Effective ReDoS Detection by Principled Vulnerability Modeling and Exploit Generation

Xinyi Wang[†§*], Cen Zhang[♮*], Yeting Li[†§✉], Zhiwu Xu[‡✉], Shuailin Huang[†§], Yi Liu[♮], Yican Yao[†§], Yang Xiao[†§], Yanyan Zou[†§], Yang Liu[♮], Wei Huo[†§]

[†]{CAS-KLONAT, BKLONSPT}, Institute of Information Engineering, CAS [♮]Nanyang Technological University
[§]School of Cyber Security, University of Chinese Academy of Sciences [‡]Shenzhen University

2023 IEEE Symposium on Security and Privacy (SP) | 978-1-6654-9336-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/SP46215.2023.10179328

***Abstract*—Regular expression Denial-of-Service (ReDoS) is one kind of algorithmic complexity attack. For a vulnerable regex, attackers can craft certain strings to trigger the super-linear worst-case matching time, which causes denial-of-service to regex engines. Various ReDoS detection approaches have been proposed recently. Among them, hybrid approaches which absorb the advantages of both static and dynamic approaches have shown their performance superiority. However, two key challenges still hinder the effectiveness of the detection: 1) Existing modelings summarize localized vulnerability patterns based on partial features of the vulnerable regex; 2) Existing attack string generation strategies are ineffective since they neglected the fact that non-vulnerable parts of the regex may unexpectedly invalidate the attack string (we name this kind of invalidation as disturbance.)**

RENGAR is our hybrid ReDoS detector with new vulnerability modeling and disturbance free attack string generator. It has the following key features: 1) Benefited by summarizing patterns from full features of the vulnerable regex, its modeling is a more precise interpretation of the root cause of ReDoS vulnerability. The modeling is more descriptive and precise than the union of existing modelings while keeping conciseness; 2) For each vulnerable regex, its generator automatically checks all potential disturbances and composes generation constraints to avoid possible disturbances.

Compared with nine state-of-the-art tools, RENGAR detects not only all vulnerable regexes they found but also 3 – 197 times more vulnerable regexes. Besides, it saves 57.41% – 99.83% average detection time compared with tools containing a dynamic validation process. Using RENGAR, we have identified 69 zero-day vulnerabilities (21 CVEs) affecting popular projects which have more than dozens of millions weekly download count.

1. Introduction

As a key artifact for pattern matching and searching, regular expressions (*abbrev.* regexes) with various extended features (*e.g.*, lookarounds, named groups and conditionals, etc. [1]) have been widely used in different fields of computer science such as programming languages, string processing tools, database query languages, and natural language processing [2]–[6]. Despite their prevalence, poorly-designed regexes can yield super-linear (*i.e.*, polynomial and exponential) matching steps [7]–[9], draining valuable computational resources. Such regexes are vulnerable to algorithmic denial-of-service (DoS) [10]–[21] attacks, also known as ReDoS attacks [7]–[9]. It was recently reported that more than 10% of regexes used in software projects exhibit super-linear worst-case behavior (*i.e.*, polynomial and exponential worst-case time complexity with respect to the given string's length), and hundreds of popular websites are threatened by ReDoS attacks [2], [5], [22]. Therefore, detection of ReDoS vulnerabilities is essential.

Plenty of works were proposed to detect ReDoS vulnerabilities including static, dynamic, and hybrid approaches. Static approaches [23]–[27] detect vulnerable regexes by searching predefined vulnerability patterns based on the abstract syntax trees (ASTs) or automata structures of the regex. They are scalable but less accurate. Dynamic approaches [28]–[31] are based on search-based fuzz testing. For a given regex, they keep generating attack strings using search-based algorithms, *e.g.*, genetic algorithms, until a testing oracle, *e.g.*, matching time, is satisfied. Dynamic approaches detect vulnerabilities in high precision. Recently, hybrid approaches [32], [33] combining static analysis and dynamic validation are proposed. First, static analysis is applied to locate candidate ReDoS vulnerable regexes. Then, a series of attack strings are generated based on the features of the vulnerable regexes. Last, they dynamically validate whether the ReDoS have been triggered by the generated strings. These hybrid approaches have made considerable strides in ReDoS detection.

Despite the noticeable progress, existing works still face two challenges. **C1.** Existing modelings summarize vulnerability patterns based on the structural features of the vulnerable loop subregexes (the subregexes that have quantifiers, abbr as LS). However, their patterns are localized and not based on a global understanding of the features of the LS. Specifically, assuming a vulnerable LS is $^\wedge(r_0|r_1)+$ where r_0, r_1 are two subregexes. Existing modelings propose patterns by summarizing features of r_0 and r_1 without considering their combinations'. Summarizing patterns without

[*]Equal Contribution.
[✉]Corresponding Authors.

DOI 10.1109/SP46215.2023.00169

a global understanding of the features limits the generality and accuracy of the summarized patterns, leading to both false positives and false negatives in candidate vulnerability locations. **C2.** Existing attack string generators have not noticed the disturbances among the subregexes inside the target regex. Commonly, the potential vulnerable subregex is identified from the regex and the generators construct strings to attack that subregex. However, the attack will become ineffective if the prefix of an attack string is unexpectedly accepted by the regex. Existing methods have not considered the acceptance caused by the disturbance from non-vulnerable subregexes. Ignoring these disturbances can significantly decrease the effectiveness of the generated attack strings.

To address the above challenges, we propose RENGAR – a ReDoS vulnerability detector with LS-based vulnerability modeling and disturbance free attack string generation strategy. **A1.** The key observation behind the modeling is that the existence of the common match strings among all subregexes of an LS and their combinations is a more precise interpretation for the root cause of a ReDoS vulnerability. Based on that understanding, RENGAR proposes three patterns that cover not only vulnerabilities inside the description scopes of existing patterns but also new vulnerabilities out of their scopes. All kinds of exponential ReDoS vulnerabilities are modeled in one pattern and polynomial vulnerabilities are summarized as two patterns according to the structural differences of their root causes. Based on the modeling, RENGAR designs an identification algorithm to locate candidate vulnerabilities. **A2.** To solve the disturbance issue, we first systematically analyze the possible disturbances. Since the common representation of a regex, *i.e.*, prefix, pathological, and suffix subregexes, only contains the vulnerability related subregexes, we propose a fine-grained representation to additionally model the disturbance-relevant subregexes. Upon that representation, we enumerate all possible disturbances among subregexes and summarize five kinds of disturbances that can invalidate the attack string. And for each kind of disturbance, we propose specific constraints to eliminate it. Based on the above, RENGAR designs an attack string generation algorithm. It first calculates the constraints for all five kinds of disturbances of a given regex, then uses them to guide the generation.

In evaluation, we compared RENGAR with nine state-of-the-art ReDoS detectors in seven datasets. The datasets include four commonly used ones and three real-world datasets scraped from package managers of Python, Java, and CSharp on a large scale. The detectors consist of tools based on static, dynamic, and hybrid approaches. The results show clear performance advantages of RENGAR: ❶ it can detect 3 – 197 times more ReDoS vulnerable regexes than the nine detectors; ❷ it detects not only all vulnerable regexes (45,367) found by the nine detectors but also vast new vulnerable regexes (130,483); ❸ its average detection time (0.5828s) is comparable with detectors of static approaches (0.0007s – 1.5499s) and is clearly less than the detectors which have a dynamic validation process (1.3685s – 336.0527s). The detailed evaluation shows that the superiority of RENGAR comes from both its vulnerability modeling and attack string generation components. RENGAR has been applied to find real-world vulnerabilities in 315 popular projects. 69 zero-day vulnerabilities including 21 CVEs have been found. These vulnerabilities cover popular projects which have more than dozens of millions of weekly download counts. We responsibly reported these vulnerabilities to the vendors and helped them to fix.

In summary, we made the following contributions:

- We identified two key challenges for detecting ReDoS vulnerabilities. One is that existing modelings are localized and only focus on the structural features of the vulnerable LS. Another is that existing attack string generators do not handle the disturbances caused by subregexes of the target regex.
- We proposed a LS-based vulnerability modeling that captures global features of the vulnerable LS and a disturbance free attack string generation strategy to solve the identified challenges. We implemented a tool called RENGAR.
- We built a new dataset containing 353,012 regexes from 668,666 real-world projects covering Python, Java, and CSharp. In all datasets, RENGAR shows superior performance advantages compared with nine state-of-the-art.
- We applied RENGAR to 315 real-world projects and identified 69 zero-day vulnerabilities (47 of them have been confirmed with 21 CVEs assigned). All the found vulnerabilities are responsibly reported.

To facilitate future research, we will release both RENGAR and our new dataset [34].

2. Preliminaries

2.1. Background

Notation The following notations will be used in this paper. Let Σ be a finite alphabet of symbols. The set of all strings over Σ is denoted by Σ^*. The empty word and the empty set are denoted by ε and $\emptyset$, respectively. We write $\mathbb{N}$ and ∞ for the set of natural numbers and the infinity, respectively.

Regular Expression (Regex) Regexes are defined on top of *classical regexes*. The syntax of classical regexes is given below:

$$r \ ::= \ \varepsilon \ | \ \emptyset \ | \ a \ | \ [C] \ | \ r|r \ | \ rr \ | \ r\{m,n\}$$

where $a \in \Sigma$, $C \subseteq \Sigma$ is a set of characters, $m \in \mathbb{N}$, $n \in \mathbb{N} \cup \{\infty\}$, and $m \leq n$. Additional operators are defined as syntactic sugaring of the above operators:

$$\begin{array}{llll} r? = r\{0,1\} & r* = r\{0,\infty\} & r\{m\} = r\{m,m\} \\ r = r\{1,1\} & r+ = r\{1,\infty\} & r\{m,\} = r\{m,\infty\} \end{array}$$

A regex over Σ is a well-formed parenthesized formula. Besides the common rules governing classical regexes (*e.g.*, $[C]$, $r|r$, rr, r_1+, $r\{m,n\}$), a regex also has the following constructs: (i) capturing group (r); (ii) non-capturing group

$(?:r)$; (iii) named capturing group $(?<name>r)$; (iv) lookarounds: positive lookahead $(?=r)$, negative lookahead $(?!r)$, positive lookbehind $(?<=r)$, and negative lookbehind $(?<!r)$; (v) anchors: Start-of-line anchor $\hat{}r$, End-of-line anchor $r\$$, word boundary $r\backslash br$, and non-word boundary $r\backslash Br$; (vi) lazy quantifier $r\{m,n\}?$; and (vii) backreferences `\i` which comprise the extended features of regexes.

The *language* $\mathcal{L}(r)$ of a regex r is the set of all strings accepted by r. A subregex of a regex r is a part of r that corresponds to a subtree in the AST. Given a regex r, if a subregex r' of r has a quantifier, then r' is called as a loop subregex (abbr as LS). For an LS, the disjunctions of LS denotes the outmost subregexes of LS divided by $|$. Furthermore, the *unfolding of LS* is a set of regexes composed of its disjunctions, that is, the set of all the iterations on its disjunctions, and the *N-unfolding of LS* denotes the set of regexes whose numbers of iterations are at most N. For example, assuming a regex r is $aa(a*|bc)+$, it has an LS $(a*|bc)+$ with the quantifier $+$. The LS has two disjunctions $a*$ and bc, and its *2-unfolding* is $\{a*, bc, a*a*, bcbc, a*bc, bca*\}$.

Regex Denial of Service (ReDoS) A regex r is *ReDoS-vulnerable* (abbr as vulnerable in following) if and only if there exists an attack string w whose matching cost $Cost(r, w)$ is not linear to the length of w. Usually, the cost is measured by the matching time or the number of backtracking of the engine. Commonly speaking, the vulnerability of a vulnerable regex r can be represented as three key subregexes: the *prefix* subregex φ_1, the *infix* or *pathological* subregex φ_2, and the *suffix* subregex φ_3, where φ_2 contains the position that causes ReDoS, φ_1 and φ_3 are subregexes before and after φ_2. Existing works [23], [24], [30]–[33] usually construct an attack string w following the structure xy^nz, where $x \in \mathcal{L}(\varphi_1)$, $y^n \in \mathcal{L}(\varphi_2)$, and $xy^nz \notin \mathcal{L}(r)$. Note that w has the following key features: ❶ y is matched by multiple distinct subregexes inside the pathological part φ_2. For example, assuming φ_2 is $(r_1|r_2|...|r_m)+$, its distinct subreregexes are r_1, r_2, ..., r_m. Then y should be a common match string among these subregexes. This guarantees y^n, *i.e.*, the repeat of y, can cause ambiguity during the match, which exponentially or polynomially increases the times of backtracking in the engine. ❷ xy^nz is not accepted by r.

2.2. Challenges

Existing works face challenges in both locating candidate vulnerabilities and triggering these vulnerabilities.

C1 Inadequate Vulnerability Modeling Existing modelings summarize vulnerability patterns based on the structural features of the vulnerable LS. The main issue of existing modelings is that their patterns are localized and lack a global understanding of the features of LS. Specifically, assuming an LS is represented as $\hat{}(\mathtt{r_0}|\mathtt{r_1})+\$$ where $\mathtt{r_0}$, $\mathtt{r_1}$ stand for two subregexes. Existing methods propose patterns based on the features of $\mathtt{r_0}$, $\mathtt{r_1}$ without considering the combinations', such as $\mathtt{r_0r_0}$, $\mathtt{r_0r_1}$, $\mathtt{r_1r_1}$ (in the *unfolding of LS*). Summarizing patterns based on a localized understanding of the features can incur inaccurate outcomes. On one hand, due to the limited understanding, the summarized patterns can be excessively strict which causes the detectors to miss vulnerabilities, *i.e.*, have false negatives. On the other hand, if the patterns are excessively loose, the detectors will falsely mark benign regex as vulnerable, *i.e.*, have false positives. TABLE 1 illustrates the limitation using examples. The second and third columns are the two latest works: ReDoSHunter [33] and Revealer [32]. The last column RENGAR shows one of our patterns. The second row details the vulnerability patterns they will use when detecting the following three regexes. **R1** and **R2** have exponential vulnerabilities and **R3** is benign. As shown in the figure, ReDoSHunter and Revealer propose patterns from different angles based on `a`, `b`, `ab` without considering their combinations. However, their patterns suffer from both false positives and false negatives. In comparison, the EOLS pattern of RENGAR outperforms them in both generality and preciseness. The superiority of EOLS comes from the fact that it models the root cause of the vulnerability as the existence of common match strings among all subregexes inside the *unfolding* of the LS, which is a more precise interpretation. Conclusively, it is interesting and promising to remodel the vulnerabilities based on the global understanding of the features of LS.

C2 Ineffective Attack String Generation Caused by Disturbances Existing attack string generation techniques suffer from generating ineffective strings caused by disturbances. Briefly, the disturbance is a phenomenon that the practical match result of an attack string does not meet the generator's expectation due to the fact that a prefix of the attack string can be accepted by the target regex. This unexpected acceptance can invalidate the attack string since it stops the exploitation of vulnerabilities inside the pathological subregex. For example, TABLE 2 shows a vulnerable regex. Existing methods generate the attack string w by calculating x, y^n, z according to the constraints in the definition. Specifically, x should be matched by φ_1, y^n is of the common match strings of the distinct subregexes of φ_2, and z is set as a value satisfying $xy^nz \notin \mathcal{L}(r)$. Usually z is a random value returned by SAT solver, *e.g.*, the '@' in table. The generator expects the y^n can repeatedly cause match ambiguity, *e.g.*, 'c' can be matched by both `[^a]` and `[^b]` which increases the times of backtracking in engine. However, as shown in the fourth column, φ_3 disturbs the expected match process and makes the attack string harmless. To avoid the disturbance, the constraint of y^n should be rewritten as $y^n \in \mathcal{L}(\mathtt{[^a]+}) \wedge y^n \in \mathcal{L}(\mathtt{[^b]+}) \wedge y^n \notin \mathcal{L}(\Sigma^*\mathtt{[^d]+})$, which equals to $y^n \in \mathcal{L}(\mathtt{d+})$. Disturbances can significantly influence the effectiveness of the attack string generators. In the example, if the disturbance is not eliminated, the possibility of generating a disturbance free attack string is much lower than $\ll 0.01\%$ (equals to $\frac{1}{|\mathcal{L}([\hat{}ab])|}$). The disturbance shown in the example is a case that the suffix φ_3 disturbs the pathological subregex φ_2. Indeed, there are more kinds of disturbances that can invalidate the generated attack strings. In summary, the disturbance is an understudied phenomenon which can significantly affect

TABLE 1: Motivation Examples of **C1**. Vuln. stands for vulnerability. TP/FP/TN/FN represents the detection result is True Positive/False Positive/True Negative/False Negative respectively.

	ReDoSHunter	Revealer	RENGAR
Matched Vuln. Pattern	Pattern EOD: R = (r1|r2|...|rk){m, n}, R is vulnerable if ∃ p ≠ q & p, q ∈ [1,k], rp.first ∩ rq.first ≠ ∅ or rp.first ∩ rq.followlast ≠ ∅	Pattern Branch-in-Loop: R = r0(r1(r2|r3)r4)*r5, R is vulnerable if r1r2r4 and r1r3r4 have a common match string	Pattern EOLS: An LS is vulnerable if any two distinct subregexes inside the *unfolding of LS* have a common match string
R1 `^(a|b|ab)+$`	`^(a|b|ab)+$` r1 r2 r3	① r0 = r1 = r4 = r5 = ε `^(a|b|ab)+$` (r2 r3) ② r0 = r1 = r4 = r5 = ε `^(ab|a|b)+$` (r2 r3) ③ r0 = r1 = r4 = r5 = ε `^(b|ab|a)+$` (r2 r3)	LS `^(a|b|ab)+$` r1 r2 r3	*Unfolding of LS*: {r1, r2, **r3**, r1r1, **r1r2**, r1r3, ...}
	∃ r1.first ∩ r3.first = {a} ≠ ∅ → Vul **TP**	∄ r1r2r4,r1r3r4 have common match → Not Vul **FN**	∃ r1r2 , r3 have common match → Vul **TP**
R2 `^(a(b|b)c)+$`		① r0 = r5 = ε `^(a(b|b)c)+$` (r1 r2 r3 r4)	LS `^(a(b|b)c)+$` r1 r2 r3 r4	*Unfolding of LS*: {**r1r2r4**, **r1r3r4**, r1r2r4r1r2r4, ...}
	Not match any pattern → Not Vul **FN**	∃ r1r2r4, r1r3r4 have common match → Vul **TP**	∃ r1r2r4, r1r3r4 have common match → Vul **TP**
R3 `^(a|ab)+$`	`^(a|ab)+$` r1 r2	① r0 = r1 = r4 = r5 = ε `^(a|ab)+$` (r2 r3)	LS `^(a|ab)+$` r1 r2	*Unfolding of LS*: {**r1**, **r2**, r1r1, r1r2, r2r1, ...}
	∃ r1.first ∩ r2.first = {a} ≠ ∅ → Vul **FP**	∄ r1r2r4, r1r3r4 have common match → Not Vul **TN**	∄ x,y; rx,ry have common match → Not Vul **TN**

Note: (i) $r.\text{first} = \{a \mid au \in \mathcal{L}(r), a \in \Sigma, u \in \Sigma^*\}$; (ii) $r.\text{followlast} = \{a \mid uav \in \mathcal{L}(r), u \in \mathcal{L}(r), u \neq \varepsilon, a \in \Sigma, v \in \Sigma^*\}$.

TABLE 2: Motivation Example of **C2**. In "Expected Match" column, ~~φ_3~~ means @ is not matched by φ_3. $y^n \rightarrow cmnMatch([\hat{}a]+, [\hat{}b]+)$ equals to $y^n \in \mathcal{L}([\hat{}a]+) \wedge y^n \in \mathcal{L}([\hat{}b]+)$.

Target Regex	Attack String Generation	Expected Match	Disturbed Match
$r \rightarrow \underbrace{a([\hat{}a]\|[\hat{}b])}_{\varphi_1}\underbrace{+}_{\varphi_2}\underbrace{[\hat{}d]}_{\varphi_3}$	$x \rightarrow a \in L(\varphi_1)$ $y^n \rightarrow cmnMatch([\hat{}a]+, [\hat{}b]+) \rightarrow [\hat{}ab]+ \in L(\varphi_2)$ $w \rightarrow xy^nz \notin L(\varphi_1\varphi_2\varphi_3)$ $\rightarrow \underbrace{accc\ldots ccc}_{y^n}@$ (x, z)	$\underbrace{accc\ldots ccc}_{\varphi_1\ \varphi_2}\underbrace{@}_{\not\varphi_3}$	$\varphi_1\varphi_2\varphi_3$ $accc\ldots ccc@$

the effectiveness of the attack string generation. It should be systematically studied and specific generation methods should be proposed.

2.3. Our Approach

A1 LS-Based Vulnerability Modeling RENGAR models vulnerabilities based on the global features of the *unfolding of LS*. Our key observation is that the existence of common match strings among all subregexes inside the *unfolding of LS* is a more precise interpretation of the root cause of a ReDoS vulnerability. Based on that finding, we checked all vulnerabilities covered in existing works [5], [23]–[27], [32], [33] and remodeled them using our observation.

As shown in TABLE 3, RENGAR proposes three patterns: EOLS (exponential vulnerability with one pathological LS), POLS (polynomial vulnerability with one pathological LS), and PTLS (polynomial vulnerability with two pathological LSes). In the name of the pattern, "exponential/polynomial" indicates its vulnerability severity. Interestingly, all exponential vulnerabilities are summarized as an EOLS pattern. This is because the key of any exponential vulnerability is that: ❶ there is a pathological subregex that can cause additional backtracking in engine when matching specific strings; ❷ the pathological subregex has a quantifier which exponentially magnifies the times of backtracking. Since LS represents a regex with quantifier, any exponential vulnerability can be modeled as an LS whose subregexes inside its *unfolding* have common match strings. Polynomial vulnerabilities are summarized as two patterns. PTLS represents a common type of polynomial vulnerabilities whose root cause involves two pathological LSes, *e.g.*, `a+a+`. POLS describes a special kind of polynomial vulnerabilities which is caused by a single pathological LS, *e.g.*, `a+$`.

Figure 1 shows a detailed scope comparison of covered vulnerabilities among ReDoSHunter, Revealer, and RENGAR. The comparison shows that patterns of RENGAR can cover not only vulnerabilities which can be described by existing patterns but also new vulnerabilities which cannot. Especially, RENGAR uses one pattern EOLS to describe all exponential vulnerabilities while other works use two or

TABLE 3: Three Types of RENGAR Vulnerability Patterns. Vuln. stands for vulnerability.

Vuln. Pattern	Vuln. Severity	Vuln. Description	Example Exhibition
❶ EOLS	Exponential	There exists a pathological LS such that any two distinct subregexes inside the *unfolding of LS* have a common match string.	`^(ab|a|b)+$` belongs to the pattern EOLS as it exists (i) a pathological LS `(ab|a|b)+`, and (ii) two distinct subregexes `(ab)+` and `(a|b)+` have common match strings $\{\texttt{ab}, \texttt{abab}, \ldots\}$.
❷ POLS	Polynomial	The regex starts with a *start-of-line*-free subregex r_1. The subregex r_1 has the following possible forms: (i) r_1 is a pathological LS; (ii) $r_1 = \beta_0\beta_1$ where β_1 is a pathological LS and $\mathcal{L}(\Sigma^*\beta_0\Sigma^*) \cap \mathcal{L}(\beta_1) \neq \emptyset$ [1].	`a(ba*)+$` belongs to the pattern POLS as it satisfies (i) `(ba*)+` is a pathological LS, and (ii) $\mathcal{L}(\Sigma^*\texttt{a}\Sigma^*) \cap \mathcal{L}(\texttt{(ba*)+}) = \{\texttt{ba}, \texttt{baa}, \ldots\} \neq \emptyset$, where $\beta_0 = \texttt{a}$ and $\beta_1 = \texttt{(ba*)+}$.
❸ PTLS	Polynomial	There exists (i) $\beta_0\beta_1$ such that β_0 and β_1 are two pathological LSes and have a common match string; (ii) $\beta_0\beta_1\beta_2$ such that β_0 and β_2 are two pathological LSes, as well as β_0, β_2, ($\beta_0\beta_1$ or $\beta_1\beta_2$) have a common match string.	`^(ab+)+a(\w+)$` belongs to the pattern PTLS as it triggers the second condition (i) `(ab+)+` and `(\w+)` are two pathological LSes, as well as (ii) `(ab+)+`, `(\w+)` and `a(\w+)` have common match strings $\{\texttt{ab}, \texttt{abb}, \ldots\}$, where $\beta_0 = \texttt{(ab+)+}$, $\beta_2 = \texttt{(\w+)}$ and $\beta_1\beta_2 = \texttt{a(\w+)}$.

TABLE 4: Analysis on Disturbances.

(a) Enumeration of Possible Disturbances. σ represents a string, "-" represents that the corresponding type of disturbance either does not exist (the condition always be false) or does not invalidate the attack string.

	Expected Match Results	Disturbed Results (Accepted by ϑ_1)	Disturbed Results (Accepted by $\varphi_1\vartheta_2\varphi_3$)	Disturbed Results (Accepted by $\varphi_1\varphi_2\varphi_3$)
$s = x[i], i \in [1, \lvert x\rvert]$	$\exists\sigma$, let $s\sigma \in \mathcal{L}(\varphi_1)$	$s \in \mathcal{L}(\vartheta_1)$ ❶	-	-
$s = x(y^n[i]), i \in [1, \lvert y^n\rvert]$	$\exists\sigma$, let $s\sigma \in \mathcal{L}(\varphi_1\varphi_2)$	$s \in \mathcal{L}(\vartheta_1)$ ❷	$s \in \mathcal{L}(\varphi_1\vartheta_2\varphi_3)$ ❹	$s \in \mathcal{L}(\varphi_1\varphi_2\varphi_3)$ ❺
$s = xy^n(z[i]), i \in [1, \lvert z\rvert]$	$\exists\sigma$, let $s\sigma \in \mathcal{L}(\varphi_1\varphi_2\neg\varphi_3)$	$s \in \mathcal{L}(\vartheta_1)$ ❸	-	-

(b) Five Disturbance Cases for Attack String Generation.

Disturbance Case	Vulnerable Regex $r = \vartheta_1\lvert(\varphi_1(\varphi_2\lvert\vartheta_2)\varphi_3)$	Failed Attack String $w = xy^nz$	Increased Constraint		Successful Attack String $w = xy^nz$
❶ if $\mathcal{L}(\vartheta_1\Sigma^*) \cap \mathcal{L}(\varphi_1) \neq \emptyset$, then ϑ_1 disturbs φ_1	`a|[ab](c+)+d`, where $\vartheta_1 = \texttt{a}$, $\varphi_1 = \texttt{[ab]}$, $\varphi_2 = \texttt{(c+)+}$, $\varphi_3 = \texttt{d}$, $\vartheta_2 = \varepsilon$	$x = \texttt{a}$, $y = \texttt{c}$, $z = \varepsilon$ where the prefix string $w[1] = \texttt{a}$ of w can be matched by ϑ_1	$x \notin \mathcal{L}(\vartheta_1\Sigma^*)$	(1)	$x =$b, $y =$c, $z = \varepsilon$
❷ if $\mathcal{L}(\vartheta_1\Sigma^*) \cap \mathcal{L}(\varphi_1\varphi_2) \neq \emptyset$, then ϑ_1 disturbs $\varphi_1\varphi_2$	`ab|a([bc]+)+d`, where $\vartheta_1 = \texttt{ab}$, $\varphi_1 = \texttt{a}$, $\varphi_2 = \texttt{([bc]+)+}$, $\varphi_3 = \texttt{d}$, $\vartheta_2 = \varepsilon$	$x = \texttt{a}$, $y = \texttt{b}$, $z = \varepsilon$ where the prefix string $w[1:2] = \texttt{ab}$ of w can be matched by ϑ_1	$xy^n \notin \mathcal{L}(\vartheta_1\Sigma^*)$	(2)	$x =$a, $y =$c, $z = \varepsilon$
❸ if $\mathcal{L}(\vartheta_1\Sigma^*) \cap \mathcal{L}(\varphi_1\varphi_2\neg\varphi_3) \neq \emptyset$, then ϑ_1 disturbs $\varphi_1\varphi_2\varphi_3$	`ab*c|a(b+)+[^cd]`, where $\vartheta_1 = \texttt{ab*c}$, $\varphi_1 = \texttt{a}$, $\varphi_2 = \texttt{(b+)+}$, $\varphi_3 = \texttt{[^cd]}$, $\vartheta_2 = \varepsilon$	$x =$a, $y =$b, $z =$c where w can be matched by ϑ_1	$xy^nz \notin \mathcal{L}(\vartheta_1\Sigma^*)$	(3)	$x =$a, $y =$b, $z =$d
❹ if $\mathcal{L}(\vartheta_2\varphi_3\Sigma^*) \cap \mathcal{L}(\varphi_2) \neq \emptyset$, then $\vartheta_2\varphi_3$ disturbs φ_2	`a(a|((a[bc])+)+)b`, where $\vartheta_1 = \varepsilon$, $\varphi_1 = \texttt{a}$, $\varphi_2 = \texttt{((a[bc])+)+}$, $\varphi_3 = \texttt{b}$, $\vartheta_2 = \texttt{a}$	$x = \texttt{a}$, $y = \texttt{ab}$, $z = \varepsilon$ where the prefix string $w[1:3] = \texttt{aab}$ of w can be matched by $\varphi_1\vartheta_2\varphi_3$	$y^n \notin \mathcal{L}(\vartheta_2\varphi_3\Sigma^*)$	(4)	$x =$a, $y =$ac, $z = \varepsilon$
❺ if $\mathcal{L}(\varphi_3\Sigma^*) \cap \mathcal{L}(\varphi_2) \neq \emptyset$, then φ_3 disturbs φ_2	`a([^a]|[^b])+[^d]`, where $\vartheta_1 = \vartheta_2 = \varepsilon$, $\varphi_1 = \texttt{a}$, $\varphi_2 = \texttt{([^a]|[^b])+}$, $\varphi_3 = \texttt{[^d]}$	$x = \texttt{a}$, $y = \texttt{c}$, $z = \texttt{d}$ where the prefix string $w[1:\lvert w\rvert-1] = \texttt{acc}\ldots\texttt{c}$ of w can be matched by $\varphi_1\varphi_2\varphi_3$	$y^n \notin \mathcal{L}(\varphi_3\Sigma^*)$	(5)	$x =$a, $y =$d, $z = \varepsilon$

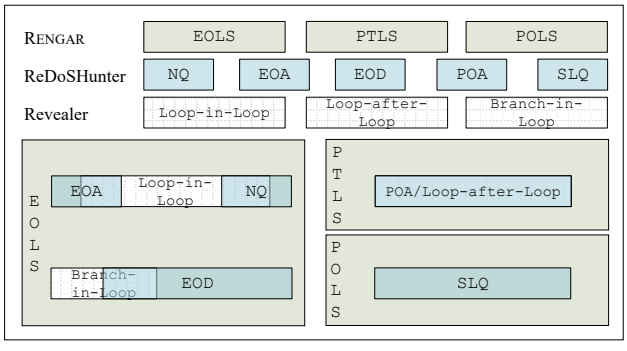

Figure 1: Vulnerability Scope Comparison for Three Approaches. The sizes of the shapes do not reflect quantitative proportions but only qualitative comparison results.

three patterns. This shows the conciseness and generality of RENGAR's pattern.

Based on the modeling, RENGAR designs algorithms to search candidate vulnerable regexes. More details are presented in Section 3.1.

A2 Disturbance Free Attack String Generation To systematically analyze the disturbances, we first need to clarify a refined structure of the vulnerability in the vulnerable regex r. It is a commonsense that the vulnerability of r is represented as three key subregexes: prefix subregex φ_1, infix/ pathological subregex φ_2, and suffix subregex φ_3. However, some subregexes of r irrelevant with the vulnerability may disturb the expected match process. For that we propose a fine-grained representation for the vulnerability of r, that is $\vartheta_1\lvert(\varphi_1(\varphi_2\lvert\vartheta_2)\varphi_3)$, where φ_1, φ_2, and φ_3 represent the same thing as before, and ϑ_1 and ϑ_2 are two disturbance subregexes which are commonly seen as irrelevant with the vulnerabilities. Note that ϑ_1 and ϑ_2 are not created, we highlight them (if they exist in r), since they may make the match not follow the expected routine. Compared with the former representation ($\varphi_1\varphi_2\varphi_3$), the fine-grained representation keeps more structural information of r. And the former can be seen as a special case of the fine-grained one. For example, in the case of regex shown in TABLE 2, both ϑ_1 and ϑ_2 are ϵ and the fine-grained representation is degraded as the former one.

According to Section 2.1, an attack string w is defined as xy^nz satisfying $w \notin \mathcal{L}(r)$, $x \in \mathcal{L}(\varphi_1)$, and $y^n \in \mathcal{L}(\varphi_2)$. Note that w is expected to be not accepted by r. However,

Algorithm 1: General Workflow of RENGAR

Input: a regex r
Output: (is vulnerable, a diagnostic list Γ)
1 $r' \leftarrow \text{shape}(r)$;
2 $\mathcal{K} \leftarrow \text{extract}(r')$;
3 **if** $|\mathcal{K}| = 0$ **then return** $(false, null)$;
4 $\mathcal{S} \leftarrow \text{LocVuln}(r', \mathcal{K})$;
5 **if** $|\mathcal{S}| = 0$ **then return** $(false, null)$;
6 $\mathcal{Q} \leftarrow \text{SolStr}(\mathcal{S})$;
7 $\Gamma \leftarrow \text{CheckReDoS}(\mathcal{Q}, r)$;
8 **if** $|\Gamma| > 0$ **then return** $(true, \Gamma)$;
9 **else return** $(false, null)$;

when the disturbance happens, w is accepted since its prefix is accepted by r (w is also a prefix of itself). For any attack string w, there are three kinds of disturbed results: its prefix s is accepted by ϑ_1, $\varphi_1\vartheta_2\varphi_3$, and $\varphi_1\varphi_2\varphi_3$. Besides, the expected matching results of s can also be divided into three cases: s is a part of x, s is a part of xy^n, and s is a part of xy^nz. TABLE 4.(a) enumerates the possible disturbances by enumerating all pairs of the expected and disturbed results for a prefix string s. Note that the table only details five of the nine pairs. The rest four either do not exist theoretically or cannot invalidate the attack string. TABLE 4.(b) formalizes the five disturbances and further illustrates them with examples. For each listed disturbance, RENGAR eliminates its disturbances by adding constraints to w. These constraints can guide the generator to generate disturbance free attack strings. The constraints are listed in the fourth column of TABLE 4.(b).

Based on the above analysis, RENGAR designs a disturbance free attack string generation algorithm. For each target regex, the algorithm calculates the existence of the five disturbances and then unifies all constraints. Last, the constraints are used to guide disturbance free attack string generation. See more details in Section 3.2.

3. Methodology

The general workflow of RENGAR is shown as Algorithm 1. RENGAR consists of three steps, namely, *candidate vulnerability searching* (Section 3.1), *attack string generation* (Section 3.2) and *dynamic validation* (Section 3.3). In the first step, RENGAR searches for possible vulnerabilities defined in our patterns based on LS(es). After that, RENGAR generates relevant attack strings for each possible vulnerability. Finally, it determines whether the regex is pathological by counting the matching steps of generated strings.

In detail, inspired from Li et al. [33], RENGAR first converts the initial regex r with backreferences into an *over-approximate* backreference-free regex r' via wrapping the content of i-th capturing group with a non-capturing group and replacing each backreference `\i` with it (line 1). Since all vulnerability patterns (*i.e.*, *EOLS*, *POLS*, and *PTLS* patterns) start from an LS, RENGAR then identifies all the LSes in the regex r' and save the identified LSes into a set

Algorithm 2: LocVuln

Input: a regex r, an LS list $\mathcal{K}$
Output: a set $\mathcal{S}$
1 $\mathcal{S} \leftarrow \{\}$
2 **foreach** $k \in \mathcal{K}$ **do**
3 $\quad \vartheta_1, \varphi_1, \varphi_2, \vartheta_2, \varphi_3 \leftarrow \text{split}(k, r)$
4 $\quad \mathcal{IR}_{eols} \leftarrow$ LocEOLS(φ_2)
5 $\quad$ **if** $\mathcal{IR}_{eols}$ *is not empty* **then**
6 $\quad\quad \mathcal{S} \xleftarrow{+} (\eta, \tau) \leftarrow ((\vartheta_1, \varphi_1, \mathcal{IR}_{eols}, \vartheta_2, \varphi_3),$ EOLS)
7 $\quad$ **if** $\varphi_1\varphi_2$ *is Start-of-Line-free* **then**
8 $\quad\quad$ **if** φ_1 *is nullable* **then**
9 $\quad\quad\quad \mathcal{IR}_{pols} \leftarrow$ LocPOLS(φ_2)
10 $\quad\quad$ **else**
11 $\quad\quad\quad \mathcal{IR}_{pols} \leftarrow$ LocPOLS(φ_1, φ_2)
12 $\quad\quad$ **if** $\mathcal{IR}_{pols}$ *is not empty* **then**
13 $\quad\quad\quad \mathcal{S} \xleftarrow{+} (\eta, \tau) \leftarrow ((\vartheta_1, \varphi_1, \mathcal{IR}_{pols}, \vartheta_2, \varphi_3),$ POLS)
14 **foreach** $k_1 \in \mathcal{K}$, $k_2 \in \mathcal{K}$, $k_1 \neq k_2$, k_1 *and* k_2 *not nested* **do**
15 $\quad$ **if** k_1 *and* k_2 *are adjacent* **then**
16 $\quad\quad \varphi_2 \leftarrow \text{splice}(k_1, k_2)$
17 $\quad\quad \mathcal{IR}_{ptls} \leftarrow$ LocPTLS(k_1, k_2)
18 $\quad$ **else**
19 $\quad\quad \rho \leftarrow \text{getSubRE}(k_1, k_2, r)$
20 $\quad\quad \varphi_2 \leftarrow \text{splice}(k_1, \rho, k_2)$
21 $\quad\quad \mathcal{IR}_{ptls} \leftarrow$ LocPTLS(k_1, ρ, k_2)
22 $\quad \vartheta_1, \varphi_1, \varphi_2, \vartheta_2, \varphi_3 \leftarrow \text{split}(\varphi_2, r)$
23 $\quad$ **if** $\mathcal{IR}_{ptls}$ *is not empty* **then**
24 $\quad\quad \mathcal{S} \xleftarrow{+} (\eta, \tau) \leftarrow ((\vartheta_1, \varphi_1, \mathcal{IR}_{pols}, \vartheta_2, \varphi_3),$ PTLS)
25 **return** $\mathcal{S}$

$\mathcal{K}$ (line 2). If $\mathcal{K}$ is empty, RENGAR returns $false$, meaning that it diagnoses the given regex as *safe* (line 3). Otherwise, it statically analyzes all the potential vulnerabilities and outputs a set $\mathcal{S}$ (*i.e.*, a five-tuple consisting of *infix subregex*, *prefix subregex*, *suffix subregex* and *two disturbance subregexes*, as well as *vuln. pattern*) (line 4). If $\mathcal{S}$ is empty, RENGAR returns $false$, indicating that it diagnoses the given regex as *safe* (line 5). Otherwise, it next generates the corresponding attack strings for each element in $\mathcal{S}$ (line 6). At last, RENGAR dynamically validates the generated attack strings and returns the confirmed vuln. information (*i.e.*, *pathological subregex*, *attack string*, *vuln. pattern*) list Γ (line 7). RENGAR returns $true$ and the vuln. list Γ if it is not empty (line 8), or returns $false$ otherwise (line 9).

3.1. Candidate Vulnerability Searching

In this section, we describe Algorithm LocVuln that circles all the candidate *pathological* subregexes, as illustrated in Algorithm 2. Before that, we extract all the LSes $\mathcal{K}$ in the regex r via calling the function $\text{extract}(r)$.

TABLE 5: Computing Rules of $\mathcal{M}axU(r)$ of a Regex r.

$\mathcal{M}axU(r)$	Regex r
0	$r = \varepsilon$, $r = (?{=}r_1)$, $r = (?!r_1)$, $r = (?<{=}r_1)$, or $r = (?\lhd r_1)$
1	$r = a \in \Sigma$ or $r = [C] \subseteq \Sigma$
$\mathcal{M}axU(r_1) + \mathcal{M}axU(r_2)$	$r = r_1 \vert r_2$, $r = r_1 r_2$, $r_1 \backslash br_2$ or $r_1 \backslash Br_2$
$\mathcal{M}axU(r_1)$	$r = \hat{}r_1$, $r = r_1\$$, $r = r_1?$, $r = (r_1)$, $r = (?:r_1)$, $r = (?<name>r_1)$, or $r = \backslash i$ (references (r_1))
$\mathcal{M}axU(r_1) \times m$	$r = r_1\{m\}$, $r = r_1\{m,n\}$, where $m = n$
$\mathcal{M}axU(r_1) \times (m+1)$	$r = r_1\{m,n\}$ where $m \neq n$, or $r = r_1\{m,\}$
$\mathcal{M}axU(r_1) \times 2$	$r = r_1*$ or $r = r_1+$

The major ideas of this algorithm are as follows. Guided by our fined-grained vulnerability representation, we partition a regex into different combinations of the prefix, infix, suffix and two disturbance subregexes (prefix, suffix, and two disturbance subregexes can be ε), and each of the combinations must satisfy that the infix subregex contains potential ReDoS vulnerabilities. In other words, the infix subregexes are the pathological subregexes. ReDoS vulnerabilities may be caused by LSes, so each LS (*i.e.*, EOLS and POLS patterns) is treated as an infix subregex. Furthermore, ReDoS vulnerabilities may be caused by the combination of any two non-ancestor-descendant[1] LSes (*i.e.*, PTLS pattern).

Specifically, LocVuln first treats each LS k as the pathological subregex (*i.e.*, infix subregex) φ_2, then extracts the prefix subregex φ_1, the suffix subregex φ_3, the disturbance subregexes ϑ_1 and ϑ_2 from r (line 3). Then, LocVuln calls function LocEOLS to check whether the infix subregex satisfies the EOLS pattern presented in TABLE 3 and returns a set $\mathcal{IR}_{eols}$ of involved regexes that share a common string (line 4), wherein the *unfolding* of k is limited by $\mathcal{M}axU(\varphi_2)$, where $\mathcal{M}axU$ is the pre-defined maximum number of unfoldings given in TABLE 5. If $\mathcal{IR}_{eols}$ is not empty, meaning that k is a possible EOLS vulnerability, then LocVuln sets τ as vulnerability pattern EOLS, creates a five-tuple η consisting of the set of involved subregexes (*i.e.*, $\mathcal{IR}_{eols}$) and the four subregexes (*i.e.*, φ_1, φ_3, ϑ_1 and ϑ_2), and further appends the five-tuple η and vulnerability pattern τ to the set $\mathcal{S}$ (line 6). Next, LocVuln calls function LocPOLS to analyze the satisfiability of pattern POLS. It is noted that $\varphi_1\varphi_2$ should be Start-of-Line-free, which conforms to the vulnerability description of pattern POLS presented in TABLE 3. If φ_1 is nullable, then case (i) is satisfied, and LocPOLS gets the set $\mathcal{IR}_{pols}$ of involved regexes from φ_2 (line 9). Otherwise, LocPOLS gets the involved regex set from φ_1 and φ_2 (*i.e.*, $\Sigma^*\varphi_1\Sigma^*$ and φ_2) (line 11). Similarly, if $\mathcal{IR}_{pols}$ is not empty, LocVuln sets τ as vuln. pattern POLS, and adds the corresponding vulnerability information into the set $\mathcal{S}$ (line 13). After that, to deal with the cases caused by the combination of two non-ancestor-descendant LSes, LocVuln considers each two distinct LSes k_1 and k_2 such that k_1 and k_2 are not nested (line 14). If the two LSes k_1 and k_2 are adjacent, according to case (i) of vulnerability pattern PTLS described in TABLE 3 (line 15), LocVuln concatenates k_1 and k_2 into the infix subregex φ_2 and calls function LocPTLS to gets the involved regex set $\mathcal{IR}_{ptls}$ from them (lines 16 – 17). Otherwise, case (ii) is satisfied. In this case, LocVuln stores the subregex between k_1 and k_2 in ρ, concatenates k_1, ρ and k_2 into the infix subregex φ_2, and get the involved regex set $\mathcal{IR}_{ptls}$ from k_1, ρ and k_2 (*i.e.*, besides k_1 and k_2, LocPTLS also considers the regexes $k_1\rho$ and ρk_2) (lines 19 – 21). Different from pattern EOLS and POLS, due to the infix subregex φ_2 is changed, LocVuln will re-separate the prefix subregex φ_1, the suffix subregex φ_3, the disturbance subregexes ϑ_1 and ϑ_2 for pattern PTLS (line 22). Likewise, LocVuln will add the vulnerability information into the set $\mathcal{S}$ if $\mathcal{IR}_{ptls}$ is not empty (line 24). Finally, LocVuln returns the set $\mathcal{S}$ (line 25).

[1]For the regex `/(a+b)+c*/`, the LS `/(a+b)+/` and the LS `/a+/` are ancestor-descendant, while the LS `/(a+b)+/` and the LS `/c*/` are non-ancestor-descendant, and the LS `/a+/` and the LS `/c*/` are non-ancestor-descendant.

3.2. Attack String Generation

In this section, we describe Algorithm SolStr that generates attack strings. One key point here is how to generate effective strings unaffected by disturbances. For this purpose, we first compose the constraints that the attack strings must satisfy, and then generate them accordingly.

Specifically, for each element consisting of a five-tuple η and vuln. pattern τ in set $\mathcal{S}$ obtained from LocVuln, SolStr first unpacks the five-tuple η, and respectively initializes the string set $\mathcal{W}$ and the constraint set $\mathcal{CS}$ as an empty set and the set $\{x \in \mathcal{L}(\varphi_1), xy^n z \notin \mathcal{L}(r)\}$ (lines 3 – 5). Then, SolStr adds the corresponding constraints to prevent the generated attack strings from being affected by disturbances, according to the disturbance cases presented in TABLE 4.(b) (lines 6 – 18). For example, if $\vartheta_1 \neq \varepsilon$ and $\mathcal{L}(\vartheta_1\Sigma^*) \cap \mathcal{L}(\varphi_1) \neq \emptyset$, SolStr adds EQ (1) to $\mathcal{CS}$ (line 8). Next, for each involved regex $ir \in \mathcal{IR}$, SolStr first makes a copy $\mathcal{CS}'$ of the current constraint set $\mathcal{CS}$ and adds the constraint $y \in \mathcal{L}(ir)$ to $\mathcal{CS}'$ (lines 20 – 21). If the vuln. pattern τ is EOLS, SolStr sets $|y^n|$ as $N_{\mathcal{E}}$, $N_{\mathcal{P}}$ otherwise (lines 22 – 23), where $N_{\mathcal{E}}$ and $N_{\mathcal{P}}$ are pre-defined numbers of repetitions for exponential ReDoS vulnerabilities and polynomial ones, respectively. After that, SolStr generates a string $w = xy^n z$ satisfying the constraint set $\mathcal{CS}'$, and adds w to $\mathcal{W}$ (line 24). Once the attack string generation for a five-tuple η is finished, SolStr stores the possible attack strings $\mathcal{W}$ and vuln. pattern τ in dictionary $\mathcal{Q}$ and associates them with the five-tuple η (line 25). Finally, SolStr returns dictionary $\mathcal{Q}$ after all elements in set $\mathcal{S}$ are processed (line 26).

3.3. Dynamic Validation

CheckReDoS validates whether the ReDoS vulnerabilities reported by Algorithm 3 are true via counting the matching steps of the corresponding attack strings.

First of all, CheckReDoS initializes a diagnostics list Γ as empty. In particular, for each candidate vulnerability $(\eta, [\mathcal{W}, \tau]) \in \mathcal{Q}$, CheckReDoS checks whether there is an

Algorithm 3: SolStr

Input: a regex r, a (five-tuple η, vuln. pattern τ) set $\mathcal{S}$
Output: a dict $\mathcal{Q}$

$\mathcal{Q} \leftarrow \{\}$
foreach *(η, τ)* $\in \mathcal{S}$ **do**
  $\vartheta_1, \varphi_1, \mathcal{IR}, \vartheta_2, \varphi_3 \leftarrow \eta$
  $\mathcal{W} \leftarrow []$
  constraint set $\mathcal{CS} \leftarrow \{x \in \mathcal{L}(\varphi_1), xy^n z \notin \mathcal{L}(r)\}$
  if $\vartheta_1 \neq \varepsilon$ **then**
    if $\mathcal{L}(\vartheta_1\Sigma^*) \cap \mathcal{L}(\varphi_1) \neq \emptyset$ **then**
      $\mathcal{CS} \overset{+}{\leftarrow}$ EQ (1)
    if $\mathcal{L}(\vartheta_1\Sigma^*) \cap \mathcal{L}(\varphi_1\varphi_2) \neq \emptyset$ **then**
      $\mathcal{CS} \overset{+}{\leftarrow}$ EQ (2)
    if $\mathcal{L}(\vartheta_1\Sigma^*) \cap \mathcal{L}(\varphi_1\varphi_2\neg\varphi_3) \neq \emptyset$ **then**
      $\mathcal{CS} \overset{+}{\leftarrow}$ EQ (3)
  if $\vartheta_2 \neq \varepsilon$ **then**
    if $\mathcal{L}(\vartheta_2\varphi_3\Sigma^*) \cap \mathcal{L}(\varphi_2) \neq \emptyset$ **then**
      $\mathcal{CS} \overset{+}{\leftarrow}$ EQ (4)
  if $\varphi_3 \neq \varepsilon$ **then**
    if $\mathcal{L}(\varphi_3\Sigma^*) \cap \mathcal{L}(\varphi_2) \neq \emptyset$ **then**
      $\mathcal{CS} \overset{+}{\leftarrow}$ EQ (5)
  foreach $ir \in \mathcal{IR}$ **do**
    $\mathcal{CS}' \leftarrow \mathcal{CS}$
    $\mathcal{CS}' \overset{+}{\leftarrow} y \in \mathcal{L}(ir)$
    if $\tau =$*EOLS* **then** $|y^n| = N_{\mathcal{E}}$;
    else $|y^n| = N_{\mathcal{P}}$;
    generate a string $w = xy^n z$ that satisfies the constraints in $\mathcal{CS}'$, and add it to $\mathcal{W}$;
  $\mathcal{Q}[\eta] \leftarrow [\mathcal{W}, \tau]$
return $\mathcal{Q}$

attack string $w \in \mathcal{W}$ whose matching steps are greater than or equal to $T = 10^5$, where T is a user-settable threshold and 10^5 is the public accepted matching steps [30], [32] for determining if the regex is pathological. If such an attack string w exists, CheckReDoS unpacks the five-tuple η and records the corresponding pathological subregex φ_2, the attack string w and the vuln. pattern τ into the list Γ, as well as stops checking the remaining strings in $\mathcal{W}$. Finally, CheckReDoS returns the set Γ.

4. Evaluation

Implementation We implemented the RENGAR prototype with over 13,904 non-comment lines of Java code, which supports detecting vulnerable regex in four programming languages including Java, Python, CSharp and JavaScript. RENGAR leverages a number of other existing tools. First, our regex parser is built on top of the ANTLR4 framework [35] and makes use of its existing functionalities, such as parse tree construction and traversal. Second, RENGAR utilizes the Z3 SMT solver [36] to deduce possible attack strings for triggering ReDoS vulnerabilities. Finally, leveraging the Java regex engine, RENGAR obtains the matching steps through instrumentation.

Evaluation Questions The evaluation aims to answer the following research questions (RQs):

- **RQ1**: How is the performance of RENGAR comparing with the state-of-the-art ReDoS detection tools?
- **RQ2**: Is RENGAR improved by solving the discussed two challenges?
- **RQ3**: How does RENGAR perform for detecting vulnerabilities in real-world applications?

4.1. Experiment Setup

Dataset Our evaluation is based on seven datasets: (i) Corpus [3], (ii) RegExLib [33], (iii) Snort [33], (iv) Regex101[2], (v) Maven, (vi) NuGet, and (vii) PyPI. The datasets (i) – (iv) are widely used datasets in previous ReDoS works. The last three datasets are built by ourselves covering the real-world regexes of public projects on a large scale. Specifically, each dataset contains the regexes of one programming language: Python, Java, and CSharp for (v), (vi), and (vii). For the dataset build process, we first collected 279,266 Python projects, 271,839 Java projects, and 117,561 CSharp projects from PyPI, Maven, and NuGet. Next, the regexes are statically extracted from these projects. For Python projects, the regexes are extracted by an AST-based regex extractor implemented by Davis et al. [5]. For Java and CSharp projects, we implemented our own regex extractors. In total, there are 353,012 regexes in our evaluation dataset. The details of these regexes can be found in TABLE 11 of Appendix 7.1.

Metrics To evaluate the effectiveness of RENGAR, we use the following evaluation metrics:

- **Precision**: The ratio of the number of true positives to the total number of the reported vulnerabilities (including true positives and false positives).
- **Recall**: The ratio of the number of true positives to the total number of all the real vulnerabilities (including true positives and false negatives).

Ground Truth The evaluation relies on knowing the ground truth of ReDoS Vulnerabilities. For these unconfirmed regexes from the above datasets, we took the following steps to set up the ground truth: ❶ We obtained a set of vulnerable regex candidates by applying RENGAR and other state-of-the-art tools (nine baselines mentioned in Section 4.2) to all the regexes inside the datasets, resulting in a set of suspicious vulnerable regexes (175,850) reported by at least one approach; ❷ We used the attack strings generated by these tools to trigger the attacks. Once an attack succeeds, it means that a PoC which proves the regex is vulnerable has been found. ❸ For the regexes (12.49%, 21,970 regexes) which tools cannot generate PoCs, a manual validation process was conducted for the classification. Three authors did the manual validation separately and any inconsistent

[2] https://regex101.com/library

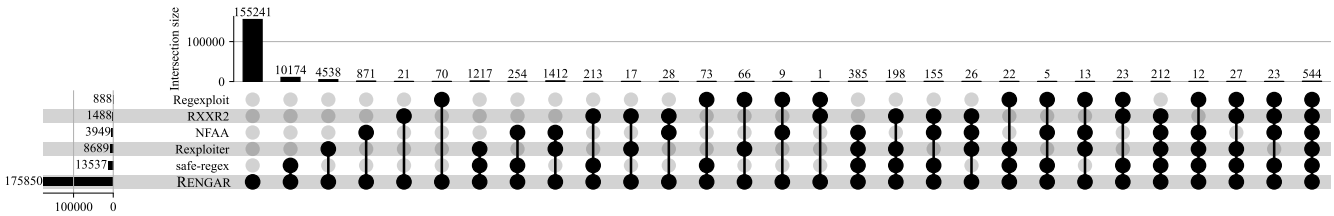

Figure 2: **RQ1** UpSet Plot for RENGAR and Five Static Detectors.

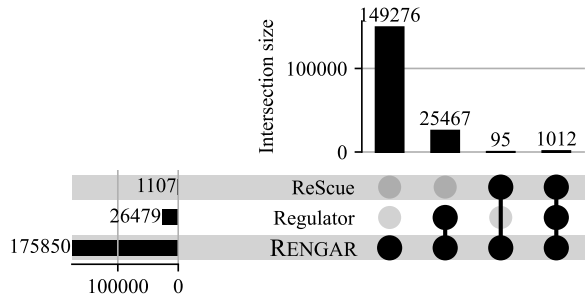

Figure 3: **RQ1** UpSet Plot for RENGAR and Two Dynamic Detectors.

judgment was discussed until reaching a consensus. One key strategy which significantly saved our efforts is that we first wrote scripts to extract the LSes inside the regexes and then manually validated the vulnerability by solely validating the LSes. In total, the manual validation costs 1.5 man-months.

Configurations We conducted experiments on a machine with 20 cores Intel Xeon Silver 4210 CPU @ 2.20GHz with 27.5MB Cache, 128GB RAM, running Windows 10 operating system. For all the experiments described below we set parameters $N_{\mathcal{P}} = 15,000$, $N_{\mathcal{E}} = 100$, and $T = 10^5$ in our algorithms. All baselines were configured in the same settings as reported in their original papers.

Key Concepts on Evaluation Results There are two concepts involved in our evaluation: "a ReDoS vulnerable regex" and "an exploitable ReDoS vulnerability". The former is the detected TP results in RQ1/RQ2 while the latter is the reported real-world vulnerability in RQ3. Generally, the existence of an exploitable ReDoS vulnerability in a real-world project requires that: ❶ The project contains a ReDoS vulnerable regex; ❷ It accepts an attacker-controllable input which can exploit that regex and cause significant consequences such as a severe performance decline. All detectors including ReDoSHunter, Revealer, and RENGAR limit the scope to the ReDoS vulnerable regex detection. Identifying exploitable ReDoS vulnerability from the regexes is out of scope and requires extensive engineering efforts.

4.2. State-of-the-Art Comparison (RQ1)

Baselines To answer RQ1, we compared RENGAR with nine state-of-the-art baselines belonging to three paradigms, *i.e.*, static approaches, dynamic approaches, and hybrid approaches. Detectors of static approaches, abbr as static detectors, include RXXR2 [23], [24], NFAA [27], Rexploiter [25], safe-regex [5], and Regexploit [26]. Detectors of dynamic approaches, abbr as dynamic detectors, include ReScue [30] and Regulator [31]. Revealer [32] and ReDoSHunter [33] are detectors of hybrid approaches (abbr as hybrid detectors), which combine static and dynamic analysis as RENGAR.

TABLE 6: The Overall Evaluation Results of **RQ1**. TY represents the type of detector, which can be S (Static Detector), or D (Dynamic Detector), or H (Hybrid Detector). TP/FP/FN stands for true positives/false positives/false negatives respectively. The best value in a column is highlighted in bold.

Approach	TY	TP	FP	FN	Precision	Recall	Avg Time (s)
RXXR2	S	1,488	105	174,362	93.41%	0.85%	0.3428
NFAA	S	3,949	284	171,901	93.29%	2.25%	1.5499
Rexploiter	S	8,689	3,450	167,161	71.58%	4.94%	0.6186
safe-regex	S	13,537	21,799	162,313	38.31%	7.70%	0.4865
Regexploit	S	888	32	174,962	96.52%	0.50%	**0.0007**
ReScue	D	1,107	**0**	174,743	**100.00%**	0.63%	3.5782
Regulator	D	26,479	**0**	149,371	**100.00%**	15.06%	336.0527
Revealer	H	4,720	**0**	171,130	**100.00%**	2.68%	1.3685
ReDoSHunter	H	44,005	**0**	131,845	**100.00%**	25.02%	1.5543
RENGAR	H	**175,850**	**0**	**0**	**100.00%**	**100.00%**	0.5828

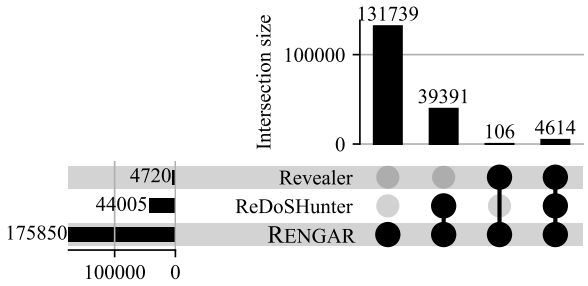

Figure 4: **RQ1** UpSet Plot for RENGAR and Two Hybrid Detectors.

Overall Results The overall evaluation results are listed in TABLE 6. RENGAR significantly outperforms all baselines in all metrics except the metric of average detection time. Similar to other baselines containing dynamic validation process, RENGAR has no false positive, *i.e.*, the precision is 100%. The recall of RENGAR is also 100%, while the highest value of nine baselines is around 25%. Note that the metrics are calculated based on the collected ground truth which may miss unknown kinds of ReDoS vulnerable regexes (See discussion in Section 5). Therefore, the 100%

TABLE 7: **RQ1** Detail Comparison with Hybrid Detectors.

Union of Hybrid Detectors (TP)	RENGAR (Unique TP/TP)	Unique TP Distribution in Static Component Comparison				Unique TP Distribution in Dynamic Component Comparison					
		Total	EOLS	POLS	PTLS	Total	Case 1	Case 2	Case 3	Case 4	Case 5
44,111	131,739/175,850	61,370	6,753 (11.00%)	39,897 (65.01%)	14,720 (23.99%)	21,803	1,527 (7.00%)	2,612 (11.98%)	4,450 (20.41%)	216 (0.99%)	12,998 (59.62%)

TABLE 8: **RQ1** Detail Comparison with Static Detectors.

Union of Static Detectors (TP)	RENGAR-Static (Unique TP/TP)	Unique TP Distribution of Vuln. Patterns		
		EOLS	POLS	PTLS
20,609	155,241/175,850	9,215 (5.94%)	121,273 (78.12%)	24,753 (15.94%)

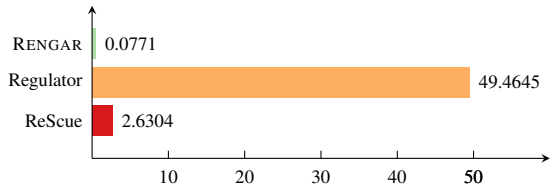

Figure 5: **RQ1** Detail Comparison on Speed with Dynamic Detectors (Seconds).

recall value does not mean that RENGAR is able to catch all kinds of ReDoS vulnerable regexes but represents that its ReDoS detection ability is greater than the union of all baselines. Another remarkable point is that RENGAR detects from 3 to 197 times more ReDoS vulnerable regexes (*i.e.*, TP in table) than the other baselines. Specifically, it detects 175,850 unique ReDoS vulnerable regexes, which detects 131,845 more than the second best tool ReDoSHunter. These comparisons clearly show that the effectiveness of RENGAR is significantly greater than all baselines. For the time cost of ReDoS detection, RENGAR has a longer average detection time than most static detectors. We believe the time cost of RENGAR is reasonable since: ❶ The static approaches do not have dynamic validation cost and will suffer from false positives; ❷ RENGAR shows a clear performance superiority compared with the dynamic and hybrid detectors which have a dynamic validation phase.

Figures 2, 3, and 4 are upset plots illustrating the details of the intersections among the ReDos vulnerable regex sets found by RENGAR and baselines. For easier interpretation of the data, the plots are split based on the detector type of the baselines. These plots have the following common features: ❶ RENGAR is the dominant tool: it not only finds all ReDoS vulnerable regexes but also uniquely finds most regexes; ❷ Without considering RENGAR, there is no dominant baseline. Specifically, excluding RENGAR, every baseline has its uniquely identified vulnerable regexes and the majority of the vulnerable regexes are uniquely identified.

Comparison by Detector Type To understand why and how RENGAR outperforms the baselines, we further collected detailed information and conducted an analysis. For the sake of clarity, the comparisons are separately discussed according to the type of baselines.

For static detectors, we compared and analyzed their ability of vulnerability patterns with RENGAR. Specifically, the static component of RENGAR (denoted as RENGAR-Static) is compared with the union of static detectors, *i.e.*, the union set of their detected vulnerable regex sets. As shown in the first two columns of TABLE 8, RENGAR-Static locates 175,850 potential vulnerable regexes while the union locates 20,609. Note that the former result is a superset of the latter, which shows that the static component of RENGAR can not only cover all ReDoS vulnerable regexes of existing detectors but also find new kinds of vulnerable regexes. Given that there are more than 10 vulnerability patterns in five static detectors while RENGAR only uses three patterns, we conclude that RENGAR has a more general and effective vulnerability modeling. The 3rd to 5th columns of TABLE 8 list the percentage of each vulnerability pattern for TP uniquely identified by RENGAR. Each pattern shows a decent contribution.

Since the existing dynamic detectors are fuzzing tools, whose mechanisms are quite different from RENGAR, we compared and analyzed their performance. As shown in Figure 3, RENGAR has found 149,276 more vulnerable regexes than the union set of dynamic detectors. For the 1,012 vulnerable regexes found by all three tools, we compared their average performance in Figure 5. The reason for the speed superiority is that RENGAR is much more focused on finding the modeled vulnerabilities while fuzzers put too much effort into the random search process.

To understand why RENGAR outperforms hybrid detectors, we compared and analyzed the effectiveness of their static and dynamic components separately. Figure 4 and the first two columns of TABLE 7 show that RENGAR can detect both the TP found by all hybrid detectors (44,111) and unique TP (131,739). The 3rd to 6th columns in TABLE 7 list the static component comparison results. Similar to the comparison with static detectors, we compared the number of TP covered by the static component of RENGAR and the union of hybrid detectors. In total, there are 61,370 TP missed by the static components of existing hybrid detectors. These missed TP can be detected by the EOLS, POLS, and PTLS patterns used in RENGAR. This proves that the vulnerability modeling of RENGAR has better generality. The last six columns of TABLE 7 show the dynamic component comparison results. Note that to fairly evaluate the detection capability of the dynamic component, we compare these tools upon the intersection set of the TP sets detected by static components of all tools. This is because adapting the existing tools to generate attack strings for vulnerable regexes not recognized by their static components can inevitably introduce design/implementation choices made by us. The intersection set has 48,576 regexes, and RENGAR uniquely identifies 21,803 (TABLE 7, seventh column). As shown in the table, all five kinds of disturbances have been

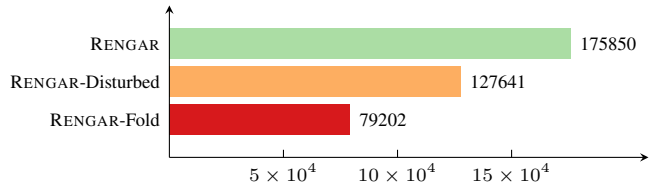

Figure 6: **RQ2** TP Comparison for RENGAR, RENGAR-*Fold*, and RENGAR-*Disturbed*. X-axis is the number of detected TP.

TABLE 9: **RQ2** TP Distribution of Vuln. Patterns.

RENGAR-Fold	RENGAR (Unique TP/TP)	EOLS	POLS	PTLS
79,202	96,648/175,850	9,239 (9.56%)	66,960 (69.28%)	20,449 (21.16%)

found in the missed TP, which shows the necessity of our disturbance free attack string generation strategy.

> **Summary to RQ1:** RENGAR shows its performance superiority by comparing it with nine state-of-the-art ReDoS detectors. Due to its vulnerability modeling and disturbance-free attack string generation techniques, it can outperform other tools in both candidate vulnerability searching and dynamic triggering.

4.3. Ablation Study (RQ2)

To understand the contributions of the techniques proposed for solving these summarized challenges, we performed ablation studies on each component. We implemented two variants of RENGAR: ❶ RENGAR-*Fold*, which replaces LS-Based vulnerability modeling by the merged modelings of ReDoSHunter and Revealer; ❷ RENGAR-*Disturbed*, which disables disturbance free attack string generation strategies, *i.e.*, it generates the attack strings without considering the disturbances. Note that using ReDosHunter and Revealer is enough to represent all hybrid and static detectors. This is because their static modelings already cover the union set of detected TPs from all static detectors. The upset plot in Appendix 7.2 provides the statistical detail.

LS-Based Vulnerability Modeling (C1) RENGAR-*Fold* is used as the baseline to study the effectiveness of our modeling. As shown in Figure 6, the amount of vulnerable regexes found by RENGAR is more than twice the amount (2.22 times) of the regexes found by RENGAR-*Fold*, which shows a significant performance advantage. The reason is that many vulnerable regexes have already been missed by RENGAR-*Fold* in the candidate vulnerability searching phase due to the lack of LS-based vulnerability modeling. To further show the contribution of the modeling, we did statistics of the vulnerability pattern information for the vulnerable regexes that only RENGAR can find. TABLE 9 lists the distribution: the POLS pattern contributes the most (68.12%) while the PTLS and EOLS patterns take the percentages of 21.16% and 9.56% separately. *Conclusively, the LS-based vulnerability modeling significantly contributes to the overall performance of* RENGAR. *Besides, in evaluation, each type of vulnerability pattern has non-negligible effects.*

TABLE 10: **RQ2** TP Distribution of Disturbance Types.

RENGAR -Disturbed	RENGAR (Unique TP/TP)	Case 1	Case 2	Case 3	Case 4	Case 5
127,641	48,209/175,850	3,572 (7.41%)	5,781 (11.99%)	11,888 (24.66%)	474 (0.98%)	26,494 (54.96%)

Disturbance Free Attack String Generation (C2) RENGAR-*Disturbed* is used as the baseline to study the effectiveness of our attack string generation technique. Figure 6 also shows the performance comparison of RENGAR and RENGAR-*Disturbed*. The amount of vulnerable regexes found by RENGAR is 175,850, which is 1.38 times of the amount of regexes found by RENGAR-*Disturbed* (127,641). In other words, even RENGAR-*Disturbed* can locate the same vulnerable candidates as RENGAR, it will falsely classify 48,209 (27.41% to all vulnerable regexes) regexes as non-vulnerable due to the failure of generating effective attack strings. The significant performance differences show the necessity for tackling the disturbance and the effectiveness of our generation strategy. TABLE 10 zooms in on the distribution of the case types of the disturbances for the vulnerable regexes that cannot be triggered by RENGAR-*Disturbed*. The case types listed in descending order are type 5, 3, 2, 1, and 4, whose proportions are 54.96%, 24.66%, 11.99%, 7.41%, and 0.98% correspondingly. *In summary, the disturbance free attack string generation strategy has a significant impact on the overall performance of* RENGAR. *And nearly all types of disturbance cases it tackles take significant portions in our evaluation.*

> **Summary to RQ2:** The evaluation proves not only the overall effectiveness of the two key components of RENGAR but also the significance of each proposed pattern and tackled disturbance case.

4.4. Real-world Application (RQ3)

To demonstrate the practicality of RENGAR, we used RENGAR on real-world projects covering multiple programming languages. Currently, we've applied RENGAR on three languages: Java, Python, and JavaScript. The projects based on the first two languages are already covered by our real-world datasets in previous evaluation. Therefore, according to the previous evaluation results, we picked the projects ranked by popularity (have high Github stars or download count) and vulnerability severity (EOLS > POLS = PTLS), manually verified whether the vulnerable regex is exploitable, reported the vulnerabilities, and helped vendors for the fix. Besides, we also evaluated RENGAR in JavaScript projects which are not included in our real-world datasets. We searched popular and vulnerable projects from NPM[3], and followed a similar process as above.

Our manual validation process has three steps: ❶ Identify whether the vulnerable regexes resides in the main

[3] https://www.npmjs.com/

```
/* Vulnerable regex */
var string = /"(?:\\.|[^"\\\r\n])*"|'(?:\\.|[^'\\\r\n])*'/;
var smartyPattern = RegExp(
      /* Match comments */
      /\{\*[\s\S]*?\*\}/.source + '|' +
      /* Match PHP tags */
      /\{php\}[\s\S]*?\{\/php\}/.source + '|' +
      /* Match smarty blocks */
      /\{(?:[^{}"']|<str>|\{(?:[^{}"']|\
<str>|\{(?:[^{}"']|<str>)*\})*\})*\}/.source
      .replace(/<str>/g,
            function(){ return string.source; }),
      'g');

/* PoC Code */
function trigger_ReDoS_during_code_highlight() {
  /* ... represents repeating \\\" 15000 times */
  \"\\\"\\\"\\\"\\\"...\\\"\\\"\\\"
}
```

Figure 7: A Vulnerable Regex in Package `prismjs`.

functionality code of the project, excluding internal code like testing scripts. ❷ Identify user-controllable input points and analyze whether the input points can reach the vulnerable regex points via data flow taint analysis, using `codeql`. ❸ Construct an exploit PoC in the format of the project input. This is the most complex and time-consuming step since building an exploit for a real-world project requires extensive project-specific domain knowledge.

After four man-months effort, we analyzed 315 real-world projects, identified and reported 69 exploitable vulnerabilities. Among the vulnerabilities, 47 of them were confirmed and fixed by vendors and 21 of them were assigned CVE numbers. Appendix 7.3, TABLE 12 lists the detail of all reported vulnerabilities.

Case Study #5 `prismjs` (6,141 K weekly download count) is a lightweight syntax highlighting package. Its multiple libraries, *e.g.*, `prism-pure`, `prism-ruby`, `prism-q`, etc, contain a common vulnerable regex in the code of the core functions, as shown in Figure 7. First, RENGAR identified that the regex starts with a *start-of-line*-free subregex $r_1 = \beta_0\beta_1$ where $\beta_0 =$ `"` (*resp.* $\beta_0 =$ `'`), $\beta_1 =$ `(?:\\.|[^"\\\r\n])*` (*resp.* $\beta_1 =$ `(?:\\.|[^'\\\r\n])*`) is an LS and $\mathcal{L}(\Sigma^*\beta_0\Sigma^*) \cap \mathcal{L}(\beta_1) = \{\texttt{\textbackslash"}, \ldots\} \neq \emptyset$ (*resp.* $\mathcal{L}(\Sigma^*\beta_0\Sigma^*) \cap \mathcal{L}(\beta_1) = \{\texttt{\textbackslash'}, \ldots\} \neq \emptyset$), which triggers vulnerability pattern POLS. Then, it generated an attack string `"'+'\\"`$\times$15000 (*resp.* `''+'\\'`$\times$15000). Finally, it verified that the matching steps of the attack string exceed the corresponding threshold $T = 10^5$ and reported that the regex is ReDoS-vulnerable. Notably, any one pattern of five static detectors and two hybrid detectors of vulnerabilities cannot capture the vulnerabilities.

Case Study #9 `node-emoji` (3,065 K weekly download count) is a nodejs project for handling simple emoji. The code in its `emoji` library contains a vulnerable regex $r = \vartheta_1|(\varphi_1(\varphi_2|\vartheta_2)\varphi_3)$ where $\vartheta_1 =$ `^[\s\uFEFF\xA0]+`, $\varphi_1 = \vartheta_2 = \epsilon$, $\varphi_2 =$ `[\s\uFEFF\xA0]+`, $\varphi_3 = \epsilon$`$`, as shown in Figure 8. Similarly, RENGAR first statically diagnosed that subregex φ_2 is pathological. If we disregard disturbance subregex ϑ_1, RENGAR generated an attack string `'\t'`$\times$15000 + `'!'` which

```
/* Vulnerable Regex */
var trimSpaceRegex = /^[\s\uFEFF\xA0]+|[\s\uFEFF\xA0]+$/g;
Emoji.replace = function replace (str, ...) {
  var replaced = emoji_related_operations(str);
  /* Code uses vulnerable regex */
  return ... ? replaced.replace(trimSpaceRegex, '')
             : replaced;
};

/* PoC Code */
var emoji = require("node-emoji");
var attack_str ='a'+'\t'.repeat(15000) + '!';
var result = emoji.replace(attack_str, '', true);
```

Figure 8: A Vulnerable Regex in Package `nodejs-tmpl`.

indeed triggers ReDoS for separate regex $(\varphi_1(\varphi_2|\vartheta_2)\varphi_3)$ while cannot trigger ReDoS for complete regex r. The reason is that the prefix string `'\t'`$\times$15000 of the generated string can be matched by the disturbance subregex ϑ_1 in regex r. So it is necessary to take into account the disturbance subregex ϑ_1. Specifically, RENGAR diagnosed $\mathcal{L}(\vartheta_1\Sigma^*) \cap \mathcal{L}(\varphi_1\varphi_2) = \{\texttt{\textbackslash t}, \ldots\} \neq \emptyset$ (*i.e.*, ϑ_1 disturbs $\varphi_1\varphi_2$), which satisfies disturbance case ❷. Therefore, RENGAR added the increased constraint $xy^n \notin \mathcal{L}(\vartheta_1\Sigma^*)$, and generated an attack string `'a'` + `'\t'`$\times$15000 + `'!'` that can trigger ReDoS for regex r.

> **Summary to RQ3:** RENGAR can be used for finding vulnerabilities in real-world projects. Benefiting from its vulnerability modeling and attack string generation techniques, it can find more kinds of vulnerabilities.

5. Threats to Validity

External Validity The effectiveness of ReDoS detection is evaluated on seven datasets, which may not be able to fully characterize the effectiveness of these approaches. In that case, the evaluation may exist bias. To alleviate the potential bias, we select the datasets as follows: Four of the datasets are the commonly used datasets in existing ReDoS detection works. The rest three datasets are large-scale real-world regexes sets covering three different languages, which are Java, Python, and CSharp. Their regexes are extracted from all public projects of popular package managers in that language. At the moment of submission, we have already built a real-world dataset of JavaScript by scraping popular projects from NPM. However, the evaluation of this dataset is too costful to be finished before submission. The JavaScript dataset contains 431,500 regexes and the evaluation of some baselines requires huge CPU resources. For example, according to our estimation, simply applying detector regulator [31] to the whole dataset can cost approximately 4.6 CPU years. We will evaluate RENGAR on more languages and update the results on our website [37].

Internal Validity The build process of the ground truth data may miss labeling some vulnerable regexes. Since there are more than half a million of regexes in all datasets, it is unrealistic to manually investigate whether they are

ReDoS vulnerable one by one. Therefore, we collected the ground truth by first adopting all ReDoS detectors covered in RQ1 to locate potentially vulnerable regexes, then manually verifying the regexes which receive discrepant results from the detectors. If there are vulnerable regexes that cannot be detected by any detector, they will be optimistically labeled as benign. This may cause the recall values of all detectors in evaluation higher than their real value.

6. Related Work

6.1. ReDoS Detection

Static Analysis RXXR2 [24], [38], extended from RXXR [23], is a static analysis tool which detects ReDoS vulnerabilities by pumping analysis. However, it is incapable to detect regexes with polynomial ReDoS vulnerabilities, and does not support extensions such as lookarounds, and backreferences. Rexploiter [25] detects ReDoS vulnerable regexes by adversarial automata construction, and it can identify polynomial ReDoS vulnerabilities. Similarly, Weideman et al. [27] detect ReDoS vulnerable regexes by NFA ambiguity analysis, and can also identify polynomial ReDoS vulnerabilities. But their work does not support most of the extensions (*e.g.*, lookarounds, backreferences, and non-capturing groups). All these static analysis methods have high false positives partly because their patterns are localized and they cannot verify the attack strings they generated.
Dynamic Analysis SDLFuzzer [28], [29] identifies ReDoS vulnerabilities by testing the matching time of regexes against a range of randomly-generated strings. Yet, it does not support most of extensions (*e.g.*, lookarounds and backreferences), making it less capable. Instead of generating random strings, ReScue [30] is designed for searching time-consuming strings. Due to the enormous string search space, it can only identify exponential or higher polynomial ReDoS vulnerabilities but is not able to detect lower polynomial or deeply hided ReDoS. Besides, the effectiveness of the genetic searching approach relies on the initialization, and is likely to trap in a local optimum, leaving the results unstable from one to the other iterations. Regulator [31] detects ReDoS vulnerabilities based on fuzzing techniques, which instruments a backtracking regex engine and implements a novel mutation strategy. However, it may cost too much time for generating attack strings. Moreover, these dynamic-based approaches usually output extremely long and randomly-generated attack strings which can hardly provide clues for the following ReDoS repair.
Hybrid approaches Revealer [32] takes a hybrid approach, which combines static and dynamic analysis, by first using an extended NFA which can support more extensions, then dynamically generating attack strings by simulating the matching process of an extended regex. Hybrid detector ReDoSHunter [33] is driven by five vulnerability patterns and uses transformations to support more extensions. However, as shown by our experiments, both of them are less powerful than RENGAR mainly due to their limitations in vulnerability modeling and attack string generation techniques.

6.2. ReDoS Prevention or Alleviation

Some works [39]–[41] try to find equivalent/approximate ReDoS-invulnerable regexes to replace the ReDoS vulnerable ones. Among them, Van der Merwe et al. [39] and Cody-Kenny et al. [40] devote to finding equivalent ReDoS-invulnerable regexes to replace the original ones. However, the exact equivalence is too strong to use in practice [5], [30], which limits their usage in real-world applications. Li et al. [41] addressed this problem by deducing anti-ReDoS regexes adopting programming-by-example algorithms. Yet the quality of anti-ReDoS regex deduced by them highly depends on the quality of user-provided examples. Based on ReDoSHunter [33], Li et al. [42] recently introduced a vulnerability analysis and repair framework benefiting from their innovative vulnerability patterns and repair patterns.

Alternatively, ReDoS attacks can also be alleviated by regex matching speedup in some special cases, *e.g.*, by parallel algorithms [43], GPU-based algorithms [44], state-merging algorithms [45], Parsing Expression Grammars (PEGs) [46]–[48], counting automata matching algorithm [49], memoization-based optimization [50] and recursion-limit/backtracking-limit/time-limit [51]–[53].

7. Conclusion

In this paper, we proposed RENGAR, a ReDoS vulnerability detector with LS-based vulnerability modeling and disturbance free attack string generation strategy. Compared with existing modelings, the modeling of RENGAR describes a broader scope of ReDoS vulnerabilities concisely. The disturbance free attack string generation strategy solves the problem that the generated attack string can be invalidated due to the disturbance among subregexes. In evaluation, RENGAR shows a clear advantage compared with nine state-of-the-art tools. It detects not only all vulnerable regexes found by these tools but also 3 – 197 times more vulnerable regexes. It is also the fastest detector compared with the detectors which contain a dynamic validation process. Using RENGAR, we have found 69 vulnerabilities in popular real-world projects including 21 CVEs.

Acknowledgment

We sincerely thank the anonymous reviewers and shepherd for their valuable comments. This work is supported in part by National Key R&D Program of China (2022YFB3103900), Strategic Priority Research Program of the CAS (XDC02030200), Chinese National Natural Science Foundation (62032010, 62202462, 61972260, and 61836005), the Special Research Assistant Program of the CAS, the Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN), NRF Investigatorship NRF-NRFI06-2020-0001, and the National Research Foundation through its National Satellite of Excellence in Trustworthy Software Systems (NSOE-TSS) project under the National Cybersecurity R&D (NCR) Grant award no. NRF2018NCR-NSOE003-0001.

References

[1] J. E. F. Friedl, *Mastering Regular Expressions - Understand Your Data and Be More Productive: for Perl, PHP, Java, .NET, Ruby, and More (3. ed.)*. O'Reilly, 2006. [Online]. Available: http://www.oreilly.de/catalog/regex3/index.html

[2] L. G. M. IV, J. Donohue, J. C. Davis, D. Lee, and F. Servant, "Regexes are Hard: Decision-making, Difficulties, and Risks in Programming Regular Expressions," in *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, 2019, pp. 415–426. [Online]. Available: https://doi.org/10.1109/ASE.2019.00047

[3] C. Chapman, P. Wang, and K. T. Stolee, "Exploring Regular Expression Comprehension," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. IEEE Computer Society, 2017, pp. 405–416. [Online]. Available: https://doi.org/10.1109/ASE.2017.8115653

[4] A. Bartoli, A. D. Lorenzo, E. Medvet, and F. Tarlao, "Inference of Regular Expressions for Text Extraction from Examples," *IEEE Trans. Knowl. Data Eng.*, vol. 28, no. 5, pp. 1217–1230, 2016. [Online]. Available: https://doi.org/10.1109/TKDE.2016.2515587

[5] 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 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. ACM, 2018, pp. 246–256. [Online]. Available: https://doi.org/10.1145/3236024.3236027

[6] J. C. Davis, L. G. M. IV, C. A. Coghlan, F. Servant, and D. Lee, "Why Aren't Regular Expressions a Lingua Franca? An Empirical Study on the Re-use and Portability of Regular Expressions," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, 2019, pp. 443–454. [Online]. Available: https://doi.org/10.1145/3338906.3338909

[7] J. Goyvaerts, "Runaway Regular Expressions: Catastrophic Backtracking," 2020, https://www.regular-expressions.info/catastrophic.html.

[8] A. Weidman, "Regular Expression Denial of Service - ReDoS," 2017, https://owasp.org/www-community/attacks/Regular_expression_Denial_of_Service_-_ReDoS.

[9] T. Kadlec, "Regular Expression Denial of Service (ReDoS) and Catastrophic Backtracking," 2017, https://snyk.io/blog/redos-and-catastrophic-backtracking/.

[10] W. Blair, A. Mambretti, S. Arshad, M. Weissbacher, W. Robertson, E. Kirda, and M. Egele, "HotFuzz: Discovering Algorithmic Denial-of-Service Vulnerabilities through Guided Micro-Fuzzing," in *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*, 2020. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/hotfuzz-discovering-algorithmic-denial-of-service-vulnerabilities-through-guided-micro-fuzzing/

[11] X. Cai, Y. Gui, and R. Johnson, "Exploiting Unix File-System Races via Algorithmic Complexity Attacks," in *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*, 2009, pp. 27–41. [Online]. Available: https://doi.org/10.1109/SP.2009.10

[12] R. M. Chang, G. Jiang, F. Ivancic, S. Sankaranarayanan, and V. Shmatikov, "Inputs of Coma: Static Detection of Denial-of-Service Vulnerabilities," in *Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF 2009, Port Jefferson, New York, USA, July 8-10, 2009*, 2009, pp. 186–199. [Online]. Available: https://doi.org/10.1109/CSF.2009.13

[13] S. A. Crosby and D. S. Wallach, "Denial of Service via Algorithmic Complexity Attacks," in *Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA, August 4-8, 2003*, 2003, pp. 29–44. [Online]. Available: https://www.usenix.org/conference/12th-usenix-security-symposium/denial-service-algorithmic-complexity-attacks

[14] K. S. Luckow, R. Kersten, and C. S. Pasareanu, "Symbolic Complexity Analysis using Context-preserving Histories," in *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*, 2017, pp. 58–68. [Online]. Available: https://doi.org/10.1109/ICST.2017.13

[15] 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, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, 2017, pp. 2155–2168.

[16] R. Smith, C. Estan, and S. Jha, "Backtracking Algorithmic Complexity Attacks against a NIDS," in *22nd Annual Computer Security Applications Conference (ACSAC 2006), 11-15 December 2006, Miami Beach, Florida, USA*, 2006, pp. 89–98. [Online]. Available: https://doi.org/10.1109/ACSAC.2006.17

[17] J. Wei, J. Chen, Y. Feng, K. Ferles, and I. Dillig, "Singularity: Pattern Fuzzing for Worst Case Complexity," in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, 2018, pp. 213–223. [Online]. Available: https://doi.org/10.1145/3236024.3236039

[18] C. Lemieux, R. Padhye, K. Sen, and D. Song, "PerfFuzz: Automatically Generating Pathological Inputs," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, 2018, pp. 254–265. [Online]. Available: https://doi.org/10.1145/3213846.3213874

[19] J. Burnim, S. Juvekar, and K. Sen, "WISE: Automated Test Generation for Worst-case Complexity," in *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, 2009, pp. 463–473. [Online]. Available: https://doi.org/10.1109/ICSE.2009.5070545

[20] K. S. Luckow, R. Kersten, and C. S. Pasareanu, "Complexity Vulnerability Analysis using Symbolic Execution," *Softw. Test. Verification Reliab.*, vol. 30, no. 7-8, 2020.

[21] Y. Noller, R. Kersten, and C. S. Pasareanu, "Badger: Complexity Analysis with Fuzzing and Symbolic Execution," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, 2018, pp. 322–332.

[22] C. Staicu and M. Pradel, "Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers," in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, 2018, pp. 361–376.

[23] J. Kirrage, A. Rathnayake, and H. Thielecke, "Static Analysis for Regular Expression Denial-of-Service Attacks," in *Network and System Security - 7th International Conference, NSS 2013, Madrid, Spain, June 3-4, 2013. Proceedings*, 2013, pp. 135–148.

[24] A. Rathnayake and H. Thielecke, "Static Analysis for Regular Expression Exponential Runtime via Substructural Logics," *CoRR*, vol. abs/1405.7058, 2014.

[25] V. Wüstholz, O. Olivo, M. J. H. Heule, and I. Dillig, "Static Detection of DoS Vulnerabilities in Programs that Use Regular Expressions," in *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II*, 2017, pp. 3–20.

[26] D. LLC, "Regexploit: DoS-able Regular Expressions," 2021, https://github.com/doyensec/regexploit.

[27] N. Weideman, B. van der Merwe, M. Berglund, and B. W. Watson, "Analyzing Matching Time Behavior of Backtracking Regular Expression Matchers by using Ambiguity of NFA," in *Implementation and Application of Automata - 21st International Conference, CIAA 2016, Seoul, South Korea, July 19-22, 2016, Proceedings*, 2016, pp. 322–334.

[28] B. Sullivan, "Regular Expression Denial of Service Attacks and Defenses," 2010, https://docs.microsoft.com/en-us/archive/msdn-magazine/2010/may/security-briefs-regular-expression-denial-of-service-attacks-and-defenses.

[29] ——, "New Tool: SDL Regex Fuzzer," 2010, http://cloudblogs.microsoft.com/microsoftsecure/2010/10/12/new-tool-sdl-regex-fuzzer.

[30] 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, ASE 2018, Montpellier, France, September 3-7, 2018*, 2018, pp. 225–235.

[31] R. McLaughlin, F. Pagani, N. Spahn, C. Kruegel, and G. Vigna, "Regulator: Dynamic Analysis to Detect ReDoS," in *31th USENIX Security Symposium, USENIX Security 2022, August 10–12, 2022*. USENIX Association, 2022, pp. 4219–4235.

[32] Y. Liu, M. Zhang, and W. Meng, "Revealer: Detecting and Exploiting Regular Expression Denial-of-Service Vulnerabilities," in *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 2021, pp. 1468–1484.

[33] Y. Li, Z. Chen, J. Cao, Z. Xu, Q. Peng, H. Chen, L. Chen, and S. Cheung, "ReDoSHunter: A Combined Static and Dynamic Approach for Regular Expression DoS Detection," in *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. USENIX Association, 2021, pp. 3847–3864.

[34] "Rengar Open Source," https://sites.google.com/view/rengar/resources.

[35] T. Parrm, "ANTLR," 2022, https://www.antlr.org.

[36] L. M. de Moura and N. S. Bjørner, "Z3: An Efficient SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, ser. Lecture Notes in Computer Science, vol. 4963. Springer, 2008, pp. 337–340. [Online]. Available: https://doi.org/10.1007/978-3-540-78800-3_24

[37] "Rengar Website," https://sites.google.com/view/rengar.

[38] A. Rathnayake, "Semantics, Analysis And Security Of Backtracking Regular Expression Matchers," Ph.D. dissertation, University of Birmingham, UK, 2015.

[39] B. van der Merwe, N. Weideman, and M. Berglund, "Turning Evil Regexes Harmless," in *Proceedings of the South African Institute of Computer Scientists and Information Technologists, SAICSIT 2017, Thaba Nchu, South Africa, September 26-28, 2017*, 2017, pp. 38:1–38:10.

[40] B. Cody-Kenny, M. Fenton, A. Ronayne, E. Considine, T. McGuire, and M. O'Neill, "A Search for Improved Performance in Regular Expressions," in *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2017, Berlin, Germany, July 15-19, 2017*, 2017, pp. 1280–1287.

[41] Y. Li, Z. Xu, J. Cao, H. Chen, T. Ge, S. Cheung, and H. Zhao, "FlashRegex: Deducing Anti-ReDoS Regexes from Examples," in *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*, 2020, pp. 659–671.

[42] Y. Li, Y. Sun, Z. Xu, J. Cao, Y. Li, R. Li, H. Chen, S. 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 2022, Boston, MA, USA, August 10-12, 2022*, K. R. B. Butler and K. Thomas, Eds. USENIX Association, 2022, pp. 4183–4200. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/li-yeting

[43] C. Lin, C. Liu, and S. Chang, "Accelerating Regular Expression Matching using Hierarchical Parallel Machines On GPU," in *Proceedings of the Global Communications Conference, GLOBECOM 2011, 5-9 December 2011, Houston, Texas, USA*. IEEE, 2011, pp. 1–5.

[44] X. Yu and M. Becchi, "GPU Acceleration of Regular Expression Matching for Large Datasets: Exploring the Implementation Space," in *Computing Frontiers Conference, CF'13, Ischia, Italy, May 14 - 16, 2013*. ACM, 2013, pp. 18:1–18:10.

[45] M. Becchi and S. Cadambi, "Memory-efficient Regular Expression Search using State Merging," in *INFOCOM 2007. 26th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 6-12 May 2007, Anchorage, Alaska, USA*. IEEE, 2007, pp. 1064–1072.

[46] B. Ford, "Parsing Expression Grammars: A Recognition-based Syntactic Foundation," in *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, 2004, pp. 111–122.

[47] S. Medeiros, F. Mascarenhas, and R. Ierusalimschy, "From Regexes to Parsing Expression Grammars," *Sci. Comput. Program.*, vol. 93, pp. 3–18, 2014.

[48] IBM, "Rosie Pattern Language (RPL)," 2020, https://rosie-lang.org/.

[49] L. Turonová, L. Holík, O. Lengál, O. Saarikivi, M. Veanes, and T. Vojnar, "Regex Matching with Counting-set Automata," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 218:1–218:30, 2020. [Online]. Available: https://doi.org/10.1145/3428286

[50] J. C. Davis, F. Servant, and D. Lee, "Using Selective Memoization to Defeat Regular Expression Denial of Service (ReDoS)," in *2021 IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, May 23-27, 2021*, 2021, p. To appear.

[51] Microsoft, "Regex class - C#," 2020, https://docs.microsoft.com/en-us/dotnet/api/system.text.regularexpressions.regex?view=net-5.0.

[52] PHP, "PHP: preg_match - Manual," 2020, https://www.php.net/manual/en/function.preg-match.php.

[53] PCRE, "PCRE - Perl Compatible Regular Expressions," 2020, https://pcre.org/.

Appendix

7.1. Detailed Evaluation Dataset

TABLE 11 presents the detailed statistics of evaluation datasets.

TABLE 11: Statistics of Used Datasets.

Name	#Regex	Description
Corpus	13,597	Regexes from scraped 3,898 Python projects [3]
RegExLib	8,699	Online regexes from regexlib.com [33]
Snort	15,355	Regexes used in the Snort NIDS [33]
Regex101	10,660	Online regexes from regex101.com
PyPI	118,104	Regexes from scraped 279,266 Python projects
Maven	136,643	Regexes from scraped 271,839 Java projects
NuGet	49,954	Regexes from scraped 117,561 CSharp projects
Total:	353,012	

7.2. Statistical Detail for RQ2 Baseline Selection

Figure 9 compares the detected candidate vulnerable regexes between the union of static components of ReDoSHunter and Revealer with the union set of all static detectors. The result shows that the former can cover all detected candidates of the latter. Therefore, using the former as the baseline in RQ2 is enough.

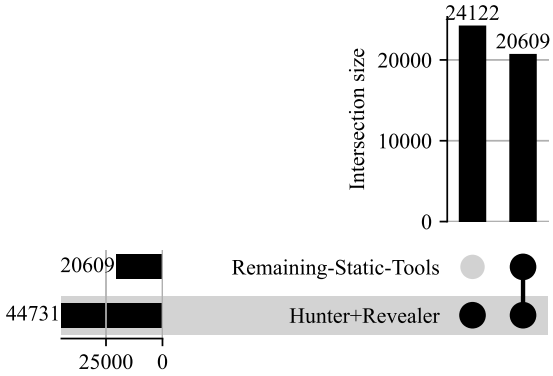

Figure 9: UpSet Plot for Static Component of Remaining Static Tools and The Combination of ReDoSHunter and Revealer.

7.3. Detail List of Reported Vulnerabilities

TABLE 12 (in the last page) lists the detailed status of all the reported vulnerabilities.

TABLE 12: Zero-Day Vulnerabilities Detected by RENGAR.

No.	Project	Weekly Downloads	Status	#Vulnerable ReDoS Regex	#Exploitable ReDoS Vulnerability	#Vulnerability Confirmed by Developers	CVE-ID
#1	minimatch	68,439 K	Confirmed & Fixed	1	1	1	CVE-2022-3517
#2	clean-css	12,803 K	Confirmed & Fixed	1	1	1	–
#3	pillow	12,664 K	Confirmed & Fixed	2	2	2	CVE-2021-23437
#4	Color-String	11,798 K	Confirmed & Fixed	3	2	2	CVE-2021-29060
#5	prismjs	6,141 K	Confirmed & Fixed	14	1	1	CVE-2021-3801
#6	mocha	5,922 K	Confirmed & Fixed	1	1	1	–
#7	js-base64	4,180 K	Confirmed & Fixed	1	1	1	–
#8	d3-color	4,039 K	Confirmed & Fixed	1	1	1	–
#9	node-emoji	3,065 K	Confirmed & Fixed	1	1	1	–
#10	pylint	3,045 K	Confirmed & Fixed	2	1	1	–
#11	stylelint	3,002 K	Confirmed & Fixed	1	1	1	–
#12	semver-regex	2,897 K	Confirmed & Fixed	1	1	1	CVE-2021-3795
#13	parse-url	2,695 K	Confirmed & Fixed	1	1	1	–
#14	IS-SVG	2,333 K	Confirmed & Fixed	3	3	3	CVE-2021-29059
#15	sphinx	2,299 K	Awaiting reply	4	1	–	–
#16	cron-parser	2,144 K	Awaiting reply	1	1	–	–
#17	async-validator	977 K	Confirmed & Fixed	1	1	1	–
#18	pdfmake	823 K	Awaiting reply	3	1	–	–
#19	configobj	691 K	Confirmed	1	1	1	–
#20	python-markdown2	670 K	Confirmed & Fixed	4	2	2	–
#21	jspdf	636 K	Confirmed & Fixed	4	1	1	–
#22	python-can	240 K	Confirmed	1	1	1	–
#23	GeoJSON.js	180 K	Awaiting reply	1	1	–	–
#24	json	136 K	Awaiting reply	3	1	–	–
#25	locutus	134 K	Confirmed & Fixed	3	1	1	CVE-2021-23392
#26	tap-mocha-reporter	124 K	Confirmed & Fixed	5	1	1	–
#27	fastest-validator	85 K	Awaiting reply	2	1	–	–
#28	zxcvbn	65 K	Confirmed	1	1	1	–
#29	Typo.js	55 K	Confirmed & Fixed	1	1	1	–
#30	uslug	34 K	Confirmed & Fixed	1	1	1	–
#31	terminal-kit	33 K	Confirmed & Fixed	2	1	1	–
#32	is-email	24 K	Confirmed & Fixed	1	1	1	CVE-2021-36716
#33	validator.js	18 K	Confirmed & Fixed	2	2	2	CVE-2021-3765
#34	validate-color	17 K	Confirmed & Fixed	1	1	1	CVE-2021-40892
#35	baron	9 K	Awaiting reply	1	1	–	–
#36	licia	5 K	Confirmed & Fixed	6	2	1	–
#37	markdown-to-json	1 K	Awaiting reply	1	1	–	–
#38	nfcpy	1 K	Awaiting reply	1	1	–	–
#39	scniro-validator	< 1 K	Confirmed & Fixed	1	1	1	CVE-2021-40901
#40	regexfn	< 1 K	Confirmed & Fixed	1	1	1	CVE-2021-40900
#41	repo-git-downloader	< 1 K	Confirmed & Fixed	7	5	1	CVE-2021-40899
#42	scaffold-helper	< 1 K	Confirmed & Fixed	2	1	1	CVE-2021-40898
#43	split-html-to-chars	< 1 K	Confirmed & Fixed	1	1	1	CVE-2021-40897
#44	todo-regex	< 1 K	Confirmed & Fixed	1	1	1	CVE-2021-40895
#45	validate-data	< 1 K	Confirmed & Fixed	1	1	1	CVE-2021-40893
#46	is-it-check	< 1 K	Confirmed & Fixed	10	1	1	–
#47	statebus	< 1 K	Awaiting reply	4	1	–	–
#48	HamlPy	< 1 K	Awaiting reply	3	1	–	–
#49	plan	< 1 K	Awaiting reply	1	1	–	–
#50	orgparse	< 1 K	Awaiting reply	3	1	–	–
#51	ford	< 1 K	Awaiting reply	5	1	–	–
#52	node-unfluff	< 1 K	Awaiting reply	4	2	–	–
#53	that-value	–	Confirmed & Fixed	1	1	1	CVE-2021-40896
#54	underscore-99xp	–	Confirmed & Fixed	1	1	1	CVE-2021-40894
#55	Vfsjfilechooser2	–	Confirmed & Fixed	1	1	1	CVE-2021-29061
#56	Prototype	–	Confirmed & Fixed	3	1	1	CVE-2020-27511
#57	autoconf-archive	–	Awaiting reply	1	1	–	–
			Total	135	69	47	