Counting in Regexes Considered Harmful: Exposing ReDoS Vulnerability of Nonbacktracking Matchers

Lenka Turoňová[1], Lukáš Holík[1], Ivan Homoliak[1], Ondřej Lengál[1], Margus Veanes[2], Tomáš Vojnar[1]
[1] *Faculty of Information Technology, Brno University of Technology, Brno, Czech Republic*
{ituronova,holik,ihomoliak,lengal,vojnar}@fit.vutbr.cz
[2] *Microsoft Research, Microsoft, Redmond, USA*
margus@microsoft.com

Abstract

In this paper, we study the performance characteristics of nonbacktracking regex matchers and their vulnerability against ReDoS (*regular expression denial of service*) attacks. We focus on their known Achilles heel, which are extended regexes that use bounded quantifiers (e.g., '`(ab){100}`'). We propose a method for generating input texts that can cause ReDoS attacks on these matchers. The method exploits the bounded repetition and uses it to force expensive simulations of the deterministic automaton for the regex. We perform an extensive experimental evaluation of our and other state-of-the-art ReDoS generators on a large set of practical regexes with a comprehensive set of backtracking and nonbacktracking matchers, as well as experiments where we demonstrate ReDoS attacks on state-of-the-art real-world security applications containing SNORT with `Hyperscan` and the HW-accelerated regex matching engine on the NVIDIA BlueField-2 card. Our experiments show that bounded repetition is indeed a notable weakness of nonbacktracking matchers, with our generator being the only one capable of significantly increasing their running time.

1 Introduction

Matching *regexes* (regular expressions) is a ubiquitous task of various software, used, e.g., for searching, data validation, detection of information leakage, parsing, replacing, data scraping, or syntax highlighting. It is commonly used and natively supported in most programming languages [7]. For instance, about 30–40 % of Java, JavaScript, and Python software uses regex matching (as reported in multiple studies; see, e.g., [10]).

Regex matching is a computationally intensive process often applied on large texts. Predictability of its efficiency has a significant impact on the overall usability of software applications. However, no matching algorithm is perfect, and an unlucky combination of a regex and text may increase the matching time by a few orders of magnitude. Unfortunately, satisfactory analytical means for distinguishing vulnerable regexes do not exist. Since very specific and rare texts may be needed to trigger an extreme behaviour, vulnerable regexes are easily missed even by thorough testing (moreover, regexes are seldom thoroughly tested, as concluded in [48, 49]). A manifestation of such vulnerability might then have serious consequences, such as a failed input validation against SQL injection or cross-site scripting attacks (cf. [52]).

Vulnerable regexes are also a doorway for denial of service attacks based on overwhelming a matching engine by crafting a vulnerability-triggering text, the so-called *ReDoS (regular expression denial of service) attacks*. For instance, in 2016, ReDoS caused an outage of StackOverflow [15] or rendered vulnerable websites that used the popular Express.js framework [4]. The fact that ReDoS is indeed a common and serious threat is argued by several works such as [10, 11]. Therefore, stress testing of regex matchers, the topic of this work, is an active research area.

Several methods and tools have been developed that attempt to determine whether a given regex is vulnerable to a ReDoS and to generate a triggering text (also referred to as *evil text* hereafter). Existing ReDoS analyzers [35, 39, 50, 53] focus on the most common family of matchers: those based on the *backtracking algorithm*.[1] These include, e.g., the regex matching engines of wide-spread programming languages .NET, Python, Perl, PHP, Java, JavaScript, and Ruby. The basic backtracking algorithm is simple and easily extensible with advanced features, however, it is at worst exponential in the text length. Regexes prone to extreme running times are easily constructed and found in practice [11]. ReDoS analyzers can often find triggering texts for regexes used in practice, and even some analytical methods for identifying regexes vulnerable to backtracking were proposed (cf. Section 3).

In contrast to the above mentioned works on vulnera-

[1] Essentially, a backtracking matcher descends through the syntactic structure of the regex, finds a mapping of the letters from the text to the atomic regex sub-expressions. Seen through the lens of a non-deterministic automaton compiled from the regex, backtracking is a depth-first exploration of the tree of all runs along the input line.

bility of backtracking-based matching, we present the first systematic study of the vulnerability of nonbacktracking *automata-based* matchers. Automata-based matchers evolved from *Thompson's algorithm* [43] (also referred to as NFA-simulation, where NFA stands for *nondeterministic finite automaton*). In essence, the algorithm is a *breadth-first* exploration of the runs of the NFA for the given regex along the input text. In combination with caching, it becomes an on-the-fly subset construction of the DFA (*deterministic finite automaton*), also called *online DFA-simulation*. Forms of online DFA-simulation are implemented in Google's `RE2` library [17], the standard GNU `grep` program [19], the `Rust` standard regex matcher [14], or Symbolic Regex Matcher (`SRM`) [38].[2] Intel's `Hyperscan` [8] uses a variation of NFA-simulation algorithm as one of its components, among a number of other techniques.

The automata-based approaches are harder to implement, and thus less flexible. On the other hand, there are years of empirical evidence showing much more stable performance of these approaches, implemented, e.g., in Google's RE2 engine [17]. Their worst-case complexity is linear in the length of the input text. Therefore, automata-based matchers are overwhelmingly preferred when avoiding regex vulnerabilities is a priority, and they are now prevailing in performance-critical industrial applications such as network intrusion detection systems (NIDSes) [25, 30] and credential scanning [27].

We present the first systematic large-scale study of vulnerability of automata-based matching, focused especially on online DFA-simulation. We focus on what seems to be the main weakness of the online DFA-simulation approach: *bounded repetition (or bounded quantifier/counting operator)*, which is a commonly used feature of extended regexes. The bounded repetition operator allows to concisely express that some pattern is repeated a specified number of times, e.g., in the regex '`(ab){100}`', the bounded quantifier '`{100}`' specifies 100 repetitions of the string '`ab`'. It has been recognized that regexes that use bounded quantifiers can suffer from performance problems both in backtracking (cf. [32]) and nonbacktracking matchers (cf. [21]). To the best of our knowledge, until now, this problem has, however, never been studied systematically, and concrete possibilities of exploiting it for ReDoS have not been analyzed.

Our approach. We present an algorithm for generating evil texts that target automata-based matchers. We target mainly matchers based on online DFA-simulation, but our techniques can also be effective with other kinds of automata-based matchers, such as `Hyperscan` (cf. Section 6.6). Our experiments confirm that our generator is the first one effective in finding evil texts for automata-based matchers.

As an example, consider the regex '`%[^\x0d\x0a]{1000}`' (from the database of regexes of the intrusion detection system SNORT [25]), which tells the matcher that after seeing '`%`', it can accept after exactly 1000 characters other than carriage return '`\x0d`' and line feed '`\x0a`'. The NFA of the regex is heavily non-deterministic and has more than 1,000 states. The minimal DFA has more than 2^{1000} states (it needs to always "remember" all positions of the character '`%`' within the last seen 1,000 characters other than '`\x0d`' and '`\x0a`'). The DFA states produced by the determinisation during matching may also be large, namely, they are sets of up to 1000 NFA states. A text on which the DFA would reach many different large DFA states is highly problematic for most matchers, backtracking as well as online DFA-based. Such a text is, however, also highly specific and the probability of generating it randomly is low (the text must contain sub-strings of 1,000 characters other then '`\x0d`' and '`\x0a`' with varying and frequent placements of '`%`'). Our evil text generator is the only automated tool we know of that can discover such text.

Our generator is based on heuristics that generate expensive runs of the DFA of the regex. Besides a general algorithm applicable to any regex, it features a heuristic specialising on bounded repetition, based on an analysis of the so-called counting-set automata [46]. Especially with extended regexes such as the regex '`%[^\x0d\x0a]{1000}`' from above, it is capable of forcing creation of many large DFA states—the number of these states may be exponential and their size may be linear in the repetition bound (i.e., 1,000 in our example), dramatically increasing the matching time. [3]

We evaluate our generator on a comprehensive database of regexes (from software projects at GitHub [12], network intrusion detection systems [2, 25, 37], detection of security breaches [20, 45], academic papers [47, 54], posts on Stack Overflow [31], and the RegExLib database [36]) against a set of major industrial regex matchers (`RE2`, `grep`, `Hyperscan` [8, 17, 19], as well as standard library matchers of `.NET`, `Python`, `Perl`, `PHP`, `Java`, `JavaScript`, `Rust`, and `Ruby`) and compare its performance against existing ReDoS generators (`RXXR2` [35], `RegexStatic` [50], `RegexCheck` [53], and `Rescue` [39]). The results of the evaluation substantiate the following conclusions, which are also the main **contributions of the paper:**

1. Bounded repetition is an Achilles heel of automata-based matchers and our novel generator is the only one that can effectively generate ReDoS texts for them.
2. On the other hand, without bounded repetition, Re-

[2] `SRM` is based on symbolic Antimirov derivatives [3] constructed on the fly, also in the spirit of online DFA construction.

[3] Bounded repetition may be expressed without the counting, by simply repeating the pattern the needed number of times, leading to the same DFA. This is, however, impractical and almost never used. The pitfalls of counting show even in the worst case complexity of the DFA and matching algorithms. In contrast to basic regexes, where the DFA is exponential and the matching time is linear to the size of the regex (when matching by automata algorithms such as online DFA simulation), bounded repetition leads to a doubly exponential DFA and singly exponential matching time. This is because the DFA for a bounded repetition is exponential in the repetition bounds (or their multiple in the case of nested bounded repetitions, as in '`((a{10}){10}){10}`'), which is again exponential in the size of their decadic numerals.

DoS generators have none or negligible success with automata-based matchers.

3. Our new ReDoS generator can indeed generate attacks on practical applications where the performance of regex matching is critical, namely on SNORT 3 with enabled `Hyperscan` [25] as well as hardware accelerated regex matching on the NVIDIA BlueField-2 DPU [29]. For both technologies, we achieved a slowdown of regex matching engines by a few orders of magnitude, tested on regexes from real-world SNORT rulesets.

Organization. After preliminaries and related work in Sections 2 and 3, we present our main technical contribution, the ReDoS generator targeting automata-based matchers, in Sections 4 and 5. Section 4.1 analyses a model of an online DFA-simulation based matcher. The analysis gives grounds to develop our novel ReDoS generator in Section 4.2, based on analysing the regex's DFA. Section 5 then presents its specialisation to bounded repetition. Section 6 details the experiments, giving evidence of vulnerability of automata-based matching against bounded repetition, including concrete practical implications, and Section 7 suggests possibilities of mitigating the implied security risks.

2 Preliminaries

We will recall needed formal concepts: words, languages, regular expressions and automata as well as the essentials of pattern matching, matching algorithms, ReDoS and the considered attacker model.

Words, languages, regular expressions. We consider a fixed finite *alphabet* of *characters/symbols* Σ (presumably a large one such as Unicode). *Words* are sequences of characters from Σ, with the empty sequence denoted by ε. *Languages* are sets of words. The operators of *concatenation* $\cdot$ and *iteration* * applied on words or languages have the usual meaning. We consider the usual basic syntax of *regular expressions* (a.k.a., *regexes*) generated by the grammar

$$R ::= \alpha \mid (R) \mid RR \mid R|R \mid R* \mid R\{\mathtt{n,m}\}$$

where $n,m \in \mathbb{N}$, $0 \leq n$, $0 < m$, $n \leq m$, and α is a *character class*, i.e, a set of characters from Σ. A character class is most often of the form `a`, `.`, $[\mathtt{a_1\text{-}b_1a_2\text{-}b_2\ldots a_n\text{-}b_n}]$, or $[\mathtt{\hat{}a_1\text{-}b_1a_2\text{-}b_2\ldots a_n\text{-}b_n}]$, denoting a singleton containing the character $a \in \Sigma$, the entire set Σ, a union of n intervals of characters, or the complement of the same, respectively.

The *language* of a regex R, denoted $L(R)$, is constructed inductively to the structure of R, from its atomic sub-expressions, character classes, using the language operations denoted by the regex combinators. They are understood as usual: two regexes in a sequence stand for the *concatenation* of their languages, '`|`' is the *choice* or *union*, '`*`' is the *iteration*, and '`{n,m}`', is the *bounded iteration*, equivalent to the union of i-fold concatenations of its operand for $n \leq i \leq m$.

Finite automata. We consider *nondeterministic finite automata (NFAs)* over Σ of the form $A = (Q, \delta, q_0, F)$ where Q is a finite set of *states*, δ is a set of *transitions* of the form $q \xrightarrow{a} r$ with $q, r \in Q$ and $a \in \Sigma$, $q_0 \in Q$ is the *initial state*, and $F \subseteq Q$ is the set of *final states*. The *language* of the automaton, denoted $L(A)$, is the set of all words $a_1 \ldots a_n$, $n \geq 0$, for which the automaton has an *accepting run*, a sequence of transitions $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} q_n$ with $q_n \in F$.

The automaton is *deterministic (DFA)* if for every state q and symbol a, δ has at most one transition $q \xrightarrow{a} r$. Any NFA can be determinised by the *subset construction*, which creates the DFA $A' = (Q', \delta', q_0', F')$ with $Q' = 2^Q$, i.e., with subsets of A as the new states, the singleton $\{q_0\}$ as the initial state q_0', with sets intersecting with F being final, i.e., $F' = \{S \subseteq Q \mid S \cap F \neq \emptyset\}$, and with the successor of a state $S \subseteq Q$ under a symbol a constructed as the set of a-successors of the NFA states in S, $S \xrightarrow{a} S' \in \delta'$ for $S' = \{s' \mid s \in S \wedge s \xrightarrow{a} s' \in \delta\}$.

Pattern matching. In its simplest form, *pattern matching* is the problem of deciding whether a given word (*line*) w has an infix conforming to a given regex R. In other words, it decides whether w can be written as a concatenation $x.v.y$ such that $v \in L(R)$, i.e., $w \in L(\mathtt{.*}R\mathtt{.*})$. *Anchors*, '`^`' at the start of the regex and '`$`' at the end, can be used to force the match v start at the beginning of the line (the prefix x is empty) or end at the end of the line (the suffix y is empty), respectively.

Besides the simplest problem of deciding whether a match appears on a single input line, which is the *single-line mode* of matching, we will also consider matching in the *multi-line mode*, in which the matcher is supposed to filter all lines of the input text that match the regex.

Approaches to pattern matching. We distinguish two families of pattern matching algorithms used in practice: backtracking and nonbacktracking automata-based algorithms.

(1) Backtracking [40] algorithms in their simplest form use a recursive procedure that descends the syntactic tree of the regex while reading the text from the left to the right and matching its characters against sub-expressions of the regex. Since disjunction and iteration offer a choice, the recursion backtracks to the last unexplored choice when the matching fails. It is in fact very similar to a depth-first exploration of all runs following the input line through an NFA corresponding to the regex. Since such matchers are conceptually very simple (a basic implementation takes a few lines of functional code (e.g. [34], page 7) and since they are processing a single path through the NFA at a time, backtracking algorithms are flexible and amenable to easy extensions with features such as priority of matched sub-expressions, submatching, or back-references. Nonetheless, as the number of NFA runs over a single line is in the worst case exponential in its length, the worst-case complexity of matching using a backtracking matcher is *exponential in the length of the text*. Extreme matching times do not occur often if regexes are written defensively, and modern implementations are fast, especially when an accepting path is guessed early. However, overlooking a dangerous regex is easy and writing such a regex intentionally is even easier. For instance, when run on the regex '`(a|b|ab)*bc`' against the input string $(ab)^n ac$ with $n = 50$, standard matchers in Java, Python, and .NET become unresponsive [34]. Examples of industrial backtracking matchers include regex matchers in the standard libraries of .NET, Python, Perl, PHP, Java, JavaScript, and Ruby.

(2) A basic and naive **automata-based** matching alternative to backtracking is the *(offline) DFA-simulation*, which is based on constructing a DFA for the regex. Having the DFA at hand is the best scenario for matching since every character is then processed in constant time by simply following the unique transition from the current DFA state to the successor. The problem is that determinisation may explode exponentially, rendering matching slow or unfeasible (the matcher may time out already during the DFA construction). This approach is therefore seldom used in practice.

A more practical alternative to DFA-simulation is based on *Thompson's algorithm* [43] aka *NFA-simulation*. NFA-simulation essentially differs from the backtracking algorithm by replacing the depth-first NFA exploration strategy by a breadth-first search strategy. Reading each symbol of the text means updating the set of all NFA states reached by runs over the so far processed prefix of the line. The time needed to process each symbol is thus linear to the size of the NFA (an iteration through all transitions over the symbol starting in the current set of states), and the entire matching is only linear in the length of the line. An advanced implementation of NFA-simulation is a part of Intel's `Hyperscan` [8] (among a number of other techniques such as advanced use of the Boyer-Moore algorithm [5] for string-matching, innovative parallelisation, or using specialised processor instructions).

A crucial ingredient for the performance of several practical matchers is caching. The reached sets of NFA states are actually states of the DFA constructed by the subset construction, while a DFA state and its successor reached after reading a symbol constitute a DFA transition. The encountered DFA states and transitions are cached. When the matching algorithm stays inside the cache of transitions, it is exactly the same as the offline DFA simulation, with constant per-character complexity. We will call the version with caching *online DFA-simulation* (following the terminology of [14]). Online DFA-simulation can achieve much better performance and especially stability and resilience against ReDoS than backtracking. The disadvantage is perhaps a less straightforward implementation, which implies lower flexibility. Also, it is not clear how to extend online DFA-simulation with advanced regex features such as back-references. Well-known examples of industrial matchers based on DFA-simulation include `RE2` [17], `grep` [19], `SRM` [38], or the regex matcher in `Rust` [14].

ReDoS and associated attacker model. This paper deals with vulnerability of regex matchers against ReDoS. Specifically, we assume a remote service utilizing a regex matcher with a set of deployed regexes that are required for the operation of the service. We assume that some of the deployed regexes contain bounded repetition. The attacker knows which regexes are deployed at the service, or has a way of informed guessing (e.g., Snort regexes are public or easily obtainable via subscription, open source web development frameworks have known regex input validators, etc.).

The attacker can access the service in a way that enables triggering remote execution of the regex matcher (with deployed regexes) on an arbitrary (i.e., provided by the attacker) input text. The goal of the attacker is to pass into the service an (evil) text that will render the service unavailable (causing a denial of service) or impose a significant performance drop due to the consumption of an exceptionally high amount of computational resources. In such cases, we say that the regex is vulnerable for the respective matcher, and we consider three different views on *vulnerability*. Given a fixed length of text, it can mean one of the following (a detailed description is given in Section 6.1):

(a) exceeding a certain time interval for processing of a text of the given length,
(b) exceeding a certain ratio of the measured time w.r.t. 'normal' time for the given matcher, or
(c) exceeding a certain ratio of the measured time w.r.t. 'normal' time for the given matcher relative to the particular regex, assuming some knowledge of a normal matching time for each regex.

3 Related Work on ReDoS

ReDoS [32] vulnerabilities have typically been attributed to backtracking-based matching, as discussed in depth in [10,11]. Backtracking regex matching engines are essentially Turing complete (cf. [24]) and therefore most analysis questions about them are difficult or undecidable. All prior research on ReDoS generators has focused on methods that attempt to generate inputs that essentially cause excessive backtracking at runtime, effectively causing non-termination of matching. Here we summarize main such approaches.

We focus mainly on *static* ReDoS generators, which analyse a regex statically, as opposed to *dynamic* generators, which analyse a profile of a regex matcher run. Static ReDoS generators are primarily based on the NFA representation of regexes [22] and exploit different techniques, such as *pumping analysis* [22, 34], *transducer analysis* [42], *adversarial automata* construction [53], and *NFA ambiguity analysis* [51]. Such techniques can be sound and even complete for certain classes of regexes. Their main disadvantages are a high rate of false positives and ineffectiveness against nonbacktracking regex matching engines. An overview of existing ReDoS generators follows:

`RegexStatic` [51] classifies the worst-case simulation cost for a regex on an input as linear, polynomial, or exponential based on how the depth-first search tree is predicted to evolve during backtracking. It supports also nonregular features like back-references.

RegexCheck [53] also identifies if a regex has linear, super-linear, or exponential time complexity based on its NFA. Moreover, it can construct an attack automaton capturing all those strings that trigger the worst-case behaviour. It also combines static and dynamic analysis to avoid false positives. It has limited support for extended (nonregular) features.

RXXR2 [34, 35] constructs an NFA from a given regex and then it searches for instances of a pattern in the NFA using an efficient pattern matching algorithm. It searches all sub-expressions for exponential vulnerability in a form of $e_1 e_2^* e_3$ where e_1 is a prefix expression, e_3 is a suffix expression, and e_2^* is a vulnerable expression. The result is an attack string $xy^n z$ such that $x \in L(e_1)$, $y \in L(e_3)$ and $xy^n z \notin L(e_1 e_2^* e_3)$.

SlowFuzz [33] is a dynamic fuzzing tool. It is based on an evolutionary fuzzer [23] that searches for those inputs that can trigger a large number of edges in the control flow graph of the program under testing. However, it lacks knowledge of regex structures, which may lead to false negatives. The results in [33] compare matching slowdown among individual iterations of the algorithm. Out of the tools mentioned here, it is the most general tool for generating evil texts, since it can handle most of the extended features supported in regexes.

Rescue [39] combines dynamic and static techniques using a genetic search algorithm as a guide. The aim is to find an input string that maximizes the number of matching steps, using regex search profiling data. The maximum string length is set to 128. A string is classified as exposing a ReDoS vulnerability if it causes more than 10^8 matching steps.

Finally, let us note that existing generators sometimes aim at extremely severe vulnerabilities, for instance, where a backtracking-based matcher gets completely stuck on a text hundreds of characters long (e.g. [39]). Automata-based matchers do not exhibit vulnerabilities this severe, but they can still be slowed down by several orders of magnitude, for which they need a long-enough input text (in the order of megabytes). These are the vulnerabilities that we target.

4 ReDoS Generation

We now discuss our ReDoS generator, i.e., a tool that generates an evil text for a given regex. We target primarily non-backtracking automata-based matchers, mainly those based on online DFA-simulation (although, as we show in Section 6, our technique works for backtracking matchers as well, and it can be tweaked to cause significant troubles also to `Hyperscan`, which uses NFA-simulation).

The generator, combined with a technique that exploits counting presented subsequently in Section 5, is the main technical contribution of our paper.

4.1 Hypothetical Matcher

We first discuss a hypothetical matcher, which will serve as a model target for our ReDoS generator described later in Section 4.2. The model was created by studying the implementations of the regex matchers in `grep`, `Rust`, `SRM`, and `RE2`. It uses online DFA-simulation with a specific management of the DFA cache, similarly to the mentioned matchers. Our model does not take into account specific advanced optimizations and implementation techniques used in real performance-oriented matchers. Taking them into account might, of course, improve the performance of the generator for a specific matcher, but our goal is a ReDoS generator that is universal and simple; therefore we use a model that captures only the most important common aspects. Despite that, the real-world matchers are quite close to this hypothetical matcher (only `Hyperscan` is related more loosely, since it uses the most radical innovations, combined with NFA-simulation instead of online DFA-simulation).

The matching algorithm and its complexity. The hypothetical matcher implements the online DFA-simulation algorithm with the following management of the cache: (i) When the cache exceeds some size, it is reset and (ii) if the cache utilization is too low or is reset too often, the matcher disables the cache completely and reverts to pure NFA-simulation.

Algorithm 1 describes the hypothetical matcher in pseudocode. It simulates a run of the DFA obtained by subset construction from the input NFA $A = (Q, \delta, q_0, F)$ along the input word w. In order to do this without constructing the entire DFA up-front, it uses the class DFA, which constructs DFA transitions and encountered DFA states lazily, on demand, and saves them for further use. Namely, it stores integer IDs of the encountered DFA states (subsets of Q) in a hash table *state2id*, paired with the inverse mapping *id2state* of the DFA states back to their IDs. A discovered DFA state is identified with the number of the so far identified states plus one (Line 17). The ID of the target state of each used DFA transition is saved in the map *successor*, accessible under the ID of the source state and the symbol on the transition. The map *final* records whether an ID belongs to a final state.

The i-th character $w[i]$ of the input line is processed in a single iteration of the **for** loop on Line 3. The cost of the iteration depends on whether the DFA transition is in the cache or not. If yes, then *successor*$[q, w[i]]$ on Line 22 simply returns the ID q' of the successor of the current state ID q. The lookup has a small constant cost (accessing the index $w[i]$ of an array of successors associated with q).

On the other hand, if the DFA transition is not cached, then it must be constructed, which is expensive: The construction requires to iterate through all $w[i]$-transitions originating from the NFA states in the current DFA state S (Line 25). The cost of this iteration depends on the size of S and the number of the used NFA transitions, both of which can be bounded by $|A|$ (the size of A, $|A| = |Q| + |\delta|$). Furthermore, the book-keeping costs of the cache of DFA states, paid after every cache miss on Line 22, is also significant (although dominated by the cost of constructing the transition on Line 25). Looking up a DFA state on Line 14 and adding a DFA state on Line 26 both take time proportional to the size of the DFA state.

Algorithm 1: Hypothetical matcher

Input : NFA $A = (Q, \delta, s_0, F)$, word w
Output : *true* iff $w \in L(A)$, otherwise *false*

1 $dfa \leftarrow$ **new** `DFA()`
2 $q \leftarrow dfa$.`init`($\{s_0\}$)
3 **for** $i \leftarrow 1$ **to** $|w|$ **do** // $O(|w| \cdot |A|)$
4   **if** $dfa.final[q]$ **then return** *true*
5     $q' \leftarrow dfa$.`get_successor_id`($q, w[i]$) // $O(|A|)$
6     $q \leftarrow q'$
7   **if** dfa.`big()` **then** $q \leftarrow dfa$.`init`($dfa.id2state[q]$)
8   **if** dfa.`ineffective()` **then** disable DFA caching
9 **return** *false*
10 **class** `DFA`**:**
11     $state2id : 2^Q \to \mathbb{N}$; $id2state : \mathbb{N} \to 2^Q$;
12     $successor : \mathbb{N} \times \Sigma \to \mathbb{N}$; $final : \mathbb{N} \to \{true, false\}$
13   **method** `get_state_id`($S \subseteq Q$)**:**
14         $q \leftarrow state2id[S]$ // $O(|S|)$
15   **if** $q =$ `None` **then**
16             $q \leftarrow state2id.cardinality + 1$
17             $state2id[S] \leftarrow q$ // $O(|S|)$
18             $id2state[q] \leftarrow S$
19             $final[q] \leftarrow (S \cap F \neq \emptyset)$
20   **return** q
21   **method** `get_successor_id`($q \in \mathbb{N}, a \in \Sigma$)**:**
22         $q' \leftarrow successor[q, a]$ // $O(1)$
23   **if** $q' =$ `None` **then**
24             $S \leftarrow id2state[q]$
25             $S' \leftarrow \{s' \mid s \in S, s \xrightarrow{a} s' \in \delta\}$ // $O(|A|)$
26             $q' \leftarrow$ `get_state_id`(S') // $O(|S'|)$
27             $successor[q, a] \leftarrow q'$
28   **return** q'
29   **method** `init`($S \subseteq Q$)**:**
30         $id2state \leftarrow state2id \leftarrow successor \leftarrow final \leftarrow \emptyset$
31   **return** `get_state_id`(S)

The complexity of matching with a high utilization of the cache is therefore approaching $O(|w|)$, but in the worst case, with a low cache utilisation, it increases to $O(|w| \cdot |A|)$. The multiplicative factor $|A|$ may be especially high with extended regexes with the bounded repetition operator, where the size of $|A|$ is linearly dependent on the repetition bounds (this is exponential in the size of the regex, assuming that the bound is given as a decadic or similar numeral). For instance, the NFA for the regex '`.*a.{k}`' needs $k+1$ states and the DFA obtained by the subset construction has 2^{k+1} states, each of them a set of up to $k+1$ states of the NFA.[4]

The algorithm manages limited resources available for the cache on Lines 7 and 8. The cache is reset on Line 7 if it grows beyond some predefined bound (given by the method *dfa*.`big()`, whose implementation would be matcher-specific). The size of the cache is computed as the sum of sizes of cached DFA states plus the number of cached transitions, $\sum\{|S| : DFA.state2id[S] \neq \texttt{None}\} + |\{(id, a) : DFA.successor[id, a] \neq \texttt{None}\}|$ (note that larger DFA states hence contribute more to the size of the cache). Line 8 may then entirely disable caching if the cache is reset too often or if its utilisation is too low (given by a matcher specific implementation of *dfa*.`ineffective()`). Disabling the cache means reverting to NFA-simulation in which every step must iterate through all NFA states in the current set and all their transitions with the current letter.

Multi-line mode. The matcher described above works in the *single-line mode*. In the *multi-line mode*, the **for** loop on Line 3 is wrapped in an iteration over all lines and every matched line is reported. Importantly, the DFA cache is *not reset* after processing one line, but is re-used when processing subsequent lines.

4.2 ReDoS Generation Algorithm

As follows from the analysis above, our best shot to stress the hypothetical matcher is to attempt to increase its runtime close to $O(|w| \cdot |A|)$ by rendering the cache ineffective and forcing construction of many large DFA states and transitions whose computation is expensive. For that, recall that every newly discovered DFA state $S \subseteq Q$ is searched for and inserted into the cache, with a cost linear to its size, and subsequently causes a cache miss and forces the construction of a transition on Line 25, with a cost linear to the number of $w[i]$-transitions starting in S. The size of S also determines the cost of looking up and inserting DFA states to the cache on Lines 14 and 26. The cost of creating the DFA transition, that is, at most the number of the NFA transitions, is usually strongly correlated with the size of the source state S (even though it is not precisely determined by it since it depends on the transition relation).

Our aim is, therefore, to produce a text that discovers many different large DFA states as fast as possible. In other words, we want to force a DFA run (or a sequence of runs in the case of multi-line matching) with a high ratio of the sum of sizes of newly discovered DFA states and the text length. We will call this ratio the *evilness* of the text. Highly evil texts cause a low cache hit/miss ratio, the cache also fills up quickly, must be reset frequently, and there is a high chance that the utilisation of the cache drops to the point where it is completely disabled.

ReDoS generator overview. Our ReDoS generator constructs a text w with high evilness as a concatenation $w_1 \cdots w_n$

[4]The '`.*`' in the regex is included for clarity, but note that it is redundant in the absence of anchors.

of lines, each line w_i generated by a run ρ_i starting at the initial state of the DFA. Each run ρ_i first takes the shortest possible path through the already visited part of the DFA to a largest discovered but so far unvisited state, referred to as the *starting state of* ρ_i, from where it navigates to new unvisited DFA states through DFA transitions chosen according to some *successor selection criterion*.

The run ρ_i is thus a concatenation $\rho_i^1.\rho_i^2$ of a prefix ρ_i^1 through already visited DFA states and a suffix ρ_i^2 through unvisited states. The criterion for navigating the second phase, that is, for selecting unvisited successors while constructing the suffix, is a parameter of the algorithm. The basic strategy, called **GREEDY**, simply selects the largest unvisited successor. (alternatives will be discussed later). This drives the exploration towards large new states. The run ρ_i then ends when it cannot continue to any unvisited and non-final state.

Avoiding final states has the following rationale. Obviously, continuing a line after reaching a final state would be counterproductive because the matcher has already returned *true*. Avoiding final states altogether additionally means that we generate only non-matching lines, which is motivated by the fact that we ideally want texts that are hard for online DFA-simulation-based as well as backtracking matchers. Non-matching lines are generally harder for backtracking matchers. They cannot terminate early after finding a single accepting NFA run but are forced to explore the entire tree of runs over the input line.

ReDoS generator in detail. We present the algorithm for generating ReDoS attacks in detail as Algorithm 2. Since constructing the entire DFA may be infeasible due to its size, the algorithm again uses the implicit DFA that is a part of the hypothetical online DFA matcher in Algorithm 1 and thus constructs only those parts of the DFA used to process the generated text.

Every iteration of the while-loop on Line 7 generates one line of the text, namely, the i-th iteration generates w_i by constructing the run ρ_i. The algorithm maintains a set *visited* of IDs of DFA states that were visited by some run ρ_i, and a set *unvisited* of IDs of discovered but yet unvisited states. The while loop terminates when there are no states remaining in *unvisited*. To select the starting state q of ρ_i (Line 8) and construct the shortest run to q quickly (via function *prefix* on Line 10), the algorithm uses a mechanism analogous to the one used in Dijkstra's algorithm for computing the shortest paths from a given source: Every discovered DFA state $p \in visited \cup unvisited$ remembers the last transition in the shortest discovered run from the initial state to p, namely, the predecessor state $pre(p)$ on the run and the symbol $\sigma(p)$ on its last transition. The state p also remembers the length (distance) $d(p)$ of the shortest run. The values of $pre(p)$, $d(p)$, and $\sigma(p)$ are updated whenever a transition to the state p is taken (Lines 18 and 19). If the run ending by that transition is shorter than the current shortest run, the function $prefix(q)$ can then construct the shortest discovered run to q in the form $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} q_n$ by taking $q_n = q$, $q_i = pre(q_{i+1})$, and $a_i = \sigma(q_i)$ for all $0 \leq i < n$, and return the word $a_1 \dots a_n$ read along this run. The starting state q of ρ_i is chosen on Line 8 from *unvisited* by selecting the DFA state (obtained as $dfa.id2state[q]$) of the largest size with the smallest distance $d(q)$.

The suffix of the run, ρ_i', is where the text supposed to increase the cost of matching is generated. The algorithm navigates through unexplored DFA states according to the strategy given by the input parameter **STRATEGY** as long as the current state q has some unexplored non-final successor p (Line 20). Namely, the for-loop on Line 13 collects into *succ* all transitions leading to non-final and not yet visited DFA states from the current state q (as pairs consisting of the target state p and symbol a). The particular transition is selected from there according to the criterion **STRATEGY** on Line 21.

Algorithm 2: DFA-based text generation

Input: An NFA $A = (Q, \delta, s_0, F)$,
successor selection criterion **STRATEGY**
Output: evil text w (concatenation of several lines)

1 $dfa \leftarrow$ **new** `DFA`
2 $q_0 \leftarrow dfa.$`init`$(\{s_0\})$
3 $unvisited.enqueue(q_0)$
4 $d(q_0) \leftarrow 0$
5 $visited \leftarrow \emptyset$
6 $w \leftarrow \varepsilon$
7 **while** $unvisited \neq \emptyset$ **do**
8   $q \leftarrow unvisited.dequeue\_nearest\_largest()$
9   $visited.add(q)$
10   $w \leftarrow w \cdot prefix(q)$
11   **while** *true* **do**
12     $succ \leftarrow \emptyset$
13   **for** $a \in \Sigma$ **do**
14       $p \leftarrow dfa.$`get_successor_id`$[q, a]$
15   **if** $dfa.final[p] \vee p \in visited$ **then continue**
16       $succ.add(p, a)$
17       $unvisited.enqueue(p)$
18   **if** $d(q) + 1 < d(p) \vee d(p) =$ `None` **then**
19         $(d(p), \sigma(p), pre(p)) \leftarrow (d(q)+1, a, q)$
20   **if** $succ = \emptyset$ **then break**
21     $(q', a) \leftarrow succ.choose($**STRATEGY**$)$
22     $unvisited.remove(q')$
23     $visited.add(q')$
24     $q \leftarrow q'$
25     $w \leftarrow w \cdot a$
26   $w \leftarrow w \cdot$ `\n`
27 **return** w

Exploration strategies. The ReDoS generation algorithm is parameterized by the strategy of exploration of unvisited DFA states, represented by the successor selection criterion **STRATEGY**. We will consider the following three strategies.

The first strategy, **Random**, picks from *succ* a random successor. This produces mostly random but still 'reasonable' texts, for which the matcher does not return *false* before the line ends, because the DFA run never leaves the area of useful DFA states. We use **Random** as the baseline to confirm that the reasoning behind our other two selection criteria, supposed to generate highly evil texts, works.

The simpler of the two strategies, **Greedy**, navigates the search towards large DFA states by always choosing the successor corresponding to the largest set of states. On the other hand, the more complex strategy **Counting** is then optimized towards generating texts for regexes with bounded repetition; it is discussed in detail in the following section.

5 ReDoS Generation for Bounded Repetition

We will now discuss the specialisation of the ReDoS generator from the previous section for regexes with counting. That is, we will specify the successor selection criterion **Counting** used as the parameter **Strategy** in Algorithm 2.

Regexes with bounded repetition are the main focus of our work since their DFAs tend to have extremely many large states. This shows even in the worst case complexity of online DFA-simulation (as well as of NFA-simulation), where processing each input character can take a number of steps *exponential* to the size of the regex (the complexity is linear to the repetition bounds, which are represented using a logarithmic number of bits). The general idea of generating evil texts for bounded repetition is the same as for normal regexes—to force many different and large DFA states. We propose an optimized strategy for navigating towards them.

Counting automata. To explain the strategy, let us first have another look at compilation of bounded repetition to automata. Since the NFAs for bounded repetition might already be too large (linear in the repetition bounds, exponential in the size of the regex), we use succinct automata with *counters* that count repetitions of the counted sub-expressions at runtime. Since the counter values are not a part of the automata control state, they are only computed at runtime, the size of these automata is independent of the counter bounds and only linear in the size of the regex.

We use a formalisation of these automata as *nondeterministic counting automata (NCAs)* from [46], which also discusses their compilation from regexes with bounded repetition. See an example NCA for the regex '`.*a.{100}`' in Figure 1a. As seen in the figure, a transition of the NCA can reset a counter to 0, keep it unchanged, increment it, and test whether its value belongs to a specified constant interval. The values of every counter c can only reach values in between 0 and some $\max_c \in \mathbb{N}$ (the maximum number which c is compared against). A run of an NCA over a word goes through a sequence of configurations, pairs of the form (q,ν) where q is a control state and ν is a counter valuation, a mapping of counters to their integer values. For instance, one of the NCA runs from Figure 1a on the word a^{100} generates configurations $(q,c=0),(s,c=0),(s,c=1),\ldots,(s,c=99)$, but the NCA can postpone the transition into s arbitrarily, leading to different values values of c. It is easy to see that one can construct an NFA whose set of states is the set of reachable configurations of an NCA; the runs of such an NFA would go precisely through the same configurations as the runs of the NCA over the same word.

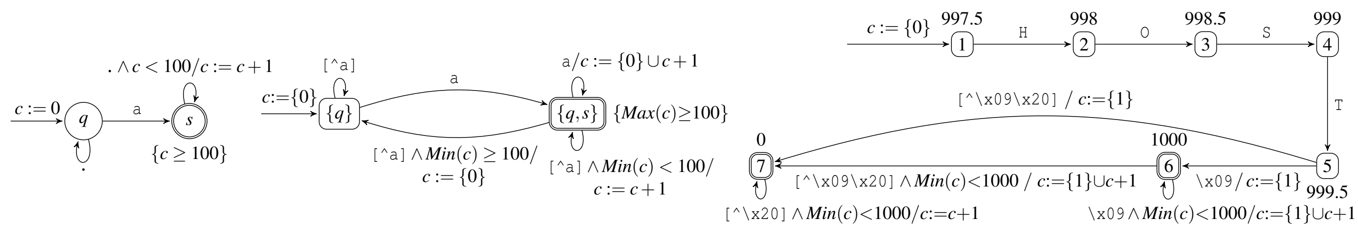
The so-called *naive determinisation* of the NCA then produces a standard DFA that would be obtained by the subset construction from the induced NFA described above. The states of the DFA are thus sets of the configurations. For the example from Figure 1a, a run of the DFA on the word a^{100} would traverse through the following sequence of DFA states (recall that each set of configurations is one state of the DFA):

$$\begin{aligned}&\{(q,c=0)\},\\&\{(q,c=0),(s,c=0)\},\\&\{(q,c=0),(s,c=0),(s,c=1)\},\\&\ldots\\&\{(q,c=0),(s,c=0),(s,c=1),\ldots,(s,c=99)\}.\end{aligned}$$

Our ReDoS generator therefore navigates through a space of such DFA states. The states may be extraordinarily large especially when the NCA configurations within them have many distinct counter values, such as in our example, where the run on the word a^{100} ends in a DFA state where the control state s is paired with 100 values.

Counting-set automata. Our heuristic for navigating through such DFAs towards large states attempts to increase the number of counter values. To do that, we take advantage of our earlier work on determinisation of NCAs into the so-called *counting-set automata (CSAs)* [46]. Namely, [46] shows how an NCA can be determinised into a CSA of a size independent of the counter bounds (unlike DFA, which may be exponentially large). The CSA is a deterministic machine that simulates the DFA but achieves succinctness by computing the counter values only at runtime, as values of a certain kind of registers. Since a single DFA state contains many counter values paired with NCA control states, these registers must be capable of holding a *set* of integer values. We call these registers, which store sets of integers, *counting sets*. A transition may then update a counting set c by incrementing all its elements, resetting it to the singleton $\{0\}$, adding the element 0 or 1 to it, and test whether the minimal or the maximal value in the set belongs to some constant interval.[5] A counting set for a counter c is also restricted to only contain values between 0 and $\max_c$ (the set-increment operation removes values greater than $\max_c$). An example of a CSA

[5]These operations can actually be implemented to work in constant time, hence simulation of CSA gives a fast matching algorithm for bounded repetition. We have implemented and tested a prototype matcher based on the CSA simulation in Section 6.



(a) NCA for '`.*a.{100}`' (b) CSA from determinization of (a) (c) CSA with weights for the regex '`^HOST\x09*[^\x20]{1000}`'

Figure 1: NCA and CSA. The transitions are labeled by their *guard*, which specifies the input character class ('$\bullet$' stands for "any character") and possibly restricts counter (or counting set) values, separated by '/' from the counter *update* (an unspecified update means that the value stays the same). In (b) and (c), the notation $\{0\} \cup c+1$ stands for the set of values obtained by *incrementing* each value in c, *adding* 0, and *removing* values larger than the upper bound of the counter, 100 for (b) and 1000 for (c). The edges denoting initial states are labelled with *initial values* of the counters. Final states are in (a) and (b) labelled with an *acceptance condition* on counters, e.g. $\{c \geq 100\}$ in (a). In (c), the final condition at states 6 and 7 is $Max(c) = 1000$.

is the automaton obtained by determinizing the NCA from Figure 1a, shown in Figure 1b. Its run on the word a^{100} would generate the following sequence of configurations:

$$\begin{array}{l}(\{q\}, c{=}\{0\}),\\ (\{q,s\}, c{=}\{0\}),\\ (\{q,s\}, c{=}\{0,1\}),\\ \dots\\ (\{q,s\}, c{=}\{0,\dots,99\}).\end{array}$$

Note that the sets of values for c precisely correspond to the values of c that s appear with in the run of the DFA shown above. The run-time configurations of a run of a CSA are (encodings of) states of the DFA that would be generated by a run reading the same word.

Navigation towards large counting sets. Since CSAs are still small (relative to the DFA), they can be pre-computed and analysed as a whole. We use such an analysis to obtain guiding criteria that lead a run through their configuration space towards configurations with many different counter values. Runs of CSAs simulate runs of DFAs, so such guiding criteria may be directly used to navigate runs of the DFAs as the successor selection criterion COUNTING.

Particularly, in the CSA for a given regex, we try to navigate towards cycles that are likely to create large counting sets. For every counter c, every cycle in the CSA is assigned a weight $weight_c$, which represents an estimate of the maximum counting set for c that iterations of the cycle can generate. The number reflects the following intuitions:

First, since the counting set c can contain only values between 0 and $\max_c$, it can have at most $\max_c+1$ elements. Second, the cycle is pumping up the set if (i) it does not reset it, (ii) it adds 0 or 1 and also increments the elements of the set (without the increment, it would be only repeatedly adding $0/1$'s to a set already containing it). Third, it is better if only a few increments happen in between additions of $0/1$'s. For instance, a cycle that increments the counting set four times per every addition of $0/1$ is actually filling it with multiples of 4, hence it can generate a set of the size at most $\frac{\max_c+1}{4}$. In summary, the weight of the cycle for the counter c is non-zero only if the cycle does not reset c and increments c at least once, and then it equals $\max_c$ multiplied by the number add_cnt_c of additions of $0/1$ to c divided by the number inc_cnt_c of increments of c, i.e. $weight_c = \frac{(\max_c+1)\cdot add\_cnt_c}{incr\_cnt_c}$. The final weight of a cycle is then computed as a sum of weights for individual counters $\sum_{c \in C} weight_c$ with C being the set of all counters used in the automaton.

The weights of cycles are assigned to states and propagated through the transitions of the CSA. Initially, all states have weight 0. We then process the cycles in the CSA one by one. For each of them, the first step is setting all weights of all states in the cycle to the maximum of their previous weight and the weight of the cycle. The weight of the cycle is then propagated backwards through paths reaching the cycle. Namely, the weight of a state r, $weight(r)$, propagates through a transition $q \text{-}\langle a \rangle\text{-}r$ so that $weight(q)$ is assigned the maximum of $weight(q)$ and $weight(r) - 0.5$. This is iterated as long as some weight can be increased. In the end, transitions with heavy target states point in the direction of short paths towards heavy cycles (the *shortness* is achieved through the subtraction of 0.5 for every transition that the weight of the cycles is propagated through).

Example 5.1. Consider the CSA for the regex '`^HOST\x09*[^\x20]{1000}`' (a simplified regex from SNORT [25]) in Figure 1c. States of the CSA have assigned weights according to the algorithm described above. Figure 2 shows the tree of DFA states obtained by Algorithm 2.

The underlying NFA would look similar as the CSA in Figure 1c, with the difference that there are copies of states 6 and 7 for each value of counter c between 1 and 1000 (and there is a nondeterministic choice over '`\x09`' in states $(6, c{=}i)$ whether to stay in $(6, c{=}i)$ or go to $(6, c{=}i+1)$).

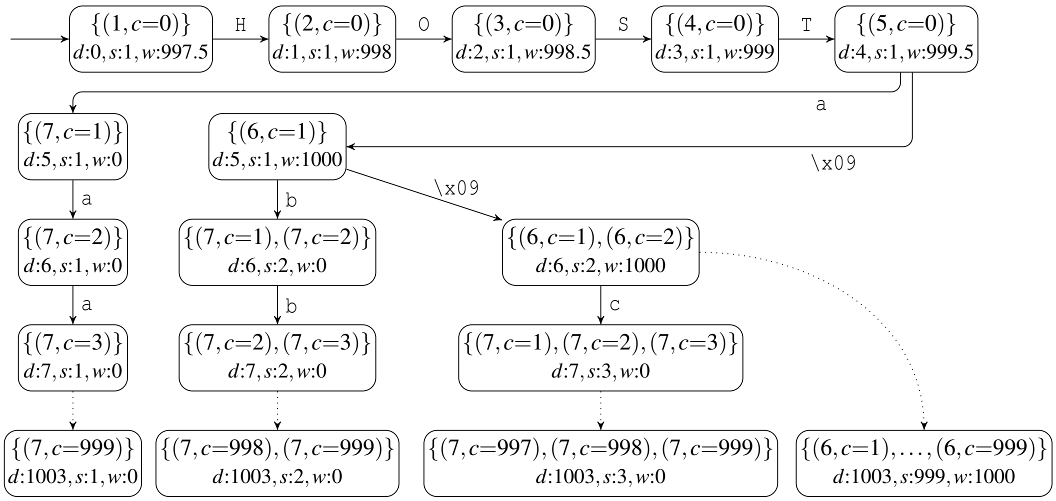


Figure 2: DFA states explored by Algorithm 2 on the regex '`^HOST\x09*[^\x20]{1000}`'.

If traversed using the **Greedy** strategy (assuming that whenever there is a choice in Figure 2 between two DFA states with the same sizes, the strategy picks the left one, e.g., when choosing between $\{(7,c{=}1)\}$ and $\{(6,c{=}2)\}$, **Greedy** would choose $\{(7,c{=}1)\}$) the traversal would first select the branch that goes to state 7 as soon as possible with DFA states of size 1 (the left-most branch), then it would select the branch with DFA states of size 2 (the second branch from the left), etc., generating the text:

```
HOSTaaa...a\n
HOST\x09bbb...b\n
HOST\x09\x09ccc...c\n
...
HOST\x09\x09...\x09yy\n
HOST\x09\x09...\x09\x09z\n
HOST\x09\x09...\x09\x09\x09\n
```

The generated text is sub-optimal because it first targets "easy" DFA states of size 1 and explores the most difficult path (with the longest sequence of `\x09`) only as the last one.

On the other hand, the **Counting** strategy avoids this by using the weights computed for the DFA states, which causes that the paths are explored in the reversed order, preferring state 6 because it has a higher weight:

```
HOST\x09\x09...\x09\x09\x09\n
HOST\x09\x09...\x09\x09z\n
HOST\x09\x09...\x09yy\n
...
HOST\x09\x09ccc...c\n
HOST\x09bbb...b\n
HOSTaaa...a\n
```

Indeed, in our experiments, for this regex, `RE2` took 23 times longer to process the text generated by **Counting** than the text generated by **Greedy**.

6 Experimental Results

We have implemented our approach in a C# prototype called `GadgetCA` and evaluated its capability of generating text causing efficiency problem (ReDoS attack) for the state-of-the-art regex matchers especially with regexes that contain a bounded repetition and compared with existing ReDoS generators.

Matchers. We experiment with the matchers introduced in Section 2. We have *automata-based* matchers `grep` [19] (version 3.3), `RE2` [17], `SRM` [38], and the standard regex matcher in `Rust` [14], all four based on online DFA-simulation, `Hyperscan` [8], which uses NFA simulation, and also the prototype matcher `CA` [46], based on counting set automata (cf. Section 5), which specialises in handling bounded quantifiers (`CA` implements offline CSA-simulation, i.e., it simulates a pre-constructed deterministic CSA on the input text). Then, representing *backtracking* matchers, we have standard library regex matching engines of a wide spectrum of programming languages: `.NET` [26], `Python` [16], `Perl` [44], `PHP` [18], `Java` [13], `JavaScript` [9], and `Ruby` [6]. We note that `grep`, `RE2`, `Rust`, and `Hyperscan` are performance-oriented matchers containing many high- and low-level optimizations.

In Section 6.6, we also experiment with the NIDS Snort [25], which internally uses `Hyperscan`, and with the hardware-accelerated regex matching engine on the NVIDIA BlueField-2 [29] card.

Except the experiments in Section 6.6, we run our benchmarks on a machine with the Intel(R) Xeon(R) CPU E3-1240 v3@3.40 GHz running Debian GNU/Linux (we run `.NET` tools on the Mono platform [1]).

Size of ReDoS text. In order to avoid low-level noise in the measured times of matchers, we generate texts of the size ~50 MB. We use this value since we observed that at around 50 MB, the ratio between the performance of a matcher on a random text and on a generated ReDoS candidate start to stabilize for many of the used matchers. Larger text sizes may still increase the slowdown, but using them would rise the cost of our experiments beyond what we can manage.

`GadgetCA`. Our generator `GadgetCA` generates a text for a potential ReDoS attack using our approach presented in

Sections 4 and 5. In particular, we run the ReDoS text generator for 10 mins or until it completely explores the state space. (We emphasize that generating the ReDoS texts is not a time critical task, since they can be prepared in advance before an attack.) Then, we take the obtained text and copy it as many times as needed in order to obtain a ~50 MB long text.

The particular ReDoS generation algorithm used depends on the chosen search strategy: **GREEDY**, **COUNTING**, **RANDOM**, or **ONELINE** (which is yet another strategy used to target SNORT's `Hyperscan` in Section 6.6).

Other generators. We compared `GadgetCA` against state-of-the-art generators, which are mainly focused on backtracking matchers (indeed, as far as we know, `GadgetCA` is the first generator targeting nonbacktracking matchers), namely `RXXR2` [35], `RegexStatic` [50], `RegexCheck` [53], and `Rescue` [39].[6] These generators use different algorithms to generate a ReDoS text. The generators may consume excessive time while analysing the regex and generating a ReDoS text, hence, we limited their running time to 10 mins (the same as for our generator). Note that all of these tools are research prototypes, so they do not support all regex features. The generators generate a ReDoS text template in the form of a triple $(\mathit{prefix}, \mathit{pump}, \mathit{suffix})$ so that a concrete ReDoS text can be obtained by instantiating $\mathit{prefix} \cdot \mathit{pump}^k \cdot \mathit{suffix}$ for some k. Therefore, we set k for each of the ReDoS texts so that $|\mathit{prefix}| + |\mathit{pump}| \cdot k + |\mathit{suffix}| \approx$ 50 MB.

Dataset. The regexes that we targeted in the experiment were selected from the following sources: (a) the database of over 500,000 real-world regexes coming from an Internet-wide analysis of regexes collected from over 190,000 software projects [12]; (b) the databases of regexes used by *network intrusion detection systems* (NIDSes), in particular, SNORT [25], Bro [37], Sagan [2], and the academic papers [47, 54]; (c) the RegExLib database of regexes [36], which is a website dedicated to regexes for various domain-specific languages (DSLs); (d) regexes from posts on Stack Overflow [31]; (e) industrial regexes from Microsoft used for security purposes [20]; and (f) industrial regexes from TrustPort [45] for detecting security breaches. This gave us a set of 609,992 regexes that we denote as ALL. We then categorized the regexes in ALL into several classes as follows:

SUPPORTED (443,265) is a subset of ALL of syntactically correct regexes without features not supported by our tool—e.g., look-arounds, back-references, etc. Moreover, our tool also does not support regexes with the bounded repetition that yield a non-uniform NCA[7] (there were 101 such regexes).

COUNTERS (47,513) is a subset of SUPPORTED containing regexes with bounded repetition. The rest of SUPPORTED is in NOCOUNTERS (395,752).

ABOVE20 (8,099) is a subset of COUNTERS with regexes where the sum of upper bounds of bounded repetition is above 20 (i.e., regexes where the use of bounded repetition may potentially lead to state space explosion). The rest of COUNTERS is put into BELOW20 (39,414).

6.1 Methodology

Let us now elaborate on the criteria we use to classify ReDoS attacks. In the literature, we found the following used criteria: Shen et al. [39] generate strings at most 128 symbols long and consider a string a ReDoS if Java's regex library matcher makes at least 10^8 steps on it. Davis et al. [11] generate strings of lengths 100 kB–1 MB and call a string a ReDoS if the matcher takes more than 10 s to match it. Staicu and Pradel [41] generate pairs of random and crafted strings of an increasing length and measure the differences of the times the matcher takes for the random and the crafted string in each pair, obtaining a sequence $d_1, d_2, \ldots, d_n$. They consider a crafted string a ReDoS if $d_1 < d_2 < \cdots < d_n$. Rathnayake and Thielecke [35], Wüstholz et al. [53], and Weideman et al. [51] define that a regex is ReDoS-vulnerable if it meets some condition that causes super-linear behaviour (they do not examine the run time of the matchers in detail).

We base our ReDoS criteria on the criteria in [11], but normalize it w.r.t. the significantly lower average matching times for automata-based matchers ([11] only considers backtracking matchers). Our ReDoS criteria are the following:

- **>10s**: The matching takes over 10 s. This corresponds to the throughput of <5 MB/s.
- **>100s**: The matching takes over 100 s. This corresponds to the throughput of <0.5 MB/s.
- **>100×AVG$_{\mathbf{REGEX}}$**: The matcher takes at least 100 times longer than usual on the given regex. The usual time is computed as the average runtime of the same matcher on 10 different ~50 MB-long random texts. This is relevant when the user has some idea about the average performance of the matcher on the regex, presumably from testing.
- **>100×AVG$_{\mathbf{MATCHER}}$**: The matcher takes at least 100 times more than usual globally. The usual time is the average time the same matcher takes on a random ~50 MB-long text across all regexes. However, we include only regexes without the anchors '`^`' and '`$`' since matching regexes with anchors in a random text mostly ends by declaring non-match after processing the first few characters. Average matching times (in seconds) for the matchers are given in Table 1.

6.2 Summary of Results

Let us quickly summarize results obtained in our experimental evaluation, described in detail in the following sections:

R1: Regexes with bounded repetition with higher bounds are potentially vulnerable to ReDoS attacks even for automata-based matchers.

[6] We do not include `SlowFuzz` [33] into the evaluation since we were not able to run it in our test environment. According to [39], `Rescue`, which we include, is more effective than `SlowFuzz`.

[7] Due to the technical difficulty of characterizing such regexes and the relatively small number of regexes affected by this, we refer the interested reader to the description in [46, Section 6.4].

Table 1: The average matching time [s] of a random 50 MB-long text for each of the matchers (averaged over all regexes)

grep	hyper-scan	re2	srm	ca	rust	ruby	php	perl	python	java	javaScript	.NET
0.04	0.07	0.14	1.02	1.32	0.07	2.13	3.10	0.09	0.69	1.11	0.93	2.59

Table 2: Numbers of regexes from ABOVE20 for which various generators successfully generated **>100s** and **>10s**-ReDoS texts. Red (darker) colour emphasizes higher numbers. For each ReDoS criterion, matchers are split into groups based on their types.

Generators		>100s-ReDoS attacks												
		grep	re2	rust	srm	hyper-scan	ca	ruby	php	perl	python	java	java-Script	.NET
GadgetCA	GREEDY	192	72	76	238	0	61	1087	1408	56	200	215	210	390
GadgetCA	COUNTING	216	110	96	272	0	45	1724	1979	89	218	242	211	419
GadgetCA	RANDOM	126	28	48	123	0	46	682	885	60	160	181	111	334
GadgetCA	ONELINE	192	17	32	23	0	56	333	40	187	433	414	378	584
RXXR2		7	0	2	0	0	1	24	0	4	30	11	11	34
RegexCheck		14	0	2	0	0	0	7	1	1	9	8	4	16
RegexStatic		34	1	5	0	0	8	160	63	69	262	253	243	285
Rescue		12	0	3	0	0	2	23	3	4	23	13	12	27
random text		52	4	11	17	0	82	33	47	23	109	162	36	231

Generators		>10s-ReDoS attacks												
		grep	re2	rust	srm	hyper-scan	ca	ruby	php	perl	python	java	java-Script	.NET
GadgetCA	GREEDY	1058	703	274	311	1	135	5050	6580	837	1027	485	955	2629
GadgetCA	COUNTING	1181	1116	295	391	3	121	5440	6289	1294	1503	532	1317	3000
GadgetCA	RANDOM	713	135	259	242	1	106	4405	5389	361	523	385	410	2025
GadgetCA	ONELINE	576	17	78	30	6	130	540	69	402	678	637	485	1448
RXXR2		11	0	2	0	0	1	26	0	5	33	12	13	35
RegexCheck		25	0	3	0	1	0	7	3	7	18	15	9	36
RegexStatic		78	1	9	0	0	19	182	70	78	287	274	254	333
Rescue		11	0	3	0	0	4	24	2	5	26	13	13	28
random text		153	10	70	27	2	137	175	47	147	272	255	228	698

R2: If a regex does not contain counting, it mostly cannot be used to perform a ReDoS attack on automata-based matchers.

R3: Our informed exploration strategy **COUNTING** is better at generating ReDoS texts than the (less informed) strategies **GREEDY** and **RANDOM**.

R4: Other state-of-the-art ReDoS generators are not able to generate ReDoS text for automata-based matchers.

R5: Our techniques can be used to attack mature real-world security solutions.

6.3 R1: Vulnerability of Counting Regexes

In our first experiment, we show that the use of bounded repetition with a higher bound in regexes creates a possible attack surface for ReDoS even for online DFA-simulation-based matchers. We used the set of regexes ABOVE20 and tried to generate ReDoS attacks using `GadgetCA` and other matchers using the methodology described above.

First, see the top part of left-hand side of Table 2, which shows how many successful **>100s**-ReDoS texts different settings of `GadgetCA` were able to generate for online DFA-simulation-based matchers. Notice that we were able to generate 216 ReDoS texts for `grep`, 110 ReDoS texts for `RE2`, 96 ReDoS texts for `Rust`, and 272 ReDoS texts for `SRM` (using the **COUNTING** strategy).

Next, the right-hand side of the table shows data for the weaker ReDoS criterion **>10s**. The number of generated successful ReDoS-texts is significantly higher: 1,181 for `grep`, 1,116 for `RE2`, 295 for `Rust`, and 391 for `SRM` (all using the **COUNTING** strategy).

Under both ReDoS criteria above, the **COUNTING** strategy achieves the best results for online DFA-simulation-based matchers and, moreover, for the **>10s** criterion also for backtracking matchers. Further, **GREEDY** obtains significantly better results than **RANDOM**, proving that our informed search strategies are better in generating hard texts than uninformed search, confirming **R3**. The table also shows that `Hyperscan`, `SRM`, and `CA` are more robust towards being attacked by our ReDoS texts: `SRM` has a special support for counters and `CA` is a matcher that uses counting set automata (cf. Section 5). We will discuss `Hyperscan` in Section 6.6. See Appendix A for examples of evil texts generated by `GadgetCA` using the **COUNTING** strategy.

In the left-hand side of Table 3, we provide a comparison of the number of **>100×AVG**$_{\mathbf{MATCHER}}$-ReDoS texts generated by the tools. Again, note that a slowdown of >100 times wrt. the global average for the matcher was achieved on many regexes for online DFA-simulation-based matchers (2,457 for `grep`, 742 for `RE2`, 1,016 for `Rust`, and 300 for `SRM`). Since the global average matching time for `PHP` was 3.1 s and we used the timeout of 300 s for matchers, in this table, the `PHP` column contains the number of timeouts instead. A more detailed analysis for other slowdown ratios is in Figure 3. Notice that although `Hyperscan` looks almost invincible in the results in Table 2, we are able to slow it down by a factor of 10–50 in many instances (543).

On the other hand, the right-hand side of Table 3 compares the numbers of generated **>100×AVG**$_{\mathbf{REGEX}}$-ReDoSes. In this case, a slowdown of >100 times wrt. the average time for the matcher and the regex was also achieved often for online DFA-simulation-based matchers (1,157 for `grep`, 1,465 for `RE2`, 1,066 for `Rust`, and 279 for `SRM`).

We conclude that many counting regexes can be successfully attacked using ReDoS texts created by our generator.

6.4 R2: Regexes Without Counting

The second experiment shows that when targeting automata-based matchers, it is indeed important to exploit counting.

Since the set SUPPORTED is too large for us to run a ReDoS generator for each regex, we use a quick filter based on the intuition that ReDoS in these matchers is caused by generating many large DFA states. Hence we run DFA construction for each regex from the set. If the construction terminates

Table 3: Numbers of regexes with successfully generated **>100×AVG$_{\mathbf{MATCHER}}$** and **>100×AVG$_{\mathbf{REGEX}}$**-ReDoS texts.

Generators		>100×AVG$_{MATCHER}$-ReDoS attacks													>100×AVG$_{REGEX}$-ReDoS attacks												
		grep	re2	rust	srm	hyper-scan	ca	ruby	php	perl	python	java	java-Script	.NET	grep	re2	rust	srm	hyper-scan	ca	ruby	php	perl	python	java	java-Script	.NET
GadgetCA	Greedy	1741	15	95	18	2	40	260	38	382	367	328	314	431	878	14	57	12	0	0	164	9	174	232	190	194	203
	Counting	2457	742	1016	300	5	67	1355	1596	1473	277	279	258	416	1157	1465	1066	279	2	3	1085	796	1252	407	142	140	171
	Random	2033	120	122	289	3	46	348	388	412	176	177	117	258	1066	320	292	130	0	0	153	156	266	91	63	60	72
	OneLine	1796	17	99	23	20	53	322	34	441	448	405	379	521	966	15	57	16	23	0	199	9	208	277	232	228	238
RXXR2		13	0	2	0	0	1	24	0	5	30	10	10	34	1	0	2	0	0	0	10	0	4	22	8	8	20
RegexCheck		104	0	5	0	1	0	7	1	7	11	8	4	14	4	0	4	0	0	0	3	0	0	4	3	2	2
RegexStatic		93	1	9	0	1	7	159	50	80	263	253	243	279	47	5	5	0	0	0	80	14	49	137	125	134	90
Rescue		12	0	3	0	0	2	23	2	5	23	13	12	26	1	2	4	0	0	1	12	2	6	15	7	6	14

with less than 1,000 states, we consider the regex safe. After 1,000 DFA states, the construction is stopped, and the regex is marked as possibly vulnerable. This test is quick, since constructing 1,000 DFA states is fast, and the vast majority of the regexes have even much smaller DFAs.

To assess the accuracy of the test in predicting that a regex is not vulnerable for automata-based matchers, we apply the test on the regexes from ABOVE20 for which we did manage to generate a ReDoS text for automata-based matchers (cf. the experiment in **R1**). From ∼2,000 of them, only `grep` and `Rust` had cases with DFA smaller than 1,000 states, namely 24 cases, 6 for `grep` and 18 for `Rust` (`RE2`, `CA`, `Hyperscan`, and `SRM` had none). These counterexample cases witness that our filter is not always right, at least for `grep` and `Rust`, and ReDoS with automata-based matchers might be possible even with small DFA. Still, the scarcity of these cases confirms that the test is a good predictor even for `grep` and `Rust`.[8]

Running the test on SUPPORTED resulted in the following numbers of regexes with DFAs with >1,000 states:

NOCOUNTERS	BELOW20	ABOVE20
175 (0.04 %)	343 (0.8 %)	1,600 (20 %)

We then used `GadgetCA` to generate ReDoS candidates for the regexes in NOCOUNTERS ∪ BELOW20 whose DFAs had more than 1,000 states. Only 7 regexes caused **>100s**-ReDoS for automata-based matchers, two for `grep`— ‘`\^.{20}\$`’ and ‘`\^_.{19}\$`’ (note that both also contain “higher bounds” for the quantifiers)—and 5 for `SRM`. A **>10s**-ReDoS was caused by 24 regexes for `grep` and ∼6 regexes for each of `RE2`, `Rust`, and `SRM`. The relative sizes of the sets indicate that regexes without higher repetition bounds are much less vulnerable to ReDoS for automata-based matchers (518 vulnerable from 435,166 in NOCOUNTERS ∪ BELOW20 while 1,600 vulnerable from 8,099 in ABOVE20).

6.5 R4: Comparison with Other Generators

Our next experiment confirms that our generator can create new ReDoS attacks much more effectively than existing tools.

First, compare the middle part of the left- and right-hand side of Table 2. For other generators, the ten-fold stronger **>100s**-ReDoS criterion makes almost no difference: they cannot find and exploit the features of the regex that make the matchers slow down (both for automata-based and backtracking matchers). The same holds for the **>100×AVG$_{\mathbf{MATCHER}}$** and **>100×AVG$_{\mathbf{REGEX}}$**-ReDoS criteria in Table 3.

Second, compare the bottom part (random text) with the middle part of the table. For counting regexes, a random text is in the majority of cases actually better in creating a ReDoS than current state-of-the-art ReDoS generators (only `RegexStatic` can keep up with the random text on some matchers). Relating this to `GadgetCA` in the top part of the table reveals that the numbers of successfully attacked regexes for the two criteria differ significantly, hence `GadgetCA` indeed succeeds in exploiting the critical feature of the regex.

Third, the comparison of the top part with the middle part of the table shows that `GadgetCA` significantly outperforms other matchers on online DFA-simulation-based matchers and most of the other generators even on backtracking matchers (the only exception being `RegexStatic`, which is comparable on some backtracking matchers).

6.6 R5: Real-World Security Solutions

Our final experiment demonstrates that the results obtained in **R1** carry over to real-world security solutions which should be prepared for being targeted by (Re)DoS. We carried out an extensive evaluation of the abilities of Snort 3 [25], a popular and often used NIDS, which internally uses `Hyperscan`, to withstand ReDoS attacks generated by `GadgetCA`. Instead of using some of the previously introduced datasets, which might contain regexes created by people unaware of the dangers of ReDoS, we used regexes from rulesets provided with Snort, which are written by security experts and tested in production. In particular, we used regexes from the following four rulesets: (i) Emerging Threats Pro, (ii) Emerging Threats 3CORESec (versions 157 and 164), and (iii) Talos LightSPD (version 2021-03-11-001). We call the obtained set of 1,112 PCRE regexes SNORT (from the original 22,425 original regexes we removed 16,094 regexes not supported by our tool, and then filtered the 1,112 regexes with quantifier bounds at least 20). The experiment was run in two different settings: (i) on a commodity `x86_64` machine with Snort using `Hyperscan` and (ii) on a computer with an NVIDIA BlueField-2 card [28], which provides its own hardware-accelerated regex matching solution.

Modified ReDoS Generator. In this experiment, we use a modified version of our ReDoS generator for the reason that although `Hyperscan`, used within Snort, can be counted

[8] The 24 cases are probably caused by specific implementation techniques or different interpretation of the regexes. The 18 cases of `Rust` seem to be related to handling of large character classes (`\w` appears in all 18 cases).

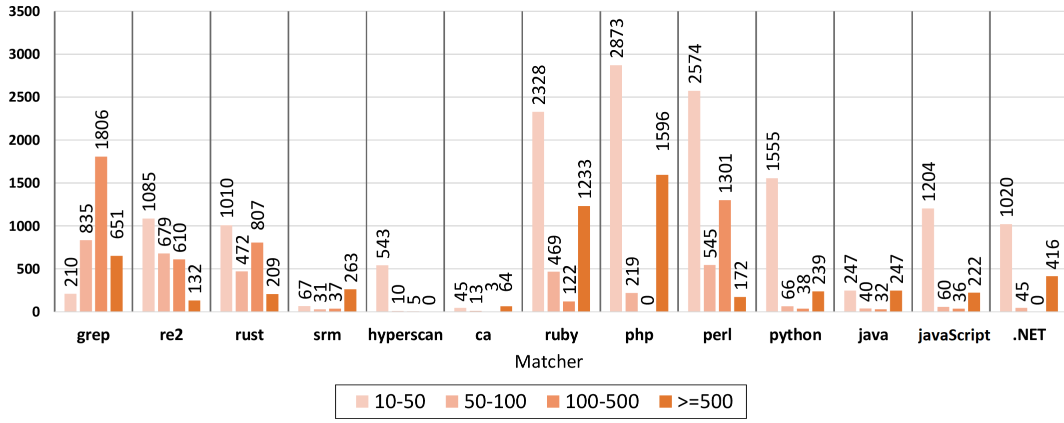


Figure 3: Histogram of ratios between times of matchers for random and ReDoS text generated by `GadgetCA`.

as an automata-based matcher, it is not based on online DFA-simulation. Experiments discussed in the previous sections indeed show that our ReDoS generator, which targets mainly online DFA-simulation, is only mildly successful with `Hyperscan`. We therefore use here a modification of `GadgetCA` tailored for `Hyperscan`.

We specifically target the following coarse abstraction of `Hyperscan`'s matching algorithm: the regex is split into a sequence of *sub-strings* (not containing any regex operator) and *sub-regexes* (or a choice of such sequences) so that a word is matched if it is a concatenation $w = v_1 \cdots v_n$ of the sub-strings of the given regex and words matched by the sub-regexes. The first phase of matching tests whether w contains all the sub-strings in the right order, by an extension of the Boyer-Moore algorithm. The second phase tests whether the remaining sub-words are matched by the respective sub-regexes. The opportunity for slowing `Hyperscan` down is in the second phase, which uses NFA-simulation to match the sub-expressions.[9]

We therefore aim at generating evil texts that contain the needed sequence of sub-strings and therefore pass the first phase of matching, and where the second phase is also hard. To do that, we use our generator to get a single evil word u over a run that takes the CSA from the initial to a final state. The word is essentially generated by the first iteration of the while-loop on Line 11 of Algorithm 2 parameterised with the strategy **Counting** (a single CSA run that aims at maximising the sizes of counting sets). The word u is then iterated to get the output text $w = uuuu\dots$ of the required length.

A word w generated this way is likely to be evil for the following two reasons: (i) every occurrence of u in w contains all sub-strings, generating many possible splits of w into the sub-strings and the parts to be matched by the sub-regexes; (ii) the word u, generated by our generator, is likely to force large DFA states, expensive for NFA simulation. Note also that, unlike for online DFA simulation, it does not matter that the encountered DFA states are likely to be found repeatedly in the repeating instances of u since NFA simulation is not caching the DFA.

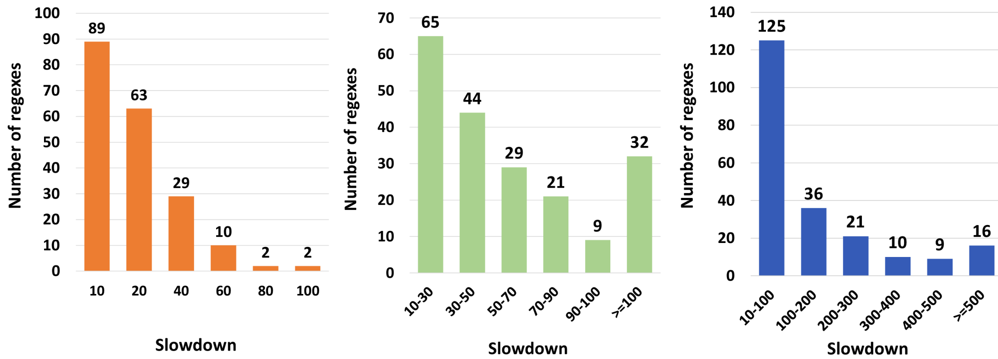
6.6.1 Snort with `Hyperscan` on `x86_64`

We installed Snort 3 with enabled performance monitor and `Hyperscan` on a commodity `x86_64` machine (we used Intel i7-10510U CPU@1.80 GHz with 4 Hyper-Threading cores). Then, we were running Snort on 100 MB-large `PCAP` files with random and ReDoS IPv4 traffic that we generated and captured the processing time of the regex matching engine, as provided in the output of the performance monitor module. We ran two experiments with two different sizes of IP packets for two selected Ethernet frames' MTUs: 1,500 B and 9,000 B (we note that bigger sizes of the payload could be used to attack Snort with TCP reassembly turned on). See Figures 4a and 4b for the slowdown that we achieved with our ReDoS text over random text.

The histograms clearly show the Snort rulesets we used contain many possibilities for slowing Snort down (see Appendix B for the most vulnerable regexes). In particular, using packet size of 1,500 B, in 43 cases we achieved a slowdown of over 40×, with 2 regexes slowing the matcher down over 100×. The number of vulnerable regexes is even higher for the packet size 9,000 B: 91 regexes yield a slowdown of over 50× and 32 regexes over 100×.

We contacted the development team of Snort and did the responsible disclosure of the discovered vulnerable regexes. Snort development team stated that the vulnerability is stemming from the `Hyperscan` library, and they mitigate it by restricting the length of packets on which the matching is performed as well as by using timeouts (the standard configuration of Snort comes with the backtracking-based PCRE engine enabled, which is, however, even more prone to attacks). This might, however, lead to skipping the malicious content that can be presented at the end of the packet/data, making the NIDS ineffective: malicious packets may get passed to applications behind the NIDS.

[9] Our abstraction of `Hyperscan` is coarse, but it is simple and sufficient for our needs: to show that methods similar to those for online DFA-simulation can be used to find vulnerabilities of `Hyperscan` too. A specialised ReDoS generator based on a more thorough analysis of `Hyperscan`'s algorithm might yield better results, but is already out of the scope of this paper.



(a) SNORT 3@`Hyperscan` (1,500 B) (b) SNORT 3@`Hyperscan` (9,000 B) (c) NVIDIA BlueField-2

Figure 4: Histograms of slowdowns for SNORT 3 with `Hyperscan` (packet sizes 1,500 B and 9,000 B) and BlueField-2 regex matching for ReDoS texts over random texts.

6.6.2 NVIDIA BlueField-2

In the second part of this experiment, we used an NVIDIA BlueField-2 data processing unit (BF2) MBF2H332A-AEEOT [29], which integrates eight 64-bit ARMv8 Cortex-A72 cores and houses two 25 GbE interfaces. BF2 provides hardware-accelerated regex matching capabilities, accessible via NVIDIA's *data plane development kit* (DPDK) [28]: in our experiments, we used the regex compiler `rxpc` and the testbed for the regex matching engine called `rxpbench`. In this experiment, we ran `rxpbench` on blocks of random and ReDoS texts of the length 100 GB (this time, we did not need to chunk the texts into packets and provided the text directly in memory) and measured the throughput of the matcher. We measured that the regex matching engine itself enables in-memory processing at ∼40 Gbps. For the evaluation, we used a subset of SNORT rules containing 617 regexes that we name SNORT-BF2 (we took all regexes from SNORT that could be compiled by `rxpc`, which does not support some advanced features of PCRE, such as negative look-ahead).

See Figure 4c for histograms of slowdowns we obtained with our ReDoS text as compared to random text. Observe that we obtained a slowdown of more than 100× on the ReDoS text in over 92 cases. Moreover, for 16 cases, we obtained a slowdown over 500× (with the highest slowdown ratio being 2,194×). See Appendix B for a list of regexes on which we obtained the largest slowdown. We have reported the vulnerability to NVIDIA, which confirmed it to be caused by a conceptual limitation of their regex matching engine. We plan to cooperate on a possible mitigation.

Our results indicate that ReDoS attacks are in general successful in slowing down the throughput of the most recent hardware utilized for NIDS in the industry. Moreover, we emphasize that for a successful ReDoS attack on an NIDS, it suffices to have a single vulnerable rule in the used rulesets.

7 Mitigation Techniques

Standard techniques for mitigation of ReDoS attacks are the following: (i) setting a resource limit (e.g., a timeout) and (ii) limiting the size of the input (e.g., to the first 100 characters) of the regex matcher. Although such techniques can avert the scenario of a server becoming unresponsive, they leave a part of the input traffic not classified and potentially harmful or unnecessarily dropped. A mitigation specific for regexes with the counting operator is to substitute it by the star `*` operator, which over-approximates the language of the original regex (this might yield other issues, such as increasing the number of false positives in an NIDS).

There are, however, two ways how users of regex matchers can mitigate the attacks without the mentioned disadvantages:

1. Use our ReDoS generator `GadgetCA` to evaluate whether a regex is ReDoS-vulnerable.
2. Use a matching algorithm that can handle counting efficiently, the one implemented in the tool CA or possibly also `SRM` (these matchers are still too immature to be used in production, but an efficient implementation of the techniques they use within `RE2` or `Hyperscan` should give rise to a robust regex matching solution).

8 Conclusion and Future Work

We have shown that nonbacktracking automata-based regex matchers, which are sometimes suggested as a mitigation of ReDoS, *are still ReDoS-vulnerable*. We have developed a method for constructing inputs for these matchers that make them perform poorly and cause significant slowdown on a large class of regexes, in particular those with counting.

In future, we plan to focus on developing robust regex matchers that could prevent these kinds of attacks. A first proof of concept is the matcher CA from [46], but the class of counting regexes it support is quite restricted; we will therefore explore formal models that can deal with more general classes of counting regexes efficiently.

Acknowledgment. We thank the reviewers for their comments on how to improve the quality of the paper and the CyberGrid group from FEEC BUT for lending us the NVIDIA BF2 card. This work was supported by the Czech Ministry of Education, Youth and Sports project LL1908 of the ERC.CZ programme, the Czech Science Foundation project 20-07487S, and the FIT BUT internal project FIT-S-20-6427.

References

[1] Mono. https://www.mono-project.com/.

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

[3] Valentin Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291 – 319, 1996.

[4] Adam Baldwin. Regular expression denial of service affecting Express.js. https://medium.com/node-security/regular-expression-denial-of-service-affecting-express-js-9c397c164c43, 2016.

[5] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.

[6] James Britt and Neurogami Secret Laboratory. Regexp - Ruby. https://ruby-doc.org/core-2.3.1/Regexp.html, 2021.

[7] Wikipedia contributors. Regular expression—wikipedia. https://en.wikipedia.org/w/index.php?title=Regular_expression&%20oldid=852858998, 2019.

[8] Intel Corporation. https://github.com/intel/hyperscan, 2021.

[9] Oracle Corporation. Regexp - JavaScript. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp, 2021.

[10] James C. Davis. Rethinking regex engines to address ReDoS. In *ESEC/FSE'19*, pages 1256–1258. ACM, 2019.

[11] James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. The impact of regular expression denial of service (ReDoS) in practice: An empirical study at the ecosystem scale. In *ESEC/FSE'18*, pages 246–256. ACM, 2018.

[12] James C. Davis, Louis G. Michael IV, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. Why aren't regular expressions a lingua franca? An empirical study on the re-use and portability of regular expressions. In *ESEC/FSE'19*, pages 1256–1258. ACM, 2019.

[13] MDN Web Docs. Class pattern - java. https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/regex/Pattern.html, 2021.

[14] docs.rs. regex - rust. https://docs.rs/regex/1.5.4/regex/, 2021.

[15] Stack Exchange. Outage postmortem. http://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016, 2016.

[16] Python Software Foundation. re - Python. https://docs.python.org/3.6/library/re.html, 2021.

[17] Google. RE2. https://github.com/google/re2.

[18] The PHP Group. PCRE patterns - PHP. https://www.php.net/manual/en/regexp.introduction.php, 2021.

[19] Mike Haertel et al. GNU grep. https://www.gnu.org/software/grep/.

[20] Lukáš Holík, Ondřej Lengál, Olli Saarikivi, Lenka Turoňová, Margus Veanes, and Tomáš Vojnar. Succinct determinisation of counting automata via sphere construction. In *Proc. of APLAS'19*, volume 11893 of *LNCS*, pages 468–489. Springer, 2019.

[21] Intel. Hyperscan 5.4 developer's reference guide, performance considerations. http://intel.github.io/hyperscan/dev-reference/performance.html, 2021.

[22] James Kirrage, Asiri Rathnayake, and Hayo Thielecke. Static analysis for regular expression denial-of-service attacks. In *NSS'13*, volume 7873 of *LNCS*, pages 135–148. Springer, 2013.

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

[24] Blake Loring, Duncan Mitchell, and Johannes Kinder. Sound regular expression semantics for dynamic symbolic execution of JavaScript. In *PLDI'19*, pages 425–438. ACM, 2019.

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

[26] Microsoft. https://docs.microsoft.com/en-us/dotnet/api/system.text.regularexpressions.regex.match, 2020.

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

[28] NVIDIA. Data plane development kit (dpdk). https://developer.nvidia.com/networking/dpdk.

[29] Nvidia. Nvidia BlueField-2 DPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-2-dpu.pdf, 2020.

[30] Open Information Security Foundation. Suricata. https://suricata.io/.

[31] Stack Overflow. Question and answer site for programmers. http://stackoverflow.com/.

[32] OWASP. Regular expression denial of service — ReDoS. https://owasp.org/www-community/attacks/Regular_expression_Denial_of_Service_-_ReDoS, 2020.

[33] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *CCS'17*, pages 2155–2168. ACM, 2017.

[34] Asiri Rathnayake. *Semantics, analysis and security of backtracking regular expression matchers*. PhD thesis, University of Birmingham, UK, 2015.

[35] Asiri Rathnayake and Hayo Thielecke. Static analysis for regular expression exponential runtime via substructural logics. *CoRR*, abs/1405.7058, 2014.

[36] RegExLib.com. The Internet's first Regular Expression Library. http://regexlib.com/.

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

[38] Olli Saarikivi, Margus Veanes, Tiki Wan, and Eric Xu. Symbolic regex matcher. In *TACAS'2019*, volume 11427 of *LNCS*, pages 372–378. Springer, 2019.

[39] Yuju Shen, Yanyan Jiang, Chang Xu, Ping Yu, Xiaoxing Ma, and Jian Lu. Rescue: crafting regular expression DoS attacks. In *ASE'18*, pages 225–235. ACM, 2018.

[40] Henry Spencer. Software solutions in C. chapter A Regular-expression Matcher, pages 35–71. Academic Press Professional, Inc., 1994.

[41] Cristian-Alexandru Staicu and Michael Pradel. Freezing the web: A study of ReDoS vulnerabilities in JavaScript-based web servers. In *USENIX'18*, pages 361–376. USENIX Association, 2018.

[42] Satoshi Sugiyama and Yasuhiko Minamide. Checking time linearity of regular expression matching based on backtracking. *IPSJ Online Transactions*, 7:82–92, 2014.

[43] Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, 1968.

[44] Iain Truskett. Perl regular expressions reference - perl. https://perldoc.perl.org/5.22.0/perlreref, 2021.

[45] TrustPort. World class cyber security. https://www.trustport.com/, 2021.

[46] Lenka Turoňová, Lukáš Holík, Ondřej Lengál, Olli Saarikivi, Margus Veanes, and Tomáš Vojnar. Regex matching with counting-set automata. *Proc. ACM Program. Lang.*, 4(OOPSLA):218:1–218:30, 2020.

[47] Milan Češka, Vojtěch Havlena, Lukáš Holík, Ondřej Lengál, and Tomáš Vojnar. Approximate reduction of finite automata for high-speed network intrusion detection. In *Proc. of TACAS'18*, volume 10806 of *LNCS*. Springer, 2018.

[48] Peipei Wang, Chris Brown, Jamie A. Jennings, and Kathryn T. Stolee. Demystifying regular expression bugs. *Empir. Softw. Eng.*, 27(1):21, 2022.

[49] Peipei Wang and Kathryn T. Stolee. How well are regular expressions tested in the wild? In *FSE'18*, pages 668–678. ACM, 2018.

[50] Nicolaas Weideman. `RegexStatic`. https://github.com/NicolaasWeideman/RegexStaticAnalysis, 2015.

[51] Nicolaas Weideman, Brink van der Merwe, Martin Berglund, and Bruce W. Watson. Analyzing matching time behavior of backtracking regular expression matchers by using ambiguity of NFA. In *CIAA'16*, volume 9705 of *LNCS*, pages 322–334. Springer, 2016.

[52] Matthias Wübbeling. Regular expression security. *ADMIN*, 55, 2020.

[53] Valentin Wüstholz, Oswaldo Olivo, Marijn J. H. Heule, and Isil Dillig. Static detection of DoS vulnerabilities in programs that use regular expressions. In *TACAS'17*, volume 10206 of *LNCS*, pages 3–20, 2017.

[54] Liu Yang, Rezwana Karim, Vinod Ganapathy, and Randy Smith. Improving NFA-based signature matching using ordered binary decision diagrams. In *Recent Advances in Intrusion Detection*, pages 58–78. Springer Berlin Heidelberg, 2010.

A Examples of Generated Evil Texts

Example 1. For the regex `Oid=[^\0D\x0A]{1000}` (originating from SNORT) `GadgetCA` (strategy: **COUNTING**) generates a text of several lines, each of the length 1,003 characters and containing full or unfinished copies of the string '`Oid=`':

`(Oid=)`^{250}`Oid`
`(Oid=)`^{249}`OidOid=`
...

Each new copy of `Oid=` adds a new value to the counting-set and since all characters of the string '`Oid=`' belong to the character class `[^\0D\x0A]`, which is being counted, all existing values in the counting-set are also incremented. The variety of full or unfinished copies of the prefix `Oid=` forces creation of many large DFA states with different counter values. The length of the shortest string matched by the regex is 1,004 characters, however, we aim at generating the longest non-matching lines, and so the length of the generated lines is 1,003 characters. The generated text is demanding for most automata-based matchers (matching time for 50 MB input: `grep`: 0.83 s, `Hyperscan`: 0.06 s, `RE2`: 228.28 s, `SRM`: 46.54 s, `CA`: 2.77 s, `Rust`: 96.7 s).

Example 2. For the regex `<[^>\x20]{500}` (originating from SNORT) `GadgetCA` generates a text containing substrings of the length 500 (the length of a minimal match is 501) with many different placements of '<':

`(<)`^{500}
`(<)`^{99}Q`(<)`^{400}
...

where Q is an arbitrary character other than '<'. This text also forces matchers to generate many DFA states with different counter values, yielding the following matching times (on 50 MB texts): `grep`: 0.11 s, `Hyperscan`: 0.1 s, `RE2`: TO, `SRM`: TO, `GadgetCA`: 2.8 s, `Rust`: 112.34 s.

B Attacks on Real-world Security Solutions

In Tables 4 and 5, we provide examples of regexes for which we managed to obtain a significant slowdown of SNORT (with `Hyperscan` as the regex matching engine) and the NVIDIA BlueField-2 DPU respectively.

Table 4: Slowdown of regex matching in Snort3 with Hyperscan on x86_64.

SID	Slowdown (MTU=9000B)	Slowdown (MTU=1500B)	Regex
46310	213.95	78.89	[?&]u=[^&\s]{35}
31068	172.32	50.49	<hostname>.{0,250}[\x60\x3b\x7c\x24\x28\x26]
2644	165.81	65.57	\(\s*TIMESTAMP\s*(\s*(\x27[^\x27]+’|\x22[^\x22]+\x22)\s*,)\s*((\x27[^\x27]{1000,})|(\x22[^\x22]{1000,}))
13364	163.52	71.15	src\s*\x3D(3D)?\s*[’"][^’"]{244}
19925	160.95	58.7	value\s*=\s*[\x27\x22][^\x27\x22]{257}
2102614	157.95	52.68	TIME_ZONE\s*=\s*((\x27[^\x27]{1000,})|(\x22[^\x22]{1000,}))
17659	157.41	79.18	\s*\x28(\x27[^\x27]{64}|\x27[^\x27]*\x27\s*,\s*\x27[^\x27]{64})
2611	157.39	49.67	USING\s*((\x27[^\x27]{1000})|(\x22[^\x22]{1000}))
46309	152.34	65.7	[?&]p=[^&\s]{260}
39982	145.5	55.61	[?&]sn=[^&]{129}
2651	140.95	51.26	NUMTO(DS|YM)INTERVAL\s*\(\s*\d+\s*,\s*((\x27[^\x27]{1000,})|(\x22[^\x22]{1000,}))
2102699	138.15	49.25	TO_CHAR\s*\(\s*SYSTIMESTAMP\s*,\s*(\x27[^\x27]{256}|\x22[^\x22]{256})
19121	136.82	63.89	SET\s*EXPLAIN\s*FILE\s*TO\s*[\x22\x27][^\x22\x27]{927}
2640	135.24	56.41	\(\s*(\x27[^\x27]*’|\x22[^\x22]+\x22)\s*,\s*(true|false)\s*,\s*((\x27[^\x27]{1000,})|(\x22[^\x22]{1000,}))
15114	135.06	51.46	embed src=\s*(\x27[^\x27]{1000}|\x22[^\x22]{1000}|[^\s\x22\x27]{1000})
29679	133.17	77.16	document\.execCommand \(\s*[\x22\x27]InsertUnorderedList[\x22\x27]\s*\)\s*\x3B.{0,250}\s*\w+\.swapNode\(\s*[A-Za-z\(\)\"\’\.\=\]{1,75}\s*\)\s*document\.execCommand\(\s*[\x22\x27]Undo[\x22\x27]\s*\)\s*\x3B
39707	131.81	48.37	folder\s*name\s*=\s*[\x22][^\x22]{200}
39709	125.72	49.06	folder\s*name\s*=\s*[\x27][^\x27]{200}
27805	123.9	45.19	\/3001[0-9A-F]{262,304}
20889	122.09	47.98	<\s*valitem[^>]*\s(value|name)\s*=\s*([\x22\x27])[^\x22\x27]{104}
16516	121.4	44.42	sys\x2eolapimpl\x5ft\x2eodcitablestart\x28[^\x2c]+\x2c[^\x2c]+\x2c\s*\x27?[^\x2c\x27]{303}
29184	120.92	54.3	encoding\x3D[\x22\x27][^\x22\x27]{1024}
14991	120.65	61.81	select\s+xmlquery\s*\x28\s*(\x27|\x22)[^\x27\x22]{512}
43005	120.52	35.49	[?&]psk=[^&]{256}
29185	118.76	50.13	version\x3D[\x22\x27][^\x22\x27]{1024}
33310	117.95	54.87	\x3C\x21ENTITY\s+.*\s+\x22\x26[^\x22]{700}
27808	110.1	29.94	\x2f\?[a-f0-9]{60,66}
42078	108.17	43.24	[?&](cmd|pwd|usr)=[^&]{64}
2488	106.01	43.94	name=\s*[^\r\n\x3b\s\x2c]{300}

Table 5: Slowdown of regex matching at an NVIDIA BlueField-2 card.

SID	Thourghput on Random Text [Gbs]	Thourghput on Redos Text [Gbs]	Slowdown	Regex
2046	41.24	0.02	2,193.76	/\sPARTIAL.*BODY\.PEEK\[[^\]]\1024\/
19213	41.19	0.02	1,681.04	/Subject\x3a\x20[^\n]*\x3fQ\x3f[^\n]{512}/
17367	40.30	0.03	1,174.83	/\d{3}\s+[^\n]{1019}/
6507	41.09	0.04	957.74	/\x2fnds[^\r\n]{1000}/
1021	41.21	0.04	956.06	/\s{230,}\.htr/
20241	40.66	0.04	947.72	/Oid\x3D[^\x0D\x0A]{1000}/
15489	40.58	0.04	920.28	/\x3cimg[^\x3e]*src\x3d(\x22|\x27)?[^\x22\x27\s]{300}/
3547	40.79	0.05	829.08	/php.*\x3f[^\n]{256}/
25586	41.03	0.06	732.67	/host=[^&]{1024}/
8060	41.31	0.06	728.49	/GET\s\x2f[^\r\n]{900}/
31354	41.14	0.06	656.15	/\x28\x3f\x3d[^)]{300}/
3149	41.22	0.06	655.34	/object\s[^>]*type\s*=\s*[\x22\x27][^\x22\x27]*\x2f{32}/
17568	41.11	0.06	641.29	/\w{3}\x25\x30\x30[^\r\n]{2000}/
4127	41.15	0.08	545.82	/\x2fnds\x2f[^&\r\n\x3b]{500}/
38287	40.97	0.08	543.40	/akey=[^&]{500}/
18484	41.14	0.08	536.42	/https?\x3a\x2f\x2f[^\n\r]{1000}/
43545	41.22	0.08	485.54	/-group[^\r\n\s]{1280}/
33310	40.96	0.09	469.76	/\x3C\x21ENTITY\s+.*\s+\x22\x26[^\x22]{700}/
2701	41.20	0.09	434.18	/sid=[^&\x3b\r\n]{255}/
2107	41.21	0.10	427.53	/\sCREATE\s[^\n]{1024}/
18579	41.18	0.10	426.76	/(Context|Action)\x3D[^\x26\x3b]{1024}/
20889	40.87	0.10	419.58	/<\s*valitem[^>]*\s(value|name)\s*=\s*([\x22\x27])[^\x22\x27]{104}/
2826	41.28	0.10	416.17	/(\(\s*(\x27[^\x27]*\x27|\x22[^\x22]+\x22)\s*,\s*(\x27[^\x27]{1075,}|\x22[^\x22]{1075,})|\(\s*(\x27[^\x27]{1075,}|\x22[^\x22]{1075,})|\(\s*((\x27[^\x27]*\x27|\x22[^\x22]+\x22)\s*,\s*){2}(\x27[^\x27]{1075,}|\x22[^\x22]{1075,}))/
2826	40.73	0.10	410.57	/(\(\s*(\x27[^\x27]*\x27|\x22[^\x22]+\x22)\s*,\s*(\x27[^\x27]{1075,}|\x22[^\x22]{1075,})|\(\s*(\x27[^\x27]{1075,}|\x22[^\x22]{1075,})|\(\s*((\x27[^\x27]*\x27|\x22[^\x22]+\x22)\s*,\s*){2}(\x27[^\x27]{1075,}|\x22[^\x22]{1075,}))/
21671	41.16	0.10	403.94	/zip\x3a\x2f\x2f[^\x0A\x20\x09\x0B\x0C\x85\x3E\x3C]{400}/
20240	41.20	0.11	375.87	/Template\x3D[^\x0D\x0A]{1000}/
27940	41.08	0.11	374.44	/password=[^\x26]{1024}/
2103070	41.00	0.11	361.25	/\sFETCH\s[^\n]{500}/
36195	41.21	0.12	338.07	/actserver=[^&]{982}/
36196	40.86	0.12	335.47	/actserver=[^&]{987}/