

PUFFERDOS: Efficient and Effective Attack String Generation for Regular Expression Denial of Service Vulnerabilities

Shangzhi Xu^{†§}, Ziqi Ding[†], Xiao Cheng^{‡*}, Yuekang Li[†], Nan Sun[†], Benjamin Turnbull[†],
Shuangxiang Kan[†], Siqu Ma^{¶*1}

[†]The University of New South Wales, [‡]Macquarie University, [§]CSIRO, [¶]The University of Wollongong

*Corresponding Authors

{z5500277, ziqi.ding1, yuekang.li, nan.sun, benjamin.turnbull, shuangxiangkan}@unsw.edu.au xiao.cheng@mq.edu.au, siqim@uow.edu.au

Abstract—Regular Expression Denial-of-Service (ReDoS) attacks constitute a critical class of resource-exhaustion vulnerabilities. In such attacks, adversaries exploit the pathological worst-case execution behavior of regular expression (regex) engines to induce highly asymmetric computational workloads, ultimately exhausting system resources and degrading service availability. To protect systems against ReDoS attacks, numerous detection techniques have been proposed that simulate the attack process by generating attack strings to proactively exploit ReDoS vulnerabilities at the early development stage and facilitate remediation. Existing techniques broadly fall into two classes: static analyses that search for pathological regex structures, and dynamic exploration methods that synthesize candidate attack strings. However, the generated attack strings are often impractical for real-world exploitation because they usually assume unrealistic input-length budgets and do not validate the effectiveness and efficiency of the attack at the program level. Therefore, many generated strings fail to trigger vulnerable regexes when applied to real-world programs, further limiting the practical utility.

To address these shortcomings, we introduce an effective and efficient attack string generator, PUFFERDOS, designed to synthesize attack inputs that are both feasible within realistic length budgets and validated at the program level, enabling effective exploitation of ReDoS vulnerabilities in real-world programs. Specifically, we first define three vulnerable patterns based on our observation and formal verification. According to the patterns, PUFFERDOS conducts a synthesis technique to generate attack strings, and then refines and validates the strings with ReDoS-specific compositional concolic execution to guarantee real-world exploitability. We evaluated PUFFERDOS on a baseline corpus of 17,962 real-world vulnerable regexes. Compared to state-of-the-art tool RENGAR, PUFFERDOS produced attack strings that achieve efficiency gains ranging from 97.2× to 3872.4×, thereby significantly enhancing practical exploitability assessment in real-world contexts. PUFFERDOS also discovered 59 exploitable ReDoS instances across 12

popular projects, which is 25 more than RENGAR.

1. Introduction

Regular expression Denial of Service (ReDoS) is an attack that occurs when pathological regular expression (regex) constructs force a backtracking engine to explore a superlinear or exponential number of matching paths, enabling crafted inputs to exhaust CPU resources and disrupt services. Its severity is amplified by the pervasive use of regexes across diverse domains of computer science, including programming languages, file and data processing, database querying, and large language model pipelines [11], [18], [62]. Recent empirical studies show that ReDoS ranks as the fourth most prevalent vulnerability in the JavaScript npm ecosystem and the sixth in the Python PyPI ecosystem [28]. According to Snyk Security Research Team’s report [63], the number of disclosed ReDoS vulnerabilities has been steadily increasing, with a notable 143% surge observed in 2018. On the Common Vulnerabilities and Exposures (CVE) database, more than 350 ReDoS vulnerabilities have been disclosed since 2016 [12], [47]. Real-world incidents further demonstrate its impact: in June 2016, a catastrophic-backtracking regex caused a 34-minute outage on Stack Overflow, and in July 2019, a faulty WAF regex triggered a 27-minute global outage at Cloudflare [27], [31].

Numerous ReDoS detection and attack string generation tools have been developed to uncover vulnerable regexes and produce corresponding proof-of-concept attack inputs, enabling developers to recognize and address potential vulnerabilities early in the development cycle. Existing methods can be broadly categorized into static analysis-guided, dynamic exploration-based, and execution-validation-based (hybrid) approaches. Static analysis-guided methods [15], [35], [53], [56] identify ReDoS vulnerabilities by matching expert-defined regex patterns and constructing corresponding attack strings based on predefined templates. Dynamic exploration-based techniques [30], [34], [40] leverage fuzzing or guided input exploration to discover inputs that cause matching times to exceed a predefined threshold. Recent approaches integrate execution validation, combining

1. S. Ma is partially funded by the Australian Research Council (ARC) Future Fellowship (FT - 250100652)

pattern-guided analysis with dynamic verification to enhance accuracy [18], [50], [74]. Moreover, for other vulnerability classes beyond ReDoS, concolic execution is widely employed to ensure real-world exploitability [30], [34], [40].

However, despite the prevalence of ReDoS and recent progress in detection or attack generation, prior studies have shown that developers often perceive ReDoS vulnerabilities as low-severity issues and tend to ignore them rather than applying fixes. The main reasons are twofold: (1) the crafted attack strings to exploit ReDoS are typically unrealistically long, rendering such attacks impractical under most real-world deployment settings [12]; and (2) as noted in recent discussions [45], many reported attack strings lack practical effectiveness because they ignore data-flow constraints in the program, preventing them from reaching and triggering the vulnerable regex in actual applications. Although traditional concolic execution can, in principle, solve such constraints and facilitate real-world exploitation, its application to ReDoS remains limited due to the intrinsic challenges of symbolically modeling complex regex semantics. Thus, beyond early detection, it is equally important to generate efficient and practical effective attack strings to bridge the gap between theoretical detection and exploitation.

To this end, we present PUFFERDOS, a hybrid ReDoS attack string generation framework that provably constructs inputs inducing substantially greater matching cost than prior approaches while preserving practical exploitability. Specifically, we first observe that selecting appropriate characters and arranging them in a specific order within an attack string can force excessive backtracking and sharply increase matching cost; we then carry out a formal analysis and prove the conditions under which this effect arises and the specific forms of character selection and ordering that induce it. Guided by the analysis, we derive three regex vulnerable patterns, design corresponding attack string generation rules, and implemented them in PUFFERDOS to construct candidate attack strings for each vulnerable regex accordingly. PUFFERDOS further refines each candidate via a ReDoS-specific compositional concolic execution to ensure exploitability under realistic constraints. In implementation, PUFFERDOS can be integrated into any ReDoS detector as a plugin attack generator with moderate additional overhead.

In our evaluation, we compare PUFFERDOS against a state-of-the-art ReDoS attack string generator RENGAR [74] on both 17,962 vulnerable regexes and 31 exploited ReDoS CVEs. The results demonstrate clear performance advantages for PUFFERDOS. Empowered by rigorous reasoning and compositional concolic refinement, PUFFERDOS generates attack strings that are on average $97.2\times$ to $3,872.4\times$ shorter than those produced by RENGAR while achieving the same matching time thresholds of 0.1s, 1s, and 10s. PUFFERDOS also successfully reproduced 96.8% of the exploited ReDoS CVEs while RENGAR can only exploit 77.4%. To further assess PUFFERDOS’s capability to exploit previously undisclosed vulnerabilities, we apply PUFFERDOS to 12 open source projects with more than 10k monthly downloads and discovered 59 exploitable ReDoS vulnerabilities across them, demonstrating PUFFERDOS’s practical

effectiveness in real-world software systems.

In summary, we make the following contributions:

- We formally prove an observed effect that selecting specific characters and ordering them in attack strings amplifies regex matching cost; based on this, we derive three regex vulnerable pattern and devise provably efficient attack string generation rules.
- We develop PUFFERDOS, a hybrid ReDoS attack string generator that encodes formal generation rules into a scalable pipeline: it identifies vulnerable patterns, synthesizes compact attack strings, and uses compositional concolic refinement and ReDoS-specific function summaries to ensure real-world exploitability.
- We evaluate PUFFERDOS on 17,962 vulnerable regexes, 31 exploited ReDoS CVEs, and 12 open-source projects. PUFFERDOS produces attack strings $97.2\times$ – $3,872.4\times$ more efficient than the baseline tool, reproduces 96.8% of exploited CVEs, and uncovers 59 previously undisclosed vulnerabilities across 12 projects; by submission, developers from six projects have validated 21 of them.

2. Background

2.1. Regular Expression Denial of Service (ReDoS)

A ReDoS attack forces a regex matcher to explore a polynomial- or exponential-scale number of alternative matches, consuming CPU resources and rendering the service unresponsive. It is triggered by supplying an input that drives the matcher into its worst-case behavior, causing repeated backtracking over all possible match paths. To execute such an attack, one must analyze the structure of regexes, identify vulnerable patterns, and generate attack strings accordingly. A formal understanding of regexes is therefore required.

Regular Expression (Regex). In formal language theory [29], [71], a regex is a formalism for describing string patterns and can be converted into an equivalent automaton [37]; it explores all possible state paths in the automaton to check whether a given string matches a specified pattern. Let Σ denote a finite alphabet, and let Σ^* be the set of all finite strings over Σ . The symbol ε represents the empty string and \emptyset the empty set. We use \mathbb{N} to denote the set of natural numbers and ∞ to represent infinity. A regex over Σ is generated by the grammar:

$$r ::= a \mid [C] \mid \varepsilon \mid \emptyset \mid (r'_1 r'_2) \mid (r'_1 \mid r'_2) \mid r'^Q.$$

where $a \in \Sigma$, $C \subseteq \Sigma$ is a character set, juxtaposition denotes concatenation and “ \mid ” denotes alternation. r' is a *subregex* of r , i.e., a syntactic fragment of r corresponding to a connected subgraph of its automaton. The language $\mathcal{L}(r) \subseteq \Sigma^*$ of a regex r is defined as the set of all strings over Σ that are accepted by r .

In practice, regexes are usually used to match inputs of varying lengths. To achieve this, a regex often contains subregexes of the form r'^Q , where $Q \in (*, ?, +, \{m, \}, \{m, n\})$ defines how characters matched by r' repeat within the strings of $\mathcal{L}(r)$: “?” denotes optional occurrence, “*” “+”

denote zero or more and one or more repetitions, respectively, and $\{m, \}$ and $\{m, n\}$ where $m, n \in \mathbb{N} \cup \{\infty\}$ denote bounded repetitions. We define such quantified subregexes as *Loop Expressions*.

Definition 1 (Loop Expression (LE)). Let r' be a subregex of r . A Loop Expression (LE) refers to a subregex r' that is annotated with a quantifier symbol or a range such as $*$, $?$, $+$, $\{m, \}$, or $\{m, n\}$. Each LE is associated with an alphabet $\Sigma(\text{LE})$, defined as the set of input symbols it can match. For example, for $r = a^*b$, the subregex $r' = a^*$ is a LE and $\Sigma(\text{LE}) = \{a\}$.

The unfolding function \mathcal{U} takes a regex r as input, recursively splits r by “|” until no further nested alternation remains, and returns a set of subregexes of r obtained through the splits. Formally, $\mathcal{U}(r) = \{r'_1, \dots, r'_k\}$. For a LE, the unfolding conditionally removes or rewrites its quantifier before splitting its body by “|”, producing $\mathcal{U}(\text{LE}) = \{r'_1, \dots, r'_l\}$.

However, as LEs can match variable length substrings, the regex matching engine must dynamically explore all potential consumption boundaries for each LE in the input string, forming the core of regex matching.

Regex Matching. Formally, a regex matching function $\mathcal{M}(r, s)$ takes a regex r and a concrete string $s \in \Sigma^*$ as input and returns a Boolean indicating whether $s \in \mathcal{L}(r)$. Spencer’s [64] (i.e., backtracking) and Thompson’s [67] algorithms represent two classical approaches to regex matching. The former performs a constrained depth-first search, whereas the latter employs a breadth-first traversal. Among them, Spencer’s algorithm is widely adopted due to its implementation simplicity [17].

Specifically, Spencer’s algorithm greedily extends each LE to consume its longest possible match, where we define the termination position as a stop point. Upon a matching failure, it backtracks each LE with an untried stop point to resume matching.

Definition 2 (Stop Point). A stop point p is an index in the input string s that marks the end of a valid match for the current LE, formally satisfying LE_i matches $s[p_{i-1} : p_i]$ and $\neg(\text{LE}_i \text{ matches } s[p_i + 1])$ when $s \in \mathcal{L}(r)$. For example, for $\text{LE} = a^*$ in $r = a^*b$ and input $s = aaa..ab$, the stop point is at index -2.

Formally, given a regex r and input $s \in \Sigma^*$, Spencer’s algorithm processes r as an ordered sequence of subregexes (r'_1, \dots, r'_k) . For each r'_i , the matcher scans s left to right, consuming the longest prefix matching r'_i . If r'_i is a LE, it greedily locates its last stop point p_i on s and then recursively matches $s[p_i + 1 :]$ against $r'_{i+1} \dots r'_k$. If the overall match fails for s , the matcher backtracks the rightmost LE with an untried stop point $p'_i < p_i$, sets p'_i as its new stop point, and resumes matching from p'_i .

ReDoS. While easy to implement, Spencer’s algorithm is vulnerable to *catastrophic backtracking*, which occurs when matcher performs polynomial or exponential backtracking before concluding $s \notin \mathcal{L}(r)$. For example, when there are k adjacent LEs in r and all of them can consume the

same substring of s with length n , the matcher must try every possible split of these n characters among the k LEs [39], yielding $O(n^{k-1})$ backtracking steps (*polynomial backtracking*). If the LEs are nested, each backtracking of an outer loop may trigger a complete re-exploration of its inner loop, doubling the search space at each level and resulting in exponential backtracking blowup, for example $O(2^n)$.

Definition 3 (ReDoS). A ReDoS is caused by adjacent LEs consume overlapping portions of the same input substring, causing the matcher to explore an exponential or polynomial number of backtracking paths before rejecting. For example, for $r = .*a.*b$ and $s_a = aa..ac$, the matcher tries different splits of “aa..a” between the two LEs before failing.

In real-world systems, when vulnerable regexes are used to process untrusted inputs, malicious attack strings can cause the system to hang by inducing excessive backtracking, leading to Denial-of-Service attacks, or ReDoS attacks.

2.2. ReDoS Attack String Generation

Given the prevalence of ReDoS vulnerabilities, many studies have proposed methods to detect such vulnerabilities and to generate corresponding attack strings for open-source software. At a high level, for each vulnerable regex r , an attack string s_a of length $|s_a|$ is generated to trigger catastrophic backtracking and exploit the denial-of-service behavior. To generate a valid s_a , prior work decomposes r into $r = \varphi_1 \varphi_2 \varphi_3$, where φ_1 is the *prefix* occurring at the start of r , φ_2 is the *infix* immediately following φ_1 and containing all LEs that induce backtracking, and φ_3 is the *suffix* occurring after φ_2 .

Definition 4 (Repetition Unit). A repetition unit y for a attack string is a nonempty string $y \in \mathcal{L}(\varphi_2)$ such that y^q is well defined for some $q \geq 1$ and whose excessive repetition causes the LEs in φ_2 to exhibit exhaustive backtracking. For example, for $r = .*a.*b$ with $s_a = aaa..ac$, the repetition unit is $y = a$.

The attack string is constructed as $s_a = x y^q z$, such that $x \in \mathcal{L}(\varphi_1)$, repetition unit $y \in \mathcal{L}(\varphi_2)$ which will be repeated q times, and $z \notin \mathcal{L}(\varphi_3)$, thus $s_a \notin \mathcal{L}(r)$, forcing exhaustive backtracking [74]. Specifically, existing attack string generation methods fall into three categories:

Static Analysis. Static analysis based generators such as regexploit detect syntactic patterns of vulnerable regexes, for example adjacent or nested LEs, and apply predefined construction rules to rapidly produce candidate attack strings [15]. Automata-based approaches instead translate the regex into an automaton and reason about matching behaviour, offering stronger formal guarantees [35], [53], [56]. While these methods are computationally cheap and produce candidates quickly, their precision is limited because they do not capture the full runtime semantics of regex matching.

Dynamic Exploration. Dynamic exploration based methods treat attack generation as a search problem, executing the regex over many mutated inputs and observing execution matching costs to identify inputs that induce high matching

cost [10], [46], [54], [61]. However, they suffer from two major limitations: (1) they require substantial computational resources to evaluate a large number of candidates, and (2) their input space coverage remains limited, which may cause them to miss inputs that trigger higher regex matching costs. **Synthesis with Executional Validation (Hybrid Methods).** To balance coverage and computational overhead, recent tools adopt executional validation mechanisms. At the regex level, hybrid approaches combine pattern-guided candidate generation with dynamic execution validation [18], [50], [74]. Candidate attack strings are first generated statically and then executed dynamically to retain only those whose matching time exceeds a predefined threshold. Although effective for trigger vulnerable regexes in isolation, these methods often fail to capture the propagation and data-flow constraints that determine whether an input can reach the vulnerable regex in real-world programs.

In traditional vulnerability exploitation, at the program level, concolic execution enables such real-world effectiveness validation by simultaneously tracking concrete inputs and symbolic expressions, and solving path constraints to explore new execution paths and trigger target behaviors (e.g. vulnerability manifest) [9], [19], [24]. While classical DART-style [14], [24] concolic executors face exponential path explosion when analyzing programs with multiple functions and branching paths, SMART-style [23] compositional concolic executors mitigates this by testing functions in isolation, summarizing their behaviors, and reusing these summaries in higher-level analyses [34]. This strategy reduces redundant exploration and improves scalability; however, the inherent difficulty of symbolically modeling regex semantics can still cause the number of summary entries to grow exponentially, reintroducing scalability challenges when exploiting ReDoS in large programs. Consequently, concolic execution is seldom used for ReDoS attack string generation, despite wide use in other vulnerability classes.

2.3. Motivation

Existing works typically generate attack strings by analyzing the regexes in isolation, producing inputs that merely exceed the time consumption threshold. This approach raises two concerns: (i) whether attack strings of similar length that **incur higher matching cost** exist, and (ii) whether generated inputs can **exploit ReDoS in real-world systems**. In other words, they focus only on the success of attack string generation but ignore both the efficiency and the practical effectiveness. Previous studies and discussions have shown that, without an efficient and effective attack string, developers will often assign the reported vulnerability low priority [12], mark it as low severity [3], [4], or reject it [5] as they consider the issue unlikely to be exploitable in real systems, leaving the underlying risk unresolved. For instance, some existing tools require inputs of 100K to 1M characters to trigger a 10-second slowdown, yet common deployment environments enforce strict input constraints such as the 8 KB HTTP header limit in nginx and Apache Tomcat [12], rendering such attacks impractical in practice.

```

# xml_decl_re = re.compile(r'\s*<?\xml[^\>]*\?>')
# doctype_lookup_re = re.compile(r'<!DOCTYPE\s+')
# tag_re = re.compile(r'<(.*?)\s.*?>.*?</.*?>')

1. def is_xml(text):
2.     """Check if a doctype exists"""
3.     if check_xml(text):
4.         return True
5.     key = hash(text)
6.     try:
7.         return _looks_like_xml_cache[key]
8.     except KeyError:
9.         text = check_doctype(text)
10.        rv = tag_re.search(text[:1000]) is not None
11.        _looks_like_xml_cache[key] = rv
12.        return rv

-----
13. def check_doctype(text):
14.     m = doctype_lookup_re.search(text)
15.     if m is not None:
16.         return re.sub(doctype_lookup_re, "", text)
17.     return text
18.
19. def check_xml(text):
20.     return xml_decl_re.match(text) is not None

-----
Attack String
# Static Generator: "<></" * n + ""
# Dynamic Generator: "<xml dha <lafaht..."
# PufferDoS: "<"*n+ "*"n+>"*n+</"*n+""

Func Summary for check_doctype
# Traditional Method: ""
'DO' in text and not 'DO' in ret
'DOCT' in text and not 'DOCT' in ret
...
'TYPE' in text and not 'TYPE' in ret
""
# PufferDoS: ""
pre: {attack_str} == text
post: {attack_str} == __ret__
""

```

Figure 1: A motivating example.

Figure 1 (a) and (b) shows a representative ReDoS vulnerability that we successfully exploited but was missed by existing approaches. It originates from a syntax-highlighting library with monthly downloads exceeding 300 million. To preserve anonymity, we obfuscated the function name and masked certain code fragments without altering the logical structure of the vulnerability. The vulnerable regex `tag_re` is used at Line 10. Existing tools leverages static analysis-guided attack string generation will decompose `tag_re` into $\varphi_1 = \emptyset$, $\varphi_2 = <(.*?)\s.*?>.*?</.*?>$ and $\varphi_3 = >$. As shown in Figure 1 (c), an attack string $s_a = x y^n z$ is constructed with $x = \epsilon$, repetition unit $y = <><\backslash$ satisfying $y \in \mathcal{L}(\varphi_2)$ and $z = \epsilon$ such that $z \notin \mathcal{L}(\varphi_3)$. Hence $s_a \notin \mathcal{L}(\text{tag_re})$ and excessive backtracking is triggered. Dynamic exploration-based generators iteratively mutate and validate candidate strings against `tag_re` until a substantial slowdown in matching is observed. However, our manual testing revealed that these generators face practical limitations. While the static analysis-guided generated attack string s_a with $|s_a| = 500,000$ induces approximately a 10 seconds hang when the regex is evaluated in isolation, the program truncates inputs to 1,000 characters at Line 10, reducing the slowdown to only 0.1s in practice. Dynamic

exploration-based generators encounter similar obstacles: truncation prevents exploits from manifesting, and earlier filters at Lines 3 and Line 9 may block randomly generated inputs before they reach Line 10. According to prior study [12], [18] and developer feedback, attacks causing slowdowns below 10 seconds, involving excessively long inputs, or failing to reach the vulnerable regex are deemed ineligible and therefore ignored.

Traditional concolic execution methods also fail to handle this example as it involves complex regexes. Although SMT solvers like Z3 provide regex support, they perform poorly when reasoning about multiple complex regex constraints simultaneously [65], which exacerbates the path explosion problem in DART. Compositional methods are likewise inadequate, as summarizing functions that embed regexes requires symbolically modeling all possible regex behaviors, causing substantial overhead when reusing function summaries. As shown in Figure 1 (c), for function `check_doctype` in Figure 1 (b), attempting to capture all regex behaviors results in large and intractable summaries.

These limitations motivate us to design a novel attack string generation approach that improves both efficiency and effectiveness. For efficiency, we seek s_a that incur higher matching time under a fixed length budget. Consider the motivating example, given `tag_re` with many adjacent LEs, a stop point p_i is selected on s_a for LE_i ; the subsequent LE_{i+1} advances from p_i to locate its own stop point p_{i+1} . When matching fails, the engine backtracks by decreasing p_{i+1} through the stop points in interval $[p_i, p_{i+1}]$. Therefore, to improve efficiency, we seek an s_a that enlarges the initial gaps between p_i and p_{i+1} , forcing each backtrack to reprocess longer input segments, thereby increasing overall matching cost. For effectiveness, once a candidate s_a is obtained, concolic execution is used to refine and validate s_a so that it reaches the r in realistic program contexts. To scale concolic execution, ReDoS-specific function summaries are synthesized to retain only summary entries that affect propagation of s_a , discarding unrelated behavior.

As shown in Figure 1 (c), a stronger s_a can be constructed by repeating each character in repetition unit $y = \langle \rangle \langle \rangle \backslash$ in isolation then concatenating the results to enlarge gaps between stop points, and finally adding a suitable prefix and a disruptive suffix. The function summary for `check_doctype` is refined to the contract: if the parameter is the attack string, then the return value must also equal to it. The abstraction eliminates in-function regex reasoning but preserves propagation semantics. We then use concolic execution on the function summary to refine s_a so it bypasses the filters at Lines 3 and 9.

During testing, the refined s_a produced a system hang exceeding 4,000 seconds when supplied as input with $|s_a| = 1,000$, demonstrating over 40,000 \times efficiency gain compared to traditional methods and confirming the vulnerability’s real-world exploitability.

3. Formal Basis for Attack String Generation

To formalize the insights from the motivating example, we present a formal analysis that yields a heuristic model of matching cost to guide our subsequent tool design. Note that, **complete proofs** of all lemmas and the theorem are presented in the Appendix A.1.

Figure 2 presents a vulnerable regex $r = a+.*(cd|e)+f$ and five corresponding representative attack strings $s_{a0}, s_{a1}, s_{a2}, s_{a3}, s_{a4}$, which are used throughout this section to illustrate each step of our verification. Among these attack strings, s_{a0} repeats its repetition unit only once while other strings repeat them multiple times. For the repetition units, LE_0 consistently selects the repetition unit $y_0 = a$. LE_1 coincides with LE_0 except in s_{a2} where it instead selects $y_1 = b$. LE_2 chooses $y_2 = cd$ in s_{a1} and s_{a2} , while choosing $y_2 = e$ in the remaining strings. Regarding structural layout, $s_{a0}, s_{a1}, s_{a2}, s_{a3}$ arranges repetition units in an interleaved manner, whereas s_{a4} concatenates them separately.

Heuristic 3.1. [Monotonic Matching Cost in Reachable Stop Points] Fix a regex r , a backtracking matcher, the set of participating LEs inside φ_2 , and the failing suffix location (i.e., where φ_3 first rejects). Let $C_r(s_a, y, n, m)$ denote the matching cost for input s_a , where $n = |s_a|$ and m is the total number of reachable stop points contributed by the participating LEs on s_a to the left of the failing location. Here y is any repetition unit used only to describe the layout of reachable stop points inside φ_2 . If two inputs s_a and s'_a keep the participating LEs and the failing location unchanged and $m' \geq m$, then

$$C_r(s'_a, y', n', m') \geq C_r(s_a, y, n, m),$$

with a strict inequality whenever $m' > m$.

Note that we assume these LEs are unbounded (e.g. with quantifier $*, ?, +$) or possess a high repetition capacity (e.g., with quantifier $\{m, n\}$ where $n - m$ is sufficiently large), such that they are capable of consuming arbitrarily long substrings within φ_2 .

To demonstrate this heuristic, we show how the number of stop points affects the matching cost. In Figure 2, for LE_0 , there are only **one** stop point in s_{a0} and **two** in s_{a1} . The regex matcher first attempts to match using the last stop point of LE_0 ; when this attempt fails, the matcher backtracks once to try the previous stop point in s_{a1} , whereas in s_{a0} the matching terminates immediately. Thus, s_{a1} incurs a higher matching cost than s_{a0} .

Lemma 3.2 (Shorter Repetition Units Increase Matching Cost). Fix a regex r and its infix φ_2 that induces backtracking. Let $x \in \mathcal{L}(\varphi_1)$ and $z \notin \mathcal{L}(\varphi_3)$ be fixed so that the resulting attack strings fail at the same suffix location. If $y_1, y_2 \in \mathcal{L}(\varphi_2)$ are feasible repetition units for all participating LEs and $|y_1| < |y_2|$, then for any fixed length budget n , constructing the infix with y_1 yields no lower matching cost than with y_2 , and the cost is strictly higher whenever the number of reachable stop points increases.

In Figure 2, s_{a2} uses $y = acd$ to construct the infix, whereas s_{a3} uses $y = ae$. As a result, for LE_0 , there are **three** stop points in s_{a2} and **four** in s_{a3} . By Heuristic 3.1 (matching cost increases with the number of reachable stop points), s_{a3} incurs a higher matching cost under a fixed attack string length.

Lemma 3.3 (Repetition Units from Intersection Increase Matching Cost). *Let r contain adjacent LE_i and LE_{i+1} with $\Sigma(LE_i) \cap \Sigma(LE_{i+1}) \neq \emptyset$. Fix $x \in \mathcal{L}(\varphi_1)$ and $z \notin \mathcal{L}(\varphi_3)$. If $y_1, y_2 \in \mathcal{L}(\varphi_2)$ are feasible repetition units such that every symbol of y_1 lies in $\Sigma(LE_i) \cap \Sigma(LE_{i+1})$ (both loops can consume y_1), while y_2 is consumable by exactly one of the two loops, i.e.,*

$$y_2 \in \Sigma(LE_i) \setminus \Sigma(LE_{i+1}) \quad \text{or} \quad y_2 \in \Sigma(LE_{i+1}) \setminus \Sigma(LE_i),$$

then for any fixed length budget n , constructing the infix with y_1 yields no lower matching cost than with y_2 , and the cost is strictly higher whenever the number of reachable stop points increases.

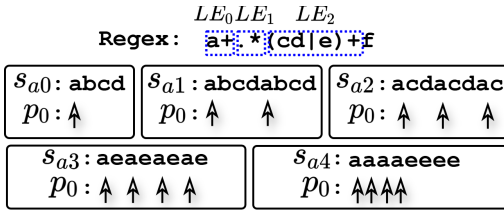


Figure 2: Illustrative example of attack string generation.

To exemplify Lemma 3.3, we consider s_{a1} and s_{a2} in Figure 2. If LE_0 and LE_1 select different repetition units, a and b respectively under intersecting alphabets condition as in s_{a1} , rather than sharing the same unit a as in s_{a2} , the stop points of LE_0 reduce from **three** in s_{a2} to **two** in s_{a1} . By Heuristic 3.1, s_{a2} incurs a higher matching cost.

Lemma 3.4 (Separated Stop Points Increase Matching Cost). *Let r contain adjacent LE_i and LE_{i+1} with $\Sigma(LE_i) \cap \Sigma(LE_{i+1}) \neq \emptyset$, and they can both consume the same substring of s_a as in Definition 3. Fix $x \in \mathcal{L}(\varphi_1)$ and $z \notin \mathcal{L}(\varphi_3)$. Denote by \mathcal{P}_i and \mathcal{P}_{i+1} the sets of stop points of LE_i and LE_{i+1} within the infix, respectively. When constructing infix, if the stop points in \mathcal{P}_i and \mathcal{P}_{i+1} are separated (i.e., all stop points of LE_i occur before those of LE_{i+1}) rather than interleaved (i.e., the stop points of the two LEs interleave along the string), then for any fixed length budget n , it yields no lower matching cost.*

For Lemma 3.4, we refer to s_{a3} and s_{a4} in Figure 2. When the regex matcher evaluates LE_0 using the last stop point in each string ($s_{a3}[6]$, $s_{a4}[3]$), a failed match causes the matcher to backtrack **once** at $s_{a3}[7]$ but **four** times at $s_{a4}[4 : 7]$ to select the stop points of LE_1 . The same trend holds when LE_0 chooses other stop points, resulting in an increased matching cost in s_{a4} .

Theorem 3.5 (Principles for Cost-Increasing Attack String Generation). *An attack string attains higher matching cost when each LE employs the shortest feasible repetition unit that is shared by as many LEs as possible. If two adjacent LEs have intersecting alphabets, a repetition unit should be selected from their alphabet intersection. Each selected repetition unit should be repeated independently and then concatenated to ensure the stop points are separated.*

4. PUFFERDOS

In this section, we present PUFFERDOS, a hybrid framework that operationalizes the formal foundations of attack generation into an efficient, structure-driven attack string generation pipeline (Section 4.2) and augments it with ReDoS-specific, compositional concolic refinement to ensure practical effectiveness (Section 4.3). Figure 3 presents an overview of PUFFERDOS, which consists of three core components.

(1) Vulnerable Regex Detection: Given a target program, we extract all regexes from the source code by locating regex-related API calls. We then apply ReDoS detectors to identify vulnerable regexes.

(2) Structure-Driven Attack String Generation: Building upon the formal foundations in Section 3, we devise a taxonomy of three regex vulnerable patterns and instantiate pattern-specific, structure-driven rules for attack string generation. For each vulnerable regex r , we parse r into subregexes, identify the looping expressions (LEs), and compute their alphabets $\Sigma(LE)$. We then classify r into the appropriate pattern and synthesize the infix under the corresponding rule. Finally, we derive the prefix and suffix from the non-loop segments (adjusting the suffix to ensure the string remains outside $\mathcal{L}(r)$) and concatenate the prefix, infix, and suffix to obtain a complete attack string.

(3) ReDoS-Specific Concolic Refinement: To ensure the generated attack string can propagate through real program and reach the vulnerable regex invocation, we first identify the target program entry points and construct call graphs to extract feasible call paths from entry point to the target regex. We then perform concolic unit testing to generate ReDoS-specific function summaries, followed by a system-level concolic execution to resolve interprocedural constraints. The attack string is refined accordingly, ensuring exploitability in realistic execution environments.

4.1. Vulnerable Regex Detection

The vulnerable regex detection component takes a target program as input, statically analyzes the source code to locate calls to regex-related APIs (e.g., matching, compilation, substitution) through function name matching, and extracts the regexes provided as string-literal arguments of the APIs.

Many regexes are not passed directly to APIs but built dynamically through variables assignment, concatenation, or interpolation. To handle these, PUFFERDOS scans source files, traverses abstract syntax trees, and tracks string provenance [15], recursively resolving variable definitions to reconstruct the concrete regex when a regex API is invoked.

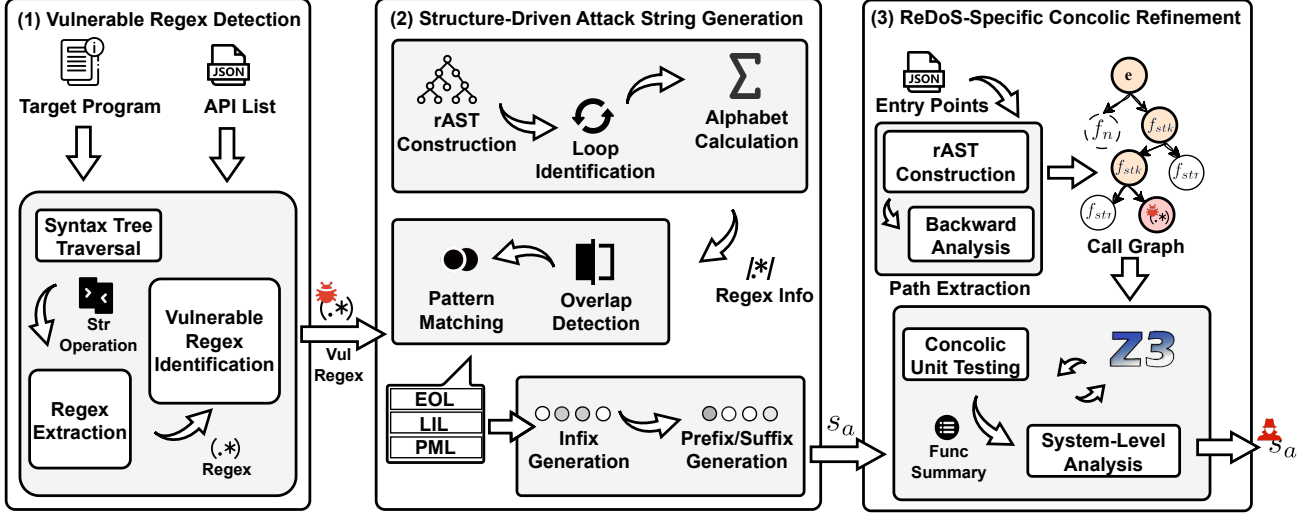


Figure 3: PUFFERDOS workflow.

For each extracted regex, PUFFERDOS applies ReDoS detectors to identify vulnerable cases and retains only those confirmed as vulnerable for subsequent analysis.

4.2. Structure-Driven Attack String Generation

Given the vulnerable regexes identified in Section 4.1, this section describes how PUFFERDOS generates a high-cost attack string for each regex. To systematically approach attack generation, we first establish a pattern taxonomy that categorizes vulnerable regex constructs by their exploitation characteristics (Section 4.2.1). We then detail how this taxonomy guides our attack string generation procedure (Section 4.2.2).

4.2.1. ReDoS Pattern Taxonomy. Building on the matching cost analysis in Section 3, we propose a taxonomy of three vulnerable patterns that (i) cover all patterns identified in prior works (see Appendix A.2) and (ii) prescribes pattern-specific attack string generation strategies. These strategies are tailored to markedly increase matching cost in backtracking regex engines.

Pattern 1 (Exponential One-Loop (EOL)). An EOL consists of a single pathological LE whose unfolding $\mathcal{U}(LE)$ contains at least two distinct subregexes r'_i, r'_j ($i \neq j$) such that $\Sigma(r'_i) \cap \Sigma(r'_j) \neq \emptyset$. For example, regex $\hat{(ab|a|b)}+ \$$ exhibits EOL pattern.

To generate attack string for r with EOLs, PUFFERDOS computes the overlap sets $\Sigma(r'_i) \cap \Sigma(r'_j) \neq \emptyset$ for each EOL. By Lemma 3.2, if only one EOL exists, it selects and repeats one of the shortest elements from its overlap set to form the infix. For multiple EOLs, by Theorem 3.5, PUFFERDOS first computes the intersection of their overlap sets. If the intersection is non-empty, the shortest element from the intersection is chosen as repetition unit y and repeated to construct infix; otherwise, PUFFERDOS repeats the shortest

element from each individual overlap set and concatenate the results to form an infix with separated stop points.

Pattern 2 (Loop-Intersect-Loop (LIL)). An LIL consists of two adjacent LEs, separated by a non-loop segment S_e , written as $r = LE_1 S_e LE_2$. It holds that $\Sigma(LE_1) \cap \Sigma(LE_2) \neq \emptyset$ and $\Sigma(S_e) \subseteq \Sigma(LE_1) \cap \Sigma(LE_2)$. Separator S_e may be empty. For example, regex $*a*$ exhibits LIL pattern.

To generate attack string, for each LIL pattern, PUFFERDOS sets repetition unit $y = S_e$ as it can be matched by both LEs and enables them to consume overlapping substrings without being obstructed by the non-loop segment. If the segment is empty, one of the shortest element from $\Sigma(LE_1) \cap \Sigma(LE_2)$ is selected instead. For multiple LILs, e.g., $r = LE_1 S_{e_1} LE_2 S_{e_2} LE_3$, by Theorem 3.5, the repetition unit y_i of each LIL pattern is repeated to form a base string $b_i = y_i^q$, and all resulting b_i are concatenated to construct the infix.

Pattern 3 (Polynomial Multi-Loop (PML)). A PML consists of k LEs, such that $\forall i \in [1, k-1], \Sigma(LE_i) \cap \Sigma(LE_{i+1}) = \emptyset$ or $\Sigma(S_e) \subsetneq \Sigma(LE_i) \cap \Sigma(LE_{i+1})$. That is, adjacent LEs have disjoint alphabets or the non-loop segment cannot be accepted by both of the LEs. For example, regex $\backslash s^* a \backslash s^*$ exhibits the PML pattern.

Here, S_e denotes the non-loop segment between each LE pair (LE_i, LE_{i+1}) . For regex with PMLs, PUFFERDOS falls back to the traditional attack string generation. We consider the target regex as $r = LE_1, \dots, LE_k$, by Lemma 3.2, PUFFERDOS selects the shortest symbol $a_i \in \Sigma(LE_i)$ for each LE, forms the base string $b = a_1 a_2 \dots a_k$ as repetition unit, and repeats the base string to build the infix.

4.2.2. Attack String Generation. Algorithm 1 shows the whole pipeline of generating attack string. It takes the identified vulnerable regexes as input. For each regex, it expands

the regex structure (Lines 4–10), matches the vulnerable patterns, and generates the corresponding s_a (Lines 11–25).

Since a regex may contain multiple alternatives introduced by the disjunction operator “|”, leading to distinct matching behaviors, we define a *regex path* (rPath) as an ordered, non-branching sequence of subregexes r' representing one specific alternative of r . Given r , PUFFERDOS expands it into a *regex Abstract Syntax Tree* (rAST), where r is the root and internal nodes denote intermediate unfoldings. Branching in the rAST arises from (i) “|” operators, with each disjunctive alternative as an independent child node, and (ii) LEs allowing zero repetitions (e.g., $.^*$ or $a\{0, 1024\}$), each producing two children, one excluding the LE to represent zero repetitions and the other including it to represent one or more repetitions. Although splitting original regex r into rPaths may omit certain branches of the original regex, any attack string synthesized for a given rPath remains a valid attack string for r , as the language of each rPath $\mathcal{L}(rPath)$ is a subset of $\mathcal{L}(r)$.

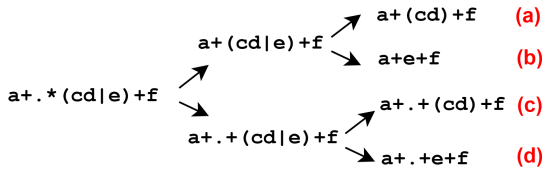


Figure 4: An illustrative example of rAST and rPath.

During expansion, PUFFERDOS checks whether a LE can be unfolded into subregexes with overlapping alphabets that meet the EOL condition defined in Pattern 1. If found, it marks and reports the EOL pattern and terminates; otherwise, decomposition continues until all leaf nodes are produced, each corresponding to one rPath. As an example, in Figure 4, the regex is expanded into an rAST with four leaf nodes, each leaf node corresponding to one rPath.

For each rPath, PUFFERDOS sequentially traverses all subregexes in it. Whenever two adjacent LE are encountered, PUFFERDOS calculates $\Sigma(\text{LE})$ and the overlap set $\Sigma(\text{LE}_i) \cap \Sigma(\text{LE}_{i+1})$. If the overlap set is non-empty, PUFFERDOS examines the separating segment between the two LEs to confirm all characters within it belong to the overlap set. A LIL pattern is identified whenever this criterion holds.

When multiple rPaths contain EOL or LIL patterns, PUFFERDOS selects the rPath with the largest number of such patterns and the shortest repetition unit; if several satisfy these criteria equally, one is chosen arbitrarily. If no rPath contains an EOL or LIL pattern, the regex is classified as PML. For example, in Figure 4, PUFFERDOS identifies that rPaths (a)–(d) all contain LIL patterns: (a) and (b) each contain one, while (c) and (d) contain two and are therefore shortlisted for comparison.

Once a vulnerable pattern is identified PUFFERDOS initiates s_a construction by generating the infix, applying pattern specific strategies for each case as described in Section 4.2.1. After generating the infix, PUFFERDOS builds the prefix by collecting all characters occurring before the first

Algorithm 1: ATTACK STRING GENERATION

```

Input:  $R$ 
Output:  $S$ 
1  $S \leftarrow \emptyset$ 
2 foreach  $r \in R$  do
3    $EOL\_flag \leftarrow \text{false}; LIL\_flag \leftarrow \text{false};$ 
4    $det\_p \leftarrow \emptyset; rAST \leftarrow r$ 
5   while  $\exists n \in rAST \wedge n.expandable = \text{true}$  do
6     if “|”  $\in n$  then
7        $rAST \leftarrow \text{GETSPLITCHILD}(n)$ 
8     else if  $n$  is LE allowing zero reps then
9        $rAST \leftarrow \text{GETLECHILD}(n)$ 
10    if  $n$  matches Pattern 1 then
11       $EOL\_flag \leftarrow \text{true}; det\_p \leftarrow n; \text{break}$ 
12  if  $EOL\_flag$  then
13     $w \leftarrow \text{INFIXEOL}(det\_p)$ 
14  else
15     $P \leftarrow \text{GETRPATH}(rAST)$ 
16    foreach  $p \in P$  do
17      if  $p$  matches Pattern 2 then
18         $Tag(p) \leftarrow LIL; LIL\_flag \leftarrow \text{true}$ 
19      if not  $LIL\_flag$  then
20         $Tag(p) \leftarrow PML$ 
21       $\hat{p} \leftarrow \text{SELECTPATH}(\{p \mid Tag(p) \neq \text{null}\})$ 
22      if  $Tag(\hat{p}) = LIL$  then
23         $w \leftarrow \text{INFIXLIL}(\hat{p})$ 
24      else
25         $w \leftarrow \text{INFIXPML}(\hat{p})$ 
26     $pre \leftarrow \text{PREFIX}(w, r); suf \leftarrow \text{SUFFIX}(w, r);$ 
27     $s_a \leftarrow pre \cdot w \cdot suf$ 
28     $S \leftarrow S \cup \{(r, s_a)\}$ 
29 return  $S$ 

```

LE. The suffix is formed from the characters after the last LE with the final character removed, ensuring $s_a \notin \mathcal{L}(r)$.

4.3. ReDoS-Specific Concolic Refinement

Once an attack string s_a is constructed, PUFFERDOS performs compositional concolic execution to systematically refine it toward inputs that trigger the vulnerable regexes.

As shown in Figure 3, PUFFERDOS takes as input the target program P , a set of vulnerable regex r , and their corresponding s_a . It operates in three phases: (1) extracting execution paths p from program entry points e to each r and record all reachable functions f . (2) performing concolic unit testing on selected functions from f along p to derive function summaries ϕ_f , and (3) composing system-level concolic execution by reusing ϕ_f to refine s_a

(1) ReDoS-Reachable Path Extraction. PUFFERDOS constructs a static call graph of P and uses developer provided entry points $E = \{e_1, \dots, e_m\}$. If no entry points supplied, PUFFERDOS heuristically selects externally invocable func-

tions (e.g. Python callables not prefixed with an underscore) that accept at least one string parameter as entry points.

For each r , PUFFERDOS traverses the call graph backwards from call sites of r to identify all $e \in E$ that can be reached, recording all interprocedurally reachable functions f as the aggregated path p for the pair (r, e) . Within each path p , functions f invoked before reaching r are classified into three categories: (i) f_{str} : functions that, before reaching r , accept a string parameter p_{str} and return a string or a constant value; (ii) f_n : functions that either do not take string parameters or do not return string or constant values before reaching r ; and (iii) f_{stk} : functions that remain on the call stack when r is reached (i.e., invoked but not returned).

(2) Function Summarization via Concolic Unit Testing. For each p , PUFFERDOS performs concolic unit testing to derive *ReDoS-specific function summary* ϕ_f that capture only the features relevant to ReDoS attack string propagation for selected functions.

The function selection follows two rules: (i) Function classified as f_{stk} are excluded, as prior work [33] indicates that these functions embody the main control-flow semantics, where abstraction could compromise precision. (ii) Functions accessing or modifying global variables are excluded, since their behaviors involve interprocedural interactions that unit testing cannot soundly capture.

For the selected functions, those classified as f_n are assigned an empty summary ($\phi_f = \emptyset$) to skip them in subsequent analysis. For each selected f_{str} , PUFFERDOS employs a heuristic approach that formulates four hypotheses: (i) s_a in p_{str} & $ret = CONST$, (ii) s_a in p_{str} & $s_a = ret$, (iii) $s_a = p_{str}$ & $s_a = ret$, (iv) s_a in p_{str} & s_a in ret , where ret represents the function return value and $CONST$ represents constant value like True or False. PUFFERDOS then performs concolic execution on each f_{str} to validate these hypotheses sequentially. If a counterexample is discovered for a hypothesis, PUFFERDOS proceeds to evaluate the next one; otherwise, the validated hypothesis is adopted as the function summary. If none of the hypotheses hold, PUFFERDOS follows traditional approaches, constructs ϕ_f by disjunct all symbolic path explored during unit testing.

(3) System-Level Concolic Execution. After deriving ϕ_f , for each path p , PUFFERDOS performs system-level concolic execution from the entry point e . It instruments P with an assertion that s_a does not reach the invocation of r . During execution, the SMT solver attempts to generate an input that falsifies this assertion while respecting the function summaries ϕ_f . If a counterexample is produced, PUFFERDOS treats it as an input that drives s_a to r under realistic execution and reports it as a refined, exploitable attack string. If no counterexample is found, the path is considered unreachable, and PUFFERDOS proceeds to the next path until either an exploitable attack string is found or all paths are exhausted.

Complexity and Correctness Discussion. The core principle of PUFFERDOS’s ReDoS-specific compositional concolic refinement is to retain execution features relevant to attack string propagation while pruning irrelevant behaviors, thereby reducing summary size, computational cost, and

path explosion. Following validation of the optimization proposed in prior work [23], we discuss the algorithm complexity and correctness of our approach.

Regarding algorithmic complexity, let P contain n functions, each with b internal paths. The DART algorithm [24] may explore up to $O(b^n)$ interprocedural paths in the worst case. The SMART [23] mitigates this via traditional function summaries; assuming there are at most b internal paths per function, whole-program analysis explores $O(n \cdot b \cdot 2^x)$ paths in the worst case, where x is the number of entries retained in each function summary. Our method reduces x by keeping only entries relevant to attack string propagation, thus lowering the overall complexity $O(n \cdot b \cdot 2^x)$.

Concerning correctness, PUFFERDOS does not modify the SMT solver, and every produced counterexample corresponds to a concrete execution; thus our refinement is *precision-preserving relative to DART* that introduce no false positives. While we cannot eliminate false negatives entirely due to the under-approximation nature of concolic execution [69], [73], [79], our approach preserves the features relevant to attack string propagation, ensuring all feasible propagation paths remain symbolically analyzable and *avoiding reductions in analyzable propagation coverage*. A concrete example illustrating the step-by-step concolic refinement process is available in Appendix A.6.

4.4. Implementation

We implement a prototype of PUFFERDOS for analyzing Python programs, motivated by prior work that identifies Python as the language most vulnerable to ReDoS [17]. PUFFERDOS is fully automated for attack string generation and concolic refinement and can be adapted to other languages by using a concolic executor for the target language, as all other components are language independent. For detecting vulnerable regexes we use regexploit [15] to extract regexes and RENGAR [74] to identify vulnerable regexes as RENGAR has been reported to achieve the best performance in prior work [74]. We then implement the rAST construction, vulnerable pattern identification and attack string generation pipeline on top of RENGAR [74]. PUFFERDOS can be integrated with any ReDoS detector. For constraint-guided refinement, PUFFERDOS uses pytype [25] to infer API argument type and identify entry points, constructs a call graph with PYCG [57] followed by a custom call flow analysis, and builds the ReDoS-specific compositional concolic execution module on CrossHair [59], a Z3 [80]-based concolic execution framework. We employ CrossHair for its active maintenance and robust Python support, while our algorithm remains compatible with any concolic execution backend. All the development, implementation and evaluation were conducted on Debian GNU/Linux 12, 32GB RAM, 1TB SSD.

5. Evaluation

We evaluate the performance of PUFFERDOS by answering the following research questions (RQs):

- **RQ1 (Efficiency):** How efficient is PUFFERDOS at producing attack strings?
- **RQ2 (Effectiveness):** How effective is PUFFERDOS at reproducing CVE-disclosed ReDoS vulnerabilities?
- **RQ3 (Overhead):** Can PUFFERDOS exploit previously undisclosed ReDoS in real-world projects?

During the experiment, we use regex matching time to measure both attack success and efficiency. To ensure stability, all reported results are averaged over three independent runs. Additionally, following common practice in program-analysis research [70], [75], attack string generator runtime overhead is evaluated by the execution time of each tool to reflect the computational cost of the analysis.

5.1. Experiment Setup

We create three datasets, a *baseline dataset*, a *CVE dataset* and a *real-world project dataset*, to separately assess the efficiency and real-world effectiveness of PUFFERDOS.

Baseline dataset. This dataset contains confirmed vulnerable regexes collected from real-world programs, which are used for assessing whether PUFFERDOS can produce attack strings. In particular, we first collected two widely used regex corpora, Corpus [16] and RENGAR dataset [1], containing 131,701 regexes collected from real-world programs. We then applied five ReDoS detectors RENGAR [74], ReScue [61], Regulator [46], Revealer [42], ReDoSHunter [38] with 100% reported precision on these two corpora to detect the vulnerable regexes. Their outputs are verified by measuring regex matching time, and eventually we confirmed **17,962 vulnerable regexes**.

CVE dataset. This dataset contains the ReDoS-related CVEs to evaluate whether the attack strings generated by PUFFERDOS can reproduce known ReDoS attacks. Specifically, we collected ReDoS instances from the National Vulnerability Database [47] (i.e., CVE-disclosed vulnerabilities). In total, to capture recent vulnerability trends and given that PUFFERDOS currently targets Python, we collected all 51 ReDoS-related CVEs reported in Python projects over the past five years. After excluding those without publicly available attack strings, **31 CVEs** remained for our final analysis, each providing verified attack strings that trigger the corresponding vulnerable regexes.

Real-world project dataset. This dataset contains previously undisclosed ReDoS vulnerabilities whose corresponding vulnerable regexes were automatically detected by the ReDoS detectors. These vulnerabilities had not been publicly reported and lacked known attack strings at the time of collection. They are used to evaluate PUFFERDOS’s capability to synthesize new exploits. To construct this dataset, we selected Python projects with at least 200 GitHub stars and 10k monthly downloads, and then randomly chose 100 projects from the top 1,000 by downloads. After manually reviewing their documentation, we excluded those that do not process any external user inputs (e.g., package management tools), leaving 74 candidates. We further examined their codebases and discarded projects without regex usage, resulting in 38 remaining projects. Finally, we applied the same five ReDoS detectors used in the baseline dataset and identified 12 projects containing vulnerable regexes.

These projects span database frameworks, LLM agents, text processing, and data analysis, with code sizes ranging from 7,860 to 443,427 lines. In total, **95 real-world vulnerable regexes** are included. Detailed statistics are provided in Appendix A.3.

5.2. Efficiency (RQ1)

To evaluate attack string generation performance, we compare PUFFERDOS with RENGAR [74], a state-of-the-art hybrid analysis tool. Since RENGAR has been shown to outperform existing approaches, we use it as our primary benchmark. We exclude EVILSTRGEN [66] from comparison as it targets non-backtracking engines and is designed to generate attack strings of at least length k [66], which is orthogonal to our goal of minimizing attack string length for backtracking engines. The experiment is carried out Python’s default regex engine `re` [55] for reflecting real-world deployment settings. For each vulnerable regex in the benchmark dataset, we ran PUFFERDOS and RENGAR independently to produce attack strings. For every (regex, tool) pair, we then determined the minimum attack string length required to push the regex match time beyond a specified threshold. Following the settings in prior usability studies [48], we set the thresholds to 0.1s, 1s, and 10s, which correspond to the three human-perceptual limits for instantaneous, uninterrupted, and sustained response times.

Specifically, given a dataset D containing N regexes, for regex $r_i \in D, i \in [0, N]$, let L_R and L_P be the minimal attack string lengths generated by RENGAR and PUFFERDOS to reach matching time cost threshold T . We use the following two metrics to evaluate the efficiency.

① We define R_T^i as the proportional increase in length required by RENGAR over PUFFERDOS to reach threshold T when attacking r_i , and \bar{R}_T denotes the mean difference across dataset D . Formally we have $R_T^i = \frac{L_R - L_P}{L_P}$ and $\bar{R}_T = \frac{1}{N} \sum_{i=0}^N R_T^i$. A positive \bar{R}_T indicates that RENGAR requires longer inputs to reach the threshold, whereas a negative value implies that PUFFERDOS does. ② We then compute the percentage P_T for which $L_R > L_P$, in order to compare the relative effectiveness of RENGAR and PUFFERDOS in reaching the threshold T across the entire dataset. To avoid trivial variations, cases with relative deviation $|L_R - L_P| / \min(L_R, L_P) < 5\%$ are regarded as equivalent and excluded from the count.

TABLE 1: Comparison between PUFFERDOS and RENGAR

Threshold	\bar{R}_T	P_T
$T = 0.1$	97.2	92.8%
$T = 1$	1128.5	92.7%
$T = 10$	3872.4	93.7%

The results are listed in Table 1. When setting the threshold to 0.1s, PUFFERDOS outperforms RENGAR in 92.8% of the cases, with RENGAR requiring attack strings that are on average 97.2× (times) longer to reach the same threshold. Such performance remains consistent as the threshold increases. At 1s, PUFFERDOS performs better in 92.7% of the cases,

TABLE 2: Results of pattern study

Pattern	$T = 0.1$		$T = 1$		$T = 10$	
	\bar{R}_T	P_T	\bar{R}_T	P_T	\bar{R}_T	P_T
EOL	0.06	100%	0.01	100%	0.01	100%
LIL	162.6	94.1%	1485.4	93.7%	4805.8	93.7%
PML	1.7	100%	1.3	100%	1.5	100%

with RENGAR needing strings that are 1128.5 \times longer; At 10s, the corresponding values are 93.7% and 3872.4, respectively. While considering the runtime overhead of string generation, RENGAR averagely takes 2.47 seconds to generate an attack string, whereas PUFFERDOS requires 5.11 seconds due to its rAST and rPath analyses for deeper structural exploration.

Scrutiny of Results. We firstly study the distribution of R_T^i across the baseline dataset. As shown in Figure 5, for the majority of regexes, RENGAR requires roughly 3x longer attack strings to reach a given threshold compared with PUFFERDOS. Nevertheless, PUFFERDOS can frequently produce exceptionally efficient attack strings, with R_T^i reaching between 10^4 and 10^7 as the threshold increases, showing that PUFFERDOS can expose weaknesses in regexes and trigger far more severe ReDoS behaviors.

To scrutinize the observations, we then randomly selected 300 regexes from the baseline dataset and manually examined their results (see Table 2). Among these, 5 (1.6%) regexes follow the EOL pattern, 133 (44.3%) the LIL pattern, and 162 (54.0%) the PML pattern. Across all thresholds, RENGAR outperforms PUFFERDOS in 9 cases, while PUFFERDOS surpasses RENGAR in 183 cases. The remaining 108 cases with results differences within 5% are omitted as comparable performance.

For regexes with EOL patterns, PUFFERDOS outperforms in two cases and achieves comparable results in three, with RENGAR requiring less than 6% longer attack strings to reach the same matching cost. For PML patterns, PUFFERDOS excels in 57 cases and comparably in 105, where RENGAR needs about twice the string length to reach the same threshold. These improvements stem from PUFFERDOS’s selection of the shortest feasible repetition unit, which increases the stop point number and matching cost. Although the gain for EOLs appears modest, it is substantial given the exponential sensitivity to input length. For LIL patterns, PUFFERDOS outperforms in 124 cases, achieving 162.6 \times , 1485.4 \times , and 4805.8 \times shorter strings than RENGAR at thresholds of $T = 0.1$, $T = 1$, and $T = 10$, respectively. Among all pattern classes, LIL contributes the most to the overall \bar{R}_T . PUFFERDOS’s advantage grows as the threshold increases as its effective separated stop point handling and shortest repetition unit selection induce non-linear backtracking growth with input length. PUFFERDOS underperforms in 9 cases: one due to a distinctive regex structure leads the engine to prematurely accept the attack string when stop points are ordered separately, and eight involving LEs that consume limited input (e.g., with quantifier $\{0,20\}$), violating Heuristic 3.1’s assumption that LEs

can consume sufficiently long substrings. However, defining a precise threshold for “sufficient consumption length” is inherently difficult, as it depends heavily on the structural complexity of each regex. Since such cases constitute only 2.7% of the total, we leave this issue for future investigation.

Evaluation across Different Regex Engines. We assess PUFFERDOS’s generality by comparing it with RENGAR across multiple regex engines in different languages, as these engines evolve independently with varying defenses and optimizations against ReDoS, and their differences in supported regex features and runtime behaviors may also affect attack performance. For instance, Java introduced a bounded caching mechanism in Java 9 to reduce backtracking, while JavaScript added a non-backtracking engine for specific patterns in 2020 [12], [13], [51]. Hence, we further include four widely used engines reported to be vulnerable [12]: Python’s regex [44], JavaScript’s RegExp [49] with the experimental ReDoS defense enabled, and Java’s java.util.regex [52] on Java 8 and Java 23 to observe the impact of engine evolution. All our experiments use the official releases as of October 2025.

The results are shown in Appendix A.4. PUFFERDOS consistently outperforms RENGAR across all regex engines, with RENGAR requiring attack strings 12.1–5051.8 \times longer to reach the same matching cost threshold on the baseline dataset. Among pattern categories, LIL regexes yield the largest improvement, making RENGAR requiring 14.9–5510.1 \times longer strings to achieve a threshold, whereas regexes with PML patterns requiring roughly 20%–180% longer strings. The improvement, however, varies across engines. PUFFERDOS achieves its greatest advantage on Java-8, whose traditional backtracking-based implementation remains highly vulnerable. In contrast, the advantage diminishes on Java-23 and Node.js-25, where the bounded caching mechanism and non-backtracking optimization partially prunes redundant backtracking paths, thereby reducing the matching costs of attack strings generated by both tools toward near-linear behavior and consequently lowering the observed \bar{R}_T and P_T . For Python’s regex engine, PUFFERDOS performs comparably to official re engine, as both engine share the same backtracking NFA core.

5.3. CVE Reproduction (RQ2)

To verify the effectiveness of the attack strings in reproducing ReDoS attacks, we utilize the CVE dataset and re-execute the attacks demonstrated in each CVE report. For comparison, we adopt RENGAR integrated with CrossHair, extended with a traditional compositional concolic execution module, as the primary baseline. We also compare the attack strings generated by PUFFERDOS with the CVE disclosed attack strings. Consistent with prior studies [12], [18], [46], [66], a ReDoS exploit is considered successful if it causes a system hang of $> 10s$ using an input smaller than 100KB.

As illustrated in Table 3. PUFFERDOS successfully reproduced 30 (96.8%) ReDoS attacks whereas RENGAR reproduced 24 (77.4%). We manually inspected the only CVE that neither tool reproduced and found that it requires

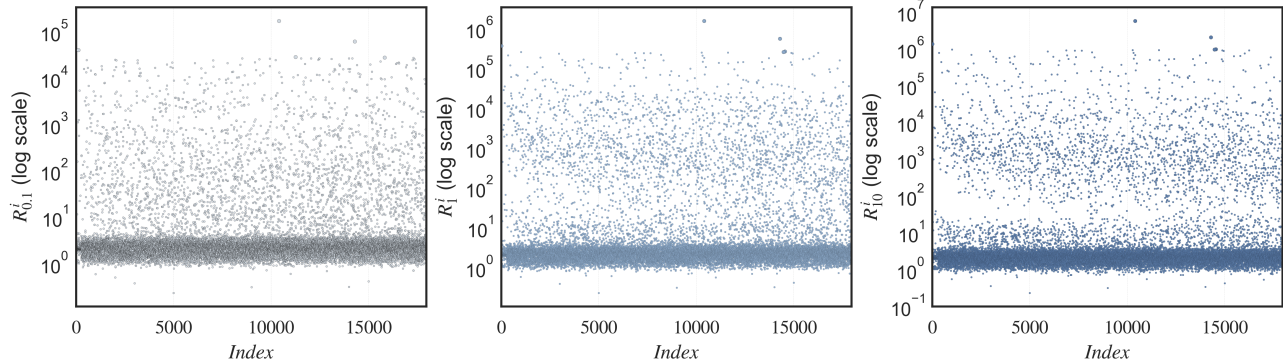


Figure 5: Distribution of \bar{R}_T on baseline. The extreme values reach 10^5 , 10^6 , and 10^7 at $T = 0.1$, 1 , and 10 , respectively.

complex user interactions and the construction of highly specific input structures. As the concolic executor does not fully model these interactions, the exploitation failed. Among the six cases where RENGAR failed but PUFFERDOS succeeded, four were caused by complex path conditions involving regex operations, leading to path explosion in traditional compositional concolic execution. PUFFERDOS uses ReDoS-specific function summaries to eliminate unnecessary behavior tracking, thereby reducing path exploration overhead. The remaining two failures resulted from the inability of RENGAR to generate sufficiently effective attack strings. PUFFERDOS instead identified that these two cases have an LIL pattern and generated attack strings that were thousands times more efficient and successfully reproduced the ReDoS attack. Notably, three CVE-disclosed attack strings failed to reproduce the attack due to excessive length, supporting prior observations that some reported attacks rely on unrealistic input sizes [12]. Two of these cases exhibit LIL patterns for which PUFFERDOS synthesized far more effective attack strings. The remaining case involves an EOL pattern: the disclosed string used an incorrect repetition unit and omitted a mandatory character, so the regex rejected the input without any backtracking.

For runtime overhead, on average, as shown in Table 3, with ReDoS-specific function summaries, PUFFERDOS reduces CVE reproduction time by 2.7 minutes per case.

TABLE 3: Comparison results on ReDoS CVEs.

	PUFFERDOS	RENGAR	Disclosed
Exploit Rate (%)	96.8	77.4	90.3
Avg. Time (min)	5.3	8.0	–

*Disclosed: attack strings documented in official CVE reports.

5.4. Real-world Effectiveness (RQ3)

We further apply PUFFERDOS to the real-world project dataset to assess its ability to detect previously undisclosed vulnerabilities. The results of exploiting undisclosed vulnerabilities are listed in Table 4. In detail, PUFFERDOS successfully triggered 59 out of 95 vulnerable regexes in 12 projects, while RENGAR + Crosshair triggered 34 of them. The ablation variant, retaining our ReDoS-specific concolic

TABLE 4: Comparison results on undisclosed ReDoS.

Target Project	PUFFERDOS		Ablation ²		RENGAR	
	ReDoS	Time ¹	ReDoS	Time	ReDoS	Time
nlk	3	23.0	2	26.8	1	28.3
python-markdown	7	4.0	4	4.3	1	5.2
pygments	15	14.6	14	19.1	14	21.4
sphinx	5	19.1	5	22.7	5	25.4
peewee	1	9.3	0	12.1	0	12.8
readme-ai	8	4.0	5	4.6	2	6.1
numpy	7	41.9	6	45.3	4	47.4
h5py	3	12.3	2	13.4	2	16.3
langchain	5	30.9	3	34.8	2	35.9
neural-compressor	1	5.1	1	6.2	0	9.3
pydantic	3	7.8	2	11.7	2	13.0
cff	1	3.2	1	5.6	1	6.3
Total	59	–	45	–	34	–
Avg.	–	14.6	–	16.1	–	18.9

1. Time in minutes. 2. Ablation: RENGAR string generation + PUFFERDOS concolic refinement.

refinement but replacing PUFFERDOS’s attack string generation with RENGAR, triggered 45 ReDoS vulnerabilities. Among them, 21 vulnerabilities in six projects have been confirmed by developers.

After manually inspecting the 36 vulnerable regexes that neither PUFFERDOS nor RENGAR could exploit, we found:

- 30 cases cannot be triggered at all, namely, these are the false positives caused by the ReDoS vulnerability detectors that we integrated. Among those 30 cases, 21 of them involve internal regexes that are unreachable from any program entry point and are not exposed to user input. For instance, numpy contains five vulnerable regexes that are used only to parse CPU version information. Nine other failures result from security filters that prevent the manifestation of ReDoS behavior. For example, python-markdown adopts the re2 [26] engine in specific modules, which enforces linear-time regex matching and eliminates backtracking-based ReDoS.
- The remaining six failed cases have either complex input structures or indirect dataflows that concolic execution cannot model. For example, in pydantic a vulnerable regex stored as an object attribute is repeatedly retrieved and reused across components, breaking concolic constraint propagation and causing the analysis to fail.

Of the 25 cases where only RENGAR failed, we found

11 cases involves complex cross-function regex operations, where summarizing function behavior led to excessively large summaries in traditional compositional concolic analyzers; these are resolved by PUFFERDOS and the ablation variant using ReDoS-specific summaries. The remaining 14 result from RENGAR’s failure to generate sufficiently efficient attack strings within length limits.

Such performance varies across projects. PUFFERDOS exhibits greater advantages as the complexity of input processing preceding regex invocation increases. For example, in pygments, a syntax-highlighting library that applies regexes directly to user input, RENGAR can easily trace regex usage and thus performs comparably to PUFFERDOS. In contrast, python-markdown preprocesses input by splitting text blocks and metadata before applying regex parsing, creating deep call chains that hinder RENGAR from identifying vulnerable paths, whereas PUFFERDOS effectively resolves them with ReDoS-specific function summaries. Manual analysis further shows that, as presented in Appendix A.3, an average of 32% of functions across all projects benefit from ReDoS-specific summaries, with variation driven by developers’ coding style like the usage of global variables.

Table 4 reports runtime overhead. PUFFERDOS requires on average 14.6 minutes per project versus 16.1 minutes for the ablation variant and 18.9 minutes for RENGAR with CrossHair, a reduction of 9.3% and 22.8% respectively. With ReDoS-specific function summaries and efficient attack string generation, PUFFERDOS propagates constraints across functions with lower overhead and exploits real-world ReDoS within practical input length limits. More detailed runtime breakdown can be found in Appendix A.5.

5.5. Threats to Validity

External Validity. Our design is grounded in formal analysis-based reasoning and is inherently language-agnostic. While we have evaluated it across multiple regex engines, the experiments primarily focus on the Python ecosystem. In addition, practical regex engines may support extended features such as lookarounds and named groups [22]. Named groups and non-capturing groups do not affect PUFFERDOS as they preserve the order and overlap structure of loop expressions; lookarounds may affect PUFFERDOS in specific cases by altering the valid alphabet of adjacent loop expressions via zero-width assertions. Nevertheless, empirical evaluation results show that PUFFERDOS remains effective across diverse real-world regexes, demonstrating strong robustness beyond its formal assumptions. Extending the formal analysis and evaluation to support advanced regex features, additional programming languages, and runtime environments is left for future work.

Internal Validity. In regex matching, full match semantics require the entire input string to satisfy the regex, while partial match semantics succeed when any substring matches the regex. The majority of previous works [38], [39], [54], [61], [74] adopt the full match setting, and PUFFERDOS

follows it for fair comparison and easier integration. We leave the support for partial match semantics as future work.

For the baseline tool, since no open-source compositional concolic execution framework exists for Python, we implemented the baseline using a conventional SMART algorithm [23], [34] following the standard methodology and employed the state-of-the-art concolic executor Crosshair to ensure fairness and implementation consistency. Besides, although taking execution time as the metric may introduce uncertainty, it is the prevailing practice in prior works [12], [18], [46], [66]; each measurement was therefore averaged over three independent runs to reduce variance. While counting regex matching steps could be more precise, no formally established criterion delineates the threshold of matching steps required to characterize a ReDoS manifestation, and most engines lack the required instrumentation, making execution time the most comparable and reproducible measure.

6. Related Work

ReDoS Attack String Generation. Traditional ReDoS attack-string generation is coupled with detection: once a regex is flagged as vulnerable, generators synthesize inputs that induce catastrophic backtracking. Prior work implements this via static, dynamic, and hybrid techniques.

Early tools such as regexploit [21] and redos-detector [36] generate attack strings using syntax-level rules for common patterns (e.g., infinite or nested loops), and Demoulin et al. [20] scale this idea with machine learning. While lightweight, these pattern-based approaches lack semantic reasoning about match execution and thus provide limited exploitability guarantees. Recent work innovatively demonstrates that compressed data direct computing can be integrated with homomorphic encryption to provide efficient and secure computation [41], [81]. Automata-based methods instead construct and analyse the automaton for a regex to detect worst-case behaviours: Rexploiter [77] synthesizes attacks via adversarial automata construction, Weideman et al. [76] speed up analysis with a prioritized NFA, and recent works [66], [68] leverage DFA techniques to produce attack strings for non-backtracking engines. However, their abstractions can still over- or under-approximate runtime behavior, and thus cannot guarantee practical accuracy.

Dynamic exploration approaches solve this problem by leveraging feedback-driven exploration of regex execution to synthesize inputs that incur high matching cost. SDL-Fuzzer [35] performs this exploration using randomly generated strings, but its unguided search leads to low efficiency and limited coverage. Regulator [46] and SlowFuzz improve upon this by adopting grey-box fuzzing, instrumenting the regex engine to guide input mutation through execution feedback. ReScue [61] employs a similar strategy, using an evolutionary genetic algorithm to evolve inputs toward higher cost. Nevertheless, these methods remain constrained by limited coverage and high time overhead.

Hybrid analysis tools combine static attack string generation with dynamic validation to balance accuracy and overhead. ReDoSHunter [38] integrates pattern-guided static

detection with dynamic validation, forming the first hybrid ReDoS analysis framework. NFAA [76] and Revealer [42] extend NFA-based static analyses using dynamic testing to confirm exploitability. RENGAR [74] further improves scalability by mitigating disturbances from subregexes, while Badger [50] integrates fuzzing with symbolic execution. Despite runtime overhead, hybrid analysis consistently delivers the best experimental performance. Yet, these methods still prioritize generating successful attack strings rather than optimizing attack efficiency or validating real-world exploitability, limiting their practical applicability.

Compositional Vulnerability Analysis. Function summaries have been widely adopted in vulnerability analysis. Early approaches aimed to summarize all behaviors of each function in the program. SMART [23] disjoined symbolic execution results over inputs and outputs, and Anand et al. [7] extended this with on-demand first-order formula refinement. FOCAL [34] applied counterexample-guided refinement in unit testing, while FunFrog [60] and HiFrog [6] built SSA-form summaries refined iteratively via counterexamples. While accurate, these methods often incur substantial summary construction and reuse overhead, which can lead to path explosion. Recent works address this through selective abstraction: generating summaries only for specific components [33], [58], abstracting array operations [8] or global variable statements [32], and extending bounded model checking to concurrent code via scheduling summaries [78]. Most closely aligned with our approach, NLP-eye [72] and Goshawk [43] restrict summaries to memory allocation and deallocation semantics to reduce size and accelerate analysis.

7. Conclusion

In this paper, we present PUFFERDOS, a hybrid framework that, guided by formal analysis, generates efficient attack strings that amplify matching cost. To guarantee real world exploitability, it incorporates a ReDoS-specific compositional concolic execution to refine and validates each attack string at program level. In evaluation, PUFFERDOS generates attack strings 97.2× to 3872.4× shorter while achieving equivalent matching cost with RENGAR. It reproduced 96.8% of the ReDoS CVE attacks and exploited 59 previously undisclosed ReDoS vulnerabilities in 12 projects, outperforming attack strings produced by state-of-the-art tools and those disclosed in CVE reports. We release PUFFERDOS at [2].

8. Ethics considerations

This work does not involve human subjects, and no personal or otherwise sensitive data were collected, created, or processed. This work raises no ethical concerns. All testing was conducted in a local offline environment with no interaction with real-world systems or user data. All previously undisclosed vulnerabilities were responsibly disclosed to developers through a coordinated process, including technical details

and proof-of-concept inputs, with sufficient time allowed for patching before public discussion.

References

- [1] “Rengar: Regular expression denial-of-service analyzer,” <https://sites.google.com/view/rengar/resources?authuser=0>.
- [2] “PUFFERDOS: Anonymous artifact repository,” <https://anonymous.4open.science/r/PufferDoS-7E0D/>.
- [3] “CVE-2017-16098: Apache Charset Regular Expression Denial of Service (ReDoS),” <https://cvefeed.io/vuln/detail/CVE-2017-16098>, 2017, affected package: charset \neq 1.0.0. Regular expression DoS via crafted input. CWE-400: Uncontrolled Resource Consumption. CVSS v3.0 Score: 7.5 (High). Published: 2018-06-07, Last Modified: 2024-11-21.
- [4] “CVE-2017-16137: Regular Expression Denial of Service in debug module (öformatter),” <https://nvd.nist.gov/vuln/detail/CVE-2017-16137>, 2017, affected product: debug (Node.js). Vulnerable versions: $\zeta=2.0.0$ $\jmath2.6.9$; $\zeta=3.0.0$ $\jmath3.1.0$; $\zeta=3.2.0$ $\jmath3.2.7$; $\zeta=4.0.0$ $\jmath4.3.1$. ReDoS issue via untrusted input into `o` formatter. Requires 50k characters to block for about 2 seconds. CWE-400. CVSS v3.0 score: 5.3 (Medium), CVSS v2.0 score: 5.0. Published: 2018-06-07.
- [5] “CVE-2023-39663: MathJax Regular Expression Denial of Service (ReDoS),” <https://www.suse.com/security/cve/CVE-2023-39663.html>, 2023, affects MathJax up to v2.7.9: two ReDoS vulnerabilities via `pattern` and `markdownPattern`. SUSE states “Does not affect SUSE products.” CVSS v3.1 Base Score: 7.5 (AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H). Overall severity: Important. Mitre/CVE page and SUSE timeline created: 2023-08-30, last modified: 2025-07-07.
- [6] L. Alt, S. Asadi, H. Chockler, K. Even Mendoza, G. Fedyukovich, A. E. Hyvärinen, and N. Sharygina, “Hifrog: Smt-based function summarization for software verification,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2017, pp. 207–213.
- [7] S. Anand, P. Godefroid, and N. Tillmann, “Demand-driven compositional symbolic execution,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 367–381.
- [8] A. Armando, M. Benerecetti, and J. Mantovani, “Counterexample-guided abstraction refinement for linear programs with arrays,” *Automated Software Engineering*, vol. 21, no. 2, pp. 225–285, 2014.
- [9] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli *et al.*, “cvc5: A versatile and industrial-strength smt solver,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2022, pp. 415–442.
- [10] E. Barlas, X. Du, and J. C. Davis, “Exploiting input sanitization for regex denial of service,” in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 883–895.
- [11] A. Bartoli, A. De Lorenzo, E. Medvet, and F. Tarlao, “Inference of regular expressions for text extraction from examples,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 5, pp. 1217–1230, 2016.
- [12] M. H. M. Bhuiyan, B. Çakar, E. H. Burmane, J. C. Davis, and C.-A. 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*, 2025, pp. 1659–1675.
- [13] M. Bidlingmaier, “An additional non-backtracking regexp engine v8,” <https://v8.dev/blog/non-backtracking-regexp>, 2021.
- [14] C. Cadar, D. Dunbar, D. R. Engler *et al.*, “Klee: unassisted and automatic generation of high-coverage tests for complex systems programs,” in *OSDI*, vol. 8, 2008, pp. 209–224.

- [15] B. Caller, L. Carettoni, and Doyensec, “regexexploit: Find regular expressions which are vulnerable to redos;” <https://github.com/doyensec/regexexploit>, 2021, release v1.0.0 (Mar 11, 2021). License: Apache-2.0.
- [16] C. Chapman and K. T. Stolee, “Exploring regular expression usage and context in python,” in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 282–293.
- [17] J. C. Davis, “Rethinking regex engines to address redos,” in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 1256–1258.
- [18] 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 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2018, pp. 246–256.
- [19] L. De Moura and N. Bjørner, “Z3: an efficient smt solver,” in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS’08/ETAPS’08. Berlin, Heidelberg: Springer-Verlag, 2008, p. 337–340.
- [20] H. M. Demoulin, I. Pedisich, N. Vasilakis, V. Liu, B. T. Loo, and L. T. X. Phan, “Detecting asymmetric application-layer {Denial-of-Service} attacks {In-Flight} with {FineLame};” in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 693–708.
- [21] doyensec, “regexexploit: Regular expression security tooling;” <https://github.com/doyensec/regexexploit>, 2025, GitHub repository.
- [22] J. Friedl, *Mastering regular expressions*. O’Reilly Media, Inc., 2006.
- [23] P. Godefroid, “Compositional dynamic test generation,” in *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2007, pp. 47–54.
- [24] P. Godefroid, N. Klarlund, and K. Sen, “Dart: directed automated random testing,” in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’05. New York, NY, USA: Association for Computing Machinery, 2005, p. 213–223. [Online]. Available: <https://doi.org/10.1145/1065010.1065036>
- [25] Google, “pytype: A static type analyzer for python,” <https://github.com/google/pytype>, 2025.
- [26] Google, “RE2: Linear-time regular expression engine,” <https://github.com/google/re2>, 2025.
- [27] J. Graham-Cumming, “Details of the cloudflare outage on july 2, 2019,” <https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/>.
- [28] M. M. Hasan, H. Li, E. Fallahzadeh, G. K. Rajbahadur, B. Adams, and A. E. Hassan, “Model context protocol (mcp) at first glance: Studying the security and maintainability of mcp servers,” *arXiv preprint arXiv:2506.13538*, 2025.
- [29] J. E. Hopcroft, R. Motwani, and J. D. Ullman, “Introduction to automata theory, languages, and computation,” Boston, MA, 2006, ISBN 978-0321455369.
- [30] X. Huang, L. Zhang, Y. Liu, P. Deng, Y. Cao, Y. Zhang, and M. Yang, “Towards automatic detection and exploitation of java web application vulnerabilities via concolic execution guided by cross-thread object manipulation,” in *34th USENIX Security Symposium (USENIX Security 25)*, 2025, pp. 8367–8384.
- [31] S. E. Inc., “Outage postmortem — july 20, 2016,” <https://stackstatus.tumblr.com/post/147710624694/outage-postmortem-july-20-2016>.
- [32] D. Kim and Y. Choi, “Model checking embedded control software using os-in-the-loop cegar,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 565–576.
- [33] Y. Kim and Y. Choi, “Pbe-based selective abstraction and refinement for efficient property falsification of embedded software,” *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 293–315, 2024.
- [34] Y. Kim, S. Hong, and M. Kim, “Target-driven compositional concolic testing with function summary refinement for effective bug detection,” in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 16–26.
- [35] J. Kiriage, A. Rathnayake, and H. Thielecke, “Static analysis for regular expression denial-of-service attacks,” in *International conference on network and system security*. Springer, 2013, pp. 135–148.
- [36] kkos, “Oniguruma: Regular expression library;” <https://github.com/kkos/oniguruma>, 2025, GitHub repository.
- [37] S. C. Kleene, *Representation of events in nerve nets and finite automata*. Princeton University Press Princeton, 1956, vol. 34.
- [38] Y. Li, Z. Chen, J. Cao, Z. Xu, Q. Peng, H. Chen, L. Chen, and S.-C. Cheung, “{ReDoSHunter}: A combined static and dynamic approach for regular expression {DoS} detection,” in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 3847–3864.
- [39] Y. Li, Y. Sun, Z. Xu, J. Cao, Y. Li, R. Li, H. Chen, S.-C. Cheung, Y. Liu, and Y. Xiao, “{RegexScalpel}: Regular expression denial of service ({{{ReDoS}}}) defense by {Localize-and-Fix};” in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 4183–4200.
- [40] C. Liu, A. Mera, E. Kirda, M. Xu, and L. Lu, “{CO3}: Concolic co-execution for firmware,” in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 5591–5608.
- [41] Y. Liu, F. Zhang, Y. Zhang, S. Ma, E. Bertino, and X. Du, “A unified framework for compressed and encrypted text direct processing,” in *ICDE*, 2026.
- [42] Y. Liu, M. Zhang, and W. Meng, “Revealer: Detecting and exploiting regular expression denial-of-service vulnerabilities,” in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1468–1484.
- [43] Y. Lyu, Y. Fang, Y. Zhang, Q. Sun, S. Ma, E. Bertino, K. Lu, and J. Li, “Goshawk: Hunting memory corruptions via structure-aware and object-centric memory operation synopsis,” in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 2096–2113.
- [44] Matthew Barnett, “Python regex module on pypi,” <https://pypi.org/project/regex/>.
- [45] D. McInerney, W. Woodruff, and A. Nygate, “Redos vulnerability reports: Relevant or noisy nuisance,” https://mlsecops.com/podcast/redos-vulnerability-reports-security-relevance-vs.-noisy- nuisance?utm_source=chatgpt.com.
- [46] R. McLaughlin, F. Pagani, N. Spahn, C. Kruegel, and G. Vigna, “Regulator: Dynamic analysis to detect {ReDoS};” in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 4219–4235.
- [47] MITRE, “Common vulnerabilities and exposures (cve),” <https://www.cve.org/>, 2025.
- [48] J. Nielsen, *Usability Engineering*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994.
- [49] Node.js Foundation, “Node.js official website,” <https://nodejs.org/en/about>.
- [50] Y. Noller, R. Kersten, and C. S. Păsăreanu, “Badger: complexity analysis with fuzzing and symbolic execution,” in *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, 2018, pp. 322–332.
- [51] OpenJDK Project, “Openjdk regular expression engine commit,” <https://github.com/openjdk/jdk/commit/b45ea89>.
- [52] Oracle Corporation, “java.util.regex package documentation,” <https://docs.oracle.com/javase/8/docs/api/java/util/regex/package-summary.html>.

- [53] F. Parolini and A. Miné, “Sound static analysis of regular expressions for vulnerabilities to denial of service attacks,” *Science of Computer Programming*, vol. 229, p. 102960, 2023.
- [54] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, “Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities,” in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 2155–2168.
- [55] Python Software Foundation, “re—regular expression operations — python 3 library reference,” <https://docs.python.org/3/library/re.html>.
- [56] A. Rathnayake and H. Thielecke, “Static analysis for regular expression exponential runtime via substructural logics,” *CoRR abs/1405.7058*, 2014.
- [57] V. Salis, T. Sotiropoulos, P. Louridas, D. Spinellis, and D. Mitropoulos, “Pygc: Practical call graph generation in python. in 2021 ieee/acm 43rd international conference on software engineering (icse),” *IEEE, IEEE, Madrid, Spain*, pp. 1646–1657, 2021.
- [58] R. Sato, H. Unno, and N. Kobayashi, “Towards a scalable software model checker for higher-order programs,” in *Proceedings of the ACM SIGPLAN 2013 workshop on Partial evaluation and program manipulation*, 2013, pp. 53–62.
- [59] P. Schanely, “Crosshair: A symbolic execution framework for python,” <https://github.com/pschanely/CrossHair>, 2025.
- [60] O. Sery, G. Fedyukovich, and N. Sharygina, “Interpolation-based function summaries in bounded model checking,” in *Haifa verification conference*. Springer, 2011, pp. 160–175.
- [61] Y. Shen, Y. Jiang, C. Xu, P. Yu, X. Ma, and J. Lu, “Rescue: crafting regular expression dos attacks,” in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 225–235.
- [62] M. L. Siddiq, J. Zhang, and J. C. D. S. Santos, “Understanding regular expression denial of service (redos): Insights from llm-generated regexes and developer forums,” in *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, 2024, pp. 190–201.
- [63] Snyk, “The state of open-source security,” <https://snyk.io/>, 2020.
- [64] H. Spencer, “A regular-expression matcher,” in *Software solutions in C*, 1994, pp. 35–71.
- [65] C. Stanford, M. Veanes, and N. Björner, “Symbolic boolean derivatives for efficiently solving extended regular expression constraints,” in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 620–635.
- [66] W. Su, H. Huang, R. Li, H. Chen, and T. Ge, “Towards an effective method of {ReDoS} detection for non-backtracking engines,” in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 271–288.
- [67] K. Thompson, “Programming techniques: Regular expression search algorithm,” *Communications of the ACM*, vol. 11, no. 6, pp. 419–422, 1968.
- [68] L. Turoňová, L. Holík, I. Homoliak, O. Lengál, M. Veanes, and T. Vojnar, “Counting in regexes considered harmful: Exposing {ReDoS} vulnerability of nonbacktracking matchers,” in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 4165–4182.
- [69] B. Vandenbogaerde, Q. Stiévenart, and C. De Roover, “Abstracting concolic execution for soft contract verification,” in *32nd Static Analysis Symposium*, 2025.
- [70] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury, “oo7: Low-overhead defense against spectre attacks via program analysis,” *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2504–2519, 2019.
- [71] H. Wang, S. Pu, G. Knezek, and J.-C. Liu, “A modular nfa architecture for regular expression matching,” in *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, 2010, pp. 209–218.
- [72] J. Wang, S. Ma, Y. Zhang, J. Li, Z. Ma, L. Mai, T. Chen, and D. Gu, “{NLP-EYE}: Detecting memory corruptions via {Semantic-Aware} memory operation function identification,” in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, 2019, pp. 309–321.
- [73] W. Wang, C. Tu, Z. Meng, W. Huang, and Y. Xiong, “Wacana: A concolic analyzer for detecting on-chain data vulnerabilities in wasm smart contracts,” *ACM Transactions on Software Engineering and Methodology*, 2024.
- [74] X. Wang, C. Zhang, Y. Li, Z. Xu, S. Huang, Y. Liu, Y. Yao, Y. Xiao, Y. Zou, Y. Liu *et al.*, “Effective redos detection by principled vulnerability modeling and exploit generation,” in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2427–2443.
- [75] G. Wei, S. Jia, R. Gao, H. Deng, S. Tan, O. Bračevac, and T. Rompf, “Compiling parallel symbolic execution with continuations,” in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1316–1328.
- [76] N. Weideman, B. Van Der Merwe, M. Berglund, and B. Watson, “Analyzing matching time behavior of backtracking regular expression matchers by using ambiguity of nfa,” in *International Conference on Implementation and Application of Automata*. Springer, 2016, pp. 322–334.
- [77] V. Wüstholtz, O. Olivo, M. J. Heule, and I. Dillig, “Static detection of dos vulnerabilities in programs that use regular expressions,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2017, pp. 3–20.
- [78] L. Yin, W. Dong, W. Liu, and J. Wang, “Scheduling constraint based abstraction refinement for weak memory models,” in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 645–655.
- [79] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, “{QSYM}: A practical concolic execution engine tailored for hybrid fuzzing,” in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 745–761.
- [80] Z3Prover, “Z3: The theorem prover,” <https://github.com/Z3Prover/z3>, 2025.
- [81] F. Zhang, J. Zhai, X. Shen, D. Wang, Z. Chen, O. Mutlu, W. Chen, and X. Du, “Tadoc: Text analytics directly on compression,” *The VLDB Journal*, vol. 30, no. 2, pp. 163–188, 2021.

Appendix A.

A.1. Formal Proofs

Proof of Lemma 3.2. By Definition 4, each boundary between consecutive copies of a feasible repetition unit y inside the infix is a reachable stop point for every participating $\mathbb{L}\mathbb{E}$ that will be considered by the backtracking matcher. For an infix of length at most n , the number of such boundaries

$$m(y) = \left\lfloor \frac{n}{|y|} \right\rfloor.$$

Therefore, if $|y_1| < |y_2|$ then $m(y_1) \geq m(y_2)$, with strict inequality whenever $\lfloor n/|y_1| \rfloor > \lfloor n/|y_2| \rfloor$. Let $s_a^{(1)} = x y_1^{q_1} z$ and $s_a^{(2)} = x y_2^{q_2} z$ be the corresponding attack strings with $|s_a^{(i)}| = n$. Under Heuristic 3.1 (matching cost is non-decreasing in the number of reachable stop points), it follows:

$$C_r(s_a^{(1)}, y_1, n, m(y_1)) \geq C_r(s_a^{(2)}, y_2, n, m(y_2)),$$

with strict inequality when $m(y_1) > m(y_2)$.

Proof of Lemma 3.3. Let $s_a^{(y)} = x y^q z$ with $|s_a^{(y)}| = n$. For a given y , let $m_i(y)$ and $m_{i+1}(y)$ denote the numbers of reachable stop points contributed by LE_i and LE_{i+1} on $s_a^{(y)}$ to the left of the failing location. Because every symbol of y_1 lies in $\Sigma(\text{LE}_i) \cap \Sigma(\text{LE}_{i+1})$, repeating y_1 creates stop points for both loops at the boundaries between consecutive copies, so $m_i(y_1) > 0$ and $m_{i+1}(y_1) > 0$ grow with q . In contrast, by construction y_2 is consumable by exactly one loop, hence on the repeated segment only one of $m_i(y_2), m_{i+1}(y_2)$ increases while the other remains zero.

Therefore the total number of reachable stop points with y_1 satisfies

$$m_i(y_1) + m_{i+1}(y_1) \geq m_i(y_2) + m_{i+1}(y_2),$$

with a strict inequality whenever the second loop contributes at least one stop point on the y_1 segment. By Heuristic 3.1 (matching cost is non-decreasing in the number of reachable stop points and strictly increases when that number increases), we obtain

$$\begin{aligned} & C_r(s_a^{(y_1)}, y_1, n, m_i(y_1) + m_{i+1}(y_1)) \\ & \geq C_r(s_a^{(y_2)}, y_2, n, m_i(y_2) + m_{i+1}(y_2)), \end{aligned}$$

with strict inequality when the total number of reachable stop points increases.

Proof of Lemma 3.4. Consider two attack strings $s_a^{(1)}$ and $s_a^{(2)}$ of equal length n that differ only in the arrangement of stop points of LE_i and LE_{i+1} . Let LE_i and LE_{i+1} have j and k stop points on both attack strings, respectively, denoted by

$$\mathcal{P}_i = \{p_{i,0}, \dots, p_{i,j-1}\}, \quad \mathcal{P}_{i+1} = \{p_{i+1,0}, \dots, p_{i+1,k-1}\}.$$

In $s_a^{(1)}$, the stop points are *interleaved*:

$$s_a^{(1)} : p_{i,0} < p_{i+1,0} < p_{i,1} < p_{i+1,1} < \dots < p_{i,j-1} < p_{i+1,k-1}$$

whereas in $s_a^{(2)}$, they are arranged in *separated* order:

$$s_a^{(2)} : p_{i,0} < \dots < p_{i,j-1} < p_{i+1,0} < \dots < p_{i+1,k-1}$$

Each time LE_i backtracks at $p_{i,l} \in \mathcal{P}_i$, LE_{i+1} backtracks at each its stop points from $p_{i,l}$ to the end of attack string. Let $M(s_a, \text{LE}_{i+1}, p_{i,l} \rightarrow n)$ denote the number of stop points of LE_{i+1} reachable from $p_{i,l}$ to the end of attack string, when stop points are interleaved as in $s_a^{(1)}$, some of LE_{i+1} 's stop points lie before $p_{i,l}$ and are unreachable during backtracking, so $M(s_a^{(1)}, \text{LE}_{i+1}, p_{i,l} \rightarrow n) \leq k$. In the separated case $s_a^{(2)}$, all k stop points remain reachable, i.e., $M(s_a^{(2)}, \text{LE}_{i+1}, p_{i,l} \rightarrow n) = k$. Hence the total number of reachable stop points satisfies

$$\begin{aligned} m^{(2)} &= \sum_{l=0}^{j-1} M(s_a^{(2)}, \text{LE}_{i+1}, p_{i,l} \rightarrow n) \\ &\geq m^{(1)} = \sum_{l=0}^{j-1} M(s_a^{(1)}, \text{LE}_{i+1}, p_{i,l} \rightarrow n) \end{aligned}$$

As LE_i backtracks over all its k own stop points in both cases, its cost is constant; thus the total matching cost

depends only on LE_{i+1} . By Heuristic 3.1 (matching cost increases with the number of reachable stop points), we obtain

$$C_r(s_a^{(2)}, y, n, m^{(2)}) \geq C_r(s_a^{(1)}, y, n, m^{(1)})$$

so that the separated arrangement incurs no lower matching cost than the interleaved one under the same length budget.

A.2. Regex Vulnerable Pattern Coverage

TABLE 5: Vulnerable patterns, examples, and coverage.

Pattern	Example	Coverage
EOL	$\wedge (ab a b) + \$$	EOLS [74], EOA, NQ, EOD [38]
LIL	$\cdot * a \cdot *$	PTLS, POLS [74], POA, SLQ [38]
PML	$\backslash s * a \backslash s * b \backslash s +$	PTLS [74], POA [38]

Table 5 summarizes how our designed patterns cover the representative vulnerable structures reported in prior work [38], [74]. Prior studies [74] have shown that these categories collectively subsume all vulnerable pattern types identified in existing literature.

A.3. Project Statistics

Table 6 summarizes statistics for the 12 real-world Python projects used in our evaluation, reporting each project's GitHub stars, LOC, number of defined functions (Func), early-returning functions ($\#f$), ReDoS-specific summarizable functions (ϕ_f), and vulnerable regexes (Vul. r). In total, the dataset comprises 95 vulnerable regexes, 68,021 defined functions, and 5,848 early-returning functions, of which 1,895 (32%) are ReDoS-specific.

A.4. Cross-engine Evaluation Results

Table 7 compares PUFFERDOS and RENGAR on the baseline dataset, where the first column lists the regex engines and the subsequent columns show results under different matching cost thresholds. Tables 8(a)–(c) present results on 300 selected regexes grouped by pattern type: (a) EOL, (b) LIL, and (c) PML.

TABLE 6: Project statistics

Project	Stars	LOC	Func	$\#f$	ϕ_f	Vul. r
nltk	14.4k	119,361	6147	678	189	8
python-markdown	2.8k	7,860	391	170	80	9
pygments	2.1k	122,102	1353	611	188	15
sphinx	7.5k	121,070	7142	1140	274	10
peewee	11.8k	40,444	3508	680	161	1
readme-ai	2.7k	10,332	585	282	32	8
numpy	30.7k	235,972	13,103	1089	446	21
h5py	2.2k	15,923	1328	485	175	3
langchain	119k	250,993	9632	474	209	14
neural-compressor	2.5k	443,427	13,530	316	43	1
pydantic	25.6k	91,150	5927	291	54	4
cffii	221	31,112	2373	632	44	1
Total	–	–	68,021	5,848	1,895	95

TABLE 7: Cross-Engine evaluation results

Engine	$T = 0.1$		$T = 1$		$T = 10$	
	\bar{R}_T	P_T	\bar{R}_T	P_T	\bar{R}_T	P_T
Python-3.12	98.8	88.0%	928.5	89.9%	2751.5	91.9%
Node.js-25	12.1	73.1%	69.3	82.9%	99.2	83.9%
Java-8	185.4	92.5%	2042.7	92.1%	5051.8	93.2%
Java-23	28.1	71.4%	162.9	73.1%	318.7	73.8%

TABLE 8: Pattern-Based cross-engine evaluation results

(a) EOL						
Engine	$T = 0.1$		$T = 1$		$T = 10$	
	\bar{R}_T	P_T	\bar{R}_T	P_T	\bar{R}_T	P_T
Python-3.12	0.01	100%	0.01	100%	0.01	100%
Node.js-25	0.00	0%	0.00	0%	0.00	0%
Java-8	0.05	100%	0.10	100%	0.10	100%
Java-23	0.00	0%	0.01	100%	0.01	100%

(b) LIL						
Engine	$T = 0.1$		$T = 1$		$T = 10$	
	\bar{R}_T	P_T	\bar{R}_T	P_T	\bar{R}_T	P_T
Python-3.12	101.3	91.2%	1021.2	91.1%	3081.8	91.7%
Node.js-25	14.9	74.4%	112.9	83.8%	175.5	88.1%
Java-8	245.2	93.6%	2359.8	93.0%	5510.1	93.2%
Java-23	38.7	74.5%	191.6	74.0%	472.3	80.8%

(c) PML						
Engine	$T = 0.1$		$T = 1$		$T = 10$	
	\bar{R}_T	P_T	\bar{R}_T	P_T	\bar{R}_T	P_T
Python-3.12	0.9	100%	1.0	100%	1.1	100%
Node.js-25	0.2	100%	0.2	100%	0.2	100%
Java-8	1.8	100%	1.4	100%	1.6	100%
Java-23	0.2	100%	0.3	100%	0.5	100%

A.5. Runtime Breakdown.

Table 9 shows the runtime breakdown. PUFFERDOS averages 1.2 min for generation and 13.4 min for concolic execution, saving 5 min over RENGAR despite slightly higher generation overhead.

TABLE 9: Runtime breakdown per project.

Target Project	PUFFERDOS			RENGAR		
	Total	Gen.	Con.	Total	Gen.	Con.
nltk	23.0	1.3	21.7	28.3	0.6	27.7
python-markdown	4.0	0.9	3.1	5.2	0.3	4.9
pygments	14.6	2.7	11.9	21.4	1.1	20.3
sphinx	19.1	0.6	18.5	25.4	0.3	25.1
peewee	9.3	0.4	8.9	12.8	0.2	12.6
readme-ai	4.0	1.8	2.2	6.1	0.7	5.4
numpy	41.9	2.3	39.6	47.4	0.9	46.5
h5py	12.3	0.5	11.8	16.3	0.2	16.1
langchain	30.9	1.6	29.3	35.9	0.7	35.2
neural-compressor	5.1	0.2	4.9	9.3	0.1	9.2
pydantic	7.8	1.1	6.7	13.0	0.4	12.6
cffi	3.2	0.4	2.8	6.3	0.2	6.1
Avg.	14.6	1.2	13.4	18.9	0.5	18.4

1. Time in minutes. 2. Con.: concolic execution.

A.6. Concolic Execution Example

We illustrate the concolic refinement of PUFFERDOS using the motivating example from Section 2.

Phase 1: Path Extraction. Given the entry point `is_xml(text)`, PUFFERDOS constructs a static call graph and traverses backward from `tag_re.search` (Line 10) to the entry. Along this path, `check_doctype`, `check_xml` and `hash` are classified as f_{str} .

Phase 2: Function Summarization. PUFFERDOS performs concolic unit testing on `check_doctype`, `check_xml`, and `hash`. Taking `check_doctype` as an example, the four hypotheses are evaluated against s_a in order:

- 1) $s_a \subseteq pstr \wedge ret = \text{const}$: *refuted*; the function does not always return a constant.
- 2) $s_a \subseteq pstr \wedge ret = s_a$: *refuted*; the return value may differ when a DOCTYPE token is present.
- 3) $s_a = pstr \wedge ret = s_a$: *validated*; when s_a contains no DOCTYPE prefix, the function returns its input unchanged.

Hypothesis (iii) is adopted as the ReDoS-specific function summary: $\varphi_f : pstr = s_a \Rightarrow ret = s_a$

Phase 3: System-Level Concolic Execution. PUFFERDOS composes the function summaries derived in Phase 2 and performs system-level concolic execution from `is_xml`. Each summarized function call is replaced by its corresponding φ_f : $\varphi_{\text{check_xml}}$ at Line 3, φ_{hash} at Line 5, and $\varphi_{\text{check_doctype}}$ at Line 9. PUFFERDOS then instruments the program with the negated assertion that s_a does not reach `tag_re.search` and invokes the SMT solver to find a counterexample. The solver reasons over the following residual constraints without symbolically executing any summarized function:

- `xml_decl_re.match(sa) = None`, to bypass the guard at Line 3.
- `sa ∉ _looks_like_xml_cache`, to bypass the cache at Line 7.
- `pstr = sa ⇒ ret = sa`, to propagate s_a through `check_doctype` at Line 9.

The solver produces a counterexample s'_a satisfying all three constraints, confirming that s'_a reaches `tag_re.search` and is reported as a refined, exploitable attack string.

Appendix B. Meta-Review

The following meta-review was prepared by the program committee for the 2026 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

B.1. Summary

The proposed tool PUFFERDOS generates length-bounded ReDoS attack strings for backtracking regex engines. Experiments show notable efficiency gains over RENGAR and report reproducing most known CVEs plus discovering new exploitable instances.

B.2. Scientific Contributions

- Creates a New Tool to Enable Future Science.
- Provides a Valuable Step Forward in an Established Field.

B.3. Reasons for Acceptance

- 1) Much shorter attack strings and large speedups compared to the baseline, improving exploit deliverability under input limits.
- 2) Useful integration of ReDoS-specific compositional concolic execution to validate exploitability in context (reachability/input constraints), reducing false positives.
- 3) Evidence of real-world impact via reproduction of known CVEs and disclosure of newly found cases.

B.4. Noteworthy Concerns

- 1) The “attack strings are too long” framing is not fully substantiated with direct evidence about attacker feasibility/developer behavior, lacking clearer, concrete threat-model justification and examples.
- 2) Analysis relies on a simplified regex model and a fixed pattern taxonomy, potentially missing ReDoS cases requiring advanced regex features.
- 3) Comparison is primarily against one baseline only (RENGAR); broader SOTA comparisons and clearer reporting of end-to-end concolic refinement cost are missing.