself through more than one distinct path while consuming the same input substring. In practice, this ambiguity usually takes form in:

- **Nested quantifiers**: a quantified group sitting inside another quantified group, e.g. `(a+)+`, `(a*)*`, `(a+)2,`.
- **Overlapping alternation under repetition**: alternatives inside a repeated group that can match overlapping substrings, e.g. `(a|aa)+`, `(a|a)*`.
- **Adjacent quantifiers with overlapping character classes**: e.g. `\s*\s*`, or the historically vulnerable email-validation pattern `(\w+\s*)+`, where `\w` and `\s` do not overlap, yet the outer + can still re-split a long run of word characters in many equivalent ways once a non-matching suffix forces the engine to backtrack.

B. Building the Attack String

Given a vulnerable pattern, an attacker builds the attack string out of two parts: first, a long *pump* substring—typically n repetitions of a character accepted by the ambiguous inner group—that maximizes the number of equivalent partitions the matcher has to try, and second, a short *suffix* that only makes the overall match fail after the matcher has already exhausted the rest of the pattern, forcing it to backtrack through every partition of the pump before giving up. For `(a+)+$`, the pump is n copies of `a` and the suffix is a single non-`a` character (say, `X`), since the trailing `$` anchor can never be satisfied once that character shows up, but the engine has no way of knowing this until it has already tried every way of partitioning the pump. This same shape is applied to other ambiguous patterns simply by replacing the pump characters or alternation choice.

V. Experiment and Analysis

A. Experimental Setup

To check the exponential blow-up described in Sections II and III against an actual running engine, an experiment was carried out using Python's built-in `re` module, which implements a classic backtracking matcher. Two patterns were compared:

- **Vulnerable pattern**: `(a+)+$`, an ambiguous nested-quantifier pattern.
- **Safe pattern**: `a+$`, which recognizes exactly the same language but contains no internal ambiguity at all.

For each input length $n \in \{15, 16, \ldots, 27\}$, an attack string made of n copies of `a` followed by a single failing character `X` was generated, so that the overall match only fails once the matcher has tried every alternative. Wall-clock matching time was measured with Python's `time.perf_counter()` on a single 'run per data' point (lengths beyond $n = 27$ were not measured, since the projected running time already crosses several minutes, which would make the experiment hard to reproduce on an ordinary laptop).

B. Results

Table II reports the measured matching time, in seconds, for the vulnerable and safe patterns at selected input lengths, along with the ratio of consecutive vulnerable-pattern timings.

TABLE II
MATCHING TIME VS. INPUT LENGTH n

n	Vulnerable (s)	Safe (s)	Ratio to $n-3$
15	0.001489	0.000044	—
18	0.010972	0.000010	7.37
21	0.090064	0.000012	8.21
24	0.758459	0.000013	8.42
27	5.903191	0.000015	7.78

Across the whole range $n = 15$ to $n = 27$, measured at single-character increments, the matching time of the vulnerable pattern grows by a factor of roughly 2.0 for every additional character, while the safe pattern's matching time just sits in the noise floor of the measurement (on the order of $10\,\mu s$), basically not caring about n at all. The total slowdown observed from $n = 15$ to $n = 27$ comes out to $5.903191/0.001489 \approx 3964$, which results in a number around 4% of the theoretical prediction $2^{12} = 4096$ for twelve extra characters—confirming the $O(2^n)$ growth predicted in Section III (2^{k-1} partitions for a run of k a's). If we follow this trend further, an input of only 50 characters would already be expected to take up to 10^4 seconds (several hours) just to fail to match—which says a lot about how short a payload an attacker actually needs to cause a multi-hour CPU stall, and is really the whole reason ReDoS is such a cheap denial-of-service method to be pulled off by an attacker.

C. A Second Pattern: Alternation-Based Ambiguity

To check that this exponential blow-up is a result of structural ambiguity in general, and not just something specific to nested quantifiers, a second vulnerable pattern was tested: `(a|aa)+$`, where the ambiguity comes from two alternatives of overlapping length (1 and 2 characters) instead of a nested quantifier. The same attack-string construction was reused: n copies of `a` followed by a disqualifying `X`. Table III reports the measured matching time.

TABLE III
MATCHING TIME FOR THE ALTERNATION-BASED PATTERN `(a|aa)+$`

n	Time (s)	Ratio to $n-1$
20	0.001066	—
24	0.009169	1.71 (avg./step)
27	0.036964	1.59 (avg./step)
30	0.148665	1.59 (avg./step)

Unlike `(a+)+$`, whose matching time doubles for every additional character, `(a|aa)+$` only grows by a factor of about 1.6 per character—close to the golden ratio $\varphi = (1+\sqrt{5})/2 \approx 1.618$. This is not a coincidence: the number of ways to split a collection of n a's into pieces of length 1 and 2 (the two alternatives of the inner group) is exactly the n-th Fibonacci number, which is well known to grow as φ^n. This second experiment shows that the growth rate of catastrophic backtracking is a result of certain combinatorics of the ambiguity inside the pattern itself: nested quantifiers over a single repeated unit give $O(2^n)$ partitions, while overlapping alternatives give the slower, but still exponential, $O(\varphi^n)$ partitions. Both are equally unacceptable in a production system, since both eventually grows exponentially as n grows, but the comparison does say something useful about how the *rate* of the explosion depends on exactly how the ambiguity is shaped, which is pretty useful to know when deciding which patterns to fix first while auditing a broken codebase.

D. Analysis

These results back up two of this paper's main points. First, the exponential blow-up is not just a theoretical worst case sitting in a textbook somewhere; it is a real and reproducible in widely used production regex engines, using a pattern that looks completely innocent at first glance. Second, the safe rewrite `a+$` is not just *asymptotically* better; across the whole tested cases, it is practically indistinguishable from constant time. This happens because removing the internal ambiguity decreases the number of equivalent partitions the matcher has to consider from 2^{n-1} down to exactly 1. This shows that fixing ReDoS at the pattern level does not require giving up or replacing the backtracking engines altogether—it only requires getting rid of the ambiguity identified in Section IV.

VI. MITIGATION STRATEGIES

A. Pattern Rewriting

As shown by the experiment above, a lot of vulnerable patterns can be rewritten into an equivalent, unambiguous pattern without changing the language they recognize, simply by getting rid of redundant nested or overlapping quantifiers (e.g. `(a+)+` $\rightarrow$ `a+`). This is the cheapest fix available and should really be the first line of defense, but it does require a developer (or a static analysis tool) to actually notice the ambiguous shape, which is not always obvious once the pattern gets more complicated than the examples used in this paper.

B. Atomic Groups and Possessive Quantifiers

Engines such as PCRE and Java's `java.util.regex` support atomic groups `(?>...)` and possessive quantifiers (`a++`, `a*+`), which tell the matcher to commit to the first successful match of the sub-pattern and never backtrack into it again. Wrapping the inner quantified group of a vulnerable pattern in an atomic group, e.g. `(?>a+)+`, removes the re-partitioning behaviour, at the cost of no longer being able to match certain edge-case inputs that actually need backtracking into that group.

C. Automaton-Based Linear-Time Engines

The most robust fix is to just use a regex engine that implements Thompson's construction (Section II) and therefore guarantees $O(nm)$ worst-case matching time no matter how the pattern is structured, at the cost of dropping support for

backreferences and most lookaround assertions. Google's RE2 library, and, for a large subset of syntax, Rust's `regex` crate, take exactly this approach [9], [10]. For things like input validation and WAF rule matching—which is precisely the use case behind the Cloudflare incident—this trade-off is almost always worth it, since backreferences are rarely something a validation rule actually needs.

D. Defensive Engineering Measures

Regardless of which regex engine is chosen, two extra defensive measures are commonly recommended: first, putting a hard timeout on every regex match that runs on untrusted input, so a suspicious match gets killed instead of being allowed to occupy a thread forever; and second, limiting the length of untrusted input before it is ever handed to a regex matcher, which directly limits the worst-case exponent as well. Neither methods actually fixes the underlying ambiguity issue, but both shrink the damage that could be done. Static analysis tools that look for ambiguous NFA structure before deployment, such as the ambiguity-based detector proposed by Weideman *et al.* [12], together with the general guidance summarized by OWASP [14], can additionally be wired into a CI pipeline to catch vulnerable patterns before they ever reach production.

E. Automated Detection Tools

Because spotting ambiguous structure by eye does not scale to a large codebase, a handful of tools exist specifically to flag ReDoS-vulnerable patterns automatically. Weideman *et al.*'s `rxxr2` statically analyzes a regex's NFA for ambiguous states reachable from themselves and reports the resulting worst-case complexity class [12]. Lighter-weight tools such as `safe-regex` (often used as an npm lint rule) and `regexploit` apply simpler heuristics—for instance, flagging any quantified group nested inside another quantified group, or any alternation with overlapping prefixes inside a repetition—and can even try to automatically synthesize a pump string and suffix of the kind described in Section IV, confirming exploitability empirically before a developer has to reason about NFA ambiguity by hand. Wiring such a tool into continuous integration, right alongside ordinary unit tests, is a fairly low-effort way to catch the structural patterns described in this paper before a vulnerable regex ever gets merged into production code.

VII. Conclusion

This paper looked at ReDoS as a direct algorithmic consequence of using exhaustive backtracking search to implement regex matching: whenever a pattern contains a quantified sub-expression that can match the same input in more than one way, a backtracking engine may have to enumerate an exponential number of equivalent partitions before it can be sure that no overall match exists, which is really just the worst case of brute-force backtracking search on an unpruned search tree, wearing a regex costume. The experiment in Section V confirmed this on an actual production-grade backtracking engine, showing that the classic pattern `(a+)+$` has matching time that doubles with every extra input character and closely tracks the theoretical 2^{k-1} partition count, while a structurally equivalent but unambiguous rewrite, `a+$`, matches in effectively constant time across the same range. Put together with documented real-world incidents such as the 2016 Stack Overflow outage and the 2019 Cloudflare outage, these results make the point that ReDoS is a practical, low-cost denial-of-service vector that comes from a well-understood and entirely avoidable algorithmic pitfall. Developers can already cut this risk down a lot today through pattern rewriting and atomic grouping, while organizations operating at scale should additionally think about migrating security-critical regex matching (e.g. WAF rules and input validators) over to automaton-based, linear-time engines, on top of match timeouts and pre-deployment static ambiguity analysis. Future work could extend this experiment to other engines (e.g. JavaScript's V8 and PCRE2) and to a wider benchmark of real-world vulnerable patterns mined from public package registries, in order to quantify how much each mitigation choice actually trades matching speed for supported regex features.

Appendix

The following Python script was used to produce the timing measurements reported in Table II and Table III, using only the standard library `re` and `time` modules so that the result reflects the backtracking engine shipped with a stock Python interpreter (no third-party regex engine was involved).

```
import re, time, sys

vulnerable = r'(a+)+$'    # or r'(a|aa)+$'
safe = r'a+$'

for n in range(15, 31, 1):
    s = "a" * n + "X"     # forces full
        backtracking,
                          # then fails to
                              match
    start = time.perf_counter()
    re.match(vulnerable, s)
    t_vuln = time.perf_counter() - start

    start = time.perf_counter()
    re.match(safe, s)
    t_safe = time.perf_counter() - start

    print(n, t_vuln, t_safe)
    if t_vuln > 5:        # abort before
        runtime
        break             # becomes
            impractical
```

Each measurement in this paper is a single wall-clock run; given that the measured times already span more than three orders of magnitude and closely track the predicted 2^{k-1} and Fibonacci growth, repeated trials were not necessary to support the qualitative conclusion. Measurements were taken on a single machine, so absolute timings are illustrative of relative growth rather than meant to be compared across hardware.

REFERENCES

[1] Cloudflare, "Details of the cloudflare outage on july 2, 2019," https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/, 2019, accessed: 2026-06-18.

[2] J. C. Davis, C. A. Coghlan, F. Servant, and D. Lee, "The impact of regular expression denial of service (redos) in practice: An empirical study at the ecosystem scale," in *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2018, pp. 246–256.

[3] C.-A. Staicu and M. Pradel, "Freezing the web: A study of redos vulnerabilities in javascript-based web servers," in *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, 2018, pp. 361–376.

[4] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *Bulletin of Mathematical Biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[5] S. C. Kleene, "Representation of events in nerve nets and finite automata," in *Automata Studies*, C. E. Shannon and J. McCarthy, Eds. Princeton University Press, 1956, pp. 3–41.

[6] K. Thompson, "Programming techniques: Regular expression search algorithm," *Communications of the ACM*, vol. 11, no. 6, pp. 419–422, 1968.

[7] R. Munir, "Algoritma backtracking (bagian 1)," https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/15-Algoritma-backtracking-(2026)-Bagian1.pdf, 2026, bahan kuliah IF2211 Strategi Algoritma, Institut Teknologi Bandung. Accessed: 2026-06-19.

[8] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 3rd ed. Addison-Wesley, 2006.

[9] R. Cox, "Regular expression matching can be simple and fast," https://swtch.com/~rsc/regexp/regexp1.html, 2007, accessed: 2026-06-18.

[10] ——, "Regular expression matching: the virtual machine approach," https://swtch.com/~rsc/regexp/regexp2.html, 2009, accessed: 2026-06-19.

[11] J. E. F. Friedl, *Mastering Regular Expressions*, 3rd ed. O'Reilly Media, 2006.

[12] N. Weideman, B. van der Merwe, M. Berglund, and B. Watson, "Analyzing matching time behavior of backtracking regular expression matchers by using ambiguity of nfa," in *Proceedings of the 21st International Conference on Implementation and Application of Automata (CIAA)*. Springer, 2016, pp. 322–334.

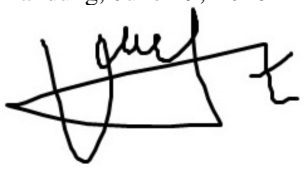
[13] N. Craver, "Stack exchange outage postmortem - july 20, 2016," https://stackstatus.tumblr.com/post/147710624694/outage-postmortem-july-20-2016, 2016, accessed: 2026-06-19.

[14] OWASP Foundation, "Regular expression denial of service - redos," https://owasp.org/www-community/attacks/Regular_expression_Denial_of_Service_-_ReDoS, oWASP Cheat Sheet Series. Accessed: 2026-06-18.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, June 19, 2026



Vincent Rionarlie (13524031)