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

Lenka Turoňová1, Lukáš Holík1, Ivan Homoliak1, Ondřej Lengál1, Margus Veanes2, Tomáš Vojnar1

1Faculty of Information Technology, Brno University of Technology, Brno, Czech Republic
{ituronova,holik,ihomoliak,lengal,vojnar}@fit.vutbr.cz
2Microsoft Research, Microsoft, Redmond, USA
margus@microsoft.com

Abstract

In this paper, we study the performance characteristics of non-backtracking 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-

---

1Essentially, 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]. 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 DFA 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.

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-

---

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

3Bounded 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 decimal 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 specialization 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 · 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

\[ R ::= \alpha \mid (R) \mid RR \mid R^* \mid R \{n,m\} \]

where \( n, m \in \mathbb{N}, 0 \leq n, 0 < m, n \leq m \), and \( \alpha \) is a character class, i.e., a set of characters from Σ. A character class is most often of the form \( a \ldots b \), \([a_1 \ldots b_1 a_2 \ldots b_2 \ldots a_n \ldots b_n]\), or \([^*a_1 \ldots b_1 a_2 \ldots b_2 \ldots a_n \ldots b_n]\), denoting a singleton containing the character \( a \in \Sigma \), the entire set \( \Sigma \), 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, ‘\( \cdot \)’ 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, \( \delta \) is a set of transitions of the form \( q \cdot (a) \cdot r \) 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 \( q_0 \cdot (a_1) \cdot q_1 \cdot (a_2) \cdot \ldots \cdot (a_n) \cdot q_n \) with \( q_n \in F \).

The automaton is deterministic (DFA) if for every state \( q \) and symbol \( a \), \( \delta \) has at most one transition \( q \cdot (a) \cdot r \). 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 \( S, S \cdot (a) \cdot (S' \in \delta') \) for \( S' = \{s' \mid s \in S \land s \cdot (a) \cdot s' \in \delta\} \).

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 \( x.v.v.y \) such that \( v \in L(R) \), i.e., \( w \in L(\cdot R \cdot \cdot) \). Anchors, ‘\( ^* \)’ at the start of the regex and ‘\( \cdot \)’ 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, sub-matching, 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 ‘(ab)*bc’ against the input string ‘(ab)*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 deterministic 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 expo-
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.

**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. $dfa \leftarrow \text{new DFA}()$
2. $q \leftarrow dfa.\text{init}([s_0])$
3. for $i \leftarrow 1$ to $|w|$ do // $O(|w| \cdot |A|)$
   4. if $dfa.\text{final}[q]$ then return true
   5. $q' \leftarrow dfa.\text{get_successor_id}(q, w[i])$ // $O(|A|)$
   6. $q \leftarrow q'$
5. if $dfa.\text{big}()$ then $q \leftarrow dfa.\text{init}(dfa.\text{id2state}[q])$
6. if $dfa.\text{ineffective}()$ then disable DFA caching
7. return false

### 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 $w_1 \cdots w_n$
of lines, each line $w_i$ generated by a run $\rho_i$ starting at the initial state of the DFA. Each run $\rho_i$ 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 $\rho_i$, from where it navigates to new unvisited DFA states through DFA transitions chosen according to some **successor selection criterion**.

The run $\rho_i$ is thus a concatenation $\rho_i = \rho_i^1 . \rho_i^2$ of a prefix $\rho_i^1$ through already visited DFA states and a suffix $\rho_i^2$ 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 $\rho_i$ 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 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 $\rho_i$. The algorithm maintains a set \textit{visited} of IDs of DFA states that were visited by some run $\rho_i$, and a set \textit{unvisited} of IDs of discovered but yet unvisited states. The while loop terminates when there are no states remaining in \textit{unvisited}. To select the starting state $q$ of $\rho_i$ (Line 8) and construct the shortest run to $q$ quickly (via function \textit{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 $p \in \textit{visited} \cup \textit{unvisited}$ remembers the last transition in the shortest discovered run from the initial state to $p$, namely, the predecessor state $\text{pre}(p)$ 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 $\text{pre}(p)$, $d(p)$, 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 \textit{prefix}(q) can then construct the shortest discovered run to $q$ in the form $q_0 \rightarrow a_1 \rightarrow q_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_n \rightarrow q_n$ by taking $q_0 = q$, $q_i = \text{pre}(q_{i+1})$, and $a_i = \sigma(q_i)$ for all $0 \leq i < n$, and return the word $a_1 \cdots a_n$ read along this run. The starting state $q$ of $\rho_i$ is chosen on Line 8 from \textit{unvisited} by selecting the DFA state (obtained as $\text{dfa.id2state}[q]$) of the largest size with the smallest distance $d(q)$.

The suffix of the run, $\rho_i'$, 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 \textit{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 \textit{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 \textit{strategy} on Line 21.

**Algorithm 2: DFA-based text generation**

**Input:** An NFA $A = (Q, \delta, s_0, F)$, successor selection criterion \textbf{strategy}.

**Output:** evil text $w$ (concatenation of several lines)

```plaintext
1   dfa ← new DFA
2   q0 ← dfa.init(\{s_0\})
3   unvisited.enqueue(q0)
4   d(q0) ← 0
5   visited ← 0
6   w ← ε
7   while unvisited ≠ ∅ do
8           q ← unvisited.dequeue_nearest_largest()
9           visited.add(q)
10          w ← w . prefix(q)
11          while true do
12               succ ← ∅
13               for a ∈ Σ do
14                   p ← dfa.get_successor_id[q, a]
15                   if dfa.final[p] ∨ p ∈ visited then continue
16                   succ.add(p, a)
17                   unvisited.enqueue(p)
18                   if d(q) + 1 < d(p) ∨ d(p) = None then
19                       (d(p), σ(p), pre(p)) ← (d(q)+1, a, q)
20               if succ = ∅ then break
21               (q′, a) ← succ.choose(strategy)
22               unvisited.remove(q′)
23               visited.add(q′)
24               q ← q′
25               w ← w . a
26          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 \textbf{strategy}. We will consider the following three strategies.
The first strategy, **RANDOM**, picks from `succest` 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 `\text{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, v)` where `q` is a control state and `v` 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):

```
\{(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)\}.
```

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.\(^3\) A counting set for a counter `c` is also restricted to only contain values between 0 and `\text{max}_c` (the set-increment operation removes values greater than `\text{max}_c`). An example of a CSA

\(^3\)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.
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:

\[
\begin{align*}
\{(q), c = \{0\}\}, \\
\{(q, s), c = \{0\}\}, \\
\{(q, s), c = \{0, 1\}\}, \\
\ldots \\
\{(q, s), c = \{0, \ldots, 99\}\}.
\end{align*}
\]

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 CSA 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 \(max_c + 4\). 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.

\[
weight_c = \frac{max_c + add_cnt_c}{inc_cnt_c}.
\]

The final weight of a cycle is then computed as a sum of weights for individual counters \(\sum_{c} weight_c\) 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 \(q \rightarrow r\) 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 \(^*\text{HOST}\x09[^\x20]*\{1000\}\) (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 \((6, c=i + 1)\)).
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 generators 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], SM [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

![Figure 2: DFA states explored by Algorithm 2 on the regex ‘^[^\x09]*[^[\x20]]1000’](image)
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 \([\text{prefix}, \text{pump}, \text{suffix}]\) so that a concrete ReDoS text can be obtained by instantiating \(\text{prefix} \cdot \text{pump}^k \cdot \text{suffix}\) for some \(k\). Therefore, we set \(k\) for each of the ReDoS texts so that \(|\text{prefix}| + |\text{pump}| \cdot k + |\text{suffix}| \approx 50\) 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 NCA7 (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 \(10^6\) 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 Thieleccke [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.
- **\(>100 \times \text{AVG}_{\text{REX}}\):** 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.
- **\(>100 \times \text{AVG}_{\text{MATCHER}}\):** 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)

<table>
<thead>
<tr>
<th>Generators</th>
<th>grep</th>
<th>hyper-</th>
<th>scan</th>
<th>re2</th>
<th>srm</th>
<th>ca</th>
<th>rust</th>
<th>ruby</th>
<th>php</th>
<th>perl</th>
<th>python</th>
<th>java</th>
<th>javaScript</th>
<th>.NET</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.04</td>
<td>0.07</td>
<td>0.14</td>
<td>1.02</td>
<td>1.32</td>
<td>3.12</td>
<td>2.13</td>
<td>3.10</td>
<td>0.09</td>
<td>0.69</td>
<td>1.11</td>
<td>0.93</td>
<td>2.59</td>
<td></td>
</tr>
</tbody>
</table>

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 type.

<table>
<thead>
<tr>
<th>Generators</th>
<th>grep</th>
<th>hyper-</th>
<th>scan</th>
<th>re2</th>
<th>srm</th>
<th>ca</th>
<th>rust</th>
<th>ruby</th>
<th>php</th>
<th>perl</th>
<th>python</th>
<th>java</th>
<th>javaScript</th>
<th>.NET</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greedy</td>
<td>192</td>
<td>72</td>
<td>76</td>
<td>238</td>
<td>0</td>
<td>61</td>
<td>109</td>
<td>1408</td>
<td>56</td>
<td>200</td>
<td>215</td>
<td>210</td>
<td>219</td>
<td>210</td>
</tr>
<tr>
<td>COUNTING</td>
<td>216</td>
<td>110</td>
<td>96</td>
<td>272</td>
<td>0</td>
<td>45</td>
<td>1724</td>
<td>1979</td>
<td>89</td>
<td>218</td>
<td>242</td>
<td>211</td>
<td>219</td>
<td>211</td>
</tr>
<tr>
<td>RANDOM</td>
<td>126</td>
<td>28</td>
<td>48</td>
<td>123</td>
<td>0</td>
<td>46</td>
<td>682</td>
<td>885</td>
<td>60</td>
<td>160</td>
<td>181</td>
<td>111</td>
<td>111</td>
<td>111</td>
</tr>
<tr>
<td>OneLine</td>
<td>192</td>
<td>17</td>
<td>32</td>
<td>23</td>
<td>0</td>
<td>56</td>
<td>333</td>
<td>40</td>
<td>187</td>
<td>433</td>
<td>414</td>
<td>378</td>
<td>584</td>
<td>584</td>
</tr>
<tr>
<td>REEDY</td>
<td>7</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>24</td>
<td>4</td>
<td>30</td>
<td>11</td>
<td>11</td>
<td>34</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>RegexCheck</td>
<td>14</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>1</td>
<td>4</td>
<td>8</td>
<td>4</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>RegexCompare</td>
<td>34</td>
<td>1</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>160</td>
<td>63</td>
<td>69</td>
<td>262</td>
<td>253</td>
<td>243</td>
<td>285</td>
<td>285</td>
</tr>
<tr>
<td>Rescue</td>
<td>12</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>23</td>
<td>3</td>
<td>4</td>
<td>23</td>
<td>13</td>
<td>12</td>
<td>27</td>
<td>27</td>
</tr>
<tr>
<td>random text</td>
<td>52</td>
<td>4</td>
<td>11</td>
<td>17</td>
<td>0</td>
<td>82</td>
<td>33</td>
<td>47</td>
<td>23</td>
<td>109</td>
<td>162</td>
<td>36</td>
<td>231</td>
<td>231</td>
</tr>
</tbody>
</table>

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 inregexes 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 differen
tsettings 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 SSM (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 SSM (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, SSM, and CA are more robust towards being attacked by our ReDoS texts: SSM has a special support for counters and CA is a matcher that uses counting set automata (cf. Section 5).

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 $>100 \times \text{AVGMatcher}$ and $>100 \times \text{AVGRegex}$-ReDoS texts.

<table>
<thead>
<tr>
<th>Generators</th>
<th>grep</th>
<th>re2</th>
<th>rust</th>
<th>smr</th>
<th>hyper-</th>
<th>scan</th>
<th>ca</th>
<th>ruby</th>
<th>php</th>
<th>perl</th>
<th>python</th>
<th>java</th>
<th>JavaScript</th>
<th>NET</th>
</tr>
</thead>
<tbody>
<tr>
<td>GREEDY</td>
<td>164</td>
<td>15</td>
<td>95</td>
<td>18</td>
<td>2</td>
<td>40</td>
<td>260</td>
<td>38</td>
<td>782</td>
<td>4</td>
<td>367</td>
<td>328</td>
<td>374,146</td>
<td>431</td>
</tr>
<tr>
<td>COUNTING</td>
<td>2457</td>
<td>724</td>
<td>1016</td>
<td>500</td>
<td>5</td>
<td>67</td>
<td>1355</td>
<td>1550</td>
<td>1873</td>
<td>277</td>
<td>279</td>
<td>258</td>
<td>416</td>
<td>416</td>
</tr>
<tr>
<td>RANDOM</td>
<td>2033</td>
<td>120</td>
<td>122</td>
<td>289</td>
<td>3</td>
<td>46</td>
<td>348</td>
<td>388</td>
<td>412</td>
<td>176</td>
<td>177</td>
<td>117</td>
<td>258</td>
<td>258</td>
</tr>
<tr>
<td>ONLINE</td>
<td>1796</td>
<td>17</td>
<td>99</td>
<td>23</td>
<td>20</td>
<td>53</td>
<td>322</td>
<td>34</td>
<td>441</td>
<td>448</td>
<td>405</td>
<td>379</td>
<td>521</td>
<td>521</td>
</tr>
<tr>
<td>NOCOUNTERS</td>
<td>13</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>24</td>
<td>0</td>
<td>5</td>
<td>30</td>
<td>10</td>
<td>10</td>
<td>34</td>
<td>34</td>
</tr>
<tr>
<td>BELOW20</td>
<td>104</td>
<td>6</td>
<td>2</td>
<td>7</td>
<td>1</td>
<td>7</td>
<td>11</td>
<td>8</td>
<td>4</td>
<td>14</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ABOVE20</td>
<td>93</td>
<td>11</td>
<td>9</td>
<td>1</td>
<td>7</td>
<td>1759</td>
<td>50</td>
<td>80</td>
<td>263</td>
<td>253</td>
<td>243</td>
<td>279</td>
<td>245</td>
<td>245</td>
</tr>
<tr>
<td>RE2</td>
<td>47</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>80</td>
<td>14</td>
<td>49</td>
<td>137</td>
<td>125</td>
<td>134</td>
<td>90</td>
<td>90</td>
</tr>
<tr>
<td>RegexCheck</td>
<td>12</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>23</td>
<td>2</td>
<td>5</td>
<td>23</td>
<td>13</td>
<td>12</td>
<td>26</td>
<td>26</td>
</tr>
<tr>
<td>Rescue</td>
<td>12</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>23</td>
<td>2</td>
<td>5</td>
<td>23</td>
<td>13</td>
<td>12</td>
<td>26</td>
<td>26</td>
</tr>
</tbody>
</table>

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—`\w{19}` and `\w{19}`—(note that both also contain “higher bounds” for the quantifiers)—and 5 for SMR. A $>10s$-ReDoS was caused by 24 regexes for grep and 6 regexes for each of RE2, Rust, and SMR. 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 matches slow down (both for automata-based and backtracking matchers). This confirms that the criteria for evaluating the number of successfully attacked regexes is comparable to the benchmarks provided.

6.6 R5: Real-World Security Solutions

Our final experiment demonstrates that the results obtained in R1 can be applied to real-world security solutions which should be prepared for being targeted by (Re)DoS. We evaluated the three popular regex engines of Snort 3 [25], which is comparable to the previous tests. In particular, we used regexes from the following four rulesets:

- Emerging Threats Pro,
- Emerging Threats 3CORESec (versions 157 and 164), and
- Talos LightSPD (version 2021-03-11-001).

We call the obtained set of 1,112 PCRE regexes Snort3 (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 efficiently, the execution times of the regexes are not a major consideration.
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 \textit{sub-strings} (not containing any regex operator) and \textit{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 repeating instances of \( u \) contains all the 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 \( 40 \times \), with 2 regexes slowing the matcher down over \( 100 \times \). The number of vulnerable regexes is even higher for the packet size 9,000 B: 91 regexes yield a slowdown of over \( 50 \times \) and 32 regexes over \( 100 \times \).

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.
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 \(\sim 40\) 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 \(\times\) on the ReDoS text in over 92 cases. Moreover, for 16 cases, we obtained a slowdown over 500 \(\times\) (with the highest slowdown ratio being 2,194 \(\times\)). 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 rule sets.

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 \(\ast\) 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 FECE 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

Peipei Wang and Kathryn T. Stolee. How well are Hardware solutions in C. chapter
Henry Spencer. Software solutions in C. chapter
Henry Spencer. Software solutions in C. chapter
Satoshi Sugiyama and Yasuhiko Minamide. Checking trustport.com/.
Henry Spencer. Software solutions in C. chapter

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 1,003 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 sub-strings of the length 500 (the length of a minimal match is 501) with many different counter values of ‘<’: ...

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.

<table>
<thead>
<tr>
<th>SID</th>
<th>Slowdown (MTU=9000B)</th>
<th>Slowdown (MTU=1500B)</th>
<th>Regex</th>
</tr>
</thead>
<tbody>
<tr>
<td>46310</td>
<td>213.95</td>
<td>78.89</td>
<td>[&amp;][&amp;]u=[^&amp;\s]{35}</td>
</tr>
<tr>
<td>31068</td>
<td>172.32</td>
<td>50.49</td>
<td>&lt;hostnametree&gt;[0,250][&amp;]60x3b[7c\x24x28x26]</td>
</tr>
<tr>
<td>2644</td>
<td>165.81</td>
<td>65.57</td>
<td>(\text{TIMESTAMP})[s,(\text{\textasciitilde})][&amp;]\texttt{TIMESTAMP}[0,256][s,(\text{\textasciitilde})][&amp;]2221[7c\x24x28x22][257]</td>
</tr>
<tr>
<td>13364</td>
<td>163.52</td>
<td>71.15</td>
<td>(\text{\textasciitilde})\texttt{TIMESTAMP}[0,256][s,(\text{\textasciitilde})][&amp;]2221[7c\x24x28x22][257]</td>
</tr>
<tr>
<td>19925</td>
<td>160.95</td>
<td>58.7</td>
<td>(\text{\textasciitilde})\texttt{TIMESTAMP}[0,256][s,(\text{\textasciitilde})][&amp;]2221[7c\x24x28x22][257]</td>
</tr>
<tr>
<td>210214</td>
<td>157.95</td>
<td>52.68</td>
<td>(\text{\textasciitilde})\texttt{TIMESTAMP}[0,256][s,(\text{\textasciitilde})][&amp;]2221[7c\x24x28x22][257]</td>
</tr>
<tr>
<td>17659</td>
<td>157.41</td>
<td>79.18</td>
<td>(\text{\textasciitilde})\texttt{TIMESTAMP}[0,256][s,(\text{\textasciitilde})][&amp;]2221[7c\x24x28x22][257]</td>
</tr>
<tr>
<td>2611</td>
<td>157.39</td>
<td>49.67</td>
<td>(\text{\textasciitilde})\texttt{TIMESTAMP}[0,256][s,(\text{\textasciitilde})][&amp;]2221[7c\x24x28x22][257]</td>
</tr>
<tr>
<td>46309</td>
<td>152.34</td>
<td>65.7</td>
<td>(\text{\textasciitilde})\texttt{TIMESTAMP}[0,256][s,(\text{\textasciitilde})][&amp;]2221[7c\x24x28x22][257]</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2046</td>
<td>41.24</td>
<td>0.02</td>
<td>2.19376</td>
<td>(\text{\textasciitilde})\texttt{PARTIAL.*BODY.PEEK}[%%][1024%]</td>
</tr>
<tr>
<td>19213</td>
<td>41.19</td>
<td>1.68104</td>
<td>7.56</td>
<td>(\text{\textasciitilde})\texttt{Subject}8\x20[%%]\texttt{XO}00[%%][512%]</td>
</tr>
<tr>
<td>17657</td>
<td>40.39</td>
<td>1.17833</td>
<td>1091</td>
<td>(\text{\textasciitilde})\texttt{ad}[%%][3%%][1019%]</td>
</tr>
<tr>
<td>6507</td>
<td>41.09</td>
<td>0.04</td>
<td>957.54</td>
<td>(\text{\textasciitilde})\texttt{2f2f}[%%][100%]</td>
</tr>
<tr>
<td>1021</td>
<td>41.21</td>
<td>0.04</td>
<td>956.06</td>
<td>(\text{\textasciitilde})\texttt{[%%][100%]}</td>
</tr>
<tr>
<td>20241</td>
<td>40.66</td>
<td>0.04</td>
<td>947.72</td>
<td>(\text{\textasciitilde})\texttt{0x3d}[%%][100%]</td>
</tr>
<tr>
<td>15409</td>
<td>40.95</td>
<td>0.04</td>
<td>920.28</td>
<td>(\text{\textasciitilde})\texttt{3c}[%%][100%]</td>
</tr>
<tr>
<td>3547</td>
<td>40.79</td>
<td>0.04</td>
<td>829.09</td>
<td>(\text{\textasciitilde})\texttt{%%}[100%]</td>
</tr>
<tr>
<td>25586</td>
<td>41.03</td>
<td>0.06</td>
<td>732.67</td>
<td>(\text{\textasciitilde})\texttt{host}[%%][1024%]</td>
</tr>
<tr>
<td>8086</td>
<td>41.31</td>
<td>0.06</td>
<td>728.49</td>
<td>(\text{\textasciitilde})\texttt{%%}[100%]</td>
</tr>
<tr>
<td>31354</td>
<td>41.14</td>
<td>0.06</td>
<td>656.15</td>
<td>(\text{\textasciitilde})\texttt{%%}[100%]</td>
</tr>
<tr>
<td>3149</td>
<td>41.22</td>
<td>0.06</td>
<td>655.34</td>
<td>(\text{\textasciitilde})\texttt{%%}[100%]</td>
</tr>
<tr>
<td>17568</td>
<td>41.11</td>
<td>0.06</td>
<td>641.29</td>
<td>(\text{\textasciitilde})\texttt{%%}[100%]</td>
</tr>
<tr>
<td>4121</td>
<td>41.15</td>
<td>0.08</td>
<td>545.82</td>
<td>(\text{\textasciitilde})\texttt{%%}[100%]</td>
</tr>
<tr>
<td>38287</td>
<td>40.97</td>
<td>0.08</td>
<td>543.40</td>
<td>(\text{\textasciitilde})\texttt{%%}[100%]</td>
</tr>
<tr>
<td>18484</td>
<td>41.14</td>
<td>0.08</td>
<td>536.42</td>
<td>(\text{\textasciitilde})\texttt{%%}[100%]</td>
</tr>
<tr>
<td>43545</td>
<td>41.22</td>
<td>0.08</td>
<td>488.54</td>
<td>(\text{\textasciitilde})\texttt{%%}[1280%]</td>
</tr>
<tr>
<td>33310</td>
<td>40.96</td>
<td>0.09</td>
<td>469.76</td>
<td>(\text{\textasciitilde})\texttt{%%}[1280%]</td>
</tr>
<tr>
<td>2701</td>
<td>41.20</td>
<td>0.09</td>
<td>434.18</td>
<td>(\text{\textasciitilde})\texttt{%%}[1280%]</td>
</tr>
<tr>
<td>2107</td>
<td>41.21</td>
<td>0.10</td>
<td>427.53</td>
<td>(\text{\textasciitilde})\texttt{%%}[1280%]</td>
</tr>
<tr>
<td>18579</td>
<td>41.18</td>
<td>0.10</td>
<td>426.76</td>
<td>(\text{\textasciitilde})\texttt{%%}[1280%]</td>
</tr>
<tr>
<td>20889</td>
<td>40.87</td>
<td>0.10</td>
<td>419.58</td>
<td>(\text{\textasciitilde})\texttt{%%}[1280%]</td>
</tr>
<tr>
<td>2826</td>
<td>41.28</td>
<td>0.10</td>
<td>416.17</td>
<td>(\text{\textasciitilde})\texttt{%%}[1280%]</td>
</tr>
<tr>
<td>2826</td>
<td>40.73</td>
<td>0.10</td>
<td>410.57</td>
<td>(\text{\textasciitilde})\texttt{%%}[1280%]</td>
</tr>
<tr>
<td>21671</td>
<td>41.16</td>
<td>0.10</td>
<td>403.94</td>
<td>(\text{\textasciitilde})\texttt{%%}[1280%]</td>
</tr>
<tr>
<td>20240</td>
<td>41.20</td>
<td>0.11</td>
<td>375.87</td>
<td>(\text{\textasciitilde})\texttt{%%}[1280%]</td>
</tr>
<tr>
<td>27940</td>
<td>41.08</td>
<td>0.11</td>
<td>374.44</td>
<td>(\text{\textasciitilde})\texttt{%%}[1280%]</td>
</tr>
<tr>
<td>210307</td>
<td>41.00</td>
<td>0.11</td>
<td>361.25</td>
<td>(\text{\textasciitilde})\texttt{%%}[1280%]</td>
</tr>
<tr>
<td>36195</td>
<td>41.21</td>
<td>0.12</td>
<td>338.07</td>
<td>(\text{\textasciitilde})\texttt{%%}[1280%]</td>
</tr>
<tr>
<td>36196</td>
<td>40.86</td>
<td>0.12</td>
<td>335.47</td>
<td>(\text{\textasciitilde})\texttt{%%}[1280%]</td>
</tr>
</tbody>
</table>