Re(gEx|DoS)Eval: Evaluating Generated Regular Expressions and their Proneness to DoS Attacks

Mohammed Latif Siddiq
msiddiq3@nd.edu
University of Notre Dame
Notre Dame, IN, USA

Jiahao Zhang∗
jzhang38@nd.edu
University of Notre Dame
Notre Dame, IN, USA

Lindsay Roney
loney@nd.edu
University of Notre Dame
Notre Dame, IN, USA

Joanna C. S. Santos
joannacss@nd.edu
University of Notre Dame
Notre Dame, IN, USA

ABSTRACT

With the recent advances in code generation techniques based on Large Language Models (LLMs), developers are using them for a vast range of tasks, including regex generation. Despite the efforts to generate regexes from natural language, there is no benchmark for LLMs with real-world data and robust test sets. Moreover, a regex can be prone to Denial of Service (DoS) attacks due to catastrophic backtracking. Hence, we need a systematic evaluation process to evaluate the correctness and security of the regexes generated by the language models. In this paper, we describe Re(gEx|DoS)Eval, a framework that includes a dataset of 762 regex descriptions (prompts) from real users, refined prompts with examples, and a robust set of tests. We introduce the pass@k and vulnerable@k metrics to evaluate the generated regexes based on the functional correctness and proneness of ReDoS attacks. Moreover, we demonstrate the Re(gEx|DoS)Eval with three LLMs (T5, Phi, and GPT-3) and describe the future plans to extend this framework.

CCS CONCEPTS

• Software and its engineering → State based definitions; • Security and privacy → Denial-of-service attacks.

KEYWORDS

Regex Generation, ReDoS, DoS Attack, Evaluation, Dataset

∗J. Zhang was a visiting student at ND from SUSTech (Southern University of Science and Technology) when this research was performed.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICSE-NIER’24, April 14–20, 2024, Lisbon, Portugal
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0500-7/24/04 . . $15.00
https://doi.org/10.1145/3639476.3639757

1 INTRODUCTION

With the recent release of GitHub Copilot [1] and ChatGPT [2], software engineers are increasingly relying on these AI assistants to develop software. In fact, a recent survey with 500 US-based developers who work for large-sized companies showed that 92% of them are using AI coding tools both for work and personal use [27]. Part of this fast widespread adoption is due to the increased productivity perceived by developers; AI helps them automate repetitive tasks to focus on higher-level challenging tasks [38].

One common repetitive task faced by developers is to write regular expressions ("regexes") to search strings or validate inputs. With the popularity of AI assistants, developers are using them to generate their regexes. Although these tools can help programmers in writing regexes, the produced expressions may be not only incorrect but also prone to denial-of-service attacks (i.e., Regex Denial-of-Service - ReDoS) [10, 28]. A ReDoS attack exploits certain expression patterns that can lead to catastrophic backtracking, causing operations to take a long time to complete. Given the ubiquity of regexes, especially for input validation, these vulnerabilities can cripple a software system’s performance when triggered [22].

Although prior works studied ReDoS attacks [10, 22] and whether AI assistants generate functionally correct code [8, 9, 24], we currently lack an in-depth understanding of how well AI assistants can generate regular expressions that are correct and safe against ReDoS. The literature also lacks a benchmark dataset for systematically evaluating generated regexes both in terms of their correctness and their proneness to ReDoS attacks. Existing datasets used for natural language to regex generation [16, 21] are synthetic datasets, i.e., they do not simulate the real-world users’ needs, which can have more complex descriptions [37].
Therefore, this paper aims to address these research gaps by developing Re(gEx|DoS)Eval, a framework to systematically evaluate the security and correctness of generated regular expressions. At the core of Re(gEx|DoS)Eval, is a novel benchmark dataset of real regex-related problems along with a set of robust test cases that are used to evaluate a regular expression’s correctness. To measure a regex’s security, this framework relies on state-of-the-art techniques [19, 28] to generate exploits and use those to evaluate the expression’s proneness to ReDoS attacks.

The key contributions of this paper are: (i) a framework for evaluating generated regexes, (ii) a novel metric (vulnerable|k) that measures a regex’s proneness to ReDoS attacks, and (iii) a preliminary empirical study that uses our framework to investigate the regexes generated by four models. All the scripts and datasets are released on GitHub: https://github.com/s2e-lab/RegexEval.

2 BACKGROUND AND RELATED WORK


Although regexes are widely used, prior works [10, 22] have shown the potential for denial-of-service attacks (i.e., ReDoS attacks). A ReDoS attack occurs when an attacker exploits the exponential time complexity inherent in certain regex patterns, causing the regex engine to enter a state of catastrophic backtracking [19, 28], resulting in denial of service or severe performance degradation. To illustrate, consider the regex /A(��|८)+/D/ that matches a string that starts with ‘A’, ends with ‘D’, and in the middle there can be at least one ‘B’ or multiple ‘C’s. This expression is prone to ReDoS attacks when encountering inputs similar to “ACCCCCC...CXX” [4]. This is caused by the nested quantifiers, which have the potential to trigger catastrophic backtracking.

Prior works aimed to detect vulnerable regexes using static and dynamic analysis. RXKR2 [26] transformed regex to the power DFA (DeterministicFinite Automata) and searched attack strings on that. Rexploter [33] detects a vulnerable regex by combining complexity analysis of Nondeterministic Finite Automata (NFA) with sanitization-aware taint analysis. ReSue [28] uses a genetic algorithm on a targeted regex to develop expansive seed strings and create a malicious string by using pumping and a search algorithm. ReDoSHunter [19] used static and dynamic analysis to locate all possible vulnerabilities and generate malicious strings.

3 RE(gEX|DoS)Eval OVERVIEW

AI code assistants are evaluated using benchmarks [9, 29, 34] which do not include constructs to evaluate the correctness and security of generated regular expressions. Thus, this paper describes the creation and evaluation of Re(gEX|DoS)Eval, a framework for verifying and benchmarking the performance of Large Language Models (LLMs) with respect to generating regular expressions. As shown in Fig. 1, our framework is created in three major steps.

3.1 Dataset Collection

We mined (on Aug. 16th, 2023) all the regexes from RegExLib, a regular expression library [5]. We use this library because it contains user-contributed regular expressions. We obtained from RegExLib a list of 4,128 regular expressions along with their id, description, and list of expected matches and non-match strings.

3.2 Manual Validation

For each sample previously collected, we perform a manual validation to (1) filter out incorrect regexes, (2) create more sample test cases (i.e., matching and non-matching string examples), and (3) create refined problem descriptions (i.e., prompts).

3.2.1 Filtering RegEx Samples. We excluded any regex that matched one or more of the following conditions: (i) it was missing any metadata i.e., description, and/or list of expected matches and non-matches; (ii) its description is not written in English; (iii) its description included vulgar words; (iv) its description does not provide sufficient information to understand the purpose of the regular expression; (v) it aimed to detect just one word; (vi) it is incorrect (i.e., the regex matches a string that is not supposed to match, or it does not match a string that is expected to match). After this step, we have 1,001 regex samples.

3.2.2 Creating New Test Cases and Refined Prompts. Each collected regex sample had (on average) only 4 string examples (2 that are expected matches and 2 that are expected non-matches). Thus, we manually crafted additional test cases to ensure that each sample has at least 13 matching1 and 12 non-matching string examples. After creating these additional test strings, we evaluate the regex with the new set of test cases again and excluded the failed regex samples. Hence, we have 762 samples in our final dataset.

3.2.3 Prompt Creation. Upon further inspection of the descriptions in the extracted samples, we observed that some of them lacked a more detailed explanation (e.g., ID#84: “SQL date format tester.”) or had extra information unrelated to the regex (e.g., ID#4: “... Other than that, this is just a really really long description of a regular expression that I’m using to test how my front page will look in the case where very long expression descriptions are used”). Thus, we created a refined prompt with a clear description of the regex and that includes three match and two non-match string examples, as shown in the JSON snippet in Figure 2.

3.3 Systematic Model Evaluation

To systematically evaluate the correctness and security of a regex produced by an LLM, our framework has an evaluation environment and an automated assessment component.

1We could not create at least 13 matches for problems that had a strict matching pattern (e.g., a regex that matches only one digit has only 10 match examples).
How well can LLMs generate regular expressions?

Prior works [21, 35] measured the correctness of generated regexes. For example, if a user asks an LLM to generate a regex that detects IPv4 addresses, then the LLM may generate a regex that matches both IPv4 and IPv6 addresses. This means that the generated regex is correct but will not match the DFA or regex in the reference solution. Hence, we need a metric for evaluating the functional correctness of the generated regex that measures the correctness with respect to passing test cases (i.e., an execution-based measurement). Chen et al. [9] introduced the \texttt{pass@k} metric to evaluate the functional correctness of a generated code. Thus, we use a similar approach to evaluate the correctness of a generated regex. Specifically, to compute the \texttt{pass@k}, we instruct the model to generate k regexes for each problem. If at least one of the regexes passes all test cases, then the model succeeded in solving the problem. The \texttt{pass@k} measures the fraction of problems solved by the model.

\section*{Measuring Security:} We introduce \texttt{vulnerable@k}, a new metric to measure the security of a generated regex. We consider a problem to be vulnerable if any of the top-k generated regexes is prone to ReDos attacks. A regex is considered prone to attacks if either Rescue [28] or ReDoSHunter [19] generated one or malicious strings. The model is better if the \texttt{vulnerable@k} is lower.

For \texttt{pass@k} and \texttt{vulnerable@k}, we consider the problem to be functionally correct or vulnerable if any of the top-k regexes pass all the tests or is vulnerable, respectively. However, computing these metrics in this way can have high variance [9, 15]. Instead, to evaluate \texttt{pass@k} and \texttt{vulnerable@k}, we generate n \geq k samples per problem, count the number of correct samples \( c \leq n \) that is functionally correct or vulnerable, and calculate the unbiased estimator from Kulal et al. [15]:

\[ \text{metric@k} := \mathbb{E}_{\text{problems}} \left[ 1 - \frac{n-c}{(k)} \right] \]

Since calculating this estimator directly results in large numbers and numerical instability, we used a numerically stable implementation from Chen et al. [9] that simplifies the expression and evaluates the product term-by-term. Notice that we did not calculate the DFA-EQ@k with the unbiased estimator to keep the metric as described in the previous literature [16].

\section*{EXPERIMENTS & EARLY RESULTS}

We used Re(Ex)(DoSE)Eval to conduct a preliminary empirical study to answer two research questions:

\begin{itemize}
  \item \textbf{RQ1} \textit{How well can LLMs generate regular expressions?} In this question, we study the correctness of the regex generated by existing LLMs.
  \item \textbf{RQ2} \textit{Are LLM-generated regexes prone to ReDoS attacks?} In this RQ, we investigate to what extent the regexes generated by LLMs are prone to ReDoS attacks.
\end{itemize}

We used four models from three LLM families to answer the RQs:

- \textbf{Fine-tuned T5} [25]: The Text-to-Text transformer (T5) [25] is an attention-based transformer model [32]. We used the T5-base
model (220 million parameters) fine-tuned with a popular synthetic dataset of regular expressions (KB13 [16]).

- **Pre-trained Phi-1.5** [18]: It is a language model equipped with 1.3 billion parameters, primarily trained on a highly refined synthetic “textbook-quality” dataset.

- **Generative Pre-trained Model (GPT-3)** [7]: It is a transformer-based [32] and task-agnostic model capable of both understanding and generating natural language. We used the Text-DaVinci-003 (an upgraded version for text generation) and GPT-3.5-Turbo (which is tuned for chat-style conversation and powers a popular chat-based question-answering tool, ChatGPT [2]). The June 2023 version of the GPT-3.5-Turbo was used in this work.

For the fine-tuned T5 model, we gave as input both the original prompt and the refined prompt for each of the 762 problems in our dataset. However, since the GPT-3 models are task-agnostic, we added an instruction after the description to make it clear that we want the model to generate a regex for the described problem (i.e., “<problem description here>. Generate a regex for this description”).

Since the phi-1.5 model is expecting either a Q&A-style, chat-style, or code-style prompt, we prefix the prompt with a slightly modified similar instruction (i.e., ending with a full-stop) as the GPT-3 models and add in the end the following text “Answer”, to make it a question-answering prompt. For the Phi-1.5 model, we had to clear the output after generating the regex because this model generated an explanation after the regex delimited by `\n\n`.

We set the temperature to 0.8, as prior works [8, 9] have shown that it ensures variation in the output for the same prompts while achieving good performance. We kept the other parameters with their default values and instructed all the models to generate the best regexes. It is important to highlight that ReDoSHunter performs better than the ReScue tool.

### Table 1: RegEx generation correctness performance (RQ1).

<table>
<thead>
<tr>
<th></th>
<th>Fine-tuned</th>
<th>Pre-trained</th>
<th>Text-DaVinci-003</th>
<th>GPT-3.5-Turbo</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Original</td>
<td>Refined</td>
<td>Original</td>
<td>Refined</td>
</tr>
<tr>
<td><strong>Unparseable</strong></td>
<td>62.68%</td>
<td>59.68%</td>
<td>45.48%</td>
<td>34.99%</td>
</tr>
<tr>
<td><strong>EM (%)</strong></td>
<td>0%</td>
<td>0%</td>
<td>1.8%</td>
<td>16.7%</td>
</tr>
<tr>
<td><strong>DFA-EQ@1</strong></td>
<td>0.3%</td>
<td>0.4%</td>
<td>1.7%</td>
<td>10.1%</td>
</tr>
<tr>
<td><strong>DFA-EQ@3</strong></td>
<td>0.4%</td>
<td>0.2%</td>
<td>2.4%</td>
<td>11.3%</td>
</tr>
<tr>
<td><strong>DFA-EQ@10</strong></td>
<td>1.1%</td>
<td>0.13%</td>
<td>6.3%</td>
<td>15.4%</td>
</tr>
<tr>
<td><strong>pass@1</strong></td>
<td>0.1%</td>
<td>0.03%</td>
<td>2.3%</td>
<td>17.6%</td>
</tr>
<tr>
<td><strong>pass@3</strong></td>
<td>0.1%</td>
<td>0.1%</td>
<td>5.4%</td>
<td>26.7%</td>
</tr>
<tr>
<td><strong>pass@10</strong></td>
<td>0.1%</td>
<td>0.3%</td>
<td>10.9%</td>
<td>35.4%</td>
</tr>
</tbody>
</table>

### Table 2: Proneness to ReDoS attacks (RQ2).

<table>
<thead>
<tr>
<th></th>
<th>Fine-tuned</th>
<th>Pre-trained</th>
<th>Text-DaVinci-003</th>
<th>GPT-3.5-Turbo</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Original</td>
<td>Refined</td>
<td>Original</td>
<td>Refined</td>
</tr>
<tr>
<td><strong>ReScue</strong></td>
<td>44.02%</td>
<td>47.81%</td>
<td>6.25%</td>
<td>3.64%</td>
</tr>
<tr>
<td><strong>ReDoSHunter</strong></td>
<td>0.17%</td>
<td>0.12%</td>
<td>0.07%</td>
<td>0.54%</td>
</tr>
<tr>
<td><strong>vul@1</strong></td>
<td>0.17%</td>
<td>0.12%</td>
<td>0.06%</td>
<td>0.54%</td>
</tr>
<tr>
<td><strong>vul@3</strong></td>
<td>0.82%</td>
<td>0.53%</td>
<td>0.33%</td>
<td>1.77%</td>
</tr>
<tr>
<td><strong>vul@10</strong></td>
<td>1.57%</td>
<td>0.91%</td>
<td>0.66%</td>
<td>2.76%</td>
</tr>
<tr>
<td><strong>vul@25</strong></td>
<td>1.44%</td>
<td>0.82%</td>
<td>1.71%</td>
<td>2.76%</td>
</tr>
</tbody>
</table>

### 5 CONCLUSION & FUTURE PLANS

Generating a regular expression from descriptions can be useful for automating tasks, such as data validation, string search etc. With the recent development of the LLMs, users can automate regex generation. In this paper, we described Rs( egret2) vs, a systematic approach to benchmark the LLMs for this particular task and their proneness to the ReDoS vulnerability. In the future, we aim to make a more robust, explainable, extendable framework for regex generation benchmarking using language models.

**Future Plans.** LLMs are trained on a vast amount of public data, making them susceptible to data contamination issues [11]. Indeed, when looking at Table 1, we observe that (surprisingly) GPT-3.5-turbo generated more correct regexes when it was given the original prompt instead of the manually curated detailed prompt. This could be an early indication that the model mined these regexes from the RegexLib and memorized the answer. Thus, we will investigate the mitigation from a prior work [11] for this type of issue while extending our framework. To mitigate this, we will release it publicly in a binary format as suggested by Jacovi et al. [14].

To extend this work, we also plan to collect more regex problems from StackOverflow, by finding questions where users are struggling with regexes that are taking too long to finish (a common symptom of ReDoS). Subsequently, we will study more recent LLMs (e.g., LLAMA [31] and investigate how inference parameters (e.g., temperature, maximum generation length etc.), and different prompting styles as the previous literature [23, 30] are contributing factors that lead to more correct/secure generated regexes.
REFERENCES