Towards an Effective Method of ReDoS Detection for Non-backtracking Engines

Weihao Su, Hong Huang, and Rongchen Li, Key Laboratory of System Software (Chinese Academy of Sciences) and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences; University of Chinese Academy of Sciences; Haiming Chen, Key Laboratory of System Software (Chinese Academy of Sciences) and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences; Tingjian Ge, Miner School of Computer & Information Sciences, University of Massachusetts, Lowell

https://www.usenix.org/conference/usenixsecurity24/presentation/su-weihao

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.
August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.
Towards an Effective Method of ReDoS Detection for Non-backtracking Engines

Weihao Su†‡∗ Hong Huang†‡∗ Rongchen Li†‡∗ Haiming Chen†∞ Tingjian Ge¶

†Key Laboratory of System Software (Chinese Academy of Sciences) and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
‡University of Chinese Academy of Sciences
¶Miner School of Computer & Information Sciences, University of Massachusetts, Lowell

Abstract
Regular expressions (regexes) are a fundamental concept across the fields of computer science. However, they can also induce the Regular expression Denial of Service (ReDoS) attacks, which are a class of denial of service attacks, caused by super-linear worst-case matching time. Due to the severity and prevalence of ReDoS attacks, the detection of ReDoS-vulnerable regexes in software is thus vital. Although various ReDoS detection approaches have been proposed, these methods have focused mainly on backtracking regex engines, leaving the problem of ReDoS vulnerability detection on non-backtracking regex engines largely open.

To address the above challenges, in this paper, we first systematically analyze the major causes that could contribute to ReDoS vulnerabilities on non-backtracking regex engines. We then propose a novel type of ReDoS attack strings that builds on the concept of simple strings. Next we propose EVILSTRGEN, a tool for generating attack strings for ReDoS-vulnerable regexes on non-backtracking engines. It is based on a novel incremental determinisation algorithm with heuristic strategies to lazily find the k-simple strings without explicit construction of finite automata. We evaluate EVILSTRGEN against six state-of-the-art approaches on a broad range of publicly available datasets containing 736,535 unique regexes. The results illustrate the significant efficacy of our tool. We also apply our tool to 85 intensively-tested projects, and have identified 34 unrevealed ReDoS vulnerabilities.

1 Introduction
Regular expressions (regexes) are widely used in various fields of computer science such as software engineering, network security, string processing, and databases [9, 18, 21, 22, 45, 66], and are supported natively or via libraries in most modern programming languages [35]. Recent studies have reported [17,21,76] that 30-40% of Java, JavaScript, and Python projects use regexes. Despite the popularity of regexes, studies have shown that Regular expression Denial-of-Service (ReDoS) attacks are a widespread and serious security problem [10,21,45,69], where attackers use malicious inputs (i.e., attack strings) to trigger super-linear worst-case matching time (e.g., quadratic or exponential time in the length of the input string) [21]. Further, the threat of ReDoS has a growing trend in recent years, as evidenced by multiple studies (e.g., [38,67]).

Much research has been done on the detection of ReDoS vulnerable regexes. However, current methods have focused mainly on backtracking regex engines, leaving the problem of ReDoS vulnerability detection on non-backtracking regex engines largely open.

Basically, regex matching algorithms can be roughly categorized into two types—those based on backtracking search [68] and those based on finite automata (i.e., non-deterministic finite automaton (NFA) or deterministic finite automaton (DFA), or both). Backtracking approaches are usually simple to implement and easily extensible to support non-regular features. However, the downside is that the worst-case time complexity can be exponential in the length of the input caused by backtracking. Backtracking engines are used in, e.g., .NET, Python, Perl, PHP, Java, JavaScript, and Ruby.

On the other hand, non-backtracking automata-based approaches, which are based on the classical theory of automata, are usually faster, yet harder to implement. Due to their high performance capabilities, automata-based matchers are employed in many modern regex engines, e.g., RE2, and the engines in Go, Rust, grep, Hyperscan, and SRM.

Since non-backtracking matchers are widely used in modern regex engines, and performance-critical industrial applications such as Network Intrusion Detection Systems (NID-Ses) [25] and Credential Scanning [58] use non-backtracking engines, it is important to systematically study the problem of ReDoS detection for non-backtracking engines.

Currently, there has only been one line of work, i.e. GadgetCA [74], in this direction, which focused on the bounded repetition (or bounded counting) operator of regexes. It ex-
experimentally showed that regexes using this operator have the potential to cause ReDoS attacks on non-backtracking matchers. However, the model used in GadgetCA is sound only on a limited subclass of regexes with counting. For example, \(*\text{aa}*\) and \(*\text{d[0,100]}*\) are not within this subclass [50]. For such regexes, GadgetCA may return wrong results. This suggests that the usability of the method proposed in [74] is quite limited. In other words, the problem of ReDoS detection on non-backtracking engines caused by the full class of regexes with counting remains unsolved.

Example 1. \texttt{classid}=\$s^*\text{x22}\text{x27}[A-Za-z\:\\{-]\{0,42\} (\text{x26}\text{x23}\text{d[2,3]}\text{x3B})\{2\}^\{\text{nx22}\text{x27}\}^\{0,25\}\}^5\$

Besides, the use of 50MB large-sized inputs as adapted in [74] to trigger ReDoS vulnerabilities in their experiments is impractical and less likely to be exploitable, as it is rare to encounter such a large input in real-world scenarios. This reflects the relatively weak ability of the strings generated by their detector to trigger ReDoS vulnerabilities, as shown in our experiments, e.g., Table 5 and Table 7.

Moreover, there are other potential factors besides bounded repetition that could contribute to ReDoS attacks for non-backtracking engines, which, however, have not been considered in the existing literature. Consider the regex in Example 2, which is used in Regex101 [4] to match cardiac surgery terms. It is \texttt{bounded_counting-free}, i.e., does not have any bounded repetition operators, yet still has a descriptional complexity blow-up up to 1,087,951 DFA states in RE2.

Example 2. \texttt{.\*cc\*.(cor\*ang\*\text{heart\*\text{icd\*\text{pace\*\text{cardi\*\text{lhc\*\text{aortic\*\text{rhc\*\text{dual\*\text{lead\*\text{cabg\*\text{trans\*\text{echo}}}}}}}}}}}

Yet GadgetCA fails to expose the vulnerability: The string generated by GadgetCA only forces RE2 to construct 1,027 DFA states and run for less than 0.17 seconds.

To address the aforementioned challenges, our research entails a systematic analysis of the diverse causes of ReDoS attacks produced by non-backtracking engines, and a search for effective techniques to analyze ReDoS attacks and generate attack strings for non-backtracking engines. Herein, we summarize the main findings of our research.

(1) \textbf{Causes}. By distinguishing whether a matcher generates the DFA in advance or not, non-backtracking matchers can be categorized into online and offline DFA matchers. Since constructing a DFA may explode doubly exponentially, the majority of non-backtracking matchers opt for online DFA matchers. We target this type of matchers and identify two classes of causes that could contribute to ReDoS vulnerabilities: (i) reaching the worst-case time complexity of NFA and online DFA matchers as well as various matching functions; and (ii) blow-up in DFA states, including determinisation blow-up, counting, and discrete character classes (§4). Considering the examples above, the ReDoS vulnerability of the regex in Example 1 is mainly resulted from the DFA state blow-up due to nested counting, while for Example 2, it is due to the use of discrete character classes.

(2) \textbf{Techniques}. At the heart of ReDoS detection are the attack strings—indeed, a regex is only considered vulnerable if such a string exists. Theoretically, finding an attack string is to find a string that triggers the worst-case complexity on regex engines. However, finding such a string is an arduous problem. Similar concerns are raised by Turoňová et al. [74]: “Such a text is, however, also highly specific and the probability of generating it randomly is low.” To effectively deduce attack strings for non-backtracking engines, our key idea is the use of the \textit{simple strings} and solving the \texttt{\textsc{k-Simple String}} problem for regexes (defined in §5). Intuitively, a simple string corresponds to an acyclic accepting path that is composed of distinct states and contains only one final state in the DFA. Thus using simple strings as attack strings will tend to force engines to construct more new DFA states to match them. We then model the attack string generation problem as the \texttt{\textsc{k-Simple String}} problem, which finds a simple string of length at least \(k\) (i.e., a \(k\)-simple string), for a user-controlled parameter \(k\). Due to the hardness of \texttt{\textsc{k-Simple String}} (Theorem 1), to effectively accelerate the solving, we propose a novel incremental determinisation algorithm with heuristic strategies to lazily find the \(k\)-simple strings \textit{without} explicit construction of finite automata.

Based on the ideas above, we propose \texttt{\textsc{EvilStrGen}}, a tool for generating attack strings for ReDoS vulnerable regexes targeting non-backtracking engines. \texttt{\textsc{EvilStrGen}} can effectively handle the full class of regexes used in non-backtracking engines and generate attack strings that cause severe ReDoS attacks on non-backtracking engines, while reducing the average length of attack strings used in GadgetCA by two orders of magnitude. Considering the Examples 1 and 2, \texttt{\textsc{EvilStrGen}} generates strings of length \(100kB\) that run in 1.61 seconds and 2.26 seconds respectively on RE2, thereby the vulnerabilities are successfully detected.

We assess \texttt{\textsc{EvilStrGen}} by comparing it to six state-of-the-art approaches on 16 regex engines (comprising 8 non-backtracking engines and 8 backtracking engines). Our analysis covers a broad range of publicly available datasets from different sources that contain 736,535 unique regexes. The results illustrate the significant efficacy of \texttt{\textsc{EvilStrGen}}.

The main contributions of this paper are listed as follows.
Novelty. We introduce a practical ReDoS detection and attack string generation method for non-backtracking engines which produces a novel form of attack strings that are effective and concise for vulnerable regexes.

Theoretical Analysis for ReDoS. We provide a comprehensive computational and descriptive complexity analysis of the causes of ReDoS vulnerabilities on non-backtracking engines in §4.

Regex Constraint Solving. We propose the $k$-SIMPLE STRING problem to model the attack string generation problem, which has not been considered by any of the regex solvers. We devise a novel incremental algorithm extended with language-theoretic heuristics to solve this problem and generate attack strings in §5.

Effectiveness and Practicality. We give a comprehensive empirical study of our tool, comparing with the current state-of-the-art ReDoS detection tools in §6. The effectiveness of our tool is demonstrated by the results. We also apply our tool to 85 intensively-tested projects, and have identified 34 unrevealed ReDoS vulnerabilities.

2 Preliminaries

We will introduce the necessary formal definitions. Let $\Sigma$ be a finite alphabet of symbols (or characters). Words (i.e., strings) are finite sequences over $\Sigma$, and languages are sets of words. The empty word and the empty language are denoted by $\epsilon$ and $\emptyset$, respectively. The size of a word $w$, denoted $|w|$, is the number of occurrences of symbols in $w$.

Regular Expression (Regex). Regexes are defined by the syntax $E ::= \epsilon | \emptyset | a | C | E|E | EE | (E) | E[m,n] | E[m,n]? | \backslash b | \text{\textbackslash} B | \text{\textbackslash} s$, where $a \in \Sigma$, $m \in \{0, 1, \ldots\}$, $n \in \{1, 2, \ldots\} \cup \{+\infty\}$, and $m \leq n$. $C \subseteq \Sigma$ are character classes, such as \text{\textbackslash}p (Greek), which denotes all symbols from the Greek alphabet. $E|E$ and $EE$ (i.e., by juxtaposition) denote the alternation and concatenation of regexes, respectively. $(E)$ denotes a capturing group, which stores the submatch matched by $E$ for extraction, replacement, etc., e.g., $a(b)$ matches an input “ab”, and $(b)$ stores “b”. $E[m,n]$ and $E[m,n]?$ denote the greedy and lazy quantifiers respectively, i.e., the counting operators. For an input “aa”, $a^{[1,2]}$ repeat matching $a$ as many times as possible to match “aa”, while $a^{[1,2]}?$ prefer fewer times to match “a”. Anchors include word boundary \text{\textbackslash}b, non-word boundary \text{\textbackslash}B, start-of-line anchor $\text{\textbackslash}S$, and end-of-line anchor $\$$. Anchors do not consume characters, but specify the non-character context. For the input “ab”, $\text{\textbackslash}b$ specifies positions 0 and 2, whose one side is a word and the other side is not. $\text{\textbackslash}B$ specifies position 1, whose both sides are words (or non-words). And $^*$ matches position 0, while $\text{\textbackslash}S$ matches position 2. Notice $E^*$, $E^+$, $E[m]$, $E[m]$? are abbreviations of $E|\epsilon$, $E^{[0,\infty]}$, $E^{[1,\infty]}$, $E[m]$ and $E[m,\infty]$ respectively. The language of a regex $E$, denoted as $L(E)$, is the set of all strings accepted by $E$.

In regex engines the matching behavior for regexes is specified through the matching functions. Intuitively, for a regex $E$ and an input string $w$, the partial matching solves the problem of sub-string matching by a regex (i.e. whether $w \in L(E|\epsilon)$), which can be implemented by prepending $\Sigma^*$ to the left of $E$ [61]; while full matching solves the regex membership problem (i.e. whether $w \in L(E)$); more details can be found in §4.1. Disambiguation rules [70] are used in engines to guarantee a unique match, e.g. Perl-style policy [3] assigns the highest priority to the left-most matches and POSIX [70] prefers the longest matches. For example, $(a|aa)^*$ matches “a” from an input “aa” in Perl style and “aa” in POSIX style.

Finite automata. An NFA is a quintuple $M = (Q, \Sigma, \delta, s, F)$, where $Q$ is a finite set of states, $\Sigma$ is an alphabet, $\delta \subseteq Q \times \Sigma \cup \{\epsilon\} \rightarrow \varphi(Q)$ is a transition function where $\varphi(Q)$ denotes the power-set of $Q$, $s \in Q$ is the starting (or initial) state, and $F \subseteq Q$ is a set of final states. The automaton is deterministic (DFA) if $\delta \subseteq Q \times \Sigma \rightarrow Q$. NFAs are denoted as $\epsilon$NFAs if the transitions can be labeled $\epsilon$, and $\epsilon$-free NFAs otherwise.

Next we discuss methods for transforming NFAs or regexes into DFAs, which is a basis of the DFA matchers. The standard method for NFA-DFA transformation is the subset construction [62]. For regex-DFA transformation, McNaughton and Yamada [57] proposed a construction ($M_{\text{regex}}$), which is used in Hyperscan [77] whenever possible. Later, another construction (which we denoted as $M_{\text{grep}}$) was implemented and described by Aho in [6], which is still being used by GNU grep [31]. $M_{\text{grep}}$ is always smaller than or equal to $M_{\text{MY}}$ [13].

Non-backtracking Regex Engines. Normally, modern non-backtracking regex engines are complex software systems, which primarily consist of DFA matchers, also possibly NFA matchers or even backtracking matchers, e.g. a backtracking matcher is used in grep to support backreferences. In this paper, we aim to find vulnerabilities for industrial-strength regex engines, which are rather generalized as opposed to those specialized to process some subclasses of regexes, e.g. [40, 43, 50]. From now on, the term regex engine(s) refers to generalized regex engine(s).

The classification of non-backtracking engines is shown in Table 1. According to the type of DFA matchers, we categorize non-backtracking engines into (i) $\epsilon$NFA-based engines, such as RE2 [34], the regex engines in Go [32], Rust [23], as well as T DFA [46] and RE2C [73]; (ii) position-based engines, which are based on $M_{\text{grep}}$, e.g. grep [31], awk [30] and sed [29] or $M_{\text{MY}}$ such as Hyperscan [77]; (iii) derivative-based engines, such as OCaml-re [75], SRM [64] and NonBacktracking [60], by extending Brzozowski’s derivatives [14], whose

3Generalized regex engines can handle the unrestricted class of regexes while specialized regex engines cannot. Though in fact, most such engines put restrictions on regexes for practical considerations, e.g. the upper bound of countings.

To differentiate the two implementations written in C#, we write C#n for NonBacktracking, and C#h for the backtracking implementation.
states are called ACI-dissimilar derivatives.

Based on whether a matcher generates the DFA before processing the input string, DFA matchers can be further categorized into matchers using ahead-of-time (offline) determinisation or just-in-time (online) determinisation (i.e., offline/online DFA matchers). Offline DFA matchers process the input strings in linear time once the DFAs are constructed [6]. However, due to the average performance issue resulted from the worst-case doubly exponentially large DFAs [27], they are rarely used in practice. In contrast, online DFA matchers construct the DFAs lazily: by each symbol in the input, at most one DFA state and transition are constructed and recorded.

**RedoS.** We give the formal definition of a regex to be RedoS vulnerable on a regex engine:

**Definition 1.** A regex $E$ is RedoS vulnerable on a regex engine $M$ if and only if there exists an input word $w$, for which $M$ can not decide whether $w \in L(E)$ in time $O(|w|)$. The word $w$ is called an attack string for $E$ on $M$.

Definition 1 is based on the observation that RedoS vulnerabilities are engine dependent, i.e., whether regexes are vulnerable is contingent on the engine that executes them.

Investigation on backtracking engines has shown that while the degree of the RedoS vulnerabilities can be exponential, most of the RedoS vulnerable regexes are in polynomial degree in the wild [21]. For non-backtracking engines, given a regex, the degree of the RedoS vulnerabilities is polynomial. In this paper, we use both machine-dependent and independent criteria to evaluate RedoS vulnerabilities.

### 3 Overview

In this section, we start with a walkthrough of the procedure of RedoS detection in EvilSTRGen, and then analyze some RedoS vulnerable regexes to show how different structures and matching functions result in vulnerability.

**RedoS Detection Framework.** Figure 1 shows how to detect RedoS vulnerabilities with our incremental determinisation algorithm to generate candidate attack strings, i.e., strings without being verified to be valid, and verify them in EvilSTRGen. The input of EVILSTRGen includes the regex, the expected attack string length, the matching function and the engine. The tool first preprocesses the input, e.g., for the functions using partial matching, a $Σ$ is prepended to the input regex, etc., in step 1 of Figure 1. Next, the tool calls the $σ$-Simple String solver INC_DET\textsuperscript{σ} to generate RedoS attack strings byte-wise with incremental determinisation based on the DFA in the corresponding engine. The search for $σ$-simple strings uses two complete and an incomplete strategies to optimize its efficacy. In step 2, the first strategy exploits the power of nondeterminism in automata theory to select the symbols that result in the largest DFA states in attack strings. The second strategy, i.e., step 3, mimics the Perl-style disambiguation rules to select the DFA state with the highest matching priority. When a considerable number of conflicts (see §5.2) is detected, step 4 uses an incomplete non-chronological backtracking to prune the state space. When the solver terminates, it outputs a $σ$-simple string or a currently longest simple string in step 5. Next in step 6, considering matching function, anchors and the size of the simple string, EvilSTRGen decides to go to the validation, or to step 7 to search repeatedly to obtain a candidate attack string of proper length by appending outputs to the former results. Finally in step 8, a validator is used to verify the string according to the criteria for RedoS on corresponding engines, and either reports the vulnerability with the effective attack string or claims the regex to be safe.

**Motivating Examples.** To illustrate how EvilSTRGen can find RedoS vulnerabilities for non-backtracking engines, we list some RedoS vulnerable regexes in Table 2. The time used to match the regexes with candidate attack strings of 100kB generated by EvilSTRGen and GadgetCA [74] under different matching functions in RE2 is shown in milliseconds.

The 1st and 2nd regexes have exponential DFA states. Both of them having over a million DFA states in RE2. However, GadgetCA cannot expose an “enough” DFA states, thus failing to report the RedoS vulnerabilities. In contrast, EvilSTRGen successfully detects the vulnerabilities, showing the effectiveness of the strings generated by EvilSTRGen.

The 3rd and 4th regexes have nested counting, which are not supported by GadgetCA. When the counting is forbidden to be nested, the doubly exponential state complexity of counting regexes [27] is never reached. Thus the RedoS detectors that only able to handle subclasses of counting regexes may cause omission of RedoS vulnerabilities or even errors.

The 5th regex has at most 651,608 DFA states in RE2, which illustrates that regexes with only unbounded countings can also be RedoS vulnerable. The 6th regex is bounded_counting-free, it is RedoS vulnerable mainly due to the nondeterminism and the discrete representation of the meta-character “.”. These facts reveal that bounded countings

---

<table>
<thead>
<tr>
<th>Engine</th>
<th>Position</th>
<th>Derivative</th>
<th>DFA States</th>
</tr>
</thead>
<tbody>
<tr>
<td>Online</td>
<td>RE2, Go, Rust</td>
<td>grep, awk, sed, Hyperscan</td>
<td>SRM, C#N</td>
</tr>
<tr>
<td>Offline</td>
<td>T DFA, RE2C</td>
<td>–</td>
<td>OCaml-re</td>
</tr>
</tbody>
</table>

---

**Table 1: Non-backtracking regex engines, categorized.**

---

**Figure 1:** The system architecture of EvilSTRGen.
are not the only Achilles'-heel of non-backtracking engines. Under full matching, the $7^{th}$, $8^{th}$ and $9^{th}$ regexes denote finite languages, i.e., only match strings of bounded length, and the engine terminates at the longest string in each language at most to match any input. Due to higher nondeterminism under partial matching, RE2 is slowed down by several orders of magnitude than full matching. Furthermore the $8^{th}$ regex also suffers from the discrete representation of multi-byte character classes in UTF-8 encoding, where $[\{p(L) \} \cup \{p(N) \}]$ requires 1,344 states in RE2. The implementation of GadgetCA does not differentiate matching functions or encoding, which results in the omission of detection.

The $9^{th}$ regex and its variants are used as the running example to explain the concepts and algorithms in this paper.

### 4 Analysis of Factors for ReDoS

This section gives an analysis of the major factors that could contribute to ReDoS vulnerabilities for non-backtracking engines, which is a prerequisite for effectively solving the ReDoS problem. From a theoretical point-of-view, we identify two classes of causes for ReDoS: Reaching the worst-case time complexity of matchers and matching functions (§4.1) and blow-up in DFA states due to nondeterminism and succinctness of counting or character classes (§4.2). Notice that the causes are not isolated or finitely enumerable into “patterns”, but entangled and interactive. To better present the effect of different causes in ReDoS, the results using the running examples as inputs in RE2 are listed in Table 3.

#### 4.1 Time Complexity Analysis of Matchers and Matching Functions

In this section, we denote the number of NFA states as $m$ and the size of input as $n$.

**Online DFA and NFA matchers.** We first analyze the worst-case time complexity of online DFA and NFA matchers. **Online DFA matchers are unsafe.** To process each symbol from the input, at most one state and transition is constructed and recorded in online DFA matchers, i.e., $O(n)$ states in total, where each matching function has a time cost $O(m^2)$ [16], which can also be optimized into quasi-linear time w.r.t. $m$, e.g. using the disjoint-set data structure [72]. If a state is already found to be matched, it processes each symbol in $O(1)$. Using this lazy evaluation technique, online DFA matchers perform well on safe inputs [44]. However the worst-case time complexity of online DFA matchers is at least $O(nn)$, plus an exponential cost for DFA cache conflict testing [16].

The worst-case time complexity of an NFA matcher is $O(m^2 n)$ as described in [42]. In [6,20], it is considered $O(nn)$, when implemented properly, e.g. using the disjoint-set data structure. Evidently the worst-case performance of online DFA matchers is worse than NFA matchers, where the latter is considered the slowest matcher in Rust [23]. Thus reaching the worst-case time complexity of online DFA matchers is a direct factor for ReDoS vulnerabilities.

**Falling back to NFAs.** Non-backtracking engines usually set bounds for the DFA cache for online DFA matchers, e.g. $\mathbb{C}m$ keeps at most 10,000 ACI-dissimilar derivatives in cache and Hyperscan’s threshold is 16,384 distinct position subsets. When the bound is reached (some are reached twice [20]), or the online DFA is considered slow, engines dump the cache and fall back to NFA matchers (or even backtracking matchers, e.g. grep). Recall that the worst-case time complexity of an NFA matcher is $O(m^2 n)$ or $O(nn)$, taking the exponential size of $m$ w.r.t. the regexes in the worst case [48] into account, falling back to NFA makes non-backtracking engines unsafe, which contributes to ReDoS vulnerabilities.

**Matching Functions.** Matching functions also have an impact on ReDoS. There are two major algorithms to implement partial matching in DFA-based regex engines: to convert a regex $E$ to $\Sigma^E$ [61], e.g. RE2, $\mathbb{C}m$, or repeating DFA match starting from each symbol in the input, e.g. grep, awk.

The first approach assures the search in texts to be performed with the same time complexity as full matching.
ever this modification brings nondeterminism into the regex: For our running example under ASCII encoding, the number of DFA states for $\langle W \setminus DS \rangle^{[5]}$ is enlarged 5.67x. This also contributes to ReDoS vulnerabilities, as demonstrated in Table 2.

The second approach repeats the search with DFAs for $O(n)$ times, making the worst case time complexity $O(mn^2)$ at least, which becomes a non-negligible factor for ReDoS vulnerabilities. Indeed, ReDoS attacks on regexes with the Starting-with-Large-Quantifier pattern [52] for backtracking engines is directly connected to this algorithm.

Extracting or replacing submatch of the subregexes in capture groups as well as the find or replace all functions use partial matching to locate the matched sub-strings, where engines repeat partial matching for $O(n)$ times, leading to the worst case time complexity at $O(mn^2)$ at least. Those may also contribute to ReDoS vulnerabilities, e.g. the submatch extraction of RE2 can be 9.13x slower than Java’s backtracking engine on ordinary inputs [36].

4.2 Descriptional Complexity Blow-up

The descriptional complexity of DFA for regexes has direct impact on the computational complexity of algorithms used in non-backtracking engines, which contributes to ReDoS.

Descriptional complexity analysis for ReDoS studies the size of automata to represent regexes, which is divided into the following topics: Transformational state complexity studies the complexity of transformations among regexes, NFAs and DFAs, while operational state complexity focuses on the state complexity of regex operators.

Determinisation Blow-up. Among transformational state complexity results, determinisation blow-up studies the NFA to DFA conversion due to nondeterminism: For an $m$ state NFA, the subset construction producing a DFA with $2^m$ states is proven worst-case optimal [62]. In practice, authors of [81] observed that regexes having exponential DFA states cause high costs both in time and storage in deep packet inspection applications. So it is a factor for ReDoS vulnerabilities.

We list some forms of regexes that have linearly sized NFAs, yet exponentially sized minimal DFAs in Table 4, which we suggest the developers to avoid in practice. Part of those is collected (and translated from automata) from the literature (where the references are given), while others are newly discovered variants. In each form, $\alpha$, $\beta$ and $\gamma$ represent disjoint subsets of $\Sigma$. Notice the minimal DFAs in this table are not completed with sink states.

Example 3. The $1^{st}$ vulnerable regex in Table 2 is selected from a real-world project in NuGet [2], which shows the impact on ReDoS due to descriptional complexity from determinisation blow-up in the wild. Consider the first example in Table 4, by substituting $\alpha$ by “$a$”, $\beta$ by “[^a]/[^a]”, and assigning $k$ as 300, $s.a.1^{[300]}$ (i.e., the $1^{st}$ regex in Table 2, considering that $a[[^a]/[^a]] = .)$ is obtained.

<table>
<thead>
<tr>
<th>Regex</th>
<th>Reference</th>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha\beta^*\alpha(\alpha\beta)^k$</td>
<td>[6, 59]</td>
<td>$2^k+1$</td>
</tr>
<tr>
<td>$\alpha\beta^*\alpha(\beta\alpha)^k$</td>
<td>[82]</td>
<td>$2^k+1$</td>
</tr>
<tr>
<td>$\alpha\beta^*\alpha(\beta\alpha)^{k+1}$</td>
<td>[49, 65]</td>
<td>$2^k+2 - 2$</td>
</tr>
<tr>
<td>$\alpha\beta^<em>\alpha(\alpha\beta)^k\alpha(\alpha\beta)^</em>\alpha$</td>
<td>[24]</td>
<td>$2^{k+1} + 1$</td>
</tr>
<tr>
<td>$\alpha\alpha(\beta\alpha)^k\alpha(\alpha\beta)^*$</td>
<td>[24]</td>
<td>$2^{k+2} + 2^{k-1} - 1$</td>
</tr>
<tr>
<td>$\alpha(\beta\alpha)^k\alpha(\alpha\beta)^*$</td>
<td>[51]</td>
<td>$2^{k+1} + 2$</td>
</tr>
<tr>
<td>$\alpha(\alpha\gamma)^k\alpha(\beta\gamma)^\alpha(\beta\gamma)^\alpha$</td>
<td>[12]</td>
<td>$2^{k+2} - 1$</td>
</tr>
<tr>
<td>$\alpha(\alpha\beta)^k\alpha(\alpha\beta)^*$</td>
<td>New variant</td>
<td>$2^{k+1} - 1$</td>
</tr>
<tr>
<td>$\alpha(\alpha\beta)^k\alpha(\alpha\beta)^*$</td>
<td>New variant</td>
<td>$2^{k+1} + 2^{k-2}$</td>
</tr>
<tr>
<td>$\alpha(\alpha\beta)^k\alpha(\alpha\beta)^*$</td>
<td>New variant</td>
<td>$2^{k+1} + 2^k + k - 2$</td>
</tr>
</tbody>
</table>

Next we discuss the impact on DFA size due to the succinctness of regex operators.

Countings. In non-backtracking engines, counting regexes are unfolded into their semantically equivalent bounded counting-free forms (some are unfolded lazily [60, 64]). However those unfolded regexes have exponential sizes in the worst case. Consider the running example, due to the unfolding, under partial matching and UTF-8 encoding, the DFA state number of $\langle W \setminus D \rangle^{[5]}$ is 1030× bigger than that of $\langle W \setminus D \rangle$, while the size of $\langle W \setminus D \rangle^{[5]}$ is only 1× larger than $\langle W \setminus D \rangle$.

Bounded counting has been considered in [74] for causing ReDoS vulnerabilities. Yet the automata that underlie the generation algorithm of GadgetCA [74] may change (i.e., over-approximate) the language of a regex, which can result in incorrect (e.g., false negative) output. Furthermore, unbounded counting in the form of $E_{(m)}$ also contributes to ReDoS vulnerabilities: see the $5^{th}$ regex in Table 2.

Nested counting could make online DFA matchers to construct DFA states in exponential size w.r.t. the regex: Consider a DFA for $a \{0.5\} \{0.5\} \{0.5\}$, $\delta(s, a) = \{(a_2, a_3, ..., a_n)\}$, requiring at least $O(k^n)$ time to compute. Notice this is only triggered by a single symbol “$a$”.

Nesting of counting and determination blow-up jointly lead to doubly-exponentially sized DFAs: For example, Gelade proved the minimal DFA of $E_n = \langle a\{b\} \{a\{b\} \{2^k\} \rangle$ has $2^{2k}$ states [27], where $E_n$ is derived from the first example in Table 4. This can be achieved by an $n$-nested counting, i.e. $E_n$ equals to $(a\{b\} \{a\{2^k\} \{2^k\} \rangle \leq 2^{2k-1}$.

Next we analyse the impact of character classes on the descriptional complexity of regexes and ReDoS.

Discrete Representations of Character Classes. Resulted from the variable length encoding of UTF-8, the Unicode to UTF-8 conversion on non-backtracking regex engines can turn multi-byte Unicode character classes into representations with large numbers of states—which we call discrete representations of character classes, which affects the performance of engines [23], and becomes a common factor for ReDoS vulnerabilities. For example, in RE2, only 8 states are enough for
encoding the full Unicode range U+0000-U+10FFFF, i.e. the meta-character “\.”, while 1,441 states are needed to encode a discrete non-ASCII character class “\W”.

A discrete representation of character classes increases the cost of computation in engines to construct DFA states, which results in ReDoS vulnerabilities, e.g. the 8th regex in Table 2. Besides, this also enlarges DFAs by a huge constant factor. For the running example under partial matching, when \W\D$ is converted from ASCII to UTF-8, its DFA is enlarged by $55\times$, for \W\D$ it is even by $8,487\times$.

Hooimeijer and Veanes [41] noted that the System.Text namespace in the .NET class library contains more than a dozen classes to deal with Unicode encoding. Because of the intricacy to process multi-byte character classes, in grep, some character classes can directly disallow DFA searches and grep degenerates into a backtracking matcher.

5 ReDoS Detection

In this section, we first introduce the notion of simple strings and the k-SIMPLE STRING problem in §5.1. Then we propose an incremental determinisation algorithm with multi-layered heuristic strategies to accelerate the search for k-simple strings in §5.2.

5.1 Simple Strings

From the complexity analysis in §4, a proper and most effective attack string for online DFA matchers should avoid being consumed by any recorded states, such that each byte of the string forces the engine to construct a new DFA state. In graph theory, finding the longest simple path between two given nodes is known to be an NP-hard problem which is even hard to approximate within a constant factor on bounded degree graphs [47]. Inspired by this problem, we propose the notion of simple strings for DFAs of regexes.

Definition 2. Given a DFA $M$ of a regex $E$, a simple string is a finite word $w \in L(E)$ where the states of the path of $w$ on $M$ contain only one final state and are pairwise distinct.

Definition 2 depends on the specific DFA construction method and the DFA is not necessarily minimal. Therefore, different DFAs of $E$ may result in different sets of simple strings of $E$, which can be incomparable w.r.t set inclusion.

There has been work concerning descriptonal complexity and properties for simple languages of automata, e.g. [28, 37]. It is known that (i) any non-null regexes have non-empty sets of simple strings; (ii) any of the simple strings of $E$ is not a prefix of any others and the path of a simple string is acyclic [37]; (iii) the size of the minimal DFA to accept all the simple strings of $M$ is proven to be exponential in the size of $M$ [37].

For any DFA $M$ of a regex $E$, the longest simple strings correspond to the longest acceptable simple paths (i.e., simple paths form the initial state to a final state) on $M$. To illustrate the hardness of finding the longest simple strings, we define the decision problem $k$-SIMPLE STRING.

Problem 1. ($k$-SIMPLE STRING) The $k$-SIMPLE STRING problem w.r.t. a DFA $M$ of a regex $E$ is to decide whether a simple string of $M$ whose length is at least $k$ exists.

Theorem 1 shows the hardness of Problem 1.

Theorem 1. $k$-SIMPLE STRING is EXPSPACE-hard.

The proof is provided in Appendix A.2.

5.2 Incremental $k$-SIMPLE STRING Solving

Due to the hardness of Problem 1, a naïve algorithm based on explicit automata construction can be very inefficient and thus impractical. The key to accelerating $k$-SIMPLE STRING solving is the lazy evaluation strategy: Algorithms should construct DFAs on demand, during the search for $k$-simple strings. Once a $k$-simple string is found, the unvisited DFA states or transitions will never be constructed.

Additionally, two partial order relations are introduced as heuristics, where $\equiv_{\text{Nondet}}$ (Equation 1 below) is used to choose the path by the highest degree of nondeterminism and $\equiv_{\text{Lex}}$ (Equation 6 below) for selecting the DFA states with the lowest matching priority. The heuristics can be applied to the specific DFAs and encodings in any engines mentioned in §2, which may introduce subtle differences in implementation details. Below we select the DFA for grep, i.e., $M_{\text{grep}}$, as an example to introduce our algorithm, i.e., Algorithm 1.

Algorithm 1 is composed of two procedures, $\text{INC\_DETECT}$ that searches for $k$-simple strings incrementally without explicit construction for automata and $\text{DECIDE}$ that answers Problem 1. $\text{INC\_DETECT}$ starts the search with a non-null regex $E$ and an integer $k$, and the output is a $k$-simple string if it exists and one of the longest simple strings otherwise. In the worst case, Algorithm 1 requires doubly exponential space to ensure completeness.

Supplementary Definitions. For any set $X$, its power-set $\wp(X)$ denotes the set of all the subsets of $X$. We write $\langle X; \leq \rangle$ for ordered sets. The composition of two functions $f : X \rightarrow Y$ and $g : Y \rightarrow Z$ is denoted as $g \circ f : X \rightarrow Z$.

Let $T$ denote true and $F$ denote false. We mark symbols in $E$ with numerical subscripts and a linear regex $\overline{E}$ is obtained, where all the symbols in $E$ occur no more than once. Let $\text{Pos}(E)$ denote the alphabet of $\overline{E}$, i.e. the positions. We use the same notation for dropping off the subscripts from linear regexes: $\overline{E} = E$.

For a regex $E$ over $\Sigma$ and a symbol $a \in \Sigma$, we define the following sets, which specify the first, the last and the characters following $a$ from $E$: $\text{First}(E) = \{ b | bw \in E, b \in \Sigma, w \notin \Sigma \}$, $\text{Last}(E) = \{ b | wb \in E, b \in \Sigma, w \notin \Sigma \}$, $\text{Follow}(E,a) = \{ b | uabv \in E, \forall v \in \Sigma, b \in \Sigma \}$. Let $0 \notin \text{Pos}(E)$. Define $\text{Follow}(E,0) = \text{First}(E)$. For $S \notin \wp(\text{Pos}(E))$, let $\text{Follow}(E,S)$ = 

USENIX Association

33rd USENIX Security Symposium 277
Figure 2: Examples of the three strategies in EVIL.STRING, where the states or transitions not yet been constructed are shown as dashed lines. The runes preferred by ⊑Nondet are highlighted in green, and the state preferred by ⊑Lex is in purple, otherwise in beige.

$\cup_{s\in S}$ Follow$(\mathcal{E}, s)$. There are various ways to compute $M_{\text{grep}}$, e.g. [6, 7, 13]. We follow the definition in [13]:

**Definition 3.** The grep DFA of a regex $\mathcal{E}$ can be defined as $M_{\text{grep}}(\mathcal{E}) = (Q_{\text{grep}}, \Sigma, \delta_{\text{grep}}, s_{\text{grep}}, F_{\text{grep}})$ where $Q_{\text{grep}} \subseteq \emptyset(\text{Pos}(\mathcal{E})) \times \{T, F\}$, $\delta_{\text{grep}}(q, a) = (\text{Follow}(\mathcal{E}, p), p \cap \text{Last}(\mathcal{E}) \neq \emptyset)$, for $q \in Q_{\text{grep}}, p = \{ s \in q \mid \exists a\in \Sigma, s_{\text{grep}}(s, a) = 3 \}$ and $a \in \Sigma$, $s_{\text{grep}} = (\text{First}(\mathcal{E}), \text{First}(\mathcal{E}) \cap \text{Last}(\mathcal{E}) \neq \emptyset)$, $F_{\text{grep}} = \{ (\_\_\_, o) \in Q_{\text{grep}} \mid o = T \}$.

**Preprocessing.** In line 3, the first preprocessing procedure of INC_DET is to unfold the regex into a bounded_counting-free regex by the rules defined in Appendix A.1. Then the linearization is applied to the character classes in Unicode following the disambiguation rules of Perl, before encoded in UTF-8, by assigning priority to the leftmost subregexes [70]: A larger number of the subscripts implies a higher matching priority. Furthermore, the lazy quantifiers are assigned with reversed priority w.r.t. greedy quantifiers according to [20].

Due to its popularity among regex engines and operating systems, we use UTF-8 standard as the default encoding. INC_DET then encodes the Unicode character classes of a regex $E$ into UTF-8 standard to obtain $\widehat{E}$. To implement this conversion, we refer to [20], where a hexadecimal-encoded range is called a *rune*. For example, [a–z] is converted to a rune $[\text{x}\text{61}\text{–}\text{x}\text{7a}]$. All the runes transformed from a character class share the same labels, i.e. numerical subscripts.

Since $\mathcal{E}$ is a static object throughout this algorithm, we directly use the addresses of each rune as positions, instead of assigning an extra integer variable. This section uses the regex $\tau = .*(\backslash W \backslash D)^{[5]} S$ from running examples to illustrate INC_DET. Example 4 shows the above processing of $\tau$.

**Example 4.** For $\tau = .*(\backslash W \backslash D)^{[5]} S$, unfolding($\tau$) = $\tau$. At the end of the linearization process, $\tau$ is represented as $\tau = (\{\text{x}\text{00}\text{–}\text{x}\text{7f}\} | \{\text{x}\text{c2}\text{–}\text{x}\text{df}\} | \{\text{x}\text{eo}\text{–}\text{x}\text{ff}\})$.

Variables are initialized in line 4, including the current prefix string $\text{wit}$, the longest current prefix $\text{ret}$, the current $M_{\text{grep}}$ state $\text{cur}$ and the set $\mathcal{Q}$ storing the visited states.

**Non-determinism-Guided Strategy.**

---

```plaintext
Algorithm 1: An incremental determinisation algorithm for k-SIMPLE STRING problem

1 INC_DET(Regex $E$, Int k) -> String
2 begin
3   $E$ = encode $\circ$ linearize $\circ$ unfold($E$);
4   $(\text{wit}, \text{ret}, \text{cur}, Q) = (\varepsilon, \varepsilon, \{0\}, \{\emptyset\})$;
5   loop:
6     for Follow($\mathcal{E}, \text{cur}$) $\neq \emptyset$ do
7       $(C, \mathcal{P}) = (\emptyset, \text{Follow}(\mathcal{E}, \text{cur}))$;
8       for $\alpha \in \text{Part}(\mathcal{P}, \nondet_s) \cup \text{Part}(\mathcal{P}, \text{Lex})$ do
9         $(q, o) = (\{p \in \mathcal{P} \mid \alpha \subseteq p\}, q \cap \text{Last}(\mathcal{E}) = \emptyset)$;
10        $C = C \cup \{ (\alpha, \text{Follow}(\mathcal{E}, q), o) \}$;
11     end
12     for $(\alpha, s) \in \{ (\alpha, \text{Lex}) \}$ do
13            if $s \notin Q$ then
14                $(\text{wit}, \text{ret}, Q) = (\text{wit} + \text{random}(s) \cup \{\mathcal{P}\})$;
15                if $\text{wit} \geq k$ then
16                    return $\text{wit}$;
17                    break;
18                else if $\text{wit} > \text{ret}$ then
19                    $\text{ret} \leftarrow \text{wit}$;
20                    goto loop;
21                else
22                    continue;
23            end
24     end
25     DECIDE(Regex $E$, Int k) -> Bool
26     begin
27        return $\text{INC_DET}(E, k) \geq k$;
28     end

Motivation. Nondeterminism is a well-established concept in automata theory [49]. Manfi [54] proved even a finite amount of nondeterminism in NFAs may lead to exponential size blow-up in DFAs. Here we introduce the degree of nondeterminism of a state.

**Definition 4.** For an $\varepsilon$-free NFA $M = (Q, \Sigma, \delta, s, F)$, the degree of nondeterminism $^\varepsilon$ of a state $q \in Q$ w.r.t a symbol $a \in \Sigma$,

---

$^\varepsilon$ Also called branching in [33].
denoted as $\text{nondet}(q,a)$, is the number of nondeterministic transitions labeled $a$ from the state $q$, which is computed as $\text{nondet}(q,a) = \#(q,a)$.

If the degrees of nondeterminism of all of the states in an e-free NFA $M$ are 1, then $M$ is a DFA and each DFA state is a singleton subset of $Q$.

**Method.** Nondeterminism-guided strategy chooses a run that has the highest degree of nondeterminism from the DFA state, which also results in possibly bigger subsequent subsets.

To exploit the power of nondeterminism on $M_{\text{grep}}(E)$, let $\text{nondet}(s,\sigma) = |\{ \text{Follow}(E,\eta) \mid s = (Q,\_), \forall q \in Q, \sigma \subseteq q\}|$, for $s \in Q_{\text{grep}}$ and $\sigma \in \wp(\Sigma)$. Next we define the following binary relation $\sqsubseteq_{\text{nondet}} \subseteq \wp(\Sigma) \times \wp(\Sigma)$: for $s \in Q_{\text{grep}}$ and $\sigma, \sigma' \in \wp(\Sigma)$, $\sigma \sqsubseteq_{\text{nondet}} \sigma'$ if and only if $\text{nondet}(s,\sigma) \leq \text{nondet}(s,\sigma')$.

$$\text{(1)}$$

Denote $\equiv_{\text{nondet}}$ as the inverse of $\sqsubseteq_{\text{nondet}}$. By substituting $\leq$ by $\equiv$ in Equation 1, an equivalence relation, denoted as $\equiv_{\text{nondet}}$, is obtained, which will be used in the second strategy below. In line 8, $\text{INC\_DET}$ first partitions $\bigcup \mathcal{P}$ into disjoint subsets according to identical successors and obtain $\text{Part}(\mathcal{P})$. For each $\alpha$ from the partially ordered run set $\langle \text{Part}(\mathcal{P});\equiv_{\text{nondet}} \rangle$, $\text{INC\_DET}$ computes the set of positions $q$ that include $\alpha$ and a Boolean variable $i$ implying if $q$ intersects $\text{Last}(E)$, in line 9. Next $\text{INC\_DET}$ associates and records $\alpha$ with the $M_{\text{grep}}$ state in $C$ in line 10.

**Example 5.** Consider a state $s = (P,F)$ from $M_{\text{grep}}(\tau)$ in Figure 2 (a), where $P = \langle \{x_0\}, \{x_1\}, \{x_2\}, \{x_4\}, \{x_5\} \rangle$. Then $\text{Part}(\mathcal{P}) = \langle \{x_0\}, \{x_2\}, \{x_4\}, \{x_5\} \rangle$. For $\mathcal{P}_{\text{grep}} = \langle \{x_0\}, \{x_2\}, \{x_5\} \rangle$, $\text{nondet}(s,\{x_0\}) = 10$, $\text{nondet}(s,\{x_2\}) = 10$, $\text{nondet}(s,\{x_5\}) = 10$.

Effect. Choosing the run that has the maximal degree of nondeterminism results in higher costs in constructing the DFA states in online DFA matchers (recall §4.1). Also higher nondeterminism tends to lead to larger successor states, making the states less likely to be identical with the already visited $M_{\text{grep}}$ states, consequently decreasing backtracking in the search and accelerating $k$-simple string generation.

**Generalized Lexical Order Strategy.**

**Motivation.** In UTF-8 encoding, each discrete character class in the partially ordered set $\langle \text{Part}(\mathcal{P});\equiv_{\text{nondet}} \rangle$ usually consists of a large number of runes, where plenty of those have equal degrees of nondeterminism. To select a run that reduces the chance of a search reaching a final state early, we introduce the generalized lexical order strategy.

**Definition 5.** (See [11].) Let $(\Sigma; \leq)$ be a partially ordered finite alphabet. The **generalized lexical order** on $\Sigma^*$ is defined as follows: Let $u = a_0,\ldots,a_n$ and $v = b_0,\ldots,b_n$ be two words (where $a_0,\ldots,a_n, b_0,\ldots,b_n \in \Sigma$), $u, v$ are ordered, i.e., $u \leq_{\text{LEX}} v$, if and only if $|u| < |v|$, or $|u| = |v|$ and (1) $u$ is a prefix of $v$ or (2) $u = pa_1x, v = pb_1y$ and $a_1 <_{\text{LEX}} b_1$, where $p$ is the longest common prefix of $u$ and $v$.

For $a, b \in \langle \text{Pos}(E);\leq_{\text{LEX}} \rangle$, define $a \equiv_{\text{LEX}} b$ to indicate that the labels of $a$ and $b$ are identical. When Algorithm 1 reaches a state $s' = (P,\_)$, for $a_0,\ldots,a_n \in \langle \text{Part}(\mathcal{P});\equiv_{\text{nondet}} \rangle$, the simple strings going through $s'$ can be represented as $u = a_0x, \ldots, a_ix, a_{i+1}y$, where $0 \leq i \leq n$. Since elements in $\langle \text{Part}(\mathcal{P});\equiv_{\text{nondet}} \rangle$ are disjoint, $p$ is the longest common prefix among $a_0x,\ldots, a_ny$. We then utilize the generalized lexical order to choose the DFA states (e.g. from $\delta_{\text{grep}}(s,\alpha_0),\ldots,\delta_{\text{grep}}(s,\alpha_n)$) that possibly lead to longer successor suffixes (e.g. the string $x$) by introducing a novel partial order between DFA states.

**Method.** Since the linearization applies to the character classes, each states in $M_{\text{grep}}(E)$ has an ordered set from $\langle \text{Pos}(E);\leq_{\text{LEX}} \rangle$ and a multiset of labels, i.e. a partially ordered mutiset of integers: The runes from the same labeled character class have identical labels, representing the same priority.

To establish the order between ordered multisets, we use the concept of the support [71] of multiset: Let $\text{Supp}(q) = \{(Q,q) \mid q \in (Q,\_))\}$, for a state $q \in Q_{\text{grep}}$. Next we define the following binary relation $\leq_{\text{LEX}} \subseteq Q_{\text{grep}}^2$.

**Definition 6.** For $q_1, q_2 \in Q_{\text{grep}}$, $p_0,\ldots,p_m \in \text{Supp}(q_1)$ and $p'_0,\ldots,p'_m \in \text{Supp}(q_2)$, $q_1 \leq_{\text{LEX}} q_2$ if and only if

1. $q_1 \neq q_2$.
2. $n < m$\

When the supports of two states are identical, $\text{INC\_DET}$ randomly chooses one to continue, since the matching priorities of those states are exactly the same. In line 11, Algorithm 1 divide the runes in $A$ into equivalence classes based on $\equiv_{\text{nondet}}$. For each candidate state sorted by $\equiv_{\text{LEX}}$ (line 12), we test whether it is already recorded in $Q$ in line 13. If not, the current state $\text{curr}$ is assigned as the candidate state $s$; moreover, $\mathcal{P}$ is recorded in $Q$, and a random symbol from $\alpha$ is appended to $\text{wit}$ in line 14. If the state $s$ is not a final state (line 15) and $\text{wit}$ has exceeded $\text{ret}$ (line 19), $\text{ret}$ is replaced with $\text{wit}$ (line 20). Then $\text{INC\_DET}$ continues the search in line 21. If $s$ is already visited, $\text{INC\_DET}$ continues the search in line 23 and chooses another pair of $(\alpha, s)$ from $\langle \{\alpha\};\equiv_{\text{nondet}};\langle \mathcal{S};\leq_{\text{LEX}} \rangle \rangle$.

**Example 6.** For the state $s$ from Example 5, $\text{Supp}(s) = \{(\{x_0\}, \{x_1\}, \{x_2\}, \{x_4\}, \{x_5\})\}$. In Figure 2 (b), $s_1$ and $s_2$ are both resulted from runs with the highest degree of nondeterminism at 10 in Figure 2 (a). Since $\text{Supp}(s_1) = \{1, 2, 4\}$ and $\text{Supp}(s_2) = \{1, 2\}$, $(s_1, s_2);\equiv_{\text{LEX}} = (s_2, s_1)$, i.e. $s_2 \equiv_{\text{LEX}} s_1$. $\text{INC\_DET}$ continues with $s_2$.

**Effect.** The effectiveness of generalized lexical order strategy is profoundly connected to Perl-style disambiguation rules, i.e. the Greedy rules, which are commonly used in non-backtracking engines. By exploiting $\equiv_{\text{LEX}}$, we maintain the consistency with engines using Greedy rules and also select the DFA states that are less likely to lead to early termination.

---

Footnote: For ease of representation, we write $\text{Supp}(s) = \{1, 2, 3\}$.  

---

USENIX Association
33rd USENIX Security Symposium 279
\[(b, Q') \leftarrow \text{AnalyzeConf}(b, \text{wit}, Q)\;\]
\[\text{if } b \geq \text{CONFLICT\_RATE} \text{ then}\]
\[\text{backtrack}(b, Q');\]

Figure 3: Code fragment for non-chronological backtracking.

**Output.** When a \(k\)-simple string is found (line 16), in line 17, the witness is returned. When the states of \(M_{\text{grep}}\) have been fully explored, yet there does not exist a \(k\)-simple string, \(\text{INC\_DET}\) returns the longest simple string found in line 24. By deciding whether the length of output from \(\text{INC\_DET}\) exceeds \(k\) in line 27, the algorithm \text{DECEDE}\) solves \(k\)-SIMPLE STRING for \(M_{\text{grep}}\) in a sound and complete manner.

**Theorem 2.** \(\text{DECEDE}(E, k)\) is sound and complete.

The proof is provided in Appendix A.3.

**Non-Chronological Backtracking.** To further improve the efficiency of our algorithm, we introduce an incomplete non-chronological backtracking algorithm [55] into \(\text{INC\_DET}\), resulting in \(\text{INC\_DET'}\). This algorithm allows \(\text{INC\_DET'}\) to jump back over several levels in the automata when a considerable number of conflicts \(^7\) are detected.

The non-chronological backtracking is applied to \(\text{INC\_DET}\) by substituting line 18 and 23 by the code fragment in Figure 3, where the integer \(b\) for the level of backtracking and state set \(Q'\) storing the currently visited states are global variables. When a conflict occurs, we analyze the conflict in line 1 by the rate between the increment in the length of \(ret\) and the increment in \(b\): If \(\text{INC\_DET'}\) executes backtracking or the increment in length of \(ret\) exceeds a threshold \(^8\), \(b\) is reset and \(Q'\) is updated as \(Q\). When the rate of conflicts exceeds \(\text{CONFLICT\_RATE}\), the search backtracks to the state implied by \(b\), where the search is continued with another choice and \(Q\) is rewritten as \(Q'\) to eliminate the pruned states. Figure 2 (c) illustrates the situation when the rate of conflict exceeds the threshold in the left part of the automaton; the algorithm prunes the state space and then jumps back several levels to find more promising branches. This mechanism accelerates the search at the cost of completeness, i.e. some of the solutions may be pruned, even though they are valid.

## 6 Evaluation

In this section, we describe the evaluation of \text{EVIL\_STR\_GEN}, our C++ implementation \(^9\) of the methodology presented.

**Research Questions.** We conducted experiments to compare \text{EVIL\_STR\_GEN} with the current state-of-the-art ReDoS detectors to answer the following research questions (RQs):

**RQ1:** How is the capacity of \text{EVIL\_STR\_GEN} compared to the state-of-the-art ReDoS detectors on large-scale real-world regex benchmarks?

**RQ2:** Specifically, how is the performance of \text{EVIL\_STR\_GEN} compared to GadgetCA [74] on the datasets that GadgetCA supports?

**RQ3:** How do different heuristic strategies described in §5 affect the efficiency of \text{EVIL\_STR\_GEN}?

**RQ4:** How is the performance of our approach for detecting vulnerabilities in real-world applications?

### 6.1 Experiment Setup

**Benchmark.** Non-backtracking engines were considered as a “safe” substitute for backtracking engines [8] to mitigate ReDoS vulnerabilities. To demonstrate that \text{EVIL\_STR\_GEN} is able to detect ReDoS in non-backtracking engines which are comparably severe to those in backtracking engines, we use SET736535, a large-scale real-world benchmark of 736,535 regexes from a wide variety of sources for evaluation.

SET736535 is obtained from a collection of 850,279 regexes composed of nine datasets: (i) Corpus [17], (ii) RegExLib [52] and (iii) Regex101 [4] from ReDoS-related literature; (iv) Large-scale regex dataset extracted from over 190,000 software projects by Davis et al. [22]; (v) Maven [1], (vi) NuGet [2] and (vii) PyPI \(^{10}\) obtained from real-world code repositories by Wang et al. [78]; (vii) Snort [25]; (viii) Zeek [26]; and (ix) Sagan [5] from network intrusion detection systems. We integrated and deduplicated the aforementioned datasets, resulting in a set of 839,670 unique regexes. By removing regexes having features not commonly supported by non-backtracking engines (e.g. lookarounds and backreferences), SET736535 is obtained.

**Engines.** We selected 8 non-backtracking engines using online DFA matchers and 8 backtracking engines for evaluating our approach. Among the engines discussed in §2, the selected non-backtracking engines include RE2, regex engine used in Rust and Go, SRM, C#\(^{11}\), awk, grep and Hyperscan. No comparison is made to sed, since GNU sed and awk share the same DFA implementation. The standard library backtracking engines for programming languages selected include Java, JavaScript, PCRE2, Perl, php, Python, Boost, and C#.

**Baselines.** To comprehensively evaluate the performance of \text{EVIL\_STR\_GEN}, we identified six state-of-the-art ReDoS detection tools for comparison, categorized into: (i) static analysis (GadgetCA [74], RegExStatic [79] and Rexploiter [80]); (ii) dynamic analysis (Regulator [56], ReScue [66]); (iii) hybrid approach (ReDoSHunter [52]). Among the baselines, the only tool designed for non-backtracking engines is GadgetCA.

**Criteria for ReDoS.** In previous literature addressing ReDoS, several different criteria have been proposed. For example \(^{11}\): Davis et al. [21] suggest that a regex is vulnerable if a string of 100K to 1M \textit{characters} takes the (backtracking) engines 10s to match; Regulator [56] uses 1M characters for 10s as the

\(^7\)A choice that coincides with a previously visited DFA state in \(Q\) or a final state before \(k\) is reached is considered a conflict.

\(^8\)Empirically we set this value to \(\frac{\text{CONFLICT\_RATE}}{2}\).

\(^9\)https://doi.org/10.5281/zenodo.11502706

\(^{10}\)https://pypi.org/

\(^{11}\)Notice all the characters and strings mentioned here are in Unicode, and a Unicode character can be as long as 4 bytes.
To obtain attack strings, ReDoSHunter [52] repeats “pump” strings 30,000 times and tests if the string takes more than 1s to match; [78] repeats pump strings 15,000 times and tests if the string takes Java’s regex engine more than $10^3$ matching steps.

To better characterize the features of ReDoS on non-backtracking engines, we propose both machine-dependent and independent ReDoS criteria as follows:

**<100kB/s**: To detect ReDoS on non-backtracking engines which are comparable to those on backtracking engines, we unified machine-dependent criteria for ReDoS on different engines by proposing a throughput threshold of 100kB/s, i.e. the engine cannot process 100kB input per second. This criterion is prominently more stringent than the lowest criterion (0.5MB/s) in [74], and conforms with the criteria for ReDoS on backtracking engines [21, 52] to be closer to practice.

**>10,000 states**: From the analysis in §4.1, the number of DFA states constructed in a non-backtracking engine to match an attack string reveals the theoretical effectiveness of the string. Considering the state cache thresholds used in engines, we introduce a novel machine-independent DFA state number measure in non-backtracking engines, i.e. an attack string of 100kB should force the engine to construct more than 10,000 distinct subsets or ACI-similar derivatives.

**Configurations.** All evaluations were performed on a PC with 3.40GHz Intel i7-6700 8 CPU and 8GB of memory, running Ubuntu 20. We deployed the baselines as their newest versions enabled, is denoted as $L$.

To maintain consistency with research on backtracking engines and avoid the highly impractical size at 50MB in [74], we keep the size of inputs at 100kB in all experiments. Analyses on related work [52, 78] and CVEs [21] show 100kB is practical to detect ReDoS for backtracking engines. It is common among network intrusion detection systems to impose length restrictions on the input. Also, inputs of 100kB can be used in web services such as those using Spread Toolkit12, etc. In EvilSTRGEN, the length is user-adjustable according to practical demands. Since baselines targeting on backtracking engines generate strings in the form of $prefix+pump^n+suffix$ for some $n$ [52], we set $n$ to satisfy $|prefix| + |pump| \times n + |suffix| \approx 100kB$.

12http://www.spread.org/index.html

### 6.2 Performance on Large-Scale Real-World Regex Benchmark

The various results under the criterion of <100kB/s for each ReDoS detector on all 16 engines for SET736535 are shown in Table 5. The numbers show the sum of ReDoS vulnerabilities found by each tool on engines and the best numbers are in bold. The results reveal that the non-backtracking engines can also be ReDoS vulnerable, where some may even perform worse than backtracking engines.

As depicted in Table 5, for all of the selected non-backtracking engines, EvilSTRGEN achieved higher effectiveness than all the baselines. In comparison with GadgetCA, using all the strategies together, EvilSTRGEN identified 1.75×, 2.29×, 5.54×, 3.30×, 3.44×, 0.31×, 0.05× and 580.75× more ReDoS vulnerabilities on RE2, Rust, Go, SRM, C#N, awk, grep and Hyperscan, respectively. Noticeably, we found 2,327 ReDoS vulnerabilities in Hyperscan, which was considered invincible in [74]. This is due to the effective encoding of UTF-8 standard and byewise generation in EvilSTRGEN, while GadgetCA, ReDoSHunter, etc. generate strings characterwise.

Furthermore, EvilSTRGEN is also the best ReDoS detector among baselines for backtracking engines in Java, JavaScript, PCRE2, Python, Boost and C#, with 13,465, 9,797, 1,633, 4,395, 3,811, and 10,947 more ReDoS vulnerabilities detected than the second best respectively.

The results also show the incapability of the detectors designed for backtracking engines to find ReDoS for non-backtracking engines. Our investigation further reveals many of the regexes in the benchmark causes awk and grep to fall back to use backtracking matchers due to reasons mentioned in §4.2, which explains the relatively better performance of some detectors for backtracking engines on those engines.

Next we give more detailed analyses of GadgetCA and EvilSTRGEN, the tools targeting non-backtracking engines. Figure 4 (a)-(h) show vulnerable regexes identified in SET736535 by each tool using an enhanced scatter plot. Each point displays a regex’s throughput rates handling strings from EvilSTRGEN (x-axis) and GadgetCA (y-axis) but groups some overlapping points into hexagonal bins and coloring them to represent density using Kernel Density Estimation (KDE), where the darker colors indicate the denser areas. A piecewise linear transformation is applied to safe regexes (both throughput ≥100kB/s) to enhance the visibility of the vulnerable ones. Points below the diagonal suggest EvilStrGen causes lower throughput, otherwise GadgetCA does. We also calculate the percentage of regexes for which EvilSTRGEN causes a more severe slowdown (e.g., “wins in 100.00% regexes”). EvilSTRGEN wins in 85.71%, 100%, 91.06%, 98.57%, 97.90%, 100%, 56.86%, 99.95% vulnerable regexes on RE2, Rust, Go, SRM, C#N, awk, grep and Hyperscan, respectively.

Figure 5 (a) displays a distribution of the number of DFA states constructed in RE2 to match the strings generated by...
both tools using KDE. The higher the curve, the higher the density. The sum of the values represented by each curve equals 1, i.e. the entire set. The distribution shows that EVIL-STRGEN can explore a larger state space on more regexes.

Furthermore, the dashed line marks the position corresponding to 10,000 states, with the area to the right of it considered to contain vulnerable regexes. EVIL-STRGEN found 1.55% of regexes vulnerable w.r.t. >10,000 states criterion, while only 0.4% of regexes are found vulnerable by GadgetCA. The results indicate that EVIL-STRGEN is generally more effective than GadgetCA, achieving an average of 17,846 DFA states and a median of 14,813 states, whereas GadgetCA has only 7,535 and 4,459 states respectively.

We further analyze ReDoS in some subclasses of regexes. Table 6 shows the total numbers of regexes with (and without) some specific structures from SET736535 (the column “Total”) and vulnerable regexes from those subclasses detected by our tool (the other columns). The statistics reveal that regexes from these subclasses are not rare to appear in practice. Owing to the effectiveness of the k-simple strings, EVIL-STRGEN is capable of exposing vulnerabilities for different engines on seemingly safe subclasses such as bounded_counting-free regexes. However, the method is still not sufficient to ensure all the vulnerabilities are found, see discussions in §6.6.

Summary. Due to its capability to detect various kinds of ReDoS vulnerabilities on non-backtracking regex engines, EVIL-STRGEN is far more effective than all baselines, detecting overall 2.16 - 3.042.41 times more ReDoS vulnerabilities. Besides, EVIL-STRGEN is also the best or close to the best detector for backtracking regex engines.

### 6.3 Comparison with GadgetCA on ABOVE20

As GadgetCA focused on ReDoS detection in a subclass of regexes, to evaluate RQ2, we fairly compare EVIL-STRGEN with GadgetCA on the ABOVE20 benchmark. ABOVE20 is a restricted dataset of 8,099 regexes with counting whose sum of upper bounds are above 20 excluding those unsupported by GadgetCA from [74].

Under the criterion of <100kB/s, the ReDoS vulnerabilities detected by both tools are depicted in Table 7. From Table 7, it can be observed that using all of the strategies collectively, EVIL-STRGEN identified 17.37%, 2.33%, 3.60%, 4.69 ×, 8.50 ×, 1.64 ×, 1.81 × and 15.00 × more ReDoS vulnerabilities compared to GadgetCA on RE2, Rust, Go, SRM, C#N, awk, grep and HyperScan, respectively.

Figure 4 (i)-(p) shows the ReDoS vulnerable regexes found by each tool in ABOVE20 under the criterion of <100kB/s. The result shows that EVIL-STRGEN wins in 94.83%, 92.59%, 85.71%, 83.64%, 92.81%, 78.75%, 77.49%, 96.42% vulnerable regexes on RE2, Rust, Go, SRM, C#N, awk, grep and HyperScan, respectively. Figure 5 (b) displays a distribution of results on ABOVE20, which shows that EVIL-STRGEN achieves an average of 54,730 DFA states and a median of 38,990 states, in contrast to GadgetCA’s average and median of only 39,300 and 29,270 states, respectively. EVIL-STRGEN found 7.2% of regexes vulnerable w.r.t. >10,000 states criterion, while only 0.74% of regexes are found vulnerable by GadgetCA.

**Summary.** EVIL-STRGEN outperforms GadgetCA in terms of both the number of vulnerabilities found among non-backtracking engines and the severity of vulnerabilities triggered on the benchmark which GadgetCA is specialized in.

### 6.4 Analysis of Different Configurations

The efficiency of different configurations of EVIL-STRGEN is depicted in Figure 6 on SET736535, measuring the average length of candidate attack strings generated by each algorithm within 50s, where ALLSTRATON achieves an average length of 19,308 bytes in 50s, while LEXICALON, NON-
DETON and BRUTE-FORCE achieve 17,602 bytes, 12,882 bytes and 9,825 bytes respectively.

In Figure 6, with more strategy enabled, the curves become flatter. Though more strategies enabled may decelerate the tool at the beginning, e.g. to generate 6,000 bytes, ALLSTRATON uses 11.37s on average, while LEXICALON, NONDETON and BRUTE-FORCE require 10.21s, 8.35s, 8.06s respectively, the curves for the other three configurations tend to stabilize in 50s, before reaching 100kB. From the overall results in Table 5 and Table 7, EVILSTRGEN is 13.4× more effective in ReDoS detection on non-backtracking engines using all our heuristics than using none.

The results also show a general improvement of each strategy on the BRUTE-FORCE algorithm. In addition, the complete algorithm LEXICALON also shows advantages over GadgetCA particularly on RE2, Go, SRM, C#N and Hyperscan on SET736535.

Summary. The results demonstrate empirically that each strategy that we have introduced provides a remarkable benefit in the efficiency of candidate attack string generation. For maximum efficiency, all of the strategies are used simultaneously in EVILSTRGEN.

6.5 Real-world Application

To illustrate the efficacy of EVILSTRGEN, it was deployed on real-world projects to identify exploitable ReDoS vulnerabilities. We have applied EVILSTRGEN on the well-starred and widely downloaded projects in Rust and Go from GitHub. The initial step involved crafting a script for extracting code segments using regexes from these projects. Next, EVILSTRGEN was employed to generate and verify attack strings according to different matching functions and engines. A manual examination was then conducted to ascertain the presence of code segment interfaces linked to the regexes that might be vulnerable to external injections.

Our investigation led to the identification and manual verification of 85 projects having potential ReDoS vulnerabilities, out of which 34 regexes were confirmed as ReDoS vulnerable within the real-world projects. Strikingly, GadgetCA was not able to recognize any of the newly revealed vulnerabilities. For details, see Appendix B.

Example 7. unknow/com is an open-source project for commonly used functions for the Go programming language. As illustrated in Figure 7, a regex used for HTML tag re-
ReDoS Detection for Backtracking engines. Previous works on ReDoS detection mainly focus on backtracking engines, which can be classified into the following methods.

**Static Analysis.** Static analysis methods for backtracking engines model ReDoS vulnerabilities but often fail due to modeling limitations, leading to high false positives and negatives. RXRX2 [63] detects ReDoS vulnerabilities by pumping analysis, struggling with polynomial ReDoS vulnerabilities. RegexStatic [79] estimates the worst case cost (linear, polynomial, or exponential) of regexes but faces challenges due to its less comprehensive modeling approach.

**Dynamic Analysis.** Dynamic analysis detects ReDoS vulnerabilities during actual runtime, having higher precision than static methods in the cost of efficiency. ReScue [66] targets time-intensive input strings using genetic algorithms, which leads to omission in lower polynomial instances due to the vast search space and the selection bias of genetic algorithms results in further limitations. Regulator [56] applies fuzz testing to regex byte-code with a mutation approach, which may cost too much time for generating attack strings.

**Hybrid Approaches.** Hybrid tools combine static and dynamic analysis. Revealer [53] uses an extended NFA based on regex engine in Java for static analysis. ReDoSHunter [52] identifies and verifies regexes with more fine-grained vulnerability patterns. Rengar [78] detects ReDoS vulnerabilities by combining “loop subregex”-based vulnerability modeling and disturbance-free attack string generation. However the existing work lacks theoretical foundations.

ReDoS Detection for Non-Backtracking engines. GadgetCA [74] is a detector targeting ReDoS vulnerabilities in non-backtracking engines, relying on the determination of counting automata to generate candidate attack strings for a restricted subclass of regexes, which may lead to false negatives in regexes that have non-uniform automata. Different from GadgetCA, EVILSTRGEN leverages a sound and complete algorithm and aims to find the “evilest” attack strings for online DFA matchers, i.e. simple strings, thus giving a better performance in detecting real-world ReDoS vulnerabilities.

## 8 Conclusion

In this paper, we presented a novel and effective approach to detect ReDoS vulnerabilities for non-backtracking regex engines. Our ReDoS detector EVILSTRGEN shows state-of-the-art performance on large-scale real-world benchmark for both non-backtracking and backtracking engines and exposes 34 new vulnerabilities in popular open-source projects. There are many promising optimizations remained to be explored, such as cooperating local search to accelerate the k-SIMPLE STRING solving and introducing symbolic execution to reveal more ReDoS vulnerabilities in programs using non-backtracking engines.
Acknowledgements. We would like to thank our shepherd and reviewers for their helpful comments and suggestions. We thank Ping Lu for his valuable advice on computational complexity and Zhiwu Xu for his careful reading and helpful comments on an earlier version of the paper. Work supported by the National Natural Science Foundation of China (Grant Nos. 62372439 and 61872339) and the Natural Science Foundation of Beijing, China (Grant No. 4232038). Tingjian Ge is supported in part by NSF grants IIS-2124704 and OAC-2106740.

References


A Omitted Definitions and Proofs

A.1 Unfolding Rules

A regex $E$ can be transformed into an equivalent bounded-counting-free regex by recursively applying the rules:

$E^0 \Rightarrow E^*, E^0[?] \Rightarrow E^{?}, E^0[0] \Rightarrow \varnothing, E^0[0][?] \Rightarrow \varnothing,$

$E^+ \Rightarrow EE^*, E^+ \Rightarrow EE^{?},

E^{m,n} \Rightarrow EE^{m-1, n-1}, E^{m,n}[?] \Rightarrow EE^{m-1, n-1}[?],$ for $n \geq m > 0,

E^{0,n} \Rightarrow E^0[0, n-1], E^{0,n}[?] \Rightarrow E^0[0, n-1][?],$ for $n > 1,

E^{m} \Rightarrow EE^{m-1}, E^{m}[?] \Rightarrow EE^{m-1}[?],$ for $m \geq 1.$

A.2 Proof for Theorem 1

**Theorem 1.** $k$-SIMPLE STRING is EXPSPACE-hard.

**Proof.** We show that $k$-SIMPLE STRING is EXPSPACE-hard by a reduction to NON-EMPTY COMPLEMENT. Given a regex $E$ defined on an alphabet $\Sigma$, the NON-EMPTY COMPLEMENT problem of $E$ is to decide whether $L(\overline{E}) \neq \varnothing.$ In [39], this problem is proven to be EXPSPACE-complete.

We note that a DFA $M$ of $E$ can be easily completed by using the number of states, and the sink state is denoted as $d$. Define the complemented $M$ as $M^c = (Q^c, \Sigma, \delta^c, s^c, F^c)$, where $F^c = (Q - F) \cup$
A.3 Proof for Theorem 2

**Theorem 2.** DECIDE\(E, k\) is sound and complete.

**Proof.** Let \(E\) be the regex and \(k\) the integer on which the algorithm is executed. First, we show the soundness of DECIDE. When DECIDE returns \(\mathbf{T}\), consequently INC\_DET returns a string with at least length \(k\). Any returned string of INC\_DET has a path on \(M_{\text{grep}}\) thus is a string in \(L(E)\). As the search avoids recording identical states by condition \(s \notin Q\) within the loop (line 13) and keep only one final state in a searching path by terminating this search once encountering a final state (line 15), thus the string is a \(k\)-simple string of \(E\), w.r.t \(M_{\text{grep}}\). When DECIDE returns \(\mathbf{F}\), INC\_DET returns a string shorter than \(k\), indicating that the string is not a \(k\)-simple string. Above all, the algorithm DECIDE is sound.

For any given input regex \(E\), INC\_DET produces a simple string (as discussed above). If the length of this output string exceeds \(k\) by the termination condition \(|\text{wit}| \geq k\), i.e., INC\_DET finds a simple string longer than \(k\) and DECIDE returns \(\mathbf{T}\). If there is no simple string of length greater than \(k\), when the finite set of states of \(M_{\text{grep}}\) is fully constructed, the algorithm terminates and returns a longest string found, which is lower than \(k\), then DECIDE returns \(\mathbf{F}\). Accordingly, DECIDE is guaranteed to terminate correctly.

From the argument above, DECIDE\(E, k\) is sound and complete in solving \(k\)-SIMPLE STRING w.r.t. \(M_{\text{grep}}\).

B Real-World Vulnerabilities

Table 8 shows ReDoS vulnerable regexes from Go and Rust projects, where their names and stars are listed. Only the source projects are shown for these regexes in supply chains.

<table>
<thead>
<tr>
<th>No.</th>
<th>Regex</th>
<th>Project</th>
<th>Stars</th>
<th>GadgetCA</th>
<th>EVILSTRGEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>`(\x1b[[b-m</td>
<td>A-H;[0-9]<em>]?;x1b[[1-6]</em>\x1b[[;f-3J`</td>
<td><code>junegunn/trf</code></td>
<td>36,859</td>
<td>✓</td>
</tr>
<tr>
<td>2</td>
<td><code>(\s)\x1b\[[=\S*\s]'\x1b\[[\S*\s]</code></td>
<td><code>unknown/com and other 56 projects</code></td>
<td>13,265 in total</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3</td>
<td><code>\$\S*(\([^\S]*\))?\x1b\[[;f-3J</code></td>
<td><code>sxyazi/yzai</code></td>
<td>5,521</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>4</td>
<td><code>\(\([^\S]*\)\)?\x1b\[[;f-3J</code></td>
<td><code>ajour/ajour</code></td>
<td>1,000</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>5</td>
<td><code>\(\([^\S]*\)\)?\x1b\[[;f-3J</code></td>
<td><code>gresnity/gore-cli</code></td>
<td>855</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>6</td>
<td><code>\(\([^\S]*\)\)?\x1b\[[;f-3J</code></td>
<td><code>felixx/ingodaten</code></td>
<td>307</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>7</td>
<td><code>\(\([^\S]*\)\)?\x1b\[[;f-3J</code></td>
<td><code>emk/subtitles-rx</code></td>
<td>285</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>8</td>
<td><code>\(\([^\S]*\)\)?\x1b\[[;f-3J</code></td>
<td><code>gwentai/skull</code></td>
<td>282</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>9</td>
<td><code>\(\([^\S]*\)\)?\x1b\[[;f-3J</code></td>
<td><code>sodafoundation/strato</code></td>
<td>231</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>10</td>
<td><code>\(\([^\S]*\)\)?\x1b\[[;f-3J</code></td>
<td><code>Away0x/gen_bbs</code></td>
<td>161</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>11</td>
<td><code>\(\([^\S]*\)\)?\x1b\[[;f-3J</code></td>
<td><code>TIBCOsoftware/mashing</code></td>
<td>86</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>12</td>
<td><code>\(\([^\S]*\)\)?\x1b\[[;f-3J</code></td>
<td><code>isto-ecosystem/isto-coredns-plugin</code></td>
<td>41</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>13</td>
<td><code>\(\([^\S]*\)\)?\x1b\[[;f-3J</code></td>
<td><code>pavekhamlin/grafana-csv-plugin</code></td>
<td>35</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>14</td>
<td><code>\(\([^\S]*\)\)?\x1b\[[;f-3J</code></td>
<td><code>kevinwati/ed2kcrawler</code></td>
<td>32</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>15</td>
<td><code>\(\([^\S]*\)\)?\x1b\[[;f-3J</code></td>
<td><code>roosta/onz</code></td>
<td>24</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>16</td>
<td><code>\(\([^\S]*\)\)?\x1b\[[;f-3J</code></td>
<td><code>Notarin/hayabusa</code></td>
<td>20</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>17</td>
<td><code>\(\([^\S]*\)\)?\x1b\[[;f-3J</code></td>
<td><code>coln35/universe</code></td>
<td>19</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>18</td>
<td><code>\(\([^\S]*\)\)?\x1b\[[;f-3J</code></td>
<td><code>psai/allwrite-docs</code></td>
<td>15</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>19</td>
<td><code>\(\([^\S]*\)\)?\x1b\[[;f-3J</code></td>
<td><code>crych/sinamg mover</code></td>
<td>15</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>20</td>
<td><code>\(\([^\S]*\)\)?\x1b\[[;f-3J</code></td>
<td><code>jvander/flags-components</code></td>
<td>13</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>21</td>
<td><code>\(\([^\S]*\)\)?\x1b\[[;f-3J</code></td>
<td><code>nimbleswimble/grin-gui</code></td>
<td>13</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>22</td>
<td><code>\(\([^\S]*\)\)?\x1b\[[;f-3J</code></td>
<td><code>fshl3gs/vertex</code></td>
<td>12</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>23</td>
<td><code>\(\([^\S]*\)\)?\x1b\[[;f-3J</code></td>
<td><code>crawlerclub/cc</code></td>
<td>11</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>24</td>
<td><code>\(\([^\S]*\)\)?\x1b\[[;f-3J</code></td>
<td><code>emunze/nextextract</code></td>
<td>11</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>25</td>
<td><code>\(\([^\S]*\)\)?\x1b\[[;f-3J</code></td>
<td><code>crawlerclub/cc</code></td>
<td>11</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>26</td>
<td><code>\(\([^\S]*\)\)?\x1b\[[;f-3J</code></td>
<td><code>Universal-Varibality-Language/evil-lsp</code></td>
<td>9</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>27</td>
<td><code>\(\([^\S]*\)\)?\x1b\[[;f-3J</code></td>
<td><code>aryamancodes/sked-search</code></td>
<td>9</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>28</td>
<td><code>\(\([^\S]*\)\)?\x1b\[[;f-3J</code></td>
<td><code>enginein/aksaratara_ruby</code></td>
<td>7</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>29</td>
<td><code>\(\([^\S]*\)\)?\x1b\[[;f-3J</code></td>
<td><code>alectomas/bit</code></td>
<td>7</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>30</td>
<td><code>\(\([^\S]*\)\)?\x1b\[[;f-3J</code></td>
<td><code>sgzhang/gdi</code></td>
<td>5</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>31</td>
<td><code>\(\([^\S]*\)\)?\x1b\[[;f-3J</code></td>
<td><code>subdgl/WFC</code></td>
<td>5</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>32</td>
<td><code>\(\([^\S]*\)\)?\x1b\[[;f-3J</code></td>
<td><code>msbosh/sol</code></td>
<td>5</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>33</td>
<td><code>\(\([^\S]*\)\)?\x1b\[[;f-3J</code></td>
<td><code>ubcsanskriti/sansscript.rb</code></td>
<td>5</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

\(^1\) ✓ denotes whether the corresponding method can successfully detect the vulnerability or not.