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 implemented 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, 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 (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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference Format:


Yanyan Jiang and Chang Xu are the corresponding authors.

1 Popular regex engines are mostly implemented in this way, e.g., the built-in regex modules in Java [3], Python [15, 27], JavaScript [31], 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, 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]:

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

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:

```python
<--!--!--!--!--!--!--!--!--!--!--!--!--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.

---

2For example, the regex (?![17]$)\(11+\)+\S$ tests whether the expression consists of a prime number of 1s.

3The developer misplaced a (\.+ in the regex, and it can be expanded in $O(2^n)$ many ways in representing a string of length $n$. 
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 development process [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 \(\epsilon\)-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 cannot 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 \(\Sigma\) (e.g., English alphabet and symbols), and a maximal input length \(\ell\), finding

\[
\text{ReDoS}(r, \Sigma, \ell) = \arg\max_{s \in \Sigma^*} \text{RegexMatchTime}_r(s).
\]

The search time \(\text{RegexMatchTime}\) 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 \(\epsilon\)-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.

![Figure 1: Overview of the ReSCue technique for automated ReDoS string generation.](image-url)
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 $\Sigma$. $s(i)$ denotes the $i$-th character in string $s$ ($1 \leq i \leq |s|$), and $(s,i)$ denotes the substring $s(i)s(i+1)\ldots s(j)$. The empty string is denoted by $\varepsilon$. For example, given $s = \text{"ReDoS"}$, $s(3) = \text{"D"}$ and $s(4:5) = \text{"os"}$. Two strings $x$ and $y$ can be concatenated, yielding $x \cdot y$ or simply $xy$. Concatenating $x = \text{"hello"}$ and $y = \text{"world"}$ yields $xy = \text{"helloworld"}$.

A regular expression $r$ is the string representation of a formal language (i.e., a set of strings) $L(r) \subseteq \Sigma^*$. For $r_{\text{hello}} = ^*\text{(\text{"hello\})[a-z]}^*(\text{5)}$, $L(r_{\text{hello}}) = \{\text{"hello"}\}^*$. Given a string $s$ and a regex $r$, the regex pattern matching problem is equivalent to testing 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 $r_{\text{hello}}$.

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 $(v_i,p_i,t_i)$ where $v_i \in V$ is an e-NFA state, $0 \leq p_i \leq |s|$ is a position in $s^i$, 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 $(v,p,t)$ from the matching stack, and pushes $f_v(s,p,t)$ onto the stack (s is the input string). Multiple values of $(v,p,t)$ 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 $(v',p+1,t)$, 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 $(v_1,p,\bot)$ then $(v_2,p,\bot)$ such that when the matching group (Node $v_2$–Node $v_3$) is successfully matched, the matching restarts at Node $v_6$.

2. For Node $v_7$ (repetition), depending on the current repetition count $t$, $(v_7,p,\bot)$ is pushed when $t = 0$ for matching the subsequent parts, otherwise $(v_7,p,t-1)$ 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.

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

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

where $x_i \in V$ is an e-NFA state and $0 \leq p_i \leq |s|$ is a position in the input string denoting the starting position of the next string pattern matching. $(x_i,p_i)$ 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 $r_{\text{hello}}$, the e-NFA shown in Figure 2, and $s_1 = \text{"helloworld"}$ and $s_2 = \text{"helloword"}$, conducting regex matching in Java yields the same traces $\tau_{r_{\text{hello}}}(s_1) = \tau_{r_{\text{hello}}}(s_2)$:

$$\{\langle v_0,0\rangle, \langle v_1,0\rangle, \langle v_2,0\rangle, \langle v_3,0\rangle, \langle v_4,5\rangle, \langle v_5,5\rangle, \langle v_6,0\rangle, \langle v_7,0\rangle, \langle v_7,1\rangle, \langle v_5,2\rangle, \langle v_7,2\rangle, \langle v_5,3\rangle, \langle v_7,3\rangle, \langle v_5,4\rangle, \langle v_7,4\rangle, \langle v_5,5\rangle, \langle v_6,5\rangle\}$$
For two traces $\tau_1(s_1) = \{(x_i, p_i)\}$ and $\tau_2(s_2) = \{(x'_i, p'_i)\}$, we consider $\tau_1(s_1)$ and $\tau_2(s_2)$ equivalent in regex matching if $|\tau_1(s_1)| = |\tau_2(s_2)|$ and $x_i = x'_i$ for all $1 \leq i \leq |\tau_1(s_1)|$. Therefore, $s_1 = \text{"hello world"}$ and $s_2 = \text{"hellopanda"}$ have equivalent traces in matching against $\tau_{\text{hello}}$.

Furthermore, we define the effective prefix $EP(s) = s(1:k)$ where $k = \max (n)$, and the redundant suffix $RS(s) = s(k+1:|s|)$ is the rest part of $s$. Obviously, $s = EP(s)RS(s)$. 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 “hello world” and “hellopanda” are “hello”.

Finally, a byproduct of regex compilation is a set of string slices $\Sigma_{\text{slice}}$, which contains substrings of $r$ used in the regex matching procedure. For example, $\Sigma_{\text{slice}}$ for $\tau_{\text{hello}}$ is (“hello”) (corresponding to the vertex $v_3$). Strings in $\Sigma_{\text{slice}}$ 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-like 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

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

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 $f(s, P)$.

#### 3.3.2 Offspring Generation

Given a population $P$, its offspring are generated by crossing over and mutating the strings in $P$.

The crossover operator $\chi(s_1, s_2)$ interlaces the effective prefixes of $s_1$ and $s_2$. In particular, $\chi(s_1, s_2)$ randomly splits $EP(s_1) = x_1y_1, EP(s_2) = x_2y_2$, and returns

$$\chi(s_1, s_2) = \{x_1y_1RS(s_2), x_2y_2RS(s_1)\}.$$  

Intuitively, the effective prefixes $EP(s_1)$ and $EP(s_2)$ 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 $\mu \in \Sigma \cup \Sigma_{\text{slice}}$ is a random string (either a random character in $\Sigma$ or a random string constant in the regex’s corresponding automaton):

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

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

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

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

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

The seeding phase tries to find a population $P_{\text{seed}}$ of strings that covers as many e-NFA states as possible and basically ignores the search time\(^6\).

The initial population of the seeding phase $P_0$ consists of constant strings $\Sigma_{\text{slice}}$ in the automaton plus some random strings in the alphabet of $\Sigma$.

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

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

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

### 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 $P_0 = P_{\text{seed}}$.

---

\(^6\)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.
The pumping phase selects such incubated string of the highest cost-effective ratio
\[ s^* = \arg\max_{s \in \mathcal{P}} f_{\text{incub}}(s) \]
and pumps it into a highly effective input string for slowing down the regex matching process. The idea (and the name) has been inspired by the Pumping Lemma, which is also used in the static detection of ReDoS vulnerabilities \cite{25}.

PUMPING LEMMA \cite{33}. For any regular language \( L \), there is a \( p \geq 0 \) such that any string \( w \in L \) of \( |w| \geq p \) can be written as \( w = xyz \), where \( y \neq \varepsilon \) and the strings \( xy, xyy, xyyyz, \ldots \) 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 \( s^* \) 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 \( f_{\text{trim}} \) 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 sub-strings in \( s(i:j) \) and tries to pump one occurrence 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 \( \tau_i(s) \).
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 \( \Sigma_{\text{slice}} \) are extracted from the 51 ice nodes.

The matching trace \( \tau_i(s) \) 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 \( (x_i,p_i) \) and maintain the pivot index \( k \), which splits the effective prefix and the redundant suffix for string \( s \), i.e., \( k = \lfloor |EP(s)| \rfloor \). 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 \cite{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/.
We collected three sets of regexes (29,088 regexes in total) from the ReScue RegLib [35], Snort [35], and GitHub open source projects, and all detected vulnerabilities are listed in Table 1, and the following techniques are compared:

- **(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 $10^6$ 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 $10^6$ 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>
<thead>
<tr>
<th>Name</th>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RegLib</td>
<td>2,992</td>
<td>Online regex examples from RegExLib.com</td>
</tr>
<tr>
<td>Snort</td>
<td>12,499</td>
<td>regexes extracted from the Snort NIDS for data packet filtering</td>
</tr>
<tr>
<td>Corpus</td>
<td>13,597</td>
<td>regexes from scraped python projects</td>
</tr>
</tbody>
</table>
Table 2: Supported regex extensions of each evaluated technique.

<table>
<thead>
<tr>
<th>Features</th>
<th>ReSCUE</th>
<th>SlowFuzz</th>
<th>RXXR2</th>
<th>Rexploiter</th>
<th>NFAA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set operations [a-z]&amp;[^aeiou]\</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Lookarounds (?=) (!?) (?&lt;=) (?&lt;!)</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Backreferences ()\1</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Non-capturing groups (?)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Named groups</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Atomic groups (?&gt;)</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Conditionals (?(?1))</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Greedy quantifiers {m,n} {n} {m}</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Lazy quantifiers ?? *? +? (?)?</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Possessive quantifiers + * ++ {n}+</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
</tbody>
</table>

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).

<table>
<thead>
<tr>
<th>Technique</th>
<th>#Vul.</th>
<th>#FP</th>
<th>TP Rate</th>
<th>Avg Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReSCUE</td>
<td>186 (82%)</td>
<td>-</td>
<td>-</td>
<td>0.6128</td>
</tr>
<tr>
<td>SlowFuzz</td>
<td>101 (44%)</td>
<td>-</td>
<td>-</td>
<td>0.5965</td>
</tr>
<tr>
<td>RXXR2</td>
<td>125 (55%)</td>
<td>80</td>
<td>61%</td>
<td>0.0025</td>
</tr>
<tr>
<td>Rexploiter</td>
<td>30 (13%)</td>
<td>-</td>
<td>-</td>
<td>0.4073</td>
</tr>
<tr>
<td>NFAA</td>
<td>0 (0%)</td>
<td>714</td>
<td>N/A</td>
<td>2.1546</td>
</tr>
<tr>
<td><strong>Summary</strong></td>
<td><strong>227 (100%)</strong></td>
<td><strong>-</strong></td>
<td><strong>-</strong></td>
<td><strong>-</strong></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Variant</th>
<th>#Vul.</th>
<th>Avg Time (s)</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReSCUE without seeding</td>
<td>152 (67%)</td>
<td>0.5061</td>
<td>57.84%</td>
</tr>
<tr>
<td>without incubating</td>
<td>99 (44%)</td>
<td>0.1273</td>
<td>89.76%</td>
</tr>
<tr>
<td>without pumping</td>
<td>49 (22%)</td>
<td>0.5705</td>
<td>90.31%</td>
</tr>
</tbody>
</table>

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 an 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.
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 by the respective developer. Some confirmed cases and (simplified) code, generated ReDoS strings are discussed as follows.

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:

```
hasStrikeSyntax(text) {
    return strikeRegex.test(text);
}
```

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.

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:

```
if (\s?\[~~\])(.*\[\s
\]*.*)*\[~~\])$/;
```

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 backtrack ing algorithm and have not considered carefully their underlying formalization [4, 35]. This fact, on one hand, makes e-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 \( \langle \text{prefix}, \text{pumpable string}, \text{suffix} \rangle \). To use this pattern, they convert given rexges 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-like 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.