ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE
ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL
ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

USENIX Security '24 Artifact Appendix: <Towards an Effective Method of ReDoS Detection for Non-backtracking Engines>

Hong Huang[†‡] Rongchen Li[†‡] Weihao Su[†‡] Haiming Chen[†✉] Tingjian Ge[¶]

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

A Artifact Appendix

A.1 Abstract

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

In our paper "Towards an Effective Method of ReDoS Detection for Non-backtracking Engines", we systematically analyze the major causes that could contribute to ReDoS vulnerabilities on non-backtracking regex engines. We then propose a novel type of ReDoS attack strings that builds on the concept of simple strings. Next we propose EVILSTRGEN, a tool for generating attack strings for ReDoS-vulnerable regexes on non-backtracking engines. It is based on a novel incremental determinisation algorithm with heuristic strategies to lazily find the k-simple strings without explicit construction of finite automata. In this artifact, we provide the source code of EVILSTRGEN, the benchmarks used in the paper and the required materials to reproduce our results as described in our paper. This appendix describes how to replicate the experiments as described in §6.2 to §6.5 in our paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

All the experiments are conducted within a Docker container, which will not perform destructive operations or disable security mechanisms to harm the computer.

A.2.2 How to access

The source code and all the experimental materials required can be found on Zenodo: https://doi.org/10.5281/zenodo.12270846. A copy of the implementation is also available at https://github.com/Anonymous89813/EvilStrGen/tree/a07d7a989d9b817524846668712a4b7e038d0226.

A.2.3 Hardware dependencies

We recommend using a machine with 3.40GHz Intel i7-6700 8 CPU, 8GB of RAM and 2TB of disk. However a CPU with higher performance could reproduce similar results in a shorter amount of time.

A.2.4 Software dependencies

Ubuntu 20.04 or an equivalent virtual machine and Docker will suffice.

A.2.5 Benchmarks

The benchmarks used in the experiments are included in the repository named `Benchmarks` as shown in Figure 1. `SET736535` is a large-scale real-world benchmark from various sources, used in the experiments **E1** and **E2** described below. We offered a script as described in **E2** below for selecting all regexes in *sub-classes* mentioned in the paper,

```
Evaluation/
...
├── Benchmarks
    ├──SET736535.txt
    ├──ABOVE20.txt
    ├──Real-World.txt
    ├──Sub-Class
...
```

Figure 1: Repository of benchmarks used for experiments.

from `SET736535`, e.g. counting-free regexes, etc. The script will automatically generate the subsets of benchmarks under `Benchmarks/Sub-Class` folder. `ABOVE20` is the dataset from [1], which GadgetCA is specialized in, and is used in **E3**. The `Real-World` benchmark is used in **E4**, which contains the collection of regexes from Table 8 in our paper.

A.3 Set-up

A.3.1 Installation

After downloading the Docker image, please refer to the "Prerequisites" section of the `README` file in the Zenodo repository to import the image. Since the working directory for subsequent commands is `/Evaluation`, execute the command: `cd /Evaluation` to make sure that your terminal is working in this directory.

A.3.2 Basic Test

We provide a basic test to quickly check all experiment scripts are functioning. To run this basic test, you can execute the command: `./BasicEvaluation.sh`. It takes about 60 minutes to run this command. If the command executes successfully, the results will be output to the `BasicResults` directory and "Successfully" will be printed in the shell.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): EvilStrGen detects more ReDoS vulnerabilities than baselines for non-backtracking regex engines on the large-scale real-world benchmark `SET736535` (§6.2 and Table 5).

(C2): EvilStrGen outperforms GadgetCA in the severity of vulnerabilities detected on the large-scale real-world benchmark `SET736535` (§6.2, Table 5 and Figure 4 (a)-(h)).

(C3): EvilStrGen is able to identify ReDoS vulnerabilities for non-backtracking regex engines on some seemingly safe regex sub-classes from the large-scale real-world benchmark `SET736535` (§6.2 and Table 6).

(C4): EvilStrGen also outperforms GadgetCA on ABOVE20, the benchmark which GadgetCA is specialized in, in terms of the number of vulnerabilities detected and the severity of vulnerabilities (§6.3, Figure 4 (i)-(p) and Table 7).

(C5): Each heuristic strategy improves the effectiveness of candidate attack string generation in EvilStrGen, that the efficiency boost provided by heuristic strategies helps EvilStrGen to detect more ReDoS vulnerabilities, on both `SET736535` and ABOVE20 (§6.4, Table 5 and Table 7).

(C6): EvilStrGen can be used to detect unrevealed ReDoS vulnerabilities when applied to practical programs using non-backtracking engines (§6.5 and Table 8).

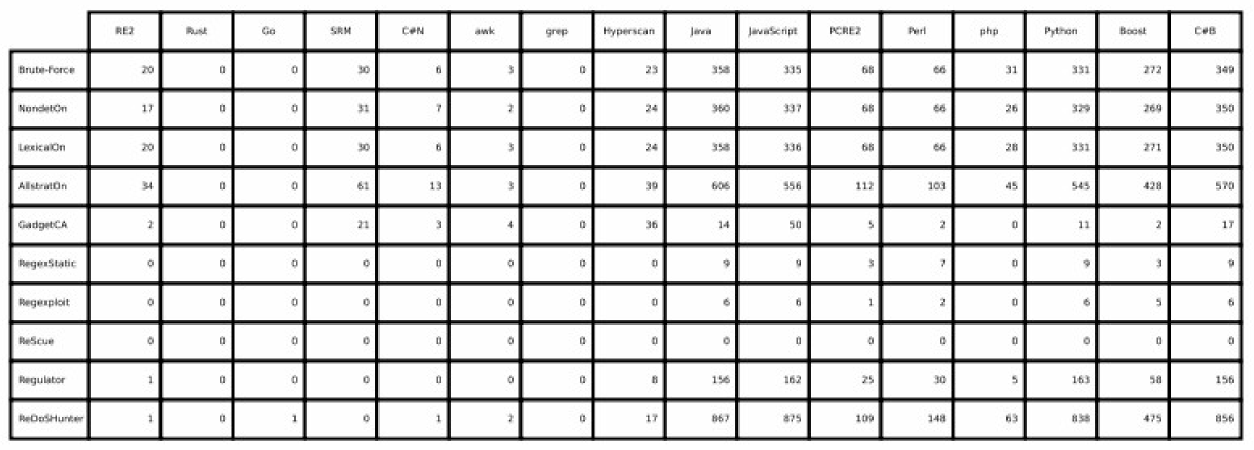
A.4.2 Experiments

We designed four experiments to demonstrate the claims **C1-C6**. Please refer to the section "Evaluating the Artifact" in the `README` file and execute the experiments as listed follows. It takes about 10 days to run all experiments in our hardware dependencies (§A.2.3). If you want to quickly check the functionality of the experiment scripts, you can follow the instructions in §A.3.2.

(E1): *[Large-Scale][5 human-minutes + over 200 compute-hours + 1TB disk]: If **E1** is successfully executed, the results are found in `E1/Table/` and `E1/Figure/`.*
Preparation: Run the command `cd /Evaluation` to enter the working directory.
Execution: Run the command `./E1.sh`.
Results: The output format of the results in `E1/Table/` is shown in Figure 2. **C1** is confirmed by crosschecking the results with Table 5 in the paper. The output format of the results in `E1/Figure/` is shown in Figure 3[1], and **C2** is confirmed by crosschecking the results in Figure 2 with Table 5 in the paper and the results in Figure 3 with Figure 4 (a)-(h) in the paper. The rows 1-4 in Figure 2 also confirm **C5** on the large-scale benchmark.

(E2): *[Sub-Classes][5 human-minutes + 40 compute-hours + 100GB disk]: If **E2** is successfully executed, the results are found in `E2/Table/`.*
Preparation: Run the command `cd /Evaluation` to enter the working directory.
Execution: Run the command `./E2.sh`.
Results: The output format of the results is shown in Figure 4 and **C3** is confirmed by crosschecking the re-

[1]Note that the results presented in `E1/Figure/` are identical in content to those in Figure 4 of the paper. However, for reasons stated in §6.2 of the paper, we have made some visual optimizations to highlight the issues we wish to address. We used hexscatter (https://ww2.mathworks.cn/matlabcentral/fileexchange/45639-hexscatter-m) and FigureBest (https://mbd.pub/o/bread/mbd-YpyUmJlq) to visualize the results as in the paper. The very same plots can be obtained by passing the time results of EvilStrGen and GadgetCA as two arrayes to hexscatter, and using the `adjustment/color mapping` function of FigureBest, if needed.



	RE2	Rust	Go	SRM	C#N	awk	grep	Hyperscan	Java	JavaScript	PCRE2	Perl	php	Python	Boost	C#B
Brute-Force	20	0	0	30	6	3	0	23	358	335	68	66	31	331	272	349
NondetOn	17	0	0	31	7	2	0	24	360	337	68	66	26	329	269	350
LexicalOn	20	0	0	30	6	3	0	24	358	336	68	66	28	331	271	350
AllstratOn	34	0	0	61	13	3	0	39	606	556	112	103	45	545	428	570
GadgetCA	2	0	0	21	3	4	0	36	14	50	5	2	0	11	2	17
RegexStatic	0	0	0	0	0	0	0	0	9	9	3	7	0	9	3	9
Regexploit	0	0	0	0	0	0	0	0	6	6	1	2	0	6	5	6
ReScue	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Regulator	1	0	0	0	0	0	0	8	156	162	25	30	5	163	58	156
ReDoSHunter	1	0	1	0	1	2	0	17	867	875	109	148	63	838	475	856

Figure 2: This figure corresponds to Table 5 in the paper.

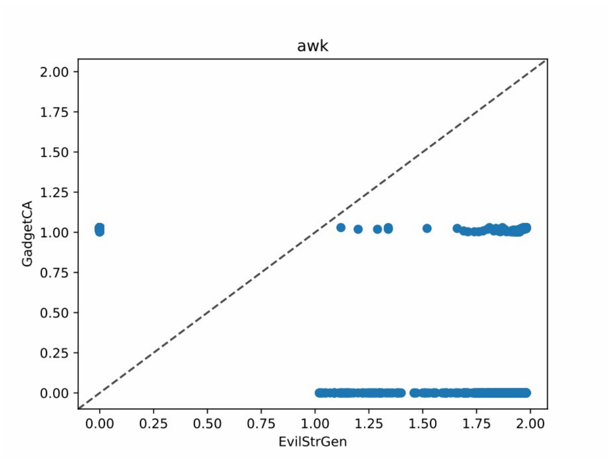
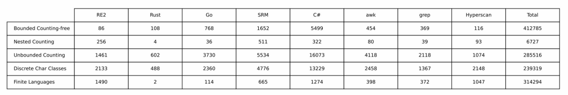
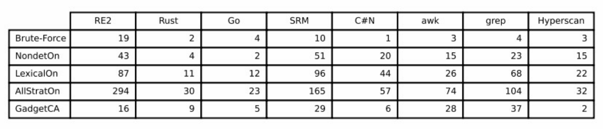


Figure 3: This figure corresponds to the Figure 4(f) in the paper.



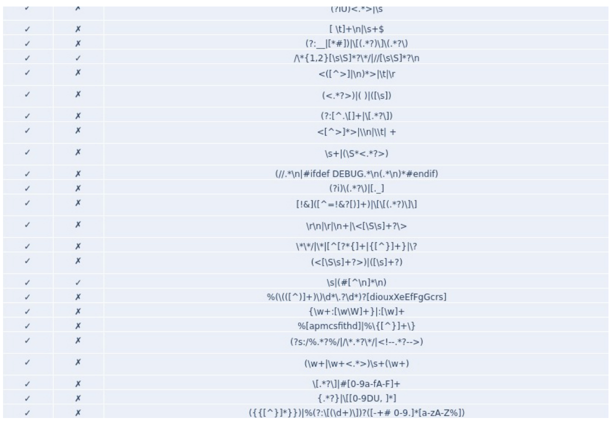
	RE2	Rust	Go	SRM	C#	awk	grep	Hyperscan	Total
Bounded Counting-free	86	108	768	1652	5499	454	369	116	412785
Nested Counting	256	4	36	511	322	80	39	93	6727
Unbounded Counting	1461	602	3730	5534	16073	4118	2118	1074	285516
Discrete Char Classes	2133	488	2360	4776	13229	2458	1367	2148	239319
Finite Languages	1490	2	114	665	1274	398	372	1047	314294

Figure 4: This figure corresponds to Table 6 in the paper.



	RE2	Rust	Go	SRM	C#N	awk	grep	Hyperscan
Brute-Force	19	2	4	10	1	3	4	3
NondetOn	43	4	2	51	20	15	23	15
LexicalOn	87	11	12	96	44	26	68	22
AllStratOn	294	30	23	165	57	74	104	32
GadgetCA	16	9	5	29	6	28	37	2

Figure 5: This figure corresponds to Table 7 in the paper.



✓	✗	(?iU)<.*>|\s
✓	✗	[\t]+\n|\s+$
✓	✗	(?:__|[*#])|\[(.*?)\]\(.*?\)
✓	✓	/*{1,2}[\s\S]*?*/|//[\s\S]*?\n
✓	✗	<([^>]|\n)*>|\t|\r
✓	✗	(<.*?>)|( )|([\s])
✓	✗	(?:[^.\[]+|\[.*?\])
✓	✗	<[^>]*>|\\n|\\t| +
✓	✗	\s+|(\S*<.*?>)
✓	✗	(//.*\n|#ifdef DEBUG.*\n(.*\n)*#endif)
✓	✗	(?i)\(.*?\)|[._]
✓	✗	[!&]([^=!&?[)]+)|\[\[(.*?)\]\]
✓	✗	\r\n|\r|\n+|\<[\S\s]+?\>
✓	✗	**/|*|[^[?*{]+|{[^}]+}|\?
✓	✗	(<[\S\s]+?>)|([\s]+?)
✓	✓	\s|(#[^\n]*\n)
✓	✗	%(\(([^)]+)\))\d*\.?\d*)?[diouxXeEfFgGcrs]
✓	✗	{\w+:[\w\W]+}|:[\w]+
✓	✗	%[apmcsfithd]|%\{[^}]+\}
✓	✗	(?s:/%.*?%/|/*.*?*/|<!--.*?-->)
✓	✗	(\w+|\w+<.*>)\s+(\w+)
✓	✗	\[.*?\]|#[0-9a-fA-F]+
✓	✗	{.*?}|\[[0-9DU,]*]
✓	✗	({{[^}]*}})|%(?:\[(\d+)\])?([-+# 0-9.]*[a-zA-Z%])

Figure 6: This figure corresponds to Table 8 in the paper.

sults with Table 6 in the paper.

(E3): *[ABOVE20][5 human-minutes + 3 compute-hours + 10GB disk]: If **E3** is successfully executed, the results are found in `E3/Table/` and `E3/Figure/`.*

Preparation: Run the command `cd /Evaluation` to enter the working directory.

Execution: Run the command `./E3.sh`.

Results: The output formats of the results in `E3/Table/` and the results in `E3/Figure/` are shown in Figure 5 and Figure 3, respectively. **C4** is confirmed by crosschecking these results with Table 7 and Figure 4 (i)-(p) in the paper. The rows 1-4 in `E3/Table/` confirm **C5** on the ABOVE20 benchmark.

(E4): *[Real-World][5 human-minutes + 1 compute-hour + 25GB disk]: If **E4** is successfully executed, the results are found in `E4/Table/`.*

Preparation: Run the command `cd /Evaluation` to enter the working directory.

Execution: Run the command `./E4.sh`.

Results: The output format of the results is shown in Figure 6 and **C6** is confirmed by crosschecking the results with Table 8 in the paper.

A.5 Notes on Reusability

EVILSTRGEN is designed to uncover ReDoS vulnerabilities for non-backtracking engines, which is commonly used in shell scripts, e.g. grep. To apply EVILSTRGEN to applications using non-backtracking engines, we provide commands to run EVILSTRGEN in Reusability section of `README` file in the Zenodo repository.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at `https://secartifacts.github.io/usenixsec2024/`.

References

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