ReScue: Crafting Regular Expression DoS Attacks*

Yuju Shen
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, China
denny.syj@hotmail.com

Yanyan Jiang
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, China
jyy@nju.edu.cn

Chang Xu
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, China
changxu@nju.edu.cn

Ping Yu
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, China
yuping@nju.edu.cn

Xiaoxing Ma
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, China
xxm@nju.edu.cn

Jian Lu
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, China
lj@nju.edu.cn

ABSTRACT

Regular expression (regex) with modern extensions is one of the most popular string processing tools. However, poorly-designed regexes can yield exponentially many matching steps, and lead to regex Denial-of-Service (ReDoS) attacks under well-conceived string inputs. This paper presents ReScue, a three-phase gray-box analytical technique, to automatically generate ReDoS strings to highlight vulnerabilities of given regexes. ReScue systematically seeds (by a genetic search), incubates (by another genetic search), and finally pumps (by a regex-dedicated algorithm) for generating strings with maximized search time. We implemenmted the ReScue tool and evaluated it against 29,088 practical regexes in real-world projects. The evaluation results show that ReScue found 49% more attack strings compared with the best existing technique, and applying ReScue to popular GitHub projects discovered ten previously unknown ReDoS vulnerabilities.

CCS CONCEPTS

• **Security and privacy → Software and application security**; *Domain-specific security and privacy architectures*; • **Software and its engineering → Search-based software engineering**; *Software defect analysis*; *Software testing and debugging*;

KEYWORDS

Regular expression, denial of service, ReDoS, genetic algorithm

ACM Reference Format:
Yuju Shen, Yanyan Jiang, Chang Xu, Ping Yu, Xiaoxing Ma, and Jian Lu. 2018. ReScue: Crafting Regular Expression DoS Attacks. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3238147.3238159

*Yanyan Jiang and Chang Xu are the corresponding authors.

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3238159

1 INTRODUCTION

Regular expression (regex), a mini-program for string pattern matching, extraction, and replacing, has been one of the most popular string processing tools. Featured with various grammatical extensions (e.g., lookarounds, named groups, atomic groups, backreferences, conditionals, and possessive quantifiers [16]), regexes are widely used in crawlers, text editors, web applications, search engines, databases, command-line utilities, to name but a few.

However, poorly-designed regexes can yield exponentially many matching steps [19], leading to timeout consequences, despite that a regex's matching usually terminates in polynomial or even linear time of its concerned pattern and given input's (a string's) length. Such timeout-triggering inputs can easily result in algorithmic denial-of-service (DoS) attacks, aka. ReDoS attacks [13, 25]. A recent report [40] showed that hundreds of popular websites are threatened by ReDoS, and it is thus natural and necessary to validate regexes against such possible vulnerabilities.

This paper presents ReScue, a three-phase gray-box analytical technique, for *automatically detecting ReDoS vulnerabilities*, i.e., finding a timeout-triggering input string for a given regex. With such a tool, developers or third-party software quality teams can perform ReDoS security checks early in the software development.

The difficulty of automated ReDoS detection is mainly attributed to the complexity of modern regex *language extensions*. A formal language of modern regexes is beyond the description range of a Thompson NFA [6, 8, 44]. Transforming such regexes to automata is already a challenge [3], and hence existing *static* analyzers (e.g., pumping analysis [25, 34, 35], transducer analysis [41], NFA ambiguity analysis [45], or adversarial automata construction [48]) are neither sound nor complete in handling these extensions. On the other hand, state-of-the-art *dynamic* analyzers (e.g., black-box fuzzers like SlowFuzz [31]) lack the understanding to the actual regex matching procedure and commonly fall short when ReDoS vulnerability is hidden in deep states of a regex's matching.

This paper tackles the ReDoS detection problem by observing that the most practical way to implement a regex engine is to adopt the *e-NFA + backtracking searching* approach[1] (*e*-NFA stands for extended non-deterministic finite-state automata), and that regexes can thus be effectively analyzed by merely *focusing on e-NFA states*.

[1] Popular regex engines are mostly implemented in this way, e.g., the built-in regex modules in Java [3], Python [15, 27], JavaScript [11], etc.

In particular, an *e*-NFA state and its corresponding recursion stack distinctly determine an intermediate state in the regex-matching process, and the whole search time in the matching is proportional to the number of *e*-NFA states ever traversed (*matching steps*) during the matching process [16]. We observe that to facilitate a successful ReDoS attack, reaching a particular *state* in an *e*-NFA is of the most importance, and that such state information is sufficient for guiding an effective ReDoS search. With such observations, the recursion stack (which is engine-dependent and difficult to encode) can be safely removed from consideration in the ReDoS detection, and targeted ReDoS strings can still be effectively generated by a genetic algorithm focusing merely on search profiling data (thus applicable to almost all major regex engines with only marginal non-intrusive changes).

This motivates for our proposal of a *gray-box* search-based algorithm for systematically finding an input string that maximizes the number of matching steps using regex search profiling data, as the following three phases explain:

- The *seeding phase* takes a regex (and its *e*-NFA) as input and uses a genetic (seeding) algorithm to search for a diverse set of seed strings that can best cover the *e*-NFA's states. We adopt existing search-based techniques [9, 20, 21, 29] and leverage a new genetic representation by maintaining a string's *effective prefix* and *redundant suffix* to better characterize the string's matching procedure.
- The *incubating phase* germinates the seed strings using a similar genetic algorithm (but with a different fitness function) to search for those attack string candidates, which maximize the ratios between the matching steps so far and their string lengths. This incubating phase resembles existing gray-box search techniques [24].
- The *pumping phase* trims and enhances the slowest string candidate in the incubated population by a local search algorithm for eventually crafting a malicious input string that will result in a significant number of matching steps in the given regex's matching, triggering a real ReDoS attack in practice.

We implemented our three-phase technique as ReScue in Java, a ReDoS detection tool that can thoroughly analyze given regexes for their ReDoS vulnerabilities, requiring merely slight profiling changes to the concerned regex engines. We evaluated ReScue using 29,088 regexes and compared ReScue against four state-of-the-art ReDoS detection techniques. We also applied ReScue to highly starred GitHub projects for ReDoS vulnerability detection.

The evaluation results show that ReScue found 49% more ReDoS strings compared with the best existing technique, and has a comparable time cost with existing native C implementation of fuzzing [31], even if our profiled Java regex engine is one or two magnitudes slower. The application to GitHub projects found ten previously unknown ReDoS vulnerabilities.

In summary, this paper made the following contributions:

- We proposed a three-phase technique for automatic analysis of regexes and systematic detection of their ReDoS vulnerabilities. The technique can be applied to all mainstream regex engines based on *e*-NFA and backtracking, to the best of our knowledge.
- We implemented ReScue in Java and evaluated it using real-world regexes. The evaluation results show that ReScue outperformed state-of-the-art techniques and detected previously unknown ReDoS vulnerabilities in popular open-source projects.

The rest of the paper is organized as follows. We provide background of regex and ReDoS in Section 2 followed by the algorithm description in Section 3. The implementation of the ReScue tool is discussed in Section 4 and its evaluation results are presented in Section 5. Related work is discussed in Section 6 and finally Section 7 concludes the paper.

2 BACKGROUND

2.1 Regexes, Regex Engines, and ReDoS

Regular expression (regex), a mini-program for compact description of formal languages [6, 8], was originated in 1950s [26] and becomes one of the most popular string processing tools. In the years of development, practitioners extended regexes with rich features to facilitate powerful and elegant handling of string pattern matching tasks: character classes, lookarounds, named groups, atomic groups, backreferences, etc. Empowered with these extensions, regex today is one of the most handy string processing tools.

Regex pattern matching is conducted with the support of a *regex engine*. Since regexes with extensions are *context-aware*[2], a modern regex could no longer be converted to a deterministic or non-deterministic finite-state automaton (as opposed to the classical textbook case [8]). As a result, regex engines must have to *search* for regex matches, and thus the state-of-the-art regex engines adopt a similar algorithmic framework as follows:

(1) Converting the given regex into a data structure which extends a non-deterministic finite automaton (i.e., an *e*-NFA), in which each vertex represents a pattern matching related operation, and
(2) Conducting a backtracking search to find whether there exists any path that matches the given input string.

Such a framework runs in polynomial (or even linear) time in most cases. However, in rare cases of a *non-match*, all possible paths would be examined and a regex engine can run in *exponential time* of the input string's length. For example, a regex from RegLib [36]:

```
^<\!\-\-(.*)+(\/){0,1}\-\->$
```

recognizes comments in an HTML file. Pattern matching with such a particular regex is usually fast even if the input is of a megabyte length. However, a Python regex engine can get stuck with the following *short* input string[3]:

```
<!--!--!--!--!--!--!--!--IamReDoS
```

Therefore, when regexes are used to process untrusted inputs (e.g., forms submitted by an Internet user), such time-out-triggering cases can make software vulnerable to Denial-of-Service attacks, or *ReDoS* attacks.

[2]For example, the regex `(?!^1?$|^(11+)\1+$)` tests whether the expression consists of a prime number of 1s.
[3]The developer misplaced a `(.*)+` in the regex, and it can be expanded in $O(2^n)$ many ways in representing a string of length n.

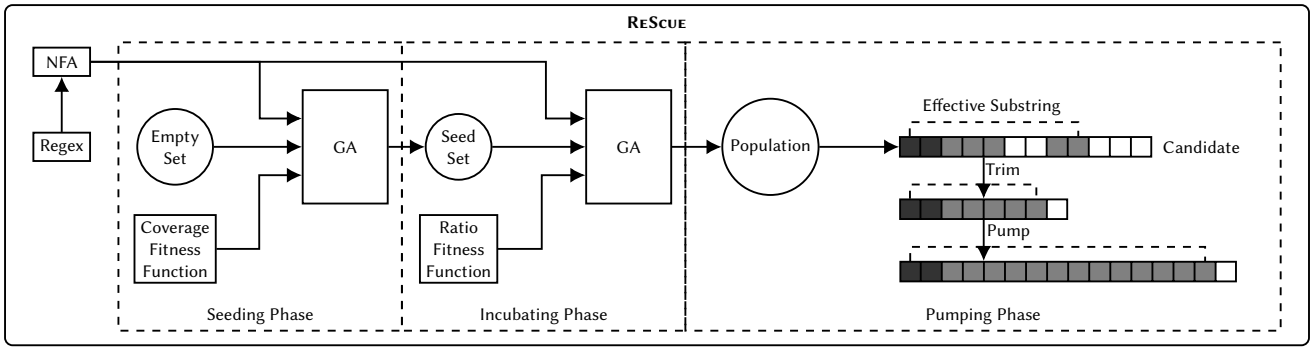

Figure 1: Overview of the ReScue technique for automated ReDoS string generation.

2.2 Automated Detection of ReDoS Vulnerabilities

Regexes are usually embedded in a piece of large software and lack thorough testing or analysis. They are thus *not* defended against ReDoS attacks [40]. A few regex engines provide matching limits (e.g., PCRE [22]) to alleviate the impact of ReDoS. However, even such a minimal effort is not widely adopted in practice[4], not to mention that setting a time limit cannot fully tackle the ReDoS issue when a server is flooded with timeout-triggering regexes.

Therefore, an automated tool for finding ReDoS strings becomes a desirable solution to warn developers of such potential vulnerabilities (and then refactoring the concerned regexes) early in the software development [40]. Depending on whether actual regex matchings are conducted, such a tool is either *static* (based purely on the given regex) or *dynamic* (based on the profiling results of regex matchings).

Static ReDoS analyses find vulnerable constructs (via static rules or patterns) in the e-NFA representation of a regex [25, 34, 35, 41, 45, 48]. Such analyses can be sound and complete for certain kinds of regexes, but can easily report false positive attack strings or miss ReDoS vulnerabilities. Taking the backreference as an example, it is extremely difficult to know what its value can be in a backreference (e.g., \1) without actually conducting the regex matching.

Dynamic ReDoS analyses, on the other hand, conduct actual regex matching and use the matching results to guide further generation of potentially vulnerable strings, e.g., fuzzing [31]. Therefore, a dynamic analyzer usually seamlessly handles regex extensions and reports only true positive ReDoS strings. The major drawback of dynamic analysis is that finding a timeout-triggering string is itself time-consuming, and thus existing black-box techniques guided by a cost-effectiveness value can miss ReDoS vulnerabilities that require a certain cost-ineffective prefix to trigger.

Our ReScue presented in this paper exploits both the static knowledge of a regex (the pumpable property of regexes) and the genetic search (guided by the profiling data collected along with the regex's matching process).

[4]For example, even popular editors like Atom and LibreOffice hang or crash when regex matching gets timeout, e.g., conducting pattern matching of `(.+)+#` by the time of this paper being written.

3 THE RESCUE ALGORITHM

This section elaborates on our three-phase ReScue algorithm for automated ReDoS detection, as illustrated in Figure 1.

3.1 Methodology Overview

The ReDoS detection is a *search* problem, i.e., given a regex r, an alphabet Σ (e.g., English alphabet and symbols), and a maximal input length ℓ, finding

$$\text{ReDoS}(r, \Sigma, \ell) = \underset{s \in \Sigma^1 \cup \ldots \cup \Sigma^\ell}{\text{argmax}} \text{RegexMatchTime}_r(s).$$

The search time RegexMatchTime depends not only on the input regex and the input string, but also on the underlying regex engine implementation (e.g., the regex search algorithm and supported regex extensions). The complexity of a regex engine implementation makes static regex analysis techniques [25, 34, 35, 41, 45, 48] neither sound nor complete. Our evaluation later shows that these techniques reported false positives and found less true positive ReDoS vulnerabilities when applied to a practical regex engine. On the other hand, existing heuristic search techniques (e.g., black-box fuzzing [31]) may also fail to recognize ReDoS vulnerabilities. Such search algorithms are usually guided by the cost-effectiveness of a string, i.e., the search time divided by the string's length. When the ReDoS vulnerability requires a less cost-effective prefix to trigger, the search is likely to be trapped in local optimum solutions, leaving the ReDoS vulnerability undetected.

These limitations of existing analysis of search techniques motivate us to design our ReScue technique to dive into the actual regex search procedure for an effective ReDoS string generation. ReScue consists of three phases:

(1) The *seeding* phase explores the regex engine's behaviors under various inputs. In particular, the seeding phase focuses on reaching as many e-NFA states of the regex as possible, *regardless* of the search time.
(2) The *incubating* phase searches for ReDoS candidate strings with maximized search time, as guided by a cost-effective measurement on the basis of the seed strings generated in the first phase. The incubating phase works on the hypothesis that ReDoS strings can be generated by successively applying mutation and crossing over operations to the seeds.

(3) The *pumping* phase finally enhances the ReDoS candidate strings by a local search of removing ReDoS-irrelevant characters and copying/pasting part of a string to obtain a truly ReDoS string of maximized search time.

Our three-phase ReScue technique inherits several important ideas from existing work. Both the seeding and incubating phases belong to genetic search extensively studied in the search-based software engineering domain [9, 20, 21, 29, 30]. In particular, the incubating phase is guided by the cost-effectiveness and resembles the black-box time complexity attack fuzzing [31]. The pumping phase generalizes the idea of static analysis for finding pumpable constructs in a regex [25] to a dynamic local search problem. Putting them together plus a seeding phase for generating useful "artifacts" in crafting ReDoS strings yields our ReScue technique, which is elaborated on as follows.

3.2 Notations and Definitions

3.2.1 Strings, Regexes, and Regex Pattern Matching. We use widely used notations for describing strings and regexes [5, 23]. A *string* s is a sequence of characters from an alphabet Σ. $s(i) \in \Sigma$ denotes the i-th character in string s ($1 \le i \le |s|$), and $s(i{:}j)$ denotes the substring $s(i)s(i+1)\ldots s(j)$. The empty string is denoted by ε. For example, given s = "ReDoS", $s(3)$ = "D" and $s(4{:}5)$ = "oS". Two strings x and y can be concatenated, yielding $x \cdot y$ or simply xy. Concatenating x = "hello" and y = "world" yields xy = "helloworld".

A *regular expression* r is the string representation of a formal language (i.e., a set of strings) $L(r) \subseteq \Sigma^*$. For

$$r_{\text{hello}} = \text{"\^{}(?=hello)[a-z]\{5\}"},$$

$L(r_{\text{hello}})$ = {"hello"}[5].

Given a string s and a regex r, the regex pattern matching problem is equivalent to test whether there exists a substring $s(i{:}j) \in L(r)$. Another relevant problem is the language inclusion test on whether $s \in L(r)$, and regex matching can be reduced to the testing of language inclusion.

3.2.2 e-NFA for Regex Matching. Modern regex engines usually first compile a regex to an *extended non-deterministic finite automaton* (*e*-NFA), which is usually implemented in a modern regex engine [16]. An *e*-NFA can be regarded as a graph $G(V, E)$ in which each vertex $v \in V$ denotes an *e*-NFA state, and each matching step (node) $v \in V$ is associated with an engine-specific function f_v to perform state transition in regex matching. For example, the Java regex engine constructs the *e*-NFA shown in Figure 2 for r_{hello}.

The regex engine conducts a matching by maintaining its matching state S, usually a stack (either explicitly or implicitly by recursion) storing a sequence of $\langle v_i, p_i, t_i\rangle$ where $v_i \in V$ is an *e*-NFA state, $0 \le p \le |s|$ is a position in s[6], and some engine-dependent states t_i. Initially, $S_0 = \{\langle v_0, 0, \bot\rangle\}$ where $v_0 \in V$ is the initial *e*-NFA state.

In each matching step, the regex engine pops $\langle v, p, t\rangle$ from the matching stack, and pushes $f_v(s, p, t)$ onto the stack (s is the input string). Multiple values of $\langle v, p, t\rangle$ may be pushed to represent *non-deterministic* choices. This stack-based mechanism provides versatile supports for regex extensions. Successful matching of a single character is simply done by pushing $\langle v', p+1, t\rangle$, while a failed matching pushes nothing (which is equivalent to backtracking). $f_v(s, p, t)$ of lookahead and repetition nodes in Figure 2 are defined as follows:

(1) For Node v_1 (lookahead), it pushes $\langle v_6, p, \bot\rangle$ then $\langle v_2, p, \bot\rangle$ such that when the matching group (Node v_2–Node v_5) is successfully matched, the matching restarts at Node v_6.
(2) For Node v_6 (repetition), depending on the current repetition count t, $\langle v_8, p, \bot\rangle$ is pushed when $t = 0$ for matching the subsequent parts, otherwise $\langle v_7, p, t-1\rangle$ is pushed for a repeated matching with a count of $t-1$.

Therefore, the regex matching procedure is sophisticated due to the runtime stack (e.g., there can be transitions from v_4 to v_6 even if they are not connected in G, as opposed to the classical NFA [8]). However, the *e*-NFA + backtracking mechanism also provides guidance towards effective search for analyzing regexes using a genetic algorithm.

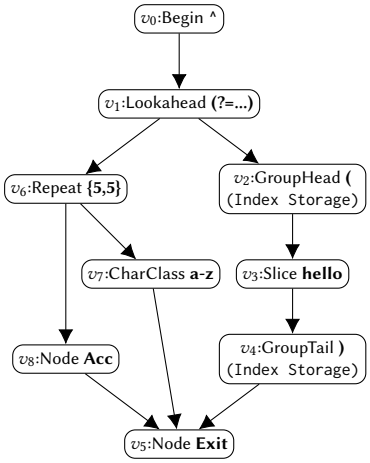

Figure 2: The compiled *e*-NFA of regex `^(?=hello)[a-z]{5}`

3.2.3 Regex Matching Traces. Though the regex matching procedure is sophisticated and engine-dependent, all known regex engines can be modified to produce *positional trace* (or simply *trace*) during matching, which is denoted by

$$\tau_r(s) = \{\langle x_0, p_0\rangle, \langle x_1, p_1\rangle, \ldots, \langle x_m, p_m\rangle\}$$

where $x_i \in V$ is an *e*-NFA state and $0 \le p_i \le |s|$ is a position in the input string denoting the starting position of the next string pattern matching. $\langle x_i, p_i\rangle$ denotes that the i-th regex matching attempt is to find a match when p_i characters in s have been matched at *e*-NFA state x_i. Essentially, the trace is obtained by *discarding* the engine-dependent states (t_i) in the matching procedure. $\tau_r(s)$ encodes sufficient information to guide an effective genetic search towards ReDoS strings in ReScue.

For r_{hello}, the *e*-NFA shown in Figure 2, and s_1 = "helloworld" and s_2 = "hellopanda", conducting regex matching in Java yields the same traces $\tau_{r_{\text{hello}}}(s_1) = \tau_{r_{\text{hello}}}(s_2)$:

$$\{\langle v_0,0\rangle, \langle v_1,0\rangle, \langle v_2,0\rangle, \langle v_3,0\rangle, \langle v_4,5\rangle, \langle v_5,5\rangle, \langle v_6,0\rangle, \langle v_7,0\rangle, \langle v_5,1\rangle,$$
$$\langle v_7,1\rangle, \langle v_5,2\rangle, \langle v_7,2\rangle, \langle v_5,3\rangle, \langle v_7,3\rangle, \langle v_5,4\rangle, \langle v_7,4\rangle, \langle v_5,5\rangle, \langle v_8,5\rangle\}.$$

[5] "(?=)" is the lookahead extension of regex.
[6] This potentially unbounded storage makes regex with extensions *context-aware*.

For two traces $\tau_r(s_1) = \{\langle x_i, p_i\rangle\}$ and $\tau_r(s_2) = \{\langle x'_i, p'_i\rangle\}$, we consider $\tau_r(s_1)$ and $\tau_r(s_2)$ *equivalent* in regex matching if $|\tau_r(s_1)| = |\tau_r(s_2)|$ and $x_i = x'_i$ for all $1 \le i \le |\tau_r(s_1)|$. Therefore, s_1 = "`helloworld`" and s_2 = "`hellopanda`" have equivalent traces in matching against r_{hello}.

Furthermore, we define the *effective prefix* $\text{EP}(s) = s(1{:}k)$ where $k = \max_{\langle x,n\rangle \in \tau_r(s)} (n)$, and the *redundant suffix* $\text{RS}(s) = s(k{+}1{:}|s|)$ is the rest part of s. Obviously, $s = \text{EP}(s)\text{RS}(s)$. We distinguish between EP and RS because regex engines sometimes first check the rest length of the input string to short path the searching, and deleting such tails may lead to an irreproducible trace. For example, the effective prefixes of both "`helloworld`" and "`hellopanda`" are "`hello`".

Finally, a byproduct of regex compilation is a set of string *slices* Σ_{slice}, which contains substrings of r used in the regex matching procedure. For example, Σ_{slice} for r_{hello} is {"`hello`"} (corresponding to the vertex v_3). Strings in Σ_{slice} are useful in the offspring production (by mutation or crossover operators) in a genetic algorithm.

3.3 The Genetic Search Framework

Both the seeding phase and the incubating phase are genetic search, which share the same classical genetic algorithm framework [47]. A genetic algorithm searches for optimal solutions in a search space by representing each solution as an individual (a chromosome-alike data structure) and maintaining an evolving population of individuals.

Given an *initial population*, the genetic algorithm iteratively breeds offspring (by *cross-over* and *mutation* operators) and produces a new generation of population under the guidance of the principle of natural selection: an individual of a higher *fitness function* value has a higher chance to survive.

Both the seeding phase and the incubating phase share the same genetic representation and offspring generation but differ in the initial population, the fitness function, and the selection strategy.

3.3.1 Genetic Representation. We directly use a string to represent each individual in the population

$$\mathcal{P} = \{s_1, s_2, \ldots, s_n\}\ (s_i \in \Sigma^*)$$

because a string naturally resembles a genome. Each generation of the population produces its offspring by random mutation and cut-concatenation of strings, and the new generation is bred by the selection under a fitness function $f(s, \mathcal{P})$.

3.3.2 Offspring Generation. Given a population $\mathcal{P}$, its offspring are generated by crossing over and mutating the strings in $\mathcal{P}$.

The crossover operator $\chi(s_1, s_2)$ interleaves the effective prefixes of s_1 and s_2. In particular, $\chi(s_1, s_2)$ randomly splits $\text{EP}(s_1) = x_1y_1$, $\text{EP}(s_2) = x_2y_2$[7], and returns

$$\chi(s_1, s_2) = \{x_1y_2\text{RS}(s_2), x_2y_1\text{RS}(s_1)\}.$$

Intuitively, the effective prefixes $\text{EP}(s_1)$ and $\text{EP}(s_2)$ are interleaved (x_1y_2 and x_2y_1) with their suffixes appended.

The mutation operator $\mu(s)$ returns a random mutant of s, where $u \in \Sigma \cup \Sigma_{\text{slice}}$ is a random string (either a random character in Σ or a random string constant in the regex's corresponding automaton):

- Appending u in $\text{EP}(s)$;
- Inserting u to a random position in $\text{EP}(s)$;
- Deleting a random substring in $\text{EP}(s)$;
- Duplicating a random substring in $\text{EP}(s)$;
- Reversing a random substring in $\text{EP}(s)$.

It is worth noting that both χ and μ operate on the effective prefix $\text{EP}(s)$ and keeps $\text{RS}(s)$ unchanged, because the redundant suffix $\text{RS}(s)$ has not been involved in the matching procedure. Nevertheless, keeping a sufficiently long $\text{RS}(s)$ is also necessary because the regex engine may check whether $\text{RS}(s)$ is sufficiently long for a successful matching.

Then the offspring are generated by applying χ and μ to randomly selected individuals in the population $\mathcal{P}$, and newly generated offspring whose trace is equivalent to an existing $\tau_r(s)$ for $s \in \mathcal{P}$ is considered redundant and is immediately discarded. Furthermore, we gave the individuals of higher fitness $f(s, \mathcal{P})$ values a higher probability in generating offspring (by crossover or mutation):

$$\mathbf{Pr}[s \in \mathcal{P} \text{ is chosen}] = \frac{f(s, \mathcal{P})}{\sum_{s' \in \mathcal{P}} f(s', \mathcal{P})}.$$

3.4 The Seeding Phase: Searching for Diverse *e*-NFA States

The seeding phase tries to find a population $\mathcal{P}_{\text{seed}}$ of strings that covers as many e-NFA states as possible and basically *ignores* the search time[8].

The initiate population of the seeding phase $\mathcal{P}_0$ consists of constant strings Σ_{slice} in the automaton plus some random strings in the alphabet of Σ.

A binary fitness function $f_{\text{seed}}(s, \mathcal{P})$ is used in the seeding phase, which summarizes whether the matching trace $\tau_r(s)$ covers some unique e-NFA states in the population $\mathcal{P}$. Formally, let $\mathcal{V}(s) = \bigcup_{\langle x,n\rangle \in \tau_r(s)} v$ denote the set of traversed e-NFA states in the regex matching procedure,

$$f_{\text{seed}}(s, \mathcal{P}) = \begin{cases} 1, & \exists v \in \mathcal{V}(s).\ v \notin \bigcup_{s' \in \mathcal{P} \setminus \{s\}} \mathcal{V}(s'); \\ 0, & \text{otherwise.} \end{cases}$$

After a new population $\mathcal{P}'$ containing crossed over and mutated strings is generated, the seeding phase selects a *minimal* set of strings $\mathcal{P}^\star \subseteq \mathcal{P}'$ as the population in the next iteration, which satisfies $\bigcup_{s \in \mathcal{P}^\star} \mathcal{V}(s) = \bigcup_{s \in \mathcal{P}} \mathcal{V}(s)$. $\mathcal{P}^\star$ is generated by sequentially considering each $s \in \mathcal{P}$ and discarding the strings without any uniquely covered e-NFA state in $\mathcal{V}(s)$.

3.5 The Incubating Phase: Searching for ReDoS Candidates

The incubating phase "incubates" the seeds to find a population of strings of maximized regex matching time. Therefore, the initial population of the incubating phase $\mathcal{P}_0 = \mathcal{P}_{\text{seed}}$.

[7] If not otherwise stated, the random selection of $x \in X$ returns each x with the same probability of $1/|X|$.

[8] In contrast to the existing technique that directly searches towards a larger running time [31], we use the seeding phase to find ReDoS strings of a long cost-ineffective prefix.

Algorithm 1: The pumping algorithm

Input: A string s and a designated length ℓ ($|s| \le \ell$)
Output: A pumped ReDoS string

1 **begin**
2 $s_{\text{trim}} \leftarrow s$;
3 **for** $i \leftarrow |S|$ **to** 1 **do**
4 $s' \leftarrow s_{\text{trim}}(1{:}i-1)s_{\text{trim}}(i+1{:}|s_{\text{trim}}|)$; // try to remove the i-th character
5 **if** $f_{incub}(s') \ge f_{incub}(s)$ **then**
6 $s_{\text{trim}} \leftarrow s'$;
7 $s \leftarrow s_{\text{trim}}$;
8 $m \leftarrow 0$; $i^\star \leftarrow 0$; $j^\star \leftarrow 0$;
9 **for** $i \leftarrow 1$ **to** $|s|$ **do**
10 **for** $j \leftarrow i+1$ **to** $|s|$ **do**
11 $s' \leftarrow s(1{:}i-1) \cdot s(i{:}j)^2 \cdot s(j+1{:}|s|)$; // repeating $s(i{:}j)$ twice
12 **if** $f_{incub}(s') > m$ **then**
13 $m \leftarrow f_{\text{incub}}(s')$; $i^\star \leftarrow i$; $j^\star \leftarrow j$;
14 $k \leftarrow \left\lfloor \frac{\ell - |s|}{j^\star - i^\star + 1} \right\rfloor$; // maximum allowed number of pumps
15 **return** $s(1{:}i^\star-1) \cdot s(i^\star{:}j^\star)^k \cdot s(j^\star+1{:}|s|)$;

The fitness function in the incubating phase

$$f_{\text{incub}}(s) = \frac{|\tau_r(s)|}{|s|}$$

measures the average cost-effectiveness of each character in s. Such a fitness function naturally guides the search towards ReDoS strings (short strings costing a substantial amount of matching steps).

Furthermore, we simultaneously maintain a *diverse* population to avoid being stuck at a local optimum. Given a newly generated population $\mathcal{P}'$, let

$$\bar{f} = \frac{1}{|\mathcal{P}'|} \sum_{s \in \mathcal{P}'} f_{\text{incub}}(s)$$

be the average of fitness function values, we first select

$$\mathcal{P}^\star = \{s \in \mathcal{P}' \mid f_{\text{incub}}(s) \ge \bar{f}\}$$

to include any string fitter than the average, and then add back a minimal set of strings to maintain exactly the same e-NFA state coverage by enumerating all strings $s \in \mathcal{P}'$ and adding s to $\mathcal{P}^\star$ if $\mathcal{V}(s) \nsubseteq \bigcup_{s^\star \in \mathcal{P}^\star} \mathcal{V}(s^\star)$.

3.6 The Pumping Phase: Enhancing ReDoS Candidates for a Successful Attack

The pumping phase selects such incubated string of the highest cost-effective ratio

$$s^\star = \operatorname*{argmax}_{s \in \mathcal{P}} f_{\text{incub}}(s)$$

and *pumps* it into a highly effective input string for slowing down the regex matching procedure. The idea (and the name) has been inspired by the Pumping Lemma, which is also used in the static detection of ReDoS vulnerabilities [25]:

Pumping Lemma [33]. *For any regular language L, there is a $p \ge 0$ such that any string $w \in L$ of $|w| \ge p$ can be written as $w = xyz$, where $y \ne \varepsilon$ and the strings $xz, xyz, xyyz, \ldots$ constructed by repeating y zero or more times are still in L.*

In other words, there may exist a substring that can be repeated and does not affect the matching results[9]. Therefore, the pumping phase trims those unnecessary characters in $s^\star$ and pumps a carefully chosen substring to finally reach a string that requires a large number of matching steps, as shown in Algorithm 1.

The trimming algorithm (Lines 2–6) removes unnecessary characters by trying to remove each single character and checks whether such a removal yields a higher fitness f_{incub} value (Line 5). Trimming is also used to reduce the length of the redundant suffix RS(s).

Then the pumping algorithm (Lines 8–15) enumerates all substrings in $s(i{:}j)$ and tries to pump one once yielding

$$s' = s(1{:}i-1) \cdot s(i{:}j)^2 \cdot s(j+1{:}|s|),$$

and finally pumps the most profitable substring (Lines 14–15) to obtain our ReDoS string.

4 IMPLEMENTATION

We implemented ReScue in Java. The implementation has approximately 3,000 lines of Java code and 500 lines of Python scripts[10], and is composed of the following components:

(1) A modified version of Java 8 `regex` standard library for collecting profiling data $\tau_r(s)$.
(2) An implementation of the genetic algorithms described in Section 3.
(3) A series of scripts for handle miscellaneous tasks, e.g. regex extraction and syntax translation.

4.1 The Modified `regex` Library

The Java `regex` standard library compiles a regex into an e-NFA, and e-NFA's state transitions are implemented by calling an e-NFA state (represented by an instance of class `Node`)'s corresponding `match` method. The slices Σ_{slice} are extracted from the `Slice` nodes.

The matching trace $\tau_r(s)$ is collected along with the regex matching each time when the `match` method of a `Node` instance is invoked. In the matching procedure, we also keep track of a list of $\langle x_i, p_i \rangle$ and maintain the pivot index k, which splits the effective prefix and the redundant suffix for string s, i.e., $k = |\text{EP}(s)|$. Such collected information suffices for conducting our three-phase ReDoS vulnerability detection.

4.2 The Genetic Algorithm Implementation

We adopt the early exit mechanism in both the seeding and incubating phases when matching r against a string $s \in \mathcal{P}$ requires more than 10^5 matching steps to terminate, and directly move on to the pumping phase. This treatment prevents the search from hanging when regex matching is extremely time-consuming.

The parameters for the genetic search are set to typical values in existing literatures [14, 38, 39]. In both the seeding and incubating

[9] Even if the Pumping Lemma does not hold for extended regexes, such a pumpable structure may exist for crafting ReDoS strings. The pumping phase searches for such structures by enumerating substrings in s.
[10] The ReScue tool is publicly available at https://2bdenny.github.io/ReScue/.

Table 1: The regex sets for evaluation.

Name	Number	Description
RegLib [35]	2,992	Online regex examples from `RegExLib.com`
Snort [35]	12,499	regexes extracted from the Snort NIDS for data packet filtering
Corpus [7]	13,597	regexes from scraped python projects

phases, we set the population size $|\mathcal{P}| = 200$. The crossover rate is set to 5% and the mutation rate is set to 10%. Both phases terminate after 200 iterations in case that the early exit mechanism is not triggered.

The pumping phase also adopts the early exit mechanism. If repeating a substring $s(i{:}j)$ yields more than 10^6 matching steps (10× larger compared with the early exit threshold in the seeding and incubating phases), the pumping phase exits earlier by returning the results of pumping $s(i{:}j)$. Finally, if a pumped string yields more than 10^8 matching steps, we consider that ReScue has successfully generated a ReDoS string.

4.3 Miscellaneous Utilities

Though we implemented the tool in Java, we also implemented scripts for extracting regexes from source code written in other programming languages (in particular, Python, JavaScript, and PHP), and converting the extracted regexes into the Java regex syntax. These scripts further helped us check the regexes from popular GitHub open source projects, and all detected vulnerabilities are discussed later in Section 5.4.

5 EVALUATION

The evaluation mainly concerns the following two aspects of ReScue:

- (*Effectiveness*) Is ReScue effective in detecting ReDoS vulnerabilities? Does it outperform existing techniques? Can it detect ReDoS vulnerabilities in real-world projects, even written in other programming languages?
- (*Efficiency*) Can ReScue run within a reasonable amount of time?

5.1 Experimental Settings

We collected three sets of regexes (29,088 regexes in total) from the evaluation of existing ReDoS vulnerability detection techniques, as listed in Table 1, and the following techniques are compared:

- ReScue presented in this paper.
- SlowFuzz [31], a dynamic genetic fuzzing technique for finding an input that maximizes the running time.
- RXXR2 [25, 35], a static ReDoS vulnerability analyzer based on pumping analysis.
- Rexploiter [48], another static ReDoS vulnerability analyzer based on pumping analysis.
- NFAA [45], a static analyzer based on matching malicious patterns in regexes.

For each regex, we run each evaluated technique under the default setting to obtain a ReDoS string of maximum length $\ell = 128$, and the results (the ReDoS string and the statistics) are collected for evaluation and comparison. This size represents a typical scenario in handling untrusted user inputs (e.g., a field in a Web form that is constrained by a moderate length limit). We consider an output ReDoS string *successfully* exploiting the ReDoS vulnerability (true positive) if it takes more than 10^8 matching steps when applied to the Java 8 regex engine.

We set a ten minute time limit to the genetic search in both ReScue and SlowFuzz. When the search exceeds the time limit, ReScue terminates regardless of its running phase while SlowFuzz outputs the current population.

Each technique's supported regex extensions are shown in Table 2. RXXR2, Rexploiter, and NFAA parse an input regex using a built-in parser and are independent of the underlying regex engine. In other words, these techniques output a potentially vulnerable string regardless of the actual matching procedure, and may produce false positives.

For SlowFuzz, we chose the one with the largest matching steps as the ReDoS string in SlowFuzz's final-round population. SlowFuzz is implemented in C and linked with the (unmodified) PCRE2 regex engine, which is different from our evaluation environment. To best alleviate such a difference for a fair comparison, we also applied ReScue generated ReDoS strings to the PCRE2 engine for cross validation of the effectiveness.

We also evaluated the effectiveness of each phase (seeding, incubating, and pumping) in ReScue by temporarily disabling each phase and study the corresponding number of successfully generated ReDoS strings.

All experiments were conducted on a server of four hexa-core Intel Xeon X7460 processors (24 cores in total) and 64GB RAM running Ubuntu 16.04.

5.2 Effectiveness Evaluation Results

5.2.1 Overall Results. The overall evaluation results are shown in Table 3. Considering the Java 8 regex engine, all ReDoS strings generated by all techniques found that 227 regexes are vulnerable (there exists a string of no more than $\ell = 128$ characters but costing more than 10^8 matching steps).

ReScue found 186 (82%) of these vulnerabilities and significantly outperforms all the other techniques. The best existing technique RXXR2 found 125 (accounting for 55% of all the known vulnerable regexes) vulnerabilities. Compared with RXXR2, ReScue found 49% more vulnerabilities. SlowFuzz also successfully identified 101 vulnerable regexes, but is even less effective compared with RXXR2.

We also observed that static analyses report false positives (dynamic analyses search towards the vulnerability criteria and always report true positives). The best existing technique RXXR2 reported 39% false positive cases, and Rexploiter and NFAA only reported a tiny fraction of true positives. This supported our claim that modern regex engines are complex and static analyses fall short.

We cross-validated the generated ReDoS strings to the PCRE2 regex engine (implemented in C), which is highly optimized and is several magnitudes faster than the Java standard `regex` library. The PCRE2 behaves quite differently than the Java regex engine, and

Table 2: Supported regex extensions of each evaluated technique.

Features	ReScue	SlowFuzz	RXXR2	Rexploiter	NFAA
Set operations `[a-z]&&[^aeiou]`	×	✓	×	×	×
Lookarounds `(?=) (?!) (?<=) (?<!)`	✓	✓	×	×	×
Backreferences `()\1`	✓	✓	✓	×	×
Non-capturing groups `(?:)`	✓	✓	✓	×	×
Named groups	✓	✓	×	×	×
Atomic groups `(?>)`	✓	✓	×	×	×
Conditionals `()?(?(1))`	×	✓	×	×	×
Greedy quantifiers `{m,n} {n} {m,}`	✓	✓	✓	✓	✓
Lazy quantifiers `?? *? +? {}?`	✓	✓	✓	✓	✓
Possessive quantifiers `?+ *+ ++ {}+`	✓	✓	×	×	×

Table 3: The overall evaluation results. #Vul. denotes the number of vulnerabilities found and #FP denotes the number of false positives (output strings that costs less than 10^8 matching steps).

Technique	#Vul.	#FP	TP Rate	Avg Time (s)
ReScue	186 (82%)	-	-	0.6128
SlowFuzz	101 (44%)	-	-	0.5965
RXXR2	125 (55%)	80	61%	0.0025
Rexploiter	30 (13%)	2152	1.3%	0.4073
NFAA	0 (0%)	714	N/A	2.1546
Summary	227 (100%)			

Table 4: The evaluation results of removing a phase. The Coverage column contains the average *e*-NFA state coverage.

Variant	#Vul.	Avg Time (s)	Coverage
ReScue	186 (82%)	0.6128	90.23%
without seeding	152 (67%)	0.5061	57.84%
without incubating	99 (44%)	0.1273	89.76%
without pumping	49 (22%)	0.5705	90.31%

almost all Java ReDoS strings quickly terminate on PCRE2. Considering the ReDoS strings for Java found by SlowFuzz and ReScue, only 6/101 (SlowFuzz) and 8/186 (ReScue) exceed a matching time of 1ms, and the median matching time of ReScue is merely 25% longer than SlowFuzz.

On the other hand, the Java regex engine is less optimized, whose behavior resembles the JavaScript/Python regex engine. Though we cannot evidently prove that ReScue is more effective in finding ReDoS instances than SlowFuzz for PCRE2, using such a less optimized engine facilitates us finding previously unknown vulnerabilities that cannot be detected by any evaluated technique.

Finally, the evaluation results also show that ReScue cannot detect *all* these known vulnerabilities. Though finding 82% of them is a quite promising result, there is still space for improvement and we will address this in the future work.

5.2.2 Effectiveness of the Phases. By removing a single phase in ReScue, we evaluated the effectiveness of each phase. The results are shown in Table 4, indicating that all the seeding, incubating, and pumping phase contributed to the effectiveness of ReScue.

Removal of the seeding phase has the least impact (18% less vulnerabilities detected) on the effectiveness. The seeding phase pays an additional 0.1 seconds time cost to drastically improve the *e*-NFA state coverage, which is worthy in analyzing complex regexes and finding potential vulnerabilities hidden in the deep states of a regex.

At first glance, one might thought that the seeding phase is not very much useful in ReScue, however, recall that in the incubating phase, we force the population to maintain a high NFA state coverage, i.e., the incubating phase *subsumes* the seeding phase to some extend. Furthermore, both seeding and incubating phases use the identical genetic representation and mutation/cross-over operators, the seeding phase uniquely contributed to 18% of ReScue discovered ReDoS strings.

Removal of the incubating has a significant impact on the effectiveness. This is expected because the incubating phase searches towards a string to maximize the matching efficiency, which is the core of a dynamic algorithmic complexity fuzzer [31].

Removal of the pumping phase has the largest impact on the effectiveness simply because we set a much smaller threshold in the incubating phase (10^5 matching steps) compared with the threshold of a successful ReDoS attack (10^8 matching steps, which is usually reached in the pumping phase). In contrast, SlowFuzz [31] directly searches for a algorithmic complexity because (1) the PCRE2 engine is hundreds times faster than the instrumented Java regex engine in ReScue; and (2) SlowFuzz's native C genetic search implementation, including mutation/cross-over operators, is much more efficient than ReScue. ReScue uses the pumping algorithm to alleviate the issues caused by the relatively inefficient implementation. The 49 ReDoS strings found by ReScue without pumping are considered relative easy cases that ReDoS could be triggered by a simple fuzzing.

Without incubating or pumping phases, ReScue would be less effective than existing techniques in finding ReDoS vulnerabilities. The incubating phase creates ReDoS attack candidates, and these strings are being further enhanced by the regex-designated pumping algorithm. Therefore, we believe that the effectiveness of ReScue is truly attributed to our carefully designed three-phase regex-aware algorithms.

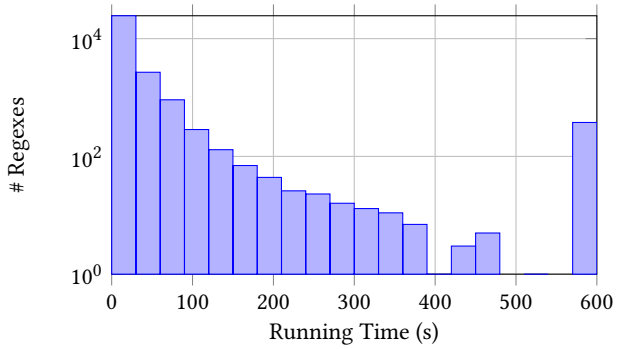

Figure 3: The histogram of ReScue runtime over all evaluated regexes

5.3 Efficiency Evaluation Results

The fifth column of Table 3 shows the average time of processing a regex, and Figure 3 displays the detailed runtime histogram statistics of ReScue. ReScue has a comparable running time with SlowFuzz (less than one second per regex), even if the Java regex engine with our profiling modifications is magnitudes slower than the native C regex engine PCRE2. We believe that such efficiency is due to our regex-designated algorithms, which quickly generate cost-effective ReDoS strings.

5.4 Case Study: ReDoS in Popular GitHub Projects

We searched the GitHub for highly starred Python and JavaScript projects that contain either `editor`, `web app`, or `database` in the project description. We downloaded 50 projects, scanned them for regexes, and tested these regexes using ReScue in the default setting.

ReScue found ten previously unknown ReDoS vulnerabilities or defects in these projects and we reported them in the projects' corresponding issue tracking systems. Six reports have been confirmed and several have already been fixed by the respective developer. Some confirmed cases and (simplified) ReScue generated ReDoS strings are discussed as follows.

`Meteor` (39k stars) is a JavaScript App Platform. One of its regexes, which is used to recognize comments in an HTML document, is found vulnerable by ReScue:

```
var _HashHTMLBlocks = function(text) {
text = text.replace(
/(\n\n[␣]{0,3}<!(--[^\r]*?--\s*)+>[␣\t]*(?=\n{2,}))/g
, hashElement);
// vulnerable to "\n\n<!--\n\n<!--------<\n\n<!--...->]Vx0
     \n\n\n------!!<\n\n\n\n]-----<!]Vx0\n\n-------<!"
```

This complex regex uses regex grammar extensions (the ?= lookaround) and thus static analyses fail to parse it. Furthermore, the ReDoS vulnerability requires both a certain prefix and a certain suffix to trigger, making a black-box fuzzing fall short. None of the evaluated techniques can detect this vulnerability except for ReScue.

`tui.editor` (6.3k stars) is a JavaScript WYSIWYG Markdown Editor. ReScue found that its syntax highlighting component may be significantly slowed down under particular input combinations:

```
const strikeRegex = /^[~~](.*[\s\n]*.*)*[~~]$/;
// vulnerable to "~&aF~&aF~&aF~&aF~&aF~&aF~&aFxDHD#"
...
const Strike = CommandManager.command('markdown', {
  ...
  hasStrikeSyntax(text) {
    return strikeRegex.test(text);
  },
  ...
}
```

`ace` (17k stars) is the `Ajax.org` Cloud9 code editor written in JavaScript. The regex for syntax highlighting may also be stuck and thus impair the entire application:

```
this.removeCapturingGroups = function(src) {
  var r = src.replace(
    /\[(?:\\.|[^\]])*?\]|\\.|\(\?[:=!]|(\()/g,
    function(x, y) {return y ? "(?:" : x;}
  );
  // vulnerable to "[[@[[@[@[_\>\?\|)\\>\?\|)\\>\?\|)
       \\...>\?\|)\>\?\|\x<,T'@[x<,T'@[_"
  return r;
};
```

`Open States` (0.5k stars) is a Python crawler for gathering official PDF files on the Internet. After scanning all regexes in the source code, ReScue found two vulnerable regexes, and one is shown as follows:

```
def scrape_lower(self):
  text = ...
  days = ...
  for day in enumerate(days[1:]):
    if day[0] % 2 == 0:
      date = day[1]
    else:
      events = re.split(r'\n((?:\w+\s?)+)\n', day[1])
      # vulnerable to "\na097a097a...097a4199613918"
```

Therefore, a malicious Web page may set up a crafted PDF file and hang the crawler.

In summary, we believe that the evaluation results confirmed the effectiveness and efficiency of ReScue in detecting ReDoS vulnerabilities. The Java implementation found 49% more vulnerabilities compared with the best evaluated technique, and has a comparable running time with a native C black-box fuzzer. The tool also detected cross-language ReDoS vulnerabilities in popular real-world projects.

6 RELATED WORK

6.1 Regex and ReDoS

Regex is widely used in real-world software projects. For example, a survey of regex usage [7] shows that regex occurs in 42% Python projects. Current regexes have multiple flavors and also have different implementations of their matching engines [16, 18]. However, some regex grammars depend heavily on a backtracking algorithm and have not considered carefully their underlying formalization [4, 35]. This fact, on one hand, makes ϵ-NFA + backtracking irreplaceable, and on the other hand, sometimes makes these matching engines suffer from ReDoS.

ReDoS [19, 46] is one form of algorithmic complexity attacks [13], and this kind of attacks can be used to craft low-bandwidth denial-of-service threats, and even be used to attack some intrusion detection systems [2, 13, 37].

6.2 Existing ReDoS Detection Techniques

Due to the existence of such ReDoS vulnerabilities, researchers have proposed various techniques for detecting potential ReDoS vulnerabilities before given regexes are fed to their matching engines. Existing ReDoS detection tools can be mainly classified into two types, as we discuss below.

6.2.1 Static Analysis. This line of work is dedicated to formalize regex grammars and statically search attack patterns from them. However, up to now there is still no static analysis tool that can formalize all regex grammars and support complete ReDoS analysis. We introduce several representative tools below.

RXXR2 [34, 35] is a ReDoS static analysis tool developed from RXXR [25]. The authors proposed a general attack string pattern in the form of (*prefix*, *pumpable string*, *suffix*). To use this pattern, they convert given regexes into an *e*-NFA structure, and with an efficient pattern matching algorithm, they search for instances of this pattern in the *e*-NFA. However, due to the restrictions of their proposed *e*-NFA converter, regexes with extended grammars is out of the scope of their analyses.

Rexploiter [48] is a tool that not only detects ReDoS vulnerabilities but also conducts taint-propagation analysis to exclude user-input unreachable regexes. However, this tool builds on a similar NFA-alike structure as in RXXR2, and this foundation makes it also unable to parse extended regexes.

Weideman et. al. [45] use a static analysis method based on the pNFA [3, 4] to develop an NFAA tool for detecting ReDoS vulnerabilities for regexes. pNFA is a new automaton model, which can describe notions like backreferences and named groups. However, new attack string patterns based on this model need exploration and improvement.

Sugiyama et. al. [41] propose detecting ReDoS vulnerabilities by simulating the regex matching process by a tree transducer. They show that tree transducing is formally equivalent to regex matching, and that ReDoS vulnerabilities can be detected by analyzing the tree transducer's size increment with trials of manual inputs. However, since this process relies on human knowledge, it is not subject to automation.

6.2.2 Dynamic Fuzzing. This line of work takes the idea of fuzz testing [17, 32] from the software testing community. With the support of runtime analysis, this kind of tool leaves the regex parsing to existing regex matching engines, and is devoted only to generating time-consuming strings for exposing potential ReDoS vulnerabilities. We also introduce several representative tools below.

SlowFuzz [31] is a general algorithmic complexity attack tool based on dynamic fuzzing. It uses an evolutionary fuzzer (based on LibFuzzer [32]) to search for those inputs that can trigger a large number of edges in the control flow graph of the program under testing. However, due to its generic nature, SlowFuzz has no knowledge about regex structures, and this fact can cause some deep states of the program unreachable, leading to false negatives.

SDLFuzzer is another dynamic fuzzer for ReDoS detection [42, 43], developed by Bryan Sullivan from Microsoft in 2010. However, this tool has no longer been maintained and has no further update since then. Therefore, the tool does not support many extended regex grammars, e.g., lookarounds, named groups, atomic groups, backreferences, and lazy or possessive quantifiers.

6.3 ReDoS Prevention or Alleviation

ReDoS attacks can also be prevented or alleviated by two treatments, namely, equivalent regex conversion and regex matching speedup, as we discuss below.

6.3.1 Equivalent Regex Conversion. This treatment tries to find equivalent invulnerable regexes to replace the original ones that are vulnerable to ReDoS attacks. For example, Cody-Kenny et. al. [10] propose applying genetic algorithms to find an equivalent regex that describes the same language as the given original regex and at the same time is free of any ReDoS vulnerability. Then the original regex can be safely replaced without affecting functionalities. However, one limitation is that due to the lack of a testing input set that can enumerate all strings in the language, the approach may not guarantee the equivalence between the newly generated regex and its corresponding original regex.

6.3.2 Regex Matching Speedup. Speedup treatments for regex matching are possible for special cases where regexes contain only simple grammars, whose matching can be optimized by dedicated techniques. For example, matching for basic regexes can be significantly speed up by parallel algorithms [28], GPU-based algorithms [49], state-merging algorithms [1], and techniques that convert NFAs to DFAs [12]. We note that this can only alleviate the ReDoS vulnerability problem, but the concerned regexes themselves are still subject to complexity attacks.

7 CONCLUSION

This paper presents ReScue, a three-phase technique, to automatically detect regular expression Denial-of-Service (ReDoS) vulnerabilities. ReScue seeds, incubates, and finally pumps an initial population of strings to obtain a ReDoS attack string whose matching time is maximized. The evaluation results show that ReScue outperforms state-of-the-art techniques in detecting ReDoS vulnerabilities and found previously unknown vulnerabilities in popular GitHub open-source projects. Finally, the evaluation results also suggest that there is space for improvement. We plan to further enhance the three-phase algorithms for more effective detection of ReDoS vulnerabilities.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their comments and suggestions. This work is supported in part by National Key R&D Program (Grant #2017YFB1001801), National Basic Research 973 Program (Grant #2015CB352202), National Natural Science Foundation (Grants #61690204, #61472174) of China, and the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

REFERENCES

[1] Michela Becchi and Srihari Cadambi. 2007. Memory-efficient regular expression search using state merging. In *Proceedings of the 26th IEEE International Conference on Computer Communications (INFOCOM '07)*. 1064–1072. https://doi.org/10.1109/INFCOM.2007.128

[2] Udi Ben-Porat, Anat Bremler-Barr, Hanoch Levy, and Bernhard Plattner. 2012. On the vulnerability of hardware hash tables to sophisticated attacks. In *International Conference on Research in Networking (Networking '12)*. 135–148. https://doi.org/10.1007/978-3-642-30045-5_11

[3] Martin Berglund, Frank Drewes, and Brink Van Der Merwe. 2014. Analyzing Catastrophic Backtracking Behavior in Practical Regular Expression Matching. In *Proceedings of the 14th International Conference on Automata and Formal Languages (AFL '14)*. 109–123.

[4] Martin Berglund and Brink van der Merwe. 2017. On the semantics of regular expression parsing in the wild. *Theoretical Computer Science* 679 (May 2017), 69–82. https://doi.org/10.1016/j.tcs.2016.09.006

[5] Anne Brüggemann-Klein. 1993. Regular expressions into finite automata. *Theoretical Computer Science* 120, 2 (Nov. 1993), 197–213. https://doi.org/10.1016/0304-3975(93)90287-4

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

[7] Carl Chapman and Kathryn T Stolee. 2016. Exploring regular expression usage and context in Python. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA '16)*. 282–293. https://doi.org/10.1145/2931037.2931073

[8] Noam Chomsky. 1956. Three models for the description of language. *IRE Transactions on Information Theory* 2, 3 (Oct. 1956), 113–124. https://doi.org/10.1109/TIT.1956.1056813

[9] John Clarke, Jose Javier Dolado, Mark Harman, Rob Hierons, Bryan Jones, Mary Lumkin, Brian Mitchell, Spiros Mancoridis, Kearton Rees, Marc Roper, et al. 2003. Reformulating software engineering as a search problem. *IEE Proceedings - Software* 150, 3 (June 2003), 161–175. https://doi.org/10.1049/ip-sen:20030559

[10] Brendan Cody-Kenny, Michael Fenton, Adrian Ronayne, Eoghan Considine, Thomas McGuire, and Michael O'Neill. 2017. A search for improved performance in regular expressions. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '17)*. 1280–1287. https://doi.org/10.1145/3071178.3071196

[11] Erik Corry, Christian P Hansen, and Lasse R H Nielsen. 2009. Irregexp, Google Chrome's New Regexp Implementation. https://blog.chromium.org/2009/02/irregexp-google-chromes-new-regexp.html

[12] Russ Cox. 2007. *Regular expression matching can be simple and fast (but is slow in Java, Perl, Php, Python, Ruby, ...)*. Technical Report.

[13] Scott A Crosby and Dan S Wallach. 2003. Denial of Service via Algorithmic Complexity Attacks. In *Proceedings of the 12th USENIX Security Symposium (USENIX Security '03)*. 29–44. https://dl.acm.org/citation.cfm?id=1251356

[14] Yuan-Shun Dai, Min Xie, Kim-Leng Poh, and Bo Yang. 2003. Optimal testing-resource allocation with genetic algorithm for modular software systems. *Journal of Systems and Software* 66, 1 (April 2003), 47–55. https://doi.org/10.1016/S0164-1212(02)00062-6

[15] Python Software Foundation. 2018. regex (PyPI). https://pypi.org/project/regex/

[16] Jeffrey E F Friedl. 2006. *Mastering Regular Expressions: Understand Your Data and Be More Productive* (3th ed.).

[17] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. 2008. Automated whitebox fuzz testing. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS '08)*. 151–166. http://www.isoc.org/isoc/conferences/ndss/08/papers/10_automated_whitebox_fuzz.pdf

[18] Jan Goyvaerts. 2018. Popular Tools, Utilities and Programming Languages That Support Regular Expressions. https://www.regular-expressions.info/tools.html

[19] Jan Goyvaerts. 2018. Runaway Regular Expressions: Catastrophic Backtracking. https://www.regular-expressions.info/catastrophic.html

[20] Mark Harman and Bryan F Jones. 2001. Search-based software engineering. *Information and Software Technology* 43, 14 (Dec. 2001), 833–839. https://doi.org/10.1016/S0950-5849(01)00189-6

[21] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. 2012. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)* 45, 1, Article 11 (Nov. 2012), 61 pages. https://doi.org/10.1145/2379776.2379787

[22] Philip Hazel. 2017. Backtracking Limit of PCRE Match. http://www.pcre.org/current/doc/html/pcre2api.html

[23] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. 2006. *Introduction to Automata Theory, Languages, and Computation* (3rd ed.).

[24] Mohd Ehmer Khan, Farmeena Khan, et al. 2012. A comparative study of white box, black box and grey box testing techniques. *International Journal of Advanced Computer Science and Applications* 3, 6 (2012), 12–15. https://doi.org/10.14569/ijacsa.2012.030603

[25] James Kirrage, Asiri Rathnayake, and Hayo Thielecke. 2013. Static analysis for regular expression denial-of-service attacks. In *Proceedings of the 7th International Conference on Network and System Security (NSS '13)*. 135–148. https://doi.org/10.1007/978-3-642-38631-2_11

[26] Stephen Cole Kleene. 1951. *Representation of Events in Nerve Nets and Finite Automata*. Technical Report.

[27] Andrew M Kuchling. 2018. Regular Expression How-To. https://docs.python.org/3/howto/regex.html

[28] Cheng-Hung Lin, Chen-Hsiung Liu, and Shih-Chieh Chang. 2011. Accelerating regular expression matching using hierarchical parallel machines on GPU. In *Proceedings of the 2011 IEEE Global Telecommunications Conference (GLOBECOM '11)*. IEEE, 1–5. https://doi.org/10.1109/GLOCOM.2011.6133663

[29] Phil McMinn. 2004. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability* 14, 2 (June 2004), 105–156. https://doi.org/10.1002/stvr.294

[30] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. 75–84. https://doi.org/10.1109/ICSE.2007.37

[31] Theofilos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. 2017. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the International Conference on Computer and Communications Security (CCS '17)*. 2155–2168. https://doi.org/10.1145/3133956.3134073

[32] LLVM Project. 2018. libFuzzer: A library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html

[33] Michael O Rabin and Dana Scott. 1959. Finite automata and their decision problems. *IBM Journal of Research and Development* 3, 2 (April 1959), 114–125. https://doi.org/10.1147/rd.32.0114

[34] Asiri Rathnayake. 2015. *Semantics, analysis and security of backtracking regular expression matchers*. Ph.D. Dissertation. Advisor(s) Thielecke, Hayo. http://etheses.bham.ac.uk/6011/

[35] Asiri Rathnayake and Hayo Thielecke. 2014. Static analysis for regular expression exponential runtime via substructural logics. (2014). arXiv:arXiv:1405.7058

[36] Juraj Hajduch (SK). 2007. RegExLib: A regular expression library. http://www.regexlib.com/REDetails.aspx?regexp_id=2598

[37] Randy Smith, Cristian Estan, and Somesh Jha. 2006. Backtracking algorithmic complexity attacks against a NIDS. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC '06)*. 89–98. https://doi.org/10.1109/ACSAC.2006.17

[38] Mozhan Soltani, Annibale Panichella, and Arie van Deursen. 2017. A guided genetic algorithm for automated crash reproduction. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. 209–220. https://doi.org/10.1109/ICSE.2017.27

[39] Praveen Ranjan Srivastava and Tai-hoon Kim. 2009. Application of genetic algorithm in software testing. *International Journal of Software Engineering and Its Applications* 3, 4 (Oct. 2009), 87–96.

[40] Cristian-Alexandru Staicu and Michael Pradel. 2018. Freezing the web: A study of redos vulnerabilities in javascript-based web servers. In *27th USENIX Security Symposium (USENIX Security '18)*.

[41] Satoshi Sugiyama and Yasuhiko Minamide. 2014. Checking time linearity of regular expression matching based on backtracking. *IPSJ Online Transactions* 7 (Nov. 2014), 82–92. https://doi.org/10.2197/ipsjtrans.7.82

[42] Bryan Sullivan. 2010. New Tool: SDL Regex Fuzzer. https://cloudblogs.microsoft.com/microsoftsecure/2010/10/12/new-tool-sdl-regex-fuzzer

[43] Bryan Sullivan. 2010. Regular Expression Denial of Service Attacks and Defenses. https://msdn.microsoft.com/en-us/magazine/ff646973.aspx

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

[45] Nicolaas Weideman, Brink van der Merwe, Martin Berglund, and Bruce Watson. 2016. Analyzing matching time behavior of backtracking regular expression matchers by using ambiguity of NFA. In *Proceedings of the International Conference on Implementation and Application of Automata (CIAA '16)*. 322–334. https://doi.org/10.1007/978-3-319-40946-7_27

[46] Adar Weidman. 2017. Regular Expression Denial of Service - ReDoS. https://www.owasp.org/index.php/Regular_expression_Denial_of_Service_-_ReDoS

[47] Darrell Whitley. 1994. A genetic algorithm tutorial. *Statistics and Computing* 4, 2 (June 1994), 65–85. https://doi.org/10.1007/BF00175354

[48] Valentin Wüstholz, Oswaldo Olivo, Marijn JH Heule, and Isil Dillig. 2017. Static detection of DoS vulnerabilities in programs that use regular expressions. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '17)*. 3–20. https://doi.org/10.1007/978-3-662-54580-5_1

[49] Xiaodong Yu and Michela Becchi. 2013. GPU acceleration of regular expression matching for large datasets: Exploring the implementation space. In *Proceedings of the ACM International Conference on Computing Frontiers*. Article 18, 10 pages. https://doi.org/10.1145/2482767.2482791