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

Lenka Turoňová    Lukáš Holík    Ivan Homoliak
Ondřej Lengál    Tomáš Vojnar
Brno University of Technology, Czech Republic

Margus Veanes
Microsoft Research, USA

USENIX Security’22
Regular Expression Denial of Service (ReDoS)

What is a ReDoS? (in this talk)

- DoS by giving a regex matcher a hard input text
Regular Expression Denial of Service (ReDoS)

What is a ReDoS? (in this talk)
- DoS by giving a regex matcher a hard input text

normal.txt (500 kB):
```
abcdefghijklmopqrstuvwxyz0123456789abcdefghijklmnopqrstuvwxyz...
```

```bash
$ time grep "a.{500}$" normal.txt
```

$ time grep "a.{500}$" evil.txt
```
real 1.63
user 1.52
sys 0.00
```

19 × slower!
(. . . and this is only a very simple example)
Regular Expression Denial of Service (ReDoS)

What is a **ReDoS**? (in this talk)
- DoS by giving a regex matcher a **hard input text**

**normal.txt (500 kB):**
```
abcdefghijklmnopqrstuvwxyz0123456789abcdefghijklmnopqrstuvwxyz0123456789
abcdeghijklmopqrstuvwxyz0123456789abcdefghijklmnopqrstuvwxyz0123456789
```

```
$ time grep "a.{500}$" normal.txt
real 0.09
user 0.08
sys 0.00
```

**evil.txt (500 kB):**
```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Regular Expression Denial of Service (ReDoS)

What is a ReDoS? (in this talk)
- DoS by giving a regex matcher a hard input text

normal.txt (500 kB):

```
abcdefgijklmnopqrstuvwxyz0123456789abcdefgijklmnopqrstuvwxyz...
```

```
$ time grep "a.{500}" normal.txt
real 0.09
user 0.08
sys 0.00
```

evil.txt (500 kB):

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa...
```

```
$ time grep "a.{500}" evil.txt
```

19 × slower!

(this may be a ReDoS)

(. . . and this is only a very simple example)
Regular Expression Denial of Service (ReDoS)

What is a ReDoS? (in this talk)
- DoS by giving a regex matcher a hard input text

normal.txt (500 kB):
```
abcdefgijklmnopqrstuvwxyz0123456789abcdefgijklmnopqrstuvwxyz...
```

$ time grep "a.\{500\}$" normal.txt
```
real 0.09
user 0.08
sys 0.00
```

evil.txt (500 kB):
```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa...
```

$ time grep "a.\{500\}$" evil.txt
```
real 1.63
user 1.52
sys 0.00
```
Regular Expression Denial of Service (ReDoS)

What is a ReDoS? (in this talk)

- DoS by giving a regex matcher a hard input text

normal.txt (500 kB):

```
abcdefghijkmnopqrstuvwxyz0123456789abcdefghijklmnoprstuv...
```

```
$ time grep "a.{500}$" normal.txt
```

```
real 0.09
user 0.08
sys 0.00
```

evil.txt (500 kB):

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa...
```

```
$ time grep "a.{500}$" evil.txt
```

```
real 1.63
user 1.52
sys 0.00
```

19× slower!
(this may be a ReDoS)
(...and this is only a very simple example)
ReDoS

Real-world threat

- **Stack Overflow**, 2016: 34 minute outage (regex "^\+$"; line "...\a")
- ReDoS vulnerability in **Express.js** (package `negotiator`), 2016
- ReDoS vulnerability in **Node.js** (package `url-regex`), 2020
- ...

Often caused by the use of backtracking matchers (PHP, JS, Perl, Ruby, .NET, ...)
ReDoS

Real-world threat

- **Stack Overflow**, 2016: 34 minute outage (regex "\+$\); line "\d+\a")
- ReDoS vulnerability in **Express.js** (package negotiator), 2016
- ReDoS vulnerability in **Node.js** (package url-regex), 2020
- ... 

Often caused by the use of backtracking matchers (PHP, JS, Perl, Ruby, .NET, ...)

**Solution:** Use nonbacktracking matchers!

---

**The Case of the Poisoned Event Handler:**
Weaknesses in the Node.js Event-Driven Architecture

James Davis
Virginia Tech
Blacksburg, VA, USA
djavis@vt.edu

Gregor Kibow
Virginia Tech
Blacksburg, VA, USA
gregor@vt.edu

Dongyoon Lee
Virginia Tech
Blacksburg, VA, USA
dongyoon@vt.edu

---

**Freezing the Web:**
A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers

Cristian-Alexandru Staica
Department of Computer Science
TU Darmstadt

Michael Pradel
Department of Computer Science
TU Darmstadt

---

**Testing Regex Generalizability And Its Implications**
A Large-Scale Many-Language Measurement Study

James C. Davis, Daniel Meyer, and Ayman M. Kasez
Department of Computer Science
Virginia Tech
(davisci,jamiey, ayman@vt.edu)

Dongyoon Lee
Department of Computer Science
Stony Brook University and Virginia Tech
dongyoon@cs.stonybrook.edu
ReDoS

Real-world threat

- **Stack Overflow**, 2016: 34 minute outage (regex "\+$"; line "\n...\n")
- ReDoS vulnerability in **Express.js** (package `negotiator`), 2016
- ReDoS vulnerability in **Node.js** (package `url-regex`), 2020
- ...

Often caused by the use of **backtracking** matchers (PHP, JS, Perl, Ruby, .NET, ...)

**Solution:** **Use nonbacktracking matchers!**

---

The Case of the Poisoned Event Handler: Weaknesses in the Node.js Event-Driven Architecture

James Davis
Virginia Tech
Blacksburg, VA, USA
davj@vt.edu

Gregor Kudow
Virginia Tech
Blacksburg, VA, USA
gkudow@vt.edu

Dongyoon Lee
Virginia Tech
Blacksburg, VA, USA
dongyoon@vt.edu

Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers

Cristian-Alexandru Staicu
Department of Computer Science
TU Darmstadt

Michael Pradel
Department of Computer Science
TU Darmstadt

Testing Regex Generalizability And Its Implications - A Large-Scale Many-Language Measurement Study

James C. Davis, Daniel Moyer, and Ayman M. Kammoun
Department of Computer Science
Virginia Tech
(davisjc1, dmoyer, akm}@vt.edu

Dongyoon Lee
Department of Computer Science
Stony Brook University and Virginia Tech
dongyoon@vt.edu

dongyoon@cs.stonybrook.edu

A hybrid regex engine except for back references, every feature of JavaScript regular expressions can be supported by a linear-time regular expression engine. As back

Another defense mechanism could be to use a more sophisticated regular expression engine that guarantees linear matching time. The problem is that these en...

... or is it?

Nonbacktracking matchers

Nonbacktracking matchers:
- **textbook**: construct NFA, **determinize** ($\mathcal{O}(2^{|A|})$), perform match — linear time
  - $\rightsquigarrow$ DFA might be too big!! (often thousands, millions of macrostates)

![Diagram of state transition](q \xrightarrow{a} r \xmapsto{a} \{q\} \xrightarrow{a} \{q, r\})
Nonbacktracking matchers

Nonbacktracking matchers:

- **textbook**: construct NFA, **determinize** ($O(2^{|A|})$), perform match — linear time
  - $\rightsquigarrow$ DFA might be too big!! (often thousands, millions of macrostates)
- **in practice** (Thompson’s algorithm):
  1. construct NFA
  2. determinize on-the-fly while doing membership test
  3. **cache!** $\rightsquigarrow O(|w|)$ average-case complexity
- **tools**: grep, re2, Rust, SRM, HYPERSCAN*

[Diagram of NFA and DFA transition]
Nonbacktracking matchers

Nonbacktracking matchers:
- **textbook**: construct NFA, **determinize** ($O(2^{|A|})$), perform match — linear time
  - DFA might be too big!! (often thousands, millions of macrostates)
- **in practice** (Thompson’s algorithm):
  1. construct NFA
  2. determinize on-the-fly while doing membership test
  3. **cache!** $\sim O(|w|)$ average-case complexity
- **tools**: grep, re2, Rust, SRM, HYPERSCAN*
- **can still run slow!** due to low cache utilization
- How can we systematically generate ReDoS texts for nonbacktracking matchers?
Nonbacktracking matchers

Nonbacktracking matchers:

- **textbook**: construct NFA, **determinize** ($O(2^{|A|})$), perform match — **linear time**
  - $\rightarrow$ DFA might be **too big**!! (often thousands, millions of macrostates)
- **in practice** (Thompson’s algorithm):
  1. construct NFA
  2. determinize **on-the-fly** while doing membership test
  3. **cache!** $\rightarrow O(|w|)$ average-case complexity
- **tools**: grep, re2, Rust, SRM, HYPERSCAN*
- **can still run slow!** due to **low cache utilization**
- How can we systematically generate ReDoS texts for nonbacktracking matchers?
- **Exploit counting!** (a.k.a. quantifiers, bounded repetition, ...)

### Counting in regexes

```
a {5,42}
b {100}
```
How much are counting regexes prone to ReDoS?

- 609,992 regexes (GitHub, SNORT, Bro, RegExLib, Microsoft, TrustPort, ...)
- removed unsupported (look-arounds/back-references/...)
- \( \sim \) 443,265 regexes
- classify according to sum of upper bounds in counting, e.g., \( a\{5, 42\} \)
- DFA Big: \( \geq 1,000 \) states (often the size of DFA cache)

<table>
<thead>
<tr>
<th>regex set</th>
<th>#</th>
<th>#DFA big</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>no counting</td>
<td>395,752</td>
<td>175</td>
<td>0.04 %</td>
</tr>
<tr>
<td>counting bounds ( \leq 20)</td>
<td>39,414</td>
<td>343</td>
<td>0.8 %</td>
</tr>
<tr>
<td>counting bounds ( &gt; 20)</td>
<td>8,099</td>
<td>1,600</td>
<td>20. %</td>
</tr>
</tbody>
</table>
ReDoS generator for nonbacktracking matchers

- generate input text by search through the DFA
  - generate non-matching text
  - prefer macrostates that are
    1. unvisited (matcher cache miss)
    2. big (hard to compute successors)

\[
\begin{align*}
\{q, r, s\} & \quad \{q, t\} & \quad \{s, t, u, v, w, z\} \\
& \quad a & \quad b & \quad c
\end{align*}
\]

- try to enforce \(O(|w| \cdot |A|)\) runtime

(A = the NFA; \(|A| = |Q| + |\Delta|\))
ReDoS generator for nonbacktracking matchers

- generate input text by search through the DFA
  - generate non-matching text
  - prefer macrostates that are
    1. unvisited (matcher cache miss)
    2. big (hard to compute successors)

- Issue: how to navigate to big macrostates?
  - DFA too big, cannot construct!
  - instead of DFA, use **Counting-Set Automaton** [OOPSLA’20]
    - allows compact deterministic representation of regexes with counting

\[
\begin{align*}
\{q, t\} &\quad \text{a} \quad \{q, r, s\} \\
{t} &\quad \text{b} \quad \{s, t, u, v, w, z\} \\
\end{align*}
\]

\[O(|w| \cdot |A|) \text{ runtime} \quad (A = \text{the NFA}; |A| = |Q| + |\Delta|)\]
ReDoS generator for nonbacktracking matchers

- generate input text by search through the DFA
  - generate non-matching text
  - prefer macrostates that are
    1. unvisited (matcher cache miss)
    2. big (hard to compute successors)

- try to enforce $O(|w| \cdot |A|)$ runtime

**Issue**: how to navigate to big macrostates?

- DFA too big, cannot construct!
- instead of DFA, use **Counting-Set Automaton** [OOPSLA’20]
  - allows compact deterministic representation of regexes with counting

**ReDoS generator** **GadgetCA**

![Diagram showing the DFA with macrostates and transitions]

(A = the NFA; $|A| = |Q| + |\Delta|$)
Experiments

Dataset:

- 609,992 regexes (GitHub, SNORT, Bro, RegExLib, Microsoft, TrustPort, . . .)
- removed unsupported (look-arounds/back-references/. . .)
- keep regexes with sum of counter bounds >20
- \( \sim 8,099 \) regexes
Experiments

Dataset:
- 609,992 regexes (GitHub, SNORT, Bro, RegExLib, Microsoft, TrustPort, ...)
- removed unsupported (look-arounds/back-references/...)
- keep regexes with sum of counter bounds >20
- \(~\) 8,099 regexes

Other generators:
- RXXR2, RegexCheck, RegexStatic, Rescue
- they target backtracking matchers
Experiments

Dataset:
- 609,992 regexes (GitHub, SNORT, Bro, RegExLib, Microsoft, TrustPort, ...)
- removed unsupported (look-arounds/back-references/...)
- keep regexes with sum of counter bounds >20
- \( \sim 8,099 \) regexes

Other generators:
- RXXR2, RegexCheck, RegexStatic, Rescue
- they target backtracking matchers

\~50 \text{MB input text} \text{ from each generator for each regex} \text{ and try on different matchers}
How many ReDoSes could we generate?

- **ReDoS**: time \textgreater 100 \times \text{longer than average} for matcher on random input
  - results for other ReDoS criteria in the paper
- **GadgetCA**: different strategies for exploring the counting-set automaton
  - **ONELINE**: special strategy to target HYPERSCAN

<table>
<thead>
<tr>
<th>Generators</th>
<th>grep</th>
<th>re2</th>
<th>rust</th>
<th>srm</th>
<th>hyper-scan</th>
<th>ca</th>
<th>ruby</th>
<th>php</th>
<th>perl</th>
<th>python</th>
<th>java</th>
<th>java-Script</th>
<th>.NET</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>COUNTING</strong></td>
<td>1157</td>
<td>1465</td>
<td>1066</td>
<td>279</td>
<td>2</td>
<td>3</td>
<td>1085</td>
<td>796</td>
<td>1252</td>
<td>407</td>
<td>142</td>
<td>140</td>
<td>171</td>
</tr>
<tr>
<td><strong>ONELINE</strong></td>
<td>966</td>
<td>15</td>
<td>57</td>
<td>16</td>
<td>23</td>
<td>0</td>
<td>199</td>
<td>9</td>
<td>208</td>
<td>277</td>
<td>232</td>
<td>228</td>
<td>238</td>
</tr>
<tr>
<td><strong>GREEDY</strong></td>
<td>878</td>
<td>14</td>
<td>57</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>164</td>
<td>9</td>
<td>174</td>
<td>232</td>
<td>190</td>
<td>194</td>
<td>203</td>
</tr>
<tr>
<td><strong>RANDOM</strong></td>
<td>1066</td>
<td>320</td>
<td>292</td>
<td>130</td>
<td>0</td>
<td>0</td>
<td>153</td>
<td>156</td>
<td>266</td>
<td>91</td>
<td>63</td>
<td>60</td>
<td>72</td>
</tr>
<tr>
<td><strong>RXXR2</strong></td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>4</td>
<td>22</td>
<td>8</td>
<td>8</td>
<td>20</td>
</tr>
<tr>
<td><strong>RegexCheck</strong></td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td><strong>RegexStatic</strong></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>
</tr>
<tr>
<td><strong>Rescue</strong></td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>12</td>
<td>2</td>
<td>6</td>
<td>15</td>
<td>7</td>
<td>6</td>
<td>14</td>
</tr>
</tbody>
</table>

(\text{ca}: our matcher based on counting-set automata)
Real-world security solutions

Real-life SNORT rule-sets (Emerging Threats Pro and 3CORESec, Talos)
- filtered out unsupported regexes and those with the sum of repetition bounds $\leq 20$
- obtained 1,112 regexes (from 22,425)
- slowdown of evil vs. random text
Real-world security solutions

Real-life SNORT rule-sets (Emerging Threats Pro and 3CORESec, Talos)
- filtered out unsupported regexes and those with the sum of repetition bounds ≤ 20
- obtained 1,112 regexes (from 22,425)
- slowdown of evil vs. random text

SNORT3@HYPERSCAN
- HYPERSCAN: no cache (modified alg.)
- TCP reassembly off
- MTU 1.5 kB and 9 kB in 100 MB files
Real-world security solutions

Real-life SNORT rule-sets (Emerging Threats Pro and 3CORESec, Talos)
- filtered out unsupported regexes and those with the sum of repetition bounds $\leq 20$
- obtained 1,112 regexes (from 22,425)
- slowdown of evil vs. random text

SNORT3@HYPERSCAN
- HYPERSCAN: no cache (modified alg.)
- TCP reassembly off
- MTU 1.5 kB and 9 kB in 100 MB files

examples

```
"[?&]u=[^&\s]{35}"
"src/s*/x3D(3D)?/s*/[’ ‘'][‘ ’]{{244}}"
"[?&](cmd|pwd|usr)=[^&]{{64}}"
```

<table>
<thead>
<tr>
<th></th>
<th>1.5 kB</th>
<th>9 kB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slowdown</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10-30</td>
<td>65</td>
<td>44</td>
</tr>
<tr>
<td>30-50</td>
<td>44</td>
<td>29</td>
</tr>
<tr>
<td>50-70</td>
<td>29</td>
<td>21</td>
</tr>
<tr>
<td>70-90</td>
<td>21</td>
<td>9</td>
</tr>
<tr>
<td>&gt;90</td>
<td>9</td>
<td>32</td>
</tr>
</tbody>
</table>

NVIDIA BlueField-2

- DPU: data processing unit (ASIC)
- 2 × 25 GbE interfaces, 8 ARM s
- HW-accelerated regex matching unit: ~40 Gbps
- 100 GB files (continuous)
- 617 regexes (from 1,112; unsupported: 495)
NVIDIA BlueField-2

- DPU: data processing unit (ASIC)
- 2 × 25 GbE interfaces, 8 ARM
- HW-accelerated regex matching unit: ~40 Gbps
- 100 GB files (continuous)
- 617 regexes (from 1,112; unsupported: 495)

**examples**

```
"\sPARTIAL.*BODY\.PEEK\[^[^\]]\{1024\}"  # 2,194×
"\s{230},\}\.htr"
"object\s[^>]*type\s*=\s*[^\x22\x27][^\x22\x27]*\x2f\{32\}"
```

**slowdown**

```
2,194×  # 2,194×
956×    # 956×
655×    # 655×
```

Conclusion

- nonbacktracking regex matchers are **NOT** a silver bullet against ReDoS
- they can still be slowed down, often by attacking **counting**, e.g., a \{100\}
- **generator** that can exploit counting — **GadgetCA**
Conclusion

- nonbacktracking regex matchers are **NOT** a silver bullet against ReDoS
- they can still be slowed down, often by attacking counting, e.g., a `{100}`
- generator that can exploit counting — **GadgetCA**
- mitigation:
  - obvious ones (time limit, input limit, disallow counting)
  - overapproximate: $a \{5, 42\} \leadsto a^*$
  - detect **vulnerable** regexes with **GadgetCA**
  - use better regex matching technology (e.g., counting-set automata)
Conclusion

- **nonbacktracking** regex matchers are **NOT** a silver bullet against ReDoS
- they can still be slowed down, often by attacking **counting**, e.g., a \{100\}
- **generator** that can exploit counting — GadgetCA

**mitigation:**
- obvious ones (time limit, input limit, disallow counting)
- overapproximate: a \{5, 42\} \mapsto a \ast
- detect **vulnerable regexes** with GadgetCA
- use **better regex matching technology** (e.g., counting-set automata)

Thank you!
Appendix
<table>
<thead>
<tr>
<th>Generators</th>
<th>grep</th>
<th>re2</th>
<th>rust</th>
<th>srm</th>
<th>hyperscan</th>
<th>ca</th>
<th>ruby</th>
<th>php</th>
<th>perl</th>
<th>python</th>
<th>java</th>
<th>java-Script</th>
<th>.NET</th>
</tr>
</thead>
<tbody>
<tr>
<td>GadgetCA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GREEDY</td>
<td>1741</td>
<td>15</td>
<td>95</td>
<td>18</td>
<td>2</td>
<td>40</td>
<td>260</td>
<td>38</td>
<td>382</td>
<td>367</td>
<td>328</td>
<td>314</td>
<td>431</td>
</tr>
<tr>
<td>COUNTING</td>
<td>2457</td>
<td>742</td>
<td>1016</td>
<td>300</td>
<td>5</td>
<td>67</td>
<td>1355</td>
<td>1596</td>
<td>1473</td>
<td>277</td>
<td>279</td>
<td>258</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>
</tr>
<tr>
<td>ONELINE</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>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RXXR2</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>
</tr>
<tr>
<td>RegexCheck</td>
<td>104</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>7</td>
<td>1</td>
<td>7</td>
<td>11</td>
<td>8</td>
<td>4</td>
<td>14</td>
</tr>
<tr>
<td>RegexStatic</td>
<td>93</td>
<td>1</td>
<td>9</td>
<td>0</td>
<td>1</td>
<td>7</td>
<td>159</td>
<td>50</td>
<td>80</td>
<td>263</td>
<td>253</td>
<td>243</td>
<td>279</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>
</tr>
<tr>
<td>Generators</td>
<td>grep</td>
<td>re2</td>
<td>rust</td>
<td>srm</td>
<td>hyper-scan</td>
<td>ca</td>
<td>ruby</td>
<td>php</td>
<td>perl</td>
<td>python</td>
<td>java</td>
<td>java-Script</td>
<td>.NET</td>
</tr>
<tr>
<td>---------------------</td>
<td>------</td>
<td>-----</td>
<td>------</td>
<td>-----</td>
<td>------------</td>
<td>-----</td>
<td>-------</td>
<td>------</td>
<td>------</td>
<td>--------</td>
<td>------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>GadgetCA</td>
<td>192</td>
<td>72</td>
<td>76</td>
<td>238</td>
<td>0</td>
<td>61</td>
<td>1087</td>
<td>1408</td>
<td>56</td>
<td>200</td>
<td>215</td>
<td>210</td>
<td>390</td>
</tr>
<tr>
<td>GREGGY</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>419</td>
</tr>
<tr>
<td>COUNTING</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>334</td>
</tr>
<tr>
<td>ONE_LINE</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>
</tr>
<tr>
<td>RXXR2</td>
<td>7</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>24</td>
<td>0</td>
<td>4</td>
<td>30</td>
<td>11</td>
<td>11</td>
<td>34</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>1</td>
<td>9</td>
<td>8</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>RegexStatic</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>
</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>
</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>
</tr>
<tr>
<td>Generators</td>
<td>grep</td>
<td>re2</td>
<td>rust</td>
<td>srm</td>
<td>hyper-scan</td>
<td>ca</td>
<td>ruby</td>
<td>php</td>
<td>perl</td>
<td>python</td>
<td>java</td>
<td>java-Script</td>
<td>.NET</td>
</tr>
<tr>
<td>----------------</td>
<td>-------</td>
<td>------</td>
<td>------</td>
<td>-----</td>
<td>------------</td>
<td>------</td>
<td>------</td>
<td>-----</td>
<td>------</td>
<td>--------</td>
<td>------</td>
<td>------------</td>
<td>------</td>
</tr>
<tr>
<td><strong>GREDY</strong></td>
<td>1058</td>
<td>703</td>
<td>274</td>
<td>311</td>
<td>1</td>
<td>135</td>
<td>5050</td>
<td>6580</td>
<td>837</td>
<td>1027</td>
<td>485</td>
<td>955</td>
<td>2629</td>
</tr>
<tr>
<td><strong>COUNTING</strong></td>
<td>1181</td>
<td>1116</td>
<td>295</td>
<td>391</td>
<td>3</td>
<td>121</td>
<td>5440</td>
<td>6289</td>
<td>1294</td>
<td>1503</td>
<td>532</td>
<td>1317</td>
<td>3000</td>
</tr>
<tr>
<td><strong>RANDOM</strong></td>
<td>713</td>
<td>135</td>
<td>259</td>
<td>242</td>
<td>1</td>
<td>106</td>
<td>4405</td>
<td>5389</td>
<td>361</td>
<td>523</td>
<td>385</td>
<td>410</td>
<td>2025</td>
</tr>
<tr>
<td><strong>ONELINE</strong></td>
<td>576</td>
<td>17</td>
<td>78</td>
<td>30</td>
<td>6</td>
<td>130</td>
<td>540</td>
<td>69</td>
<td>402</td>
<td>678</td>
<td>637</td>
<td>485</td>
<td>1448</td>
</tr>
<tr>
<td><strong>RXXR2</strong></td>
<td>11</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>26</td>
<td>0</td>
<td>5</td>
<td>33</td>
<td>12</td>
<td>13</td>
<td>35</td>
</tr>
<tr>
<td><strong>RegexCheck</strong></td>
<td>25</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>7</td>
<td>3</td>
<td>7</td>
<td>18</td>
<td>15</td>
<td>9</td>
<td>36</td>
</tr>
<tr>
<td><strong>RegexStatic</strong></td>
<td>78</td>
<td>1</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>19</td>
<td>182</td>
<td>70</td>
<td>78</td>
<td>287</td>
<td>274</td>
<td>254</td>
<td>333</td>
</tr>
<tr>
<td><strong>Rescue</strong></td>
<td>11</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>24</td>
<td>2</td>
<td>5</td>
<td>26</td>
<td>13</td>
<td>13</td>
<td>28</td>
</tr>
<tr>
<td>random text</td>
<td>153</td>
<td>10</td>
<td>70</td>
<td>27</td>
<td>2</td>
<td>137</td>
<td>175</td>
<td>47</td>
<td>147</td>
<td>272</td>
<td>255</td>
<td>228</td>
<td>698</td>
</tr>
</tbody>
</table>
Counting-set automaton with weights

CSA with weights for the regex "^HOST\x09*[^\x20]\{1000\}"
DFA states explored by our algorithm on the regex "\(^\text{HOST}\backslash x09*[^\x20]\{1000\}"