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

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., nondeterministic 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 (NIDSes) [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-

[*]These authors contributed equally to this work.

perimentally 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, $.^*(aa)^{\{100\}}$ and $.^*a^{\{0,100\}}a^{\{200\}}$ [1] 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 shows a vulnerable regex from the intrusion detection system Snort [25], which specifies the correct string format of classids[2]. Since regexes with *nested* countings, e.g. $(\backslash x26\backslash x23\backslash d^{\{2,3\}}\backslash x3B)^{\{2\}}$, are forbidden from the subclass mentioned above, GadgetCA reports an internal error.

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

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 single 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 *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. $.^*cc.^*(cor.^*ang.^*|heart.^*|icd.^*|pace.^*|cardi.^*|l$ $hc.^*|aortic.^*|.rhc.^*|dual.^*lead.^*|cabg.^*|trans.^*).^*echo$

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) **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) **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 *simple strings* and solving the 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 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 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 *without* explicit construction of finite automata.

Based on the ideas above, we propose EVILSTRGEN, a tool for generating attack strings for ReDoS vulnerable regexes targeting non-backtracking engines. 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, 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 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 EVILSTRGEN.

The main contributions of this paper are listed as follows.

[1] Where $.^*$ accepts any string and the bounded repetition $E^{\{m,n\}}/E^{\{m\}}$ accepts (m to n)/m repetitions of E for finite numbers m,n with $0 \le m \le n$.

[2] Where $\backslash s^*$ matches any whitespace characters 0 or more times, $\backslash xh_1h_2$ is the hexadecimal representation of the ASCII code "h_1h_2", [] represents a character class, which matches any character inside [], so $[\backslash x22\backslash x27]$ matches either a double quotation mark or a single quotation mark, similarly $[A\text{–}Za\text{–}z\backslash :\backslash\{-]$ matches an uppercase or lowercase English letters or ":" or "{" or "–", while $[\wedge\backslash x22\backslash x27]$ matches any character other than a double quotation mark or a single quotation mark, $\backslash d$ matches any digital number.

- **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 descriptional 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 Σ be a finite *alphabet* of symbols (or characters). *Words* (i.e. *strings*) are finite sequences over Σ, and *languages* are sets of words. The *empty word* and the *empty language* are denoted by ϵ and $\varnothing$, 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 $E ::= \epsilon \mid \varnothing \mid a \mid C \mid E|E \mid EE \mid (E) \mid E^{\{m,n\}} \mid E^{\{m,n\}?}$ $\mid \backslash b \mid \backslash B \mid \hat{} \mid \$$, 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 $\backslash p\{Greek\}$, 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* $\backslash b$, *non-word boundary* $\backslash B$, *start-of-line anchor* ^, and *end-of-line anchor* \$. Anchors do not consume characters, but specify the non-character context. For the input "ab", $\backslash b$ specifies positions 0 and 2, whose one side is a word and the other side is not. $\backslash B$ specifies position 1, whose both sides are words (or non-words). And ^ matches position 0, while \$ matches position 2. Notice $E^?$, E^*, E^+, $E^{\{m\}}$, $E^{\{m,\}}$ are abbreviations of $E|\epsilon$, $E^{\{0,+\infty\}}$, $E^{\{1,+\infty\}}$, $E^{\{m,m\}}$ and $E^{\{m,+\infty\}}$ respectively. The *language* of a regex E, denoted as $\mathcal{L}(E)$, 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 $w \in \mathcal{L}(\Sigma^* E \Sigma^*)$), which can be implemented by prepending Σ^* to the left of E [61]; while *full matching* solves the *regex membership* problem (i.e. whether $w \in \mathcal{L}(E)$); 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 $\mathcal{M} = (Q, \Sigma, \delta, s, F)$, where Q is a finite set of states, Σ is an alphabet, $\delta \subseteq Q \times \Sigma \cup \{\epsilon\} \to \wp(Q)$ is a transition function where $\wp(Q)$ 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 $\delta \subseteq Q \times \Sigma \to Q$. NFAs are denoted as ϵNFAs if the transitions can be labeled ϵ, and ϵ-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 ($\mathcal{M}_{MY}$), which is used in Hyperscan [77] whenever possible. Later, another construction (which we denoted as $\mathcal{M}_{grep}$) was implemented and described by Aho in [6], which is still being used by GNU grep [31]. $\mathcal{M}_{grep}$ is always smaller than or equal to $\mathcal{M}_{MY}$ [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*[3] 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) ϵNFA-based engines, such as RE2 [34], the regex engines in Go [32] , Rust [23], as well as TDFA [46] and RE2C [73]; (ii) position-based engines, which are based on $\mathcal{M}_{grep}$, e.g., grep [31], awk [30] and sed [29] or $\mathcal{M}_{MY}$ such as Hyperscan [77]; (iii) derivative-based engines, such as OCaml-re [75], SRM [64] and NonBacktracking[4] [60], by extending Brzozowski's derivatives [14], whose

[3] Generalized 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.

[4] To differentiate the two implementations written in C#, we write $C\#_N$ for NonBacktracking, and $C\#_B$ for the backtracking implementation.

Table 1: Non-backtracking regex engines, categorized.

	ϵNFA	Position	Derivative
Online	RE2, Go, Rust	grep, awk, sed, Hyperscan	SRM, C#$_N$
Offline	TDFA, RE2C	–	OCaml-re

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 $w \in \mathcal{L}(E)$ 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 analyse 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 k-SIMPLE STRING solver INC_DET$'$ to generate ReDoS attack strings *bytewise* with incremental determinisation based on the DFA in the corresponding engine. The search for k-simple strings uses two complete and an incomplete

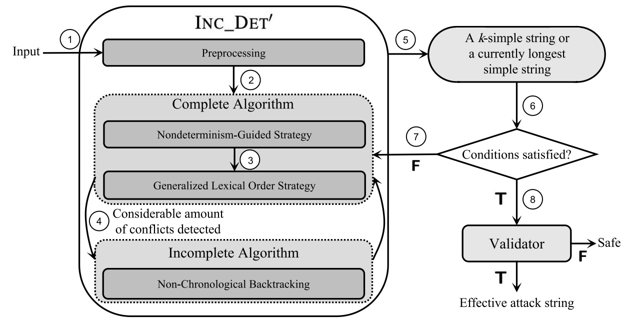

Figure 1: The system architecture of EVILSTRGEN.

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 k-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 1^{st} and 2^{nd} 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 3^{rd} and 4^{th} 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 5^{th} regex has at most 651,608 DFA states in RE2, which illustrates that regexes with only *unbounded countings* can also be ReDoS vulnerable. The 6^{th} 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 2: Time [ms] used to match examples of ReDoS vulnerable regexes by RE2 under different function calls.

No.	Regex	Source	`RE2::FullMatch`		`RE2::PartialMatch`	
			EvilStrGen	GadgetCA	EvilStrGen	GadgetCA
1	$.^{*}a.^{\{300\}}$	[2]	6,273	124	2,154	283
2	$\hat{}.^{*}[aA][pP][oO][pP][\hat{}\backslash x0a]^{\{256\}}$	[26]	1,167	39	1,192	52
3	$\backslash A(\backslash".^{\{60,\}}\backslash"\backslash n)^{\{2\}}\backslash Z$	[22]	7,314	3	7,875	201
4	$classid{=}\backslash s^{*}[\backslash x22\backslash x27]\backslash s^{*}[A\text{-}Za\text{-}z\backslash :\backslash\{\text{-}]^{\{0,42\}}$ $((\backslash x26\backslash x23\backslash d^{\{2,3\}}\backslash x3B)^{\{2\}}[\hat{}\backslash x22\backslash x27]^{\{0,25\}})^{\{5\}}$	[25]	38	*error*	1,612	*error*
5	$filename\backslash x3d[\hat{}\backslash r\backslash n]^{*}(\backslash x2e[\hat{}\backslash x22\backslash x27\backslash r\backslash n]^{\{18,\}}[\backslash x22\backslash x27])$	[25]	5,451	7	5,642	273
6	$.^{*}cc.^{*}(cor.^{*}ang.^{*}\vert heart.^{*}\vert icd.^{*}\vert pace.^{*}\vert cardi.^{*}\vert lhc.^{*}$ $\vert aortic.^{*}\vert .rhc.^{*}\vert dual.^{*}lead.^{*}\vert cabg.^{*}\vert trans.^{*}).^{*}echo$	[4]	2,176	6	2,261	167
7	$t[\backslash d\vert\backslash w]^{\{32\}}$	[2]	2	2	3,706	21
8	$[\backslash p\{L\}\backslash p\{N\}\text{-}\llcorner\!\lrcorner]^{\{1,25\}}\$$	[1]	7	5	2,355	64
9	$(\backslash W\backslash D)^{\{5\}}\$$	Synthetic	3	3	5,789	78

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 $[\backslash p\{L\}\backslash p\{N\}\text{-}\llcorner\!\lrcorner]$ 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 transition 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 recorded by the matcher, 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 $O(mn)$, plus an exponential cost for DFA cache conflict testing [16].

Table 3: The numbers of DFA states of two regexes under different functions and encoding in RE2.

Function	$\backslash W\backslash D\$$		$(\backslash W\backslash D)^{\{5\}}\$$	
	ASCII	UTF-8	ASCII	UTF-8
`RE2::FullMatch`	4	399	12	1,987
`RE2::PartialMatch`	12	**659**	80	**678,980**

The worst-case time complexity of an NFA matcher is $O(m^2n)$ as described in [42]. In [6,20], it is considered $O(mn)$, 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. $C\#_N$ 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 $O(m^2n)$ or $O(mn)$, 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, $C\#_N$, 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. However this modification brings nondeterminism into the regex: For our running example under ASCII encoding, the number of DFA states for $\backslash W \backslash D\$$ triples under partial matching, while that of $(\backslash W \backslash D)^{\{5\}}\$$ is enlarged $5.67\times$. 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 $9.13\times$ 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, α, β and γ represent disjoint subsets of Σ. 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 α by "a", β by "[^$a\backslash n$]", and assigning k as 300, $.^*a.^{\{300\}}$ (i.e., the 1^{st} regex in Table 2, considering that $a|[\hat{}a\backslash n] = .$) is obtained.

Table 4: Examples of regular languages E_k whose minimal DFAs have exponential numbers of states for $k \geq 1$.

Regex	Reference	States
$(\alpha\|\beta)^*\alpha(\alpha\|\beta)^k$	[6,59]	2^{k+1}
$(\alpha\|\beta)^*\alpha(\beta\alpha^*)^k((\beta\alpha^*)^{k+1})^*$	[82]	2^{k+1}
$(\alpha\|\beta)^{\leq k}\alpha(\alpha\|\beta)^k$	[49,65]	$2^{k+2}-2$
$(\alpha\|\beta)^*\alpha(\alpha\|\beta)^k\alpha(\alpha\|\beta)^*$	[24]	$2^{k+1}+1$
$(\alpha^*(\alpha\beta^*)^k\alpha)^*$	[24]	$2^{k+1}+2^{k-1}-1$
$(\alpha\|(\alpha\beta^*)^k\alpha)^*$	[51]	$2^{k+1}-1$
$(\alpha\|\beta)^*(\alpha\|\gamma)\alpha^*((\beta\|\gamma)\alpha^*)^k$	[12]	$2^{k+2}-1$
$(\alpha^*\|(\alpha\beta^*)^k\alpha)^*$	**New variant**	$2^{k+1}-1$
$(\alpha\|(\alpha\beta^?)^k\alpha)^*$	**New variant**	$2^{k+1}+2^{k-2}$
$(\alpha^*(\alpha\beta^?)^k\alpha)^*$	**New variant**	$2^{k+1}+2^k-2$
$\alpha^*((\alpha\beta^?)^k\alpha)^*$	**New variant**	$2^{k+1}+2^k+k-2$

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 has 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 $(\backslash W \backslash D)^{\{5\}}\$$ is $1030\times$ bigger than that of $\backslash W \backslash D\$$, while the size of $(\backslash W \backslash D)^{\{5\}}\$$ is only $1\times$ larger than $\backslash W \backslash D\$$.

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 $a^{\{0,k\}\{0,k\}\overset{n-3}{\frown\cdots}\{0,k\}}$, $\delta(s,a) = (\{a_2, a_3, \ldots, a_{k^n}\})$, requiring at least $O(k^n)$ time to compute. Notice this is only triggered by a single symbol "a".

Nesting of counting and determinisation blow-up jointly lead to doubly-exponentially sized DFAs: For example, Gelade proved the minimal DFA of $E_n{=}(a|b)^*a(a|b)^{\{2^n\}}$ has 2^{2^n} 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 $(a|b)^*a(\overset{n-3}{\frown\cdots}(((a|b)^{\{2\}})^{\{2\}})\overset{n-3}{\frown\cdots})^{\{2\}}$.

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 "$\backslash W$".

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 8^{th} regex in Table 2. Besides, this also enlarges DFAs by a huge constant factor. For the running example under partial matching, when $\backslash W \backslash D\$$ is converted from ASCII to UTF-8, its DFA is enlarged by $55\times$, for $(\backslash W \backslash D)^{\{5\}}\$$ 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 $\mathcal{M}$ of a regex E, a *simple string* is a finite word $w \in \mathcal{L}(E)$ where the states of the path of w on $\mathcal{M}$ 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 descriptional 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 $\mathcal{M}$ is proven to be exponential in the size of $\mathcal{M}$ [37].

For any DFA $\mathcal{M}$ of a regex E, the *longest* simple strings of E correspond to the longest acceptable simple paths (i.e., simple paths form the initial state to a final state) on $\mathcal{M}$. 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 $\mathcal{M}$ of a regex E is to decide whether a simple string of $\mathcal{M}$ 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 $\sqsupseteq_{\mathsf{Nondet}}$ (Equation 1 below) is used to choose the path by the highest degree of nondeterminism and $\sqsubseteq_{\mathsf{Lex}}$ (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., $\mathcal{M}_{grep}$, as an example to introduce our algorithm, i.e., Algorithm 1.

Algorithm 1 is composed of two procedures, Inc_Det that searches for k-simple strings incrementally without explicit construction for automata and Decide that answers Problem 1. Inc_Det 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 $f : X \to Y$ and $g : Y \to Z$ is denoted as $g \circ f : X \to Z$.

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

For a regex E over Σ and a symbol $a \in \Sigma$, we define the following sets, which specify the first, the last and the characters following a from E: $\mathsf{First}(E) = \{b \mid bw \in \mathcal{L}(E), b \in \Sigma, w \in \Sigma^*\}$, $\mathsf{Last}(E) = \{b \mid wb \in \mathcal{L}(E), b \in \Sigma, w \in \Sigma^*\}$, $\mathsf{Follow}(E,a) = \{b \mid uabv \in \mathcal{L}(E), u, v \in \Sigma^*, b \in \Sigma\}$. Let $0 \notin \mathsf{Pos}(E)$, define $\mathsf{Follow}(\overline{E}, 0) = \mathsf{First}(\overline{E})$. For $S \in \wp(\mathsf{Pos}(E))$, let $\mathsf{Follow}(\overline{E}, S) =$

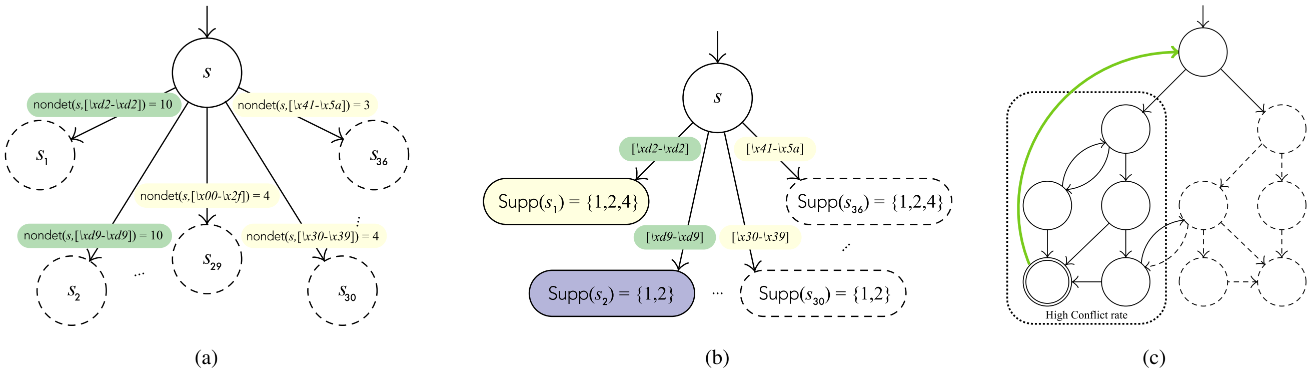

Figure 2: Examples of the three strategies in EVILSTRGEN, where the states or transitions not yet been constructed are shown as dashed lines. The runes preferred by $\sqsupseteq_{\mathsf{Nondet}}$ are highlighted in green, and the state preferred by $\sqsubseteq_{\mathsf{Lex}}$ is in purple, otherwise in beige.

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

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

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 $\overline{E}$. To implement this conversion, we refer to [20], where a hexadecimal-encoded range is called a *rune*. For example, $[a\text{–}z]$ is converted to a rune $[\backslash x61\text{–}\backslash x7a]$. All the runes transformed from a character class share the same labels, i.e. numerical subscripts.

Since $\overline{E}$ 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 $\tau = .^*(\backslash W\backslash D)^{\{5\}}\$$ from running examples to illustrate INC_DET. Example 4 shows the above processing of τ.

Example 4. For $\tau = .^*(\backslash W\backslash D)^{\{5\}}\$$, $\mathsf{unfold}(\tau) = .^*\backslash W\backslash D\backslash W\backslash D\backslash W\backslash D\backslash W\backslash D\backslash W\backslash D\$$, $\mathsf{linearize}\circ\mathsf{unfold}(\tau) = .^*_1\backslash W_2\backslash D_3\backslash W_4\backslash D_5\backslash W_6\backslash D_7\backslash W_8\backslash D_9\backslash W_{10}\backslash D_{11}\$_{12}$. $\overline{\tau} = ([\backslash x00\text{–}\backslash x7f]_1 | [\backslash xc2\text{–}\backslash xdf]_1|\ldots)^*[\backslash x00\text{–}\backslash x2f]_2\ldots\$_{12}$.

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

Nondeterminism-Guided Strategy.

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

1 INC_DET(Regex E, Int k) $\rightarrow$ String
2 **begin**
3 $\quad \overline{E} \leftarrow \mathsf{encode} \circ \mathsf{linearize} \circ \mathsf{unfold}(E)$;
4 $\quad (wit, ret, cur, Q) \leftarrow (\epsilon, \epsilon, \{0\}, \{\varnothing\})$;
5 $\quad$ *loop:*
6 $\quad$ **for** $\mathsf{Follow}(\overline{E}, cur) \neq \varnothing$ **do**
7 $\quad\quad (\mathcal{C}, \mathcal{P}) \leftarrow (\varnothing, \mathsf{Follow}(\overline{E}, cur))$;
8 $\quad\quad$ **for** $\alpha \in \langle \mathsf{Part}(\mathcal{P}); \sqsupseteq_{\mathsf{Nondet}} \rangle$ **do**
9 $\quad\quad\quad (q, o) \leftarrow (\{p \in \mathcal{P} \mid \alpha \subseteq \overline{p}\}, q \cap \mathsf{last}(\overline{E}) = \varnothing)$;
10 $\quad\quad\quad \mathcal{C} \leftarrow \mathcal{C} \cup \{(\alpha, (\mathsf{Follow}(\overline{E}, q), o))\}$;
11 $\quad\quad$ **for** $([\mathcal{A}]_{\equiv_{\mathsf{Nondet}}}, \mathcal{S}) \subseteq \mathcal{C}$ **do**
12 $\quad\quad\quad$ **for** $(\alpha, s) \in ([\mathcal{A}]_{\equiv_{\mathsf{Nondet}}}, \langle \mathcal{S}; \sqsubseteq_{\mathsf{Lex}} \rangle)$ **do**
13 $\quad\quad\quad\quad$ **if** $s \notin Q$ **then**
14 $\quad\quad\quad\quad\quad (wit, cur, Q) \leftarrow$
$\quad\quad\quad\quad\quad\quad (wit + \mathsf{random}(\alpha), \mathcal{P}, Q \cup \{\mathcal{P}\})$;
15 $\quad\quad\quad\quad\quad$ **if** $s \in F_{grep}$ **then**
16 $\quad\quad\quad\quad\quad\quad$ **if** $|wit| \geq k$ **then**
17 $\quad\quad\quad\quad\quad\quad\quad$ **return** *wit*;
18 $\quad\quad\quad\quad\quad\quad$ **break**;
19 $\quad\quad\quad\quad\quad$ **else if** $|wit| > |ret|$ **then**
20 $\quad\quad\quad\quad\quad\quad ret \leftarrow wit$;
21 $\quad\quad\quad\quad\quad$ **goto** *loop*;
22 $\quad\quad\quad\quad$ **else**
23 $\quad\quad\quad\quad\quad$ **continue**;
24 $\quad$ **return** *ret*;
25 DECIDE(Regex E, Int k) $\rightarrow$ Bool
26 **begin**
27 $\quad$ **return** $|\text{INC\_DET}(E,k)| \geq k$;

Motivation. Nondeterminism is a well-established concept in automata theory [49]. Maňdl [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 ϵ-free NFA $\mathcal{M} = (Q, \Sigma, \delta, s, F)$, the *degree of nondeterminism*[5] of a state $q \in Q$ w.r.t a symbol $a \in \Sigma$,

[5] Also called *branching* in [33].

denoted as $\mathsf{nondet}(q,a)$, is the number of nondeterministic transitions labeled a from the state q, which is computed as $\mathsf{nondet}(q,a) = |\delta(q,a)|$.

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

Method. Nondeterminism-guided strategy chooses a rune 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 $\mathcal{M}_{grep}(E)$, let $\mathsf{nondet}(s,\sigma) = |\{\mathsf{Follow}(\overline{E},\overline{q}) \mid s = (\mathcal{Q},\_), \forall q \in \overline{\mathcal{Q}}, \sigma \subseteq q\}|$, for $s \in Q_{grep}$ and $\sigma \in \wp(\Sigma)$. Next we define the following binary relation $\sqsubseteq_{\mathsf{Nondet}} \subseteq \wp(\Sigma) \times \wp(\Sigma)$: for $s \in Q_{grep}$ and $\sigma, \sigma' \in \wp(\Sigma)$, $\sigma \sqsubseteq_{\mathsf{Nondet}} \sigma'$ if and only if

$$\mathsf{nondet}(s,\sigma) \leq \mathsf{nondet}(s,\sigma'). \tag{1}$$

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

Example 5. Consider a state $s = (\mathcal{P}, \mathbf{F})$ from $\mathcal{M}_{grep}(\tau)$ in Figure 2 (a), where $\mathcal{P} = \{[\backslash x00\text{-}\backslash x7f]_1, [\backslash xc2\text{-}\backslash xdf]_1, \ldots, [\backslash xf4\text{-}\backslash xf4]_3\}$. Then $\mathsf{Part}(\overline{\mathcal{P}}) = \{[\backslash x00\text{-}\backslash x2f], \ldots, [\backslash xf5\text{-}\backslash xff]\}$. For $[\backslash x41\text{-}\backslash x5a]$, ..., $[\backslash xd2\text{-}\backslash xd2], [\backslash xd9\text{-}\backslash xd9] \in \mathsf{Part}(\overline{\mathcal{P}})$, $\mathsf{nondet}(s,[\backslash xd2\text{-}\backslash xd2]) = 10$, $\mathsf{nondet}(s,[\backslash xd9\text{-}\backslash xd9]) = 10$, ..., and $\mathsf{nondet}(s,[\backslash x41\text{-}\backslash x5a]) = 2$. $\langle \mathsf{Part}(\overline{\mathcal{P}}); \sqsupseteq_{\mathsf{Nondet}} \rangle = \{[\backslash xd2\text{-}\backslash xd2], [\backslash xd9\text{-}\backslash xd9], \ldots, [\backslash x41\text{-}\backslash x5a]\}$.

Effect. Choosing the rune 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 $\mathcal{M}_{grep}$ states, consequently decreasing backtracks 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 $\langle \mathsf{Part}(\overline{\mathcal{P}}); \sqsupseteq_{\mathsf{Nondet}} \rangle$ usually consists of a large number of runes, where plenty of those have equal degrees of nondeterminism. To select a rune that reduces the chance of a search reaching a final state early, we introduce the generalized lexical order strategy.

Definition 5. (See [11].) Let $\langle \Sigma; \leqq \rangle$ be a partially ordered finite alphabet. The *generalized lexical order* on Σ^* is defined as follows: Let $u = a_0 \ldots a_h$ and $v = b_0 \ldots b_k$ be two words (where $a_0, \ldots, a_h, b_0, \ldots, b_k \in \Sigma$). u, v are ordered, i.e., $u \leq_{\mathsf{Lex}} v$, if and only if $|u| < |v|$, or $|u| = |v|$ and (1) u is a prefix of v or (2) $u = pa_ix$, $v = pb_iy$ and $a_i <_{\mathsf{Lex}} b_i$, where p is the longest common prefix of u and v.

For $a, b \in \langle \mathsf{Pos}(E); \leq_{\mathsf{Lex}} \rangle$, define $a \equiv_{\mathsf{Lex}} b$ to indicate that the labels of a and b are identical. When Algorithm 1 reaches a state $s' = (P', \_)$, for $a_0, \ldots, a_n \in \langle \mathsf{Part}(\overline{P'}); \sqsupseteq_{\mathsf{Nondet}} \rangle$, the simple strings going through s' can be represented as $u = pa_ix$, where $0 \leq i \leq n$. Since elements in $\langle \mathsf{Part}(\overline{P'}); \sqsupseteq_{\mathsf{Nondet}} \rangle$ are disjoint, p is the longest common prefix among $pa_0, \ldots, pa_n$. We then utilize the generalized lexical order to choose the DFA states (e.g. from $\delta_{grep}(s', a_0), \ldots, \delta_{grep}(s', a_n)$) 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 $\mathcal{M}_{grep}(E)$ has an ordered set from $\langle \mathsf{Pos}(E); \leq_{\mathsf{Lex}} \rangle$ and a multiset of labels, i.e. a partially ordered multiset 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 $\mathsf{Supp}(q) = \{[Q]_{\equiv_{\mathsf{Lex}}} \mid q = (Q, \_)\}$, for a state $q \in Q_{grep}$. Next we define the following binary relation $\sqsubseteq_{\mathsf{Lex}} \subseteq Q^2_{grep}$:

Definition 6. For $q_1, q_2 \in Q_{grep}$, $p_0, \ldots, p_n \in \mathsf{Supp}(q_1)$ and $p'_0, \ldots, p'_m \in \mathsf{Supp}(q_2)$, $q_1 \sqsubseteq_{\mathsf{Lex}} q_2$ if and only if

1. $q_1 \in F_{grep} \Leftarrow q_2 \in F_{grep}$,
2. $n < m \vee n = m \wedge \exists 0 < j \leq n, \forall i < j, p'_i = p_i \wedge p'_j <_{\mathsf{Lex}} p_j$.

When the supports of two states are identical, 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 $\mathcal{A}$ into *equivalence classes* based on $\equiv_{\mathsf{Nondet}}$. For each candidate state sorted by $\sqsubseteq_{\mathsf{Lex}}$ (line 12), we start to test whether it is already recorded in $\mathcal{Q}$ in line 13. If not, the current state *cur* is assigned as the candidate state s; moreover, $\mathcal{P}$ is recorded in $\mathcal{Q}$, and a random symbol from α is appended to *wit* in line 14. If the state s is not a final state (line 15) and $|wit|$ has exceeded $|ret|$ (line 19), *ret* is replaced with *wit* (line 20). Then INC_DET continues the search in line 21. If s is already visited, INC_DET continues the search in line 23 and chooses another pair of (α, s) from $([\mathcal{A}]_{\equiv_{\mathsf{Nondet}}}, \langle \mathcal{S}; \sqsubseteq_{\mathsf{Lex}} \rangle)$.

Example 6. For the state s from Example 5, $\mathsf{Supp}(s) = \{\{[\backslash x00\text{-}\backslash x7f]_1, \ldots, [\backslash xe1\text{-}\backslash xef]_1\}, \{[\backslash x00\text{-}\backslash x2f]_2, \ldots, [\backslash xf4\text{-}\backslash xf4]_2\}, \{[\backslash x00\text{-}\backslash x2f]_3, \ldots, [\backslash xf4\text{-}\backslash xf4]_3\}\}$[6]. In Figure 2 (b), s_1 and s_2 are both resulted from runes with the highest degree of nondeterminism at 10 in Figure 2 (a). Since $\mathsf{Supp}(s_1) = \{1,2,4\}$ and $\mathsf{Supp}(s_2) = \{1,2\}$, $\langle \{s_1, s_2\}; \sqsubseteq_{\mathsf{Lex}} \rangle = \{s_2, s_1\}$, i.e. $s_2 \sqsubseteq_{\mathsf{Lex}} s_1$. 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 $\sqsubseteq_{\mathsf{Lex}}$, we maintain the consistency with engines using Greedy rules and also select the DFA states that are less likely to lead to early termination.

[6] For ease of representation, we write $\mathsf{Supp}(s) = \{1,2,3\}$.

```
1 (b, Q') ← AnalyzeConfl(b, wit, Q);
2 if b ≥ CONFLICT_RATE then
3     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 $\mathcal{M}_{grep}$ have been fully explored, yet there does not exist a k-simple string, INC_DET returns the longest simple string found in line 24. By deciding whether the length of output from INC_DET exceeds k in line 27, the algorithm DECIDE solves k-SIMPLE STRING for $\mathcal{M}_{grep}$ in a sound and complete manner.

Theorem 2. DECIDE(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 INC_DET, resulting in INC_DET$'$. This algorithm allows 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 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 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 *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 EVILSTRGEN, our C++ implemention[9] of the methodology presented.

Research Questions. We conducted experiments to compare EVILSTRGEN with the current state-of-the-art ReDoS detectors to answer the following research questions (RQs):

RQ1: How is the capacity of EVILSTRGEN compared to the state-of-the-art ReDoS detectors on large-scale real-world regex benchmarks?

RQ2: Specifically, how is the performance of EVILSTRGEN compared to GadgetCA [74] on the datasets that GadgetCA supports?

RQ3: How do different heuristic strategies described in §5 affect the efficiency of EVILSTRGEN?

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 EVILSTRGEN 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#$_\mathrm{N}$, 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 EVILSTRGEN, 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 *characters* takes the (backtracking) engines 10*s* to match; Regulator [56] uses 1M characters for 10*s* 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 $\frac{k}{20}$.

[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.

criterion; To obtain attack strings, ReDoSHunter [52] repeats "pump" strings 30,000 times and tests if the string takes engine more than 1s to match; [78] repeats pump strings 15,000 times and tests if the string takes Java's regex engine more than 10^5 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-disimilar 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 stable releases and configured in the settings reported in their original documents. A 10 minutes timeout was used. We cross-verified the strings generated by each detector for concrete vulnerabilities on engines to establish the ground truth.

We experiment with different configurations of our algorithm as follows. We denote the algorithm with all three strategies off as BRUTE-FORCE; When only the nondeterminism-guided strategy is enabled, the configuration is denoted as NONDETON; The tool with both nondeterminism-guided and generalized lexical order strategies enabled is denoted as LEXICALON; The default configuration in EVILSTRGEN, with all strategies enabled, is denoted as ALLSTRATON.

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 Toolkit[12], 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 $|prefix| + |pump| \times n + |suffix| \approx 100$kB.

[12] http://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#$_{\mathrm{N}}$, 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 bytewise 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 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 $\geq$100kB/s) to enhance the visibility of the vulnerable ones. Points below the diagonal suggest EvilStrGen causes lower throughput, otherwise GadgetCA does. We also calculate the percentage of regexes for which EVILSTRGEN causes a more severe slowdown (e.g., "wins in 100.00% regexes"). EVILSTRGEN wins in 85.71% , 100%, 91.06%, 98.57%, 97.90%, 100%, 56.86%, 99.95% vulnerable regexes on RE2, Rust, Go, SRM, C#$_{\mathrm{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

Table 5: Comparison on the effectiveness of different configurations of EVILSTRGEN and baselines on SET736535.

		Non-backtracking Regex Engines								Backtracking Regex Engines							
		RE2	Rust	Go	SRM	$C\#_N$	awk	grep	Hyperscan	Java	JavaScript	PCRE2	Perl	php	Python	Boost	$C\#_B$
EVILSTRGEN	BRUTE-FORCE	175	29	196	149	182	211	179	41	4,142	3,873	1,438	1,202	121	3,982	800	3,334
	NONDETON	470	80	536	653	499	586	651	219	11,318	10,617	3,907	3,267	338	10,945	2,194	9114
	LEXICALON	1,296	221	1,470	2,672	7,636	2,115	1,417	731	31,090	29,106	10,783	8,933	928	30,017	5,940	24,936
	ALLSTRATON	**3,479**	**728**	**4,737**	**8,404**	**23,872**	**5,156**	**3,018**	**2,327**	**51,365**	**48,120**	**17,890**	14,816	1,531	**49,605**	**9,820**	**41,236**
GadgetCA		1,264	226	724	1,954	5,370	3,938	2,884	4	10,305	9,230	9825	2,289	237	10,360	899	11,335
RegexStatic		6	6	6	3	40	137	154	0	3,516	3,937	816	1,988	475	3,797	935	3,458
Regexploit		2	0	1	0	21	23	26	0	1,552	1,637	493	668	42	1,837	900	1545
ReScue		1	0	4	0	9	1	1	1	142	190	41	45	2	200	118	174
Regulator		243	241	478	17	257	449	517	5	34,076	35,516	6,639	4,285	1,287	34,335	4,083	19,060
ReDoSHunter		311	253	299	69	324	1,967	1,623	3	37,900	38,323	16,257	**16,280**	**1,554**	45,210	6,009	30,289

Table 6: Performance of EVILSTRGEN (ALLSTRATON) on different subclasses of regexes from SET736535.

	RE2	Rust	Go	SRM	$C\#_N$	awk	grep	Hyperscan	Total
Bounded_Counting-free	86	108	768	1,652	5,499	454	369	116	412,785
Nested Counting	256	4	36	51	322	80	39	93	6,727
Unbounded Counting	1,461	602	3,730	5,534	16,073	4,118	2,118	1,074	285,516
Discrete Char Classes	2,133	488	3,260	4,776	13,229	2,458	1,367	2,148	239,319
Finite Languages	1,490	2	114	665	1,274	398	372	1,047	314,294

Table 7: Comparison on the effectiveness of different configurations of EVILSTRGEN and GadgetCA on ABOVE20.

		RE2	Rust	Go	SRM	$C\#_N$	awk	grep	Hyperscan
EVILSTRGEN	BRUTE-FORCE	19	2	4	10	1	3	4	3
	NONDETON	43	4	2	51	20	15	23	15
	LEXICALON	87	11	12	96	44	26	68	22
	ALLSTRATON	**294**	**30**	**23**	**165**	**57**	**74**	**104**	**32**
GadgetCA		16	9	5	29	6	28	37	2

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 EVILSTRGEN 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. EVILSTRGEN 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 EVILSTRGEN 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. Owning to the effectiveness of the *k*-simple strings, EVILSTRGEN 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, EVILSTRGEN is far more effective than all baselines, detecting overall $2.16-3{,}042.41\times$ times more ReDoS vulnerabilities. Besides, EVILSTRGEN 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 EVILSTRGEN 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, EVILSTRGEN 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 EVILSTRGEN 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 EVILSTRGEN 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. EVILSTRGEN 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. EVILSTRGEN 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 EVILSTRGEN 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-

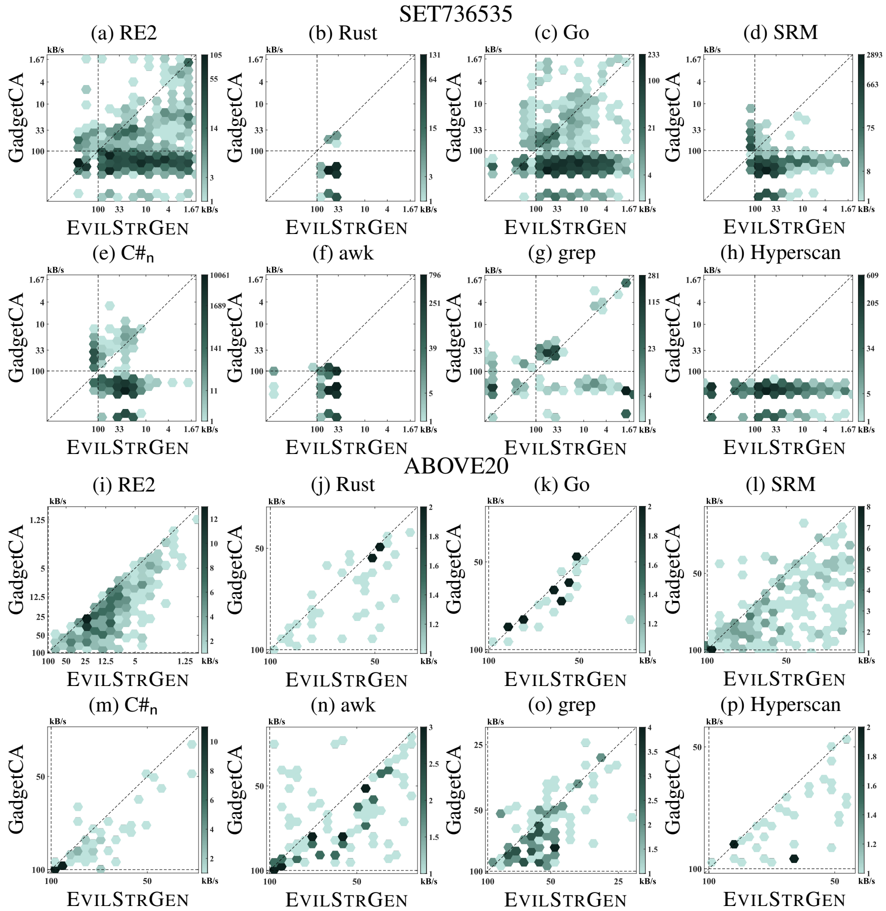

Figure 4: Throughput rate [kB/s] comparison of non-backtracking engines to match strings generated by EVILSTRGEN and GadgetCA.

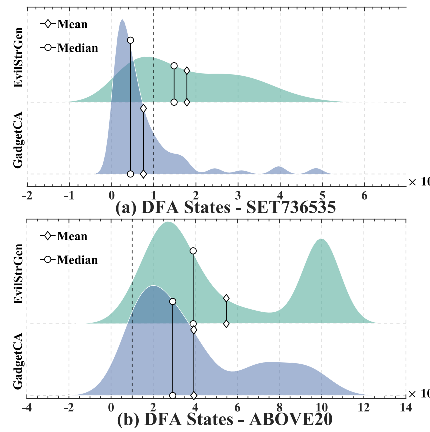

Figure 5: DFA states in RE2 to match strings generated by EVILSTRGEN and GadgetCA.

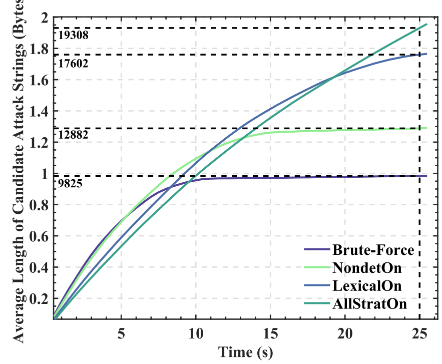

Figure 6: The average length of candidate attack strings generated by different configurations of EVILSTRGEN on SET736535.

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. `unknwon/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 vulnerability
func StripTags(src string) string {
    re := regexp.MustCompile('(?s)<(?:style|script)[^<>]*>.*?</(?:style|script)
        >|</?[a-z][a-z0-9]*[^<>]*>|<!–.*?–>')
    src = re.ReplaceAllString(src, "")
  ...
}
// PoC: An 10MB input attack string was injected into CLI interface of the program
, taking 769s for the program to process.
StripTags(" <!- -\x04+\x05\x16\x0B<!- -<!- -X\x1A?<!- -LE*\x1A<!- -I\x09\x18
    \xF1<!- -\x04+\x05\x16\x0<!- -<!- -X\x1A?<!- -LE*\x1A<!- -I\x09\x18
    \x3C< <!- -\x04+\x05\x16\x0B... ")
```

Figure 7: A vulnerable regex in package `unknwon/com`.

moval is vulnerable to ReDoS. A PoC targeting this vulnerability significantly slows down the program, with the throughput rate of the engine dropping below 11kB/s and causing a 99% CPU load on average during the search on the attack string. This vulnerability affects 54 projects, posing significant risks, especially to cloud services, by potentially causing hardware failures or system shutdowns. Key affected projects include `seccome/Ehoney`, a honeypot management system, and `huaweicloud/external-sfs`, a cloud storage solution on Huawei Cloud, where disruptions could severely impact network security and data availability. Additionally, vulnerabilities in cloud management platforms like `nttcom/terraform-provider-ecl` could lead to extensive service outages and endangering data integrity.

Summary. The results show the high usability of EVILSTRGEN to reveal ReDoS vulnerabilities for programs using non-backtracking engines in practice.

6.6 Discussion

Limitations. Though EVILSTRGEN performs well in practice, there are following directions to improve. Firstly, the algorithm selection in engines, e.g. from DFA to NFA, is rather complex. As shown in the experiment, attack strings generated for backtracking engines might also perform well on grep. EVILSTRGEN should be combined with a non-trivial extension to **dynamic symbolic execution** to monitor the engines and know from which byte the different matchers are used and switch to generating targeted attack strings. Secondly, EVILSTRGEN selects the runes by the degree of nondeterminism *greedily*. However this does not guarantee to **maximize the degree of nondeterminism** in the whole attack string, or even performs worse than baselines. To avoid this, more comprehensive branching strategies should be introduced to avoid finding sub-optimal attack strings.
Discussion for Practioners. Cox has given several good suggestions on using non-backtracking engines in [20]. Apart from these, we strongly suggest using regexes from some rather safe subclasses of regexes for online DFA matchers, such as deterministic regular expressions [15, 19]. We also suggest to *not* using the structures and operators that may result in high descriptional complexity as mentioned in §4.2.

7 Related Work

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. RXXR2 [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 determinisation 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

[1] Maven. https://maven.apache.org/.

[2] NuGet. https://www.nuget.org/.

[3] Pcre - perl compatible regular expressions. http://www.pcre.org/.

[4] Regex101. https://regex101.com.

[5] The Sagan Log Analysis Engine. https://quadrantsec.com/sagan_log_analysis_engine/.

[6] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2006.

[7] A. Asperti, C. S. Coen, and E. Tassi. Regular Expressions, *au point*, 2010. arXiv:1010.2604[cs].

[8] E. Barlas, X. Du, and J. C. Davis. Exploiting Input Sanitization for Regex Denial of Service. In *ICSE 2022*, pages 883–895, 2022.

[9] A. Bartoli, A. de Lorenzo, E. Medvet, and F. Tarlao. Inference of Regular Expressions for Text Extraction from Examples. *IEEE Trans. Knowl. Data Eng.*, 28(5):1217–1230, 2016.

[10] M. H. M. Bhuiyan, A. S. Parthasarathy, N. Vasilakis, M. Pradel, and C.-A. Staicu. SECBENCH.JS: An Executable Security Benchmark Suite for Server-Side JavaScript. In *ICSE 2023*, pages 1059–1070, 2023.

[11] J.-C. Birget. Partial Orders on Words, Minimal Elements of Regular Languages, and State Complexity. *Theor. Comput. Sci.*, 119(2):267–291, 1993.

[12] H. Bordihn, M. Holzer, and M. Kutrib. Determination of Finite Automata Accepting Subregular Languages. *Theor. Comput. Sci.*, 410(35):3209–3222, 2009.

[13] S. Broda, M. Holzer, E. Maia, N. Moreira, and R. Reis. A Mesh of Automata. *Inf. Comput.*, 265:94–111, 2019.

[14] J. A. Brzozowski. Derivatives of Regular Expressions. *J. ACM*, 11(4):481–494, 1964.

[15] A. Brüggemann-Klein. Regular Expressions into Finite Automata. *Theor. Comput. Sci.*, 120:197–213, 1993.

[16] J.-M. Champarnaud. Subset Construction Complexity for Homogeneous Automata, Position Automata and ZPC-Structures. *Theor. Comput. Sci.*, 267(1-2):17–34, 2001.

[17] C. Chapman and K. T. Stolee. Exploring Regular Expression Usage and Context in Python. In *ISSTA 2016*, pages 282–293, 2016.

[18] C. Chapman, P. Wang, and K. T. Stolee. Exploring Regular Expression Comprehension. In *ASE 2017*, pages 405–416, 2017.

[19] H. Chen and P. Lu. Checking Determinism of Regular Expressions with Counting. *Inf. Comput.*, 241:302–320, 2015.

[20] R. Cox. Regular Expression Matching in the Wild, 2010. https://swtch.com/~rsc/regexp/regexp3.html.

[21] J. C. Davis, C. A. Coghlan, F. Servant, and D. Lee. The Impact of Regular Expression Denial of Service (ReDoS) in Practice: an Empirical Study at the Ecosystem Scale. In *ESEC/FSE 2018*, pages 246–256, 2018.

[22] J. C. Davis, L. G. Michael IV, C. A. Coghlan, F. Servant, and D. Lee. Why Aren't Regular Expressions a Lingua Franca? An Empirical Study on the Re-Use and Portability of Regular Expressions. In *ESEC/FSE 2019*, pages 443–454, 2019.

[23] Docs.rs. "regex - Rust", 2023. https://docs.rs/regex/1.9.1/regex/.

[24] K. Ellul, B. Krawetz, J. O. Shallit, and M.-w. Wang. Regular Expressions: New Results and Open Problems. *J. Autom. Lang. Comb.*, 10(4):407–437, 2005.

[25] M. Roesch et al. Snort: A Network Intrusion Detection and Prevention System. http://www.snort.org.

[26] R. Sommer et al. The Bro Network Security Monitor. http://www.bro.org.

[27] W. Gelade. Succinctness of Regular Expressions with Interleaving, Intersection and Counting. *Theor. Comput. Sci.*, 411(31):2987–2998, 2010.

[28] D. Giammarresi, J.-L. Ponty, and D. Wood. Thompson Languages. In *Jewels are Forever, Contributions on Theoretical Computer Science in Honor of Arto Salomaa*, pages 16–24. 1999.

[29] GNU. "GNU Sed: a stream editor", 2020. https://www.gnu.org/software/sed/manual/.

[30] GNU. "Gawk: Effective AWK Programming", 2023. https://www.gnu.org/software/gawk/manual/.

[31] GNU. "GNU Grep: Print lines matching a pattern", 2023. https://www.gnu.org/software/grep/manual/.

[32] Go. Go regexp, 2023. https://pkg.go.dev/regexp.

[33] J. Goldstine, C.M.R. Kintala, and D. Wotschke. On measuring nondeterminism in regular languages. *Inf. Comput.*, 86(2):179–194, 1990.

[34] Google. RE2, 2023. https://github.com/google/re2.

[35] J. Goyvaerts. Popular tools, utilities and programming languages that support regular expressions, 2021. https://www.regular-expressions.info/tools.html.

[36] S. Haber, W. Horne, P. Manadhata, M. Mowbray, and P. Rao. Efficient Submatch Extraction for Practical Regular Expressions. In *LATA 2013*, pages 323–334, 2013.

[37] Y.-S. Han, G. Trippen, and D. Wood. Simple-Regular Expressions and Languages. *J. Autom. Lang. Comb.*, 12(1-2):181–194, 2007.

[38] S. A. Hassan, Z. Aamir, D. Lee, J. C. Davis, and F. Servant. Improving Developers' Understanding of Regex Denial of Service Tools through Anti-Patterns and Fix Strategies. In *S&P 2023*, pages 1238–1255, 2023.

[39] M. Holzer and M. Kutrib. The Complexity of Regular(-like) Expressions. In *DLT 2010*, pages 16–30, 2010.

[40] L. Holík, J. Síč, L. Turoňová, and T. Vojnar. Fast Matching of Regular Patterns with Synchronizing Counting. In *FoSSaCS 2023*, pages 392–412, 2023.

[41] P. Hooimeijer and M. Veanes. An Evaluation of Automata Algorithms for String Analysis. In *VMCAI 2011*, pages 248–262, 2011.

[42] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[43] D. Hovland. Regular Expressions with Numerical Constraints and Automata with Counters. In *ICTAC 2009*, volume 5684, pages 231–245. 2009.

[44] A. Hume. A Tale of Two Greps. *Softw. - Pract. Exp.*, 18(11):1063–1072, 1988.

[45] L. G. Michael IV, J. Donohue, J. C. Davis, D. Lee, and F. Servant. Regexes are Hard: Decision-Making, Difficulties, and Risks in Programming Regular Expressions. In *ASE 2019*, pages 415–426, 2019.

[46] Christopher K. regex-tdfa: A new all haskell tagged dfa regex engine, inspired by libtre., 2023. https://github.com/haskell-hvr/regex-tdfa.

[47] D. Karger, R. Motwani, and G. D. S. Ramkumar. On Approximating the Longest Path in a Graph. *Algorithmica*, 18(1):82–98, 1997.

[48] P. Kilpeläinen and R. Tuhkanen. Regular Expressions with Numerical Occurrence Indicators - preliminary results. In *SPLST'03*, pages 163–173, 2003.

[49] C. M. R. Kintala and D. Wotschke. Amounts of Nondeterminism in Finite Automata. *Acta Inform.*, 13(2):199–204, 1980.

[50] A. Le Glaunec, L. Kong, and K. Mamouras. Regular Expression Matching using Bit Vector Automata. In *OOPSLA 2023*, pages 492–521, 2023.

[51] H. Leung. Separating Exponentially Ambiguous Finite Automata from Polynomially Ambiguous Finite Automata. *SIAM J. Comput.*, 27(4):1073–1082, 1998.

[52] Y. Li, Z. Chen, J. Cao, Z. Xu, Q. Peng, H. Chen, L. Chen, and S. C. Cheung. ReDoSHunter: A Combined Static and Dynamic Approach for Regular Expression DoS Detection. In *USENIX Security '21*, pages 3847–3864, 2021.

[53] Y. Liu, M. Zhang, and W. Meng. Revealer: Detecting and Exploiting Regular Expression Denial-of-Service Vulnerabilities. In *S&P 2021*, pages 1468–1484, 2021.

[54] R. Mańdl. Precise bounds associated with the subset construction on various classes of nondeterministic finite automata. In *CISS 1973*, pages 263–267, 1973.

[55] J. P. Marques Silva and K. A. Sakallah. Grasp - A New Search Algorithm for Satisfiability. In *The Best of ICCAD*, pages 73–89. 2003.

[56] R. McLaughlin, F. Pagani, N. Spahn, C. Kruegel, and G. Vigna. Regulator: Dynamic Analysis to Detect ReDoS. In *USENIX Security '22*, pages 4219–4235, 2022.

[57] R. McNaughton and H. Yamada. Regular Expressions and State Graphs for Automata. *IRE Trans. Electron. Comput.*, 9:39–47, 1960.

[58] Microsoft. CredScan, 2021. https://secdevtools.azurewebsites.net/helpcredscan.html.

[59] F. R. Moore. On the Bounds for State-Set Size in the Proofs of Equivalence Between Deterministic, Nondeterministic, and Two-Way Finite Automata. *IEEE Trans. Comput.*, C-20(10):1211–1214, 1971.

[60] D. Moseley, M. Nishio, Jose Perez R., O. Saarikivi, S. Toub, M. Veanes, T. Wan, and E. Xu. Derivative Based Nonbacktracking Real-World Regex Matching with Backtracking Semantics. *PLDI 2023*, 7:1026–1049, 2023.

[61] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences*. Cambridge University Press, 2002.

[62] M. O. Rabin and D. Scott. Finite Automata and Their Decision Problems. *IBM J. Res. Dev.*, 3(2):114–125, 1959.

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

[64] O. Saarikivi, M. Veanes, T. Wan, and E. Xu. Symbolic Regex Matcher. In *TACAS 2019*, pages 372–378, 2019.

[65] K. Salomaa and S. Yu. NFA to DFA Transformation for Finite Languages. In *WIA 1996*, pages 149–158, 1996.

[66] Y. Shen, Y. Jiang, C. Xu, P. Yu, X. Ma, and J. Lu. Rescue: Crafting Novel Regular Expression DoS Attacks. In *ASE 2018*, pages 225–235, 2018.

[67] Snyk. The State of Open-source Security, 2020. https://snyk.io/.

[68] H. Spencer. A Regular Expression Matcher. In *Software Solutions in C*, pages 35–71. 1994.

[69] C.-A. Staicu and M. Pradel. Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers. In *USENIX Security '18*, pages 361–376, 2018.

[70] M. Sulzmann and K. Z. M. Lu. POSIX Regular Expression Parsing with Derivatives. In *FLOPS 2014*, pages 203–220, 2014.

[71] A. Syropoulos. Mathematics of Multisets. In *Multiset Processing*, pages 347–358, 2001.

[72] R. E. Tarjan. Efficiency of a Good But Not Linear Set Union Algorithm. *J. ACM*, 22(2):215–225, 1975.

[73] U. Trofimovich. RE2C: A Lexer Generator Based on Lookahead-TDFA. *Softw. Impacts*, 6:100027, 2020.

[74] L. Turoňová, L. Holík, I. Homoliak, O. Lengál, M. Veanes, and T. Vojnar. Counting in Regexes Considered Harmful: Exposing ReDoS Vulnerability of Nonbacktracking Matchers. In *USENIX Security '22*, pages 4165–4182, 2022.

[75] Jerome V. Ocaml-re, 2023. https://github.com/ocaml/ocaml-re.

[76] P. Wang and K. T. Stolee. How Well are Regular Expressions Tested in the Wild? In *ESEC/FSE 2018*, pages 668–678, 2018.

[77] X. Wang, Y. Hong, H. Chang, G. Langdale, and J. Hu. Hyperscan: A Fast Multi-pattern Regex Matcher for Modern CPUs. In *NSDI 19*, pages 631–648, 2019.

[78] X. Wang, C. Zhang, Y. Li, Z. Xu, S. Huang, Y. Liu, Y. Yao, Y. Xiao, Y. Zou, Y. Liu, and W. Huo. Effective ReDoS Detection by Principled Vulnerability Modeling and Exploit Generation. In *S&P 2023*, pages 2427–2443, 2023.

[79] N. Weideman, B. van der Merwe, M. Berglund, and B. Watson. Analyzing Matching Time Behavior of Backtracking Regular Expression Matchers by Using Ambiguity of NFA. In *CIAA 2016*, pages 322–334, 2016.

[80] V. Wüstholz, O. Olivo, M. J. H. Heule, and I. Dillig. Static Detection of DoS Vulnerabilities in Programs that Use Regular Expressions. In *TACAS 2017*, pages 3–20.

[81] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz. Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection. In *ANCS06*, pages 93–102, 2006.

[82] S. Yu, Q. Zhuang, and K. Salomaa. The State Complexities of Some Basic Operations on Regular Languages. *Theor. Comput. Sci.*, 125(2):315–328, 1994.

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:
$E^{\{0,\}} \to E^*$, $E^{\{0,\}?} \to E^{*?}$, $E^{\{0,0\}} \to \varnothing$, $E^{\{0,0\}?} \to \varnothing$,
$E^+ \to EE^*$, $E^{+?} \to EE^{*?}$,
$E^{\{m,n\}} \to EE^{\{m-1,n-1\}}$, $E^{\{m,n\}?} \to EE^{\{m-1,n-1\}?}$, for $n \geq m > 0$,
$E^{\{0,n\}} \to E^?E^{\{0,n-1\}}$, $E^{\{0,n\}?} \to E^{??}E^{\{0,n-1\}?}$, for $n > 1$,
$E^{\{m,\}} \to EE^{\{m-1,\}}$, $E^{\{m,\}?} \to EE^{\{m-1,\}?}$, 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 Σ, the NON-EMPTY COMPLEMENT problem of E is to decide whether $\mathcal{L}(\neg E) \neq \varnothing$. In [39], this problem is proven to be EXPSPACE-complete.

We note that a DFA $\mathcal{M}$ of E can be easily completed [42], thus we assume that all DFAs under consideration are complete and the sink state is denoted as d. Define the complemented $\mathcal{M}$ as $\mathcal{M}^c = (Q^c, \Sigma, \delta^c, s^c, F^c)$, where $F^c = (Q - F) \cup$

$\{d\}$. We modify the Definition 2 as a simple string w^c is a finite word $w \in \mathcal{L}(\mathcal{M}^c)$, where the states of the path of w on $\mathcal{M}^c$ of E contains only one final state and are pairwise distinct. This transformation is done in polynomial time. Therefore, any 0-simple string w^c of $\mathcal{M}^c$ of E satisfies $w^c \in \mathcal{L}(\neg E)$, i.e. an witness of NON-EMPTY COMPLEMENT. □

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 **T**, consequently INC_DET returns a string with at least length k. Any returned string of INC_DET has a path on $\mathcal{M}_{grep}$ thus is a string in $\mathcal{L}(E)$. 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 $\mathcal{M}_{grep}$. When DECIDE returns **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 $|wit| \geq k$, i.e., INC_DET finds a simple string longer than k and DECIDE returns **T**. If there is no simple string of length greater than k, when the finite set of states of $\mathcal{M}_{grep}$ is fully constructed, the algorithm terminates and returns a longest string found, which is lower than k, then DECIDE returns **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. $\mathcal{M}_{grep}$. □

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 8: Real-world ReDoS vulnerable regexes and corresponding projects.

No.	Regex	Project	Stars	GadgetCA	EVILSTRGEN
1	\x1bPtmux;\x1b\x1b.*?[^\x1b]\x1b\\\|\x1b(_G|P[0-9;]*q).*?\x1b\\\r?|\x1b]1337;.*?\a	junegunn/fzf	56,859	✗	✔
2	(?s)<(?:style|script)[^<>]*>.*?</(?:style|script)>|</?[a-z][a-z0-9]*[^<>]*>|<!--.*?-->	unknwon/com and other 56 projects	13,265 in total	✗	✔
3	\$(?:\{([^}]+)\}|([a-zA-Z\d_]+))	sxyazi/yazi	5,521	✗	✔
4	(?:[^“”])*[^,]”|“(?:[^“”])*[^,]”	jdkato/prose and other 4 projects	3,136 in total	✗	✔
5	<[^>]*>|&#?\w+;|[gl]t;	ajour/ajour	1,000	✗	✔
6	\$(\$|[a-zA-Z0-9_]+|\(([^)]+\)|\{[^}]+\})	getsentry/sentry-cli	855	✗	✔
7	^-i␣|␣–indexonly␣and␣-x␣|␣–indexfirst␣can't␣be␣present␣at␣the ␣same␣time.␣Try␣to␣remove␣the␣-x␣|␣–indexfirst␣flag.*	feliixx/mgodatagen	307	✗	✔
8	(\([^)]*\)|♪[^♪]*♪|[A-Z]{2,}␣?:)	emk/subtitles-rs	285	✗	✔
9	^([a-zA-Z0-9-_./]+)␣m\\\|([^|]+)\\\|([is]{0,2})(?:␣(.*))?$	qiwentaidi/Slack	282	✗	✔
10	(?i)X-Amz-Signature|File|Policy|X-Ignore-.+	sodafoundation/strato	231	✗	✔
11	\r\n|\r|\n+|\<[\S\s]+?\>	Away0x/gin_bbs	161	✗	✔
12	(\[\"(.*?)\"\])|(\[\'(.*?)\'\])	TIBCOSoftware/mashling	86	✗	✔
13	(\(.*\))|([.,␣]+(com|inc)[␣,.]*$)\|(␣)\|(^@)\|([&@].*)$	istio-ecosystem/istio-coredns-plugin	41	✗	✔
14	\$(\w+)|\[\[([\s\S]+?)(?::(\w+))?\]\]|\${(\w+)(?:\.([^:^\}]+))?(?::(\w+))?}/g	paveldanilin/grafana-csv-plugin	35	✗	✔
15	<([^>]|\n)*>|\t|\r	kevinwatt/ed2kcrawler	32	✗	✔
16	(\x1b[^m]*m|\x1b\[\d+C)	roosta/oozz	24	✗	✔
17	\x1B(?:\[[0-?]*[-␣/]*[@-~]|_[^\\]*;[^\\]*\\)	Notarin/hayabusa	20	✗	✔
18	\s*impl(<.*?>)?\s+(\w+)(<.*?>)?(\s+for\s+(\w+))?	colin353/universe	19	✗	✔
19	^\\\|(\\d+)\\\|\\W+?(.+)$	psaia/allwrite-docs	15	✗	✔
20	!\[.*?\]\((.*?)\)|<img.*?src=[\'\"](.*?)[\'\"].*?>	cnych/sinaimgmover	14	✗	✔
21	(\[\"(.*?)\"\])|(\[\'(.*?)\'\])	jvanderl/flogo-components	13	✗	✔
22	<[^>]*>|&#?\w+;|[gl]t;	mimblewimble/grin-gui	13	✗	✔
23	\$\$|\$(\w+)|\$\{(\w+)(?::-([^}]+)?)?\}	f1shl3gs/vertex	12	✗	✔
24	(?ims).+?src=\s*?"(.+?)"|'(.+?)'	crawlerclub/ce	11	✗	✔
25	<[^>]*>|\\n|\\t|␣+	emiruz/textextract	11	✗	✔
26	(?ims).+?src=\s*?"(.+?)"|'(.+?)'+	crawlerclub/ce	11	✗	✔
27	\$\d+|\$\{\d+:.+}	Universal-Variability-Language/uvl-lsp	9	✗	✔
28	(\[http[\S]+)(\{\{)|(\}\})|(\[\[)|(\]\])|(==)|:|\\||*	aryamancodes/xkcd-search	9	✗	✔
29	##.*?##|\{#.*?#\}	enginestein/aksarantara.ruby	7	✗	✔
30	**/|*|[^[?*{]+|{[^}]+}|\?	alecthomas/bit	7	✗	✔
31	/*{1,2}[\s\S]*?*/|//[\s\S]*?\n	sjqzhang/gdi	5	✗	✔
32	*|\[([^\]]*)\]	subdgtl/WFC	5	✗	✔
33	(,|#[^\n]*\n)	mrshoe/sol	5	✗	✔
34	##.*?##|\{#.*?#\}	ubcsanskrit/sanscript.rb	5	✗	✔

[1] ✔ and ✗ denote whether the corresponding method can successfully detect the vulnerability or not.