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

Mohammed Latif Siddiq, Jiahao Zhang, Lindsay Roney, and Joanna C. S. Santos

{msiddiq3,jzhang38,lroney,joannacss}@nd.edu

Department of Computer Science, University of Notre Dame

Notre Dame, IN, USA

ABSTRACT

With the recent development of large language model-based text and code generation technologies, users are using them for a vast range of tasks, including regex generation. Despite the efforts to generate regexes from natural language, there is no prompt 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 short 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 large language model families *i.e.*, T5, Phi, and GPT-3, and describe the plan for the future extension of 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

ACM Reference Format:

Mohammed Latif Siddiq, Jiahao Zhang, Lindsay Roney, and Joanna C. S. Santos. 2023. Re(gEx|DoS)Eval: Evaluating Generated Regular Expressions and their Proneness to DoS Attacks. In *Proceedings of the 46th International Conference on Software Engineering, NIER Track (ICSE-NIER'24).* ACM, New York, NY, USA, 5 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

With the recent release of GitHub Copilot [1] and ChatGPT [2], software engineers increasingly rely 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 [28]. 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 [39].

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 relying on them to generate their regular expressions. Although these tools can help programmers in writing regexes, the produced expressions may be not only textitincorrect but also prone to *denial-of-service attacks* (*i.e.*, Regex Denial-of-Service - ReDoS) [10, 29]. 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 application performance and render systems unresponsive when triggered [23].

Although existing literature studied ReDoS attacks [10, 23] and whether AI assistants generate functionally correct code [8, 9, 25], 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 [17, 22] are *synthetic* datasets, *i.e.*, they do not simulate the real-world users' needs, which can have more complex descriptions [38].

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 [20, 29] to generate exploits and use those to evaluate the expression's proneness to ReDoS attacks.

The key contributions of this short paper are: *(i)* a framework for evaluating generated regex, *(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 here: https://github.com/s2e-lab/RegexEval.

ICSE-NIER'24, April 14–20, 2024, Lisbon, Portugal
© 2023 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

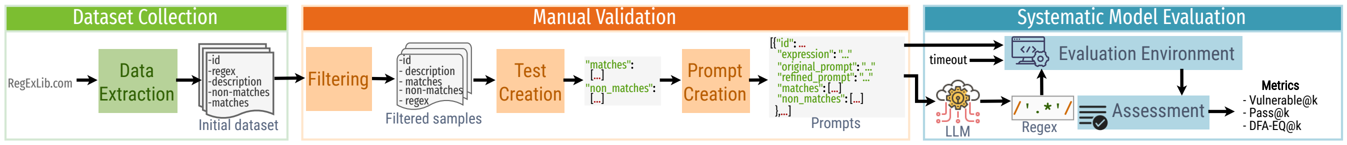

Figure 1: Overview of our Framework RE(GEX|DOS)EVAL

2 BACKGROUND AND RELATED WORK

A **Regular expression** (**RegEx**) is a pattern that can be used to search, replace, or match strings within text [15]. Given its usefulness, prior works focused on automated regex generation from examples or natural language descriptions. For example, AlphaRegex [18] generated regexes by pruning out a large search space based on approximation. RegexGenerator++ [6] used a genetic algorithm to generate a regex. Kushman and Barzilay [17] used semantic unification to generate a regex from natural language. Deep-Regex [22] used an LSTM-based [12] approach with a new comparatively large synthetic dataset. SemRegex [37] used reinforcement learning with Deep-Regex [22]. TransRegex [21] improves the policy gradient by ensuring the validity of generated regex by rewarding the model for generating valid regex. InfeRe [36] used the chain of inference by considering generating a regex from a description as a sequential cognitive process.

Although regexes are widely used, prior research has shown the potential for denial-of-service attacks conducted through regular expressions (*i.e.*, ReDoS attacks) [10, 23]. 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*** [20, 29], resulting in denial of service or severe performance degradation.

To illustrate, consider the regex `/A(B|C+)+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 `"ACCCCCCC...CX"` [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. RXXR2 [27] transformed regex to the power DFA and searched attack strings on that. Rexploiter [34] detects a vulnerable regex by combining complexity analysis of Nondeterministic Finite Automata (NFA) with sanitization-aware taint analysis. ReScue [29] 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 [20] used static analysis to locate all possible vulnerabilities and generate corresponding string triggers. The possible vulnerabilities were then assigned time complexities and a dynamic analysis was performed on the identified vulnerabilities to verify their risk.

3 RE(GEX|DOS)EVAL OVERVIEW

AI code assistants are evaluated using benchmarks [9, 30, 35] which do not include constructs to evaluate the correctness and security of regular expressions within the generated code. 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 matching[1] 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 sample, 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.

[1]We 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).

```
{"id": 879,
 "expression":"^([1-9]{0,1})([0-9]{1})(\\.[0-9])?$",
 "original_prompt":"Matches numbers 0 through 99.9. Allows only one preceding zero and does not
 require the decimal point",
 "refined_prompt":"Matches numbers in the format of a single digit, with an optional leading di-
 git in the range 1-9 and an optional decimal part consisting of a period followed by one digit.
 - Match examples: \"1\", \"1.1\", \"0.1\"
 - Non-match examples: \"01\", \"01.1\"",
 "matches":["1", "1.1", "0.1", "4.8", "6.5", "1.4", "8.4", "9.8", "3.2", ...],
 "non_matches":["01", "01.1", "0.10", "000151", "0051156", "0.215413", ...] }
```

Figure 2: Example of a RegEx Problem from our Dataset

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.

3.3.1 Evaluation Environment. Our framework provides the runtime environment to compile and test the generated regexes. This environment is composed of a Python script that takes as input both the regexes generated by an LLM and the list of strings that are expected to match/not match for the regexes. This script first ***compiles*** each generated regex using `re` library [3]. If the compilation is successful, it ***tests*** the generated regex with a user-defined *timeout* using the strings from the list of *matches* and *non-matches*. Moreover, this script also runs two state-of-the-art techniques (Rescue [29] and ReDoSHunter [20]) to generate strings that trigger catastrophic backtracking. If these tools successfully generate a malicious string, that means the generated regex is prone to ReDos attacks. Thus, for each generated regex, the output of our evaluation environment component is a list of *failed/passed tests* and the *malicious strings* generated by Rescue [29] and/or ReDoSHunter (if any).

3.3.2 Regex Assessment. This component computes three metrics to measure the *correctness* and one metric for *security* of the generated regexes.

►**Measuring Correctness:** Prior works [22, 36] measured the correctness of a generated regex based on *(i)* the ratio of exact matching (EM) to the reference solution and *(ii)* the equivalence between the regex's and the solution's deterministic finite automaton (DFA-EQ). Thus, this component computes these metrics as follows:

–**Exact Match (EM)**: It measures how many generated regexes are equal to the solution in our dataset (*e.g.*, if 15 out of 20 generated regexes are equal to their reference solution, then EM=75%).

–**DFA-EQ@k**: Two regexes are semantically equivalent if they have the same minimal DFA [13]. Thus, the DFA-EQ@k measures the percentage of instances in which there is *at least one* regex that is semantically equivalent to the ground truth among the top k-produced regexes. To clarify, consider that we have 10 problems and a model produces 10 regexes for each of these problems. If there are 6 problems for which at least one minimal DFA matched within the top 5 solutions, then the DFA-EQ@5 score will be 60%.

Although the EM and DFA-EQ@k metrics were widely used in prior works, they cannot account for the large and complex space of regexes that are functionally equivalent to a reference solution [38]. For example, if a user asks an LLM to generate a regex that detects an IP address (but does not clarify their hidden intention is to detect 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 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 **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 *pass@k* measures the fraction of problems solved by the model.

►**Measuring Security:** We introduce a new metric (**vulnerable@k**) 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 [29] or ReDoSHunter [20] is able to generate one or malicious strings. For this metric, the model is better if the score is lower.

For *pass@k* and *vulnerable@k*, we consider the problem to be functionally correct or vulnerable if any of the top-k regexes pass all the tests or are vulnerable, respectively. However, computing these metrics in this way can have high variance [9, 16]. Instead, to evaluate *pass@k* and *vulnerable@k*, we generate n ≥ k samples per task, count the number of correct samples $c \leq n$ that is functionally correct or vulnerable, and calculate the unbiased estimator from Kulal *et al.* [16]: $metric@k := \mathbb{E}_{problems}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right]$.

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 [17].

4 EXPERIMENTS & EARLY RESULTS

We used our framework to conduct a preliminary study to answer two research questions:

RQ1 ***How well can LLMs generate regular expressions?*** In this question, we study the *correctness* of the regex generated by existing LLMs.

RQ2 ***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.

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

– **Fine-tuned T5** [26]: The Text-to-Text transformer (T5) [26] is an attention-based transformer model [33]. We used the T5-base model (220 million parameters) fine-tuned with a popular synthetic dataset (KB13 [17]).

– **Pre-trained Phi-1.5** [19]: The Phi-1.5 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 [33] 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 (This instruction was "*<prompt description here>. Generate a regex for this description:*") after the prompt to make it clear that we want the model to generate a regex for the described problem. Since the ***phi-1.5 model*** is expecting either a Q&A-style, chat-style, or code-style prompt, we prefix the prompt with the 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 billion model, we had to clear the data 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 **10** regular expressions with a maximum of 128 new tokens for each of the 762 problems in our framework. After generating the regexes, we used our approach's evaluation environment to compile and test all the expressions. Consistent with prior works [20, 36], we used a 60-second timeout. In our experiment, we use $k = 1, 3$ and 10 to compute the **DFA-EQ@K**, **PASS@K**, and **VULNERABLE@K**.

4.1 RQ1: RegEx Generation Performance

Table 1: RegEx generation performance (RQ1).

	Fine-tuned T5		Pre-trained Phi-1.5		Text-DaVinci-003		GPT-3.5-Turbo	
	Original	Refined	Original	Refined	Original	Refined	Original	Refined
Unparseable	62.68%	59.68%	24.58%	34.99%	14.57%	14.73%	11.55%	11.71%
EM (%)	0%	0%	1.1%	1.8%	16.7%	23.4%	33.9%	33.1%
DFA-EQ@1	0.3%	0.0%	1.4%	0.0%	7.6%	10.1%	11.8%	10.4%
DFA-EQ@3	0.4%	0.0%	2.4%	2.4%	11.3%	14.3%	15.5%	15.0%
DFA-EQ@10	1.1%	0.13%	6.3%	5.6%	15.4%	17.2%	19.2%	18.8%
pass@1	0.1%	0.03%	2.3%	1.7%	17.6%	27.0%	33.4%	33.6%
pass@3	0.1%	0.1%	5.4%	4.3%	26.7%	37.4%	44.5%	44.4%
pass@10	0.1%	0.3%	10.9%	9.7%	35.4%	46.3%	52.5%	52.9%

Table 1 shows the percentage of regexes that were unparsable out of 7,620 (= 762 problems × 10 outputs) regexes generated by each model. The GPT-3 models (Text-DaVinci-003 and GPT-3.5-Turbo) generated more compilable and successful regexes from the perspective of passing the test cases. It is also noticeable that the refined prompts improve the performance of the Text-DaVinci-003 model, whereas, for the GPT-3.5-Turbo, the performance is similar for both types of prompts. This phenomenon is the opposite in the Phi-1.5 model *i.e.*, the original prompts are performing *better* than the refined prompts. Another thing is that the value of the DFA-EQ@k metric is lower than the pass@k metric. That indicates that despite not having semantic similarity, different regexes can solve problems described by natural language.

4.2 RQ2: Proneness to ReDoS

Table 2 summarizes the results. The first two rows indicate the percentage of vulnerable regexes identified by each tool out of 7,620 regexes. The next three rows indicate the ***vulnerable@k*** score from ReScue [29], and the following three rows present the ***vulnerable@k*** score from ReDoSHunter [20]. We found that the fine-tuned T5 not only has a lower performance based on correctness but it also generates more ReDoS-vulnerable regexes. Phi-1.5 and both GPT-3 models produce a similar number of ReDoS vulnerable regexes. It is important to highlight that ReDoSHunter performs better than the ReScue tool.

Table 2: Proneness to ReDoS attacks (RQ2)

	Fine-Tuned T5		Pre-trained Phi-1.5		Text-DaVinci -003		GPT-3.5-Turbo	
	Original	Refined	Original	Refined	Original	Refined	Original	Refined
ReScue	0.17%	0.12%	0.16%	0.07%	0.54%	0.28%	0.28%	0.22%
ReDoSHunter	44.02%	47.81%	6.25%	5.12%	3.64%	3.56%	3.50%	3.19%
vul@1$_{Rescue}$	0.17%	0.12%	0.16%	0.06%	0.54%	0.28%	0.28%	0.22%
vul@3$_{Rescue}$	0.82%	0.53%	0.79%	0.33%	1.77%	1.09%	1.14%	1.00%
vul@10$_{Rescue}$	1.57%	0.91%	1.57%	0.66%	2.76%	1.71%	1.84%	1.71%
vul@1$_{ReDoSHunter}$	44.23%	47.81%	6.25%	5.12%	3.64%	3.56%	3.50%	3.19%
vul@3$_{ReDoSHunter}$	67.52%	72.53%	16.42%	13.14%	6.90%	5.88%	6.20%	5.73%
vul@10$_{ReDoSHunter}$	84.78%	87.53%	38.45%	30.31%	11.68%	9.58%	9.84%	9.19%

5 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 [32] and investigate how inference parameters (*e.g.*, temperature, maximum generation length *etc.*), and different prompting styles as the previous literature [24, 31] are contributing factors that lead to more correct/secure generated regexes.

6 CONCLUSION

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 Re(gEx|DoS)Eval, 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.

REFERENCES

[1] 2022. GitHub Copilot. Accessed Dec 7, 2022. https://github.com/features/copilot

[2] 2023. Chat completions. Accessed Mar 25, 2023. https://platform.openai.com/docs/guides/chat

[3] 2023. re — Regular expression operations. https://docs.python.org/3/library/re.html [Online; accessed 11. Sep. 2023].

[4] 2023. ReDoS | Tutorials & Examples | Snyk Learn. https://learn.snyk.io/lesson/redos [Online; accessed 15. Sep. 2023].

[5] 2023. Regular Expression Library. https://regexlib.com [Online; accessed 11. Sep. 2023].

[6] Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. 2016. Inference of Regular Expressions for Text Extraction from Examples. *IEEE Transactions on Knowledge and Data Engineering* 28, 5 (2016), 1217–1230.

[7] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 1877–1901.

[8] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023. CodeT: Code Generation with Generated Tests. In *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=ktrw68Cmu9c

[9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]

[10] James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. 2018. The Impact of Regular Expression Denial of Service (ReDoS) in Practice: An Empirical Study at the Ecosystem Scale. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, 246–256.

[11] Jamie Hayes and Olga Ohrimenko. 2018. Contamination attacks and mitigation in multi-party machine learning. *Advances in neural information processing systems* 31 (2018).

[12] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-term Memory. *Neural computation* 9 (12 1997), 1735–80.

[13] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. 2001. Introduction to automata theory, languages, and computation. *Acm Sigact News* 32, 1 (2001), 60–65.

[14] Alon Jacovi, Avi Caciularu, Omer Goldman, and Yoav Goldberg. 2023. Stop uploading test data in plain text: Practical strategies for mitigating data contamination by evaluation benchmarks. *arXiv preprint arXiv:2305.10160* (2023).

[15] SC Kleene. 1951. Representation of Events in Nerve Nets and Finite Automata. *Automata Studies: Annals of Mathematics Studies. Number 34* 34 (1951), 3.

[16] Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. 2019. SPoC: Search-based Pseudocode to Code. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc.

[17] Nate Kushman and Regina Barzilay. 2013. Using semantic unification to generate regular expressions from natural language. North American Chapter of the Association for Computational Linguistics (NAACL).

[18] Mina Lee, Sunbeom So, and Hakjoo Oh. 2016. Synthesizing Regular Expressions from Examples for Introductory Automata Assignments. *SIGPLAN Not.* 52, 3 (oct 2016), 70–80.

[19] Yuanzhi Li, Sébastien Bubeck, Ronen Eldan, Allie Del Giorno, Suriya Gunasekar, and Yin Tat Lee. 2023. Textbooks Are All You Need II: phi-1.5 technical report. arXiv:2309.05463 [cs.CL]

[20] Yeting Li, Zixuan Chen, Jialun Cao, Zhiwu Xu, Qiancheng Peng, Haiming Chen, Liyuan Chen, and Shing-Chi Cheung. 2021. ReDoSHunter: A Combined Static and Dynamic Approach for Regular Expression DoS Detection. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 3847–3864.

[21] Yeting Li, Shuaimin Li, Zhiwu Xu, Jialun Cao, Zixuan Chen, Yun Hu, Haiming Chen, and Shing-Chi Cheung. 2021. TransRegex: Multi-Modal Regular Expression Synthesis by Generate-and-Repair. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 1210–1222.

[22] Nicholas Locascio, Karthik Narasimhan, Eduardo DeLeon, Nate Kushman, and Regina Barzilay. 2016. Neural Generation of Regular Expressions from Natural Language with Minimal Domain Knowledge. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics (ACL), Austin, Texas, 1918–1923.

[23] Louis G. Michael, James Donohue, James C. Davis, Dongyoon Lee, and Francisco Servant. 2019. Regexes are Hard: Decision-Making, Difficulties, and Risks in Programming Regular Expressions. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 415–426.

[24] Noor Nashid, Mifta Sintaha, and Ali Mesbah. 2023. Retrieval-Based Prompt Selection for Code-Related Few-Shot Learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2450–2462.

[25] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. *ICLR* (2023).

[26] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research* 21, 140 (2020), 1–67. http://jmlr.org/papers/v21/20-074.html

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

[28] Inbal Shani. 2023. Survey reveals AI's impact on the developer experience | The GitHub Blog. *GitHub Blog* (June 2023). https://github.blog/2023-06-13-survey-reveals-ais-impact-on-the-developer-experience/#methodology

[29] Yuju Shen, Yanyan Jiang, Chang Xu, Ping Yu, Xiaoxing Ma, and Jian Lu. 2018. ReScue: crafting regular expression DoS attacks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 225–235.

[30] Mohammed Latif Siddiq and Joanna C. S. Santos. 2022. SecurityEval Dataset: Mining Vulnerability Examples to Evaluate Machine Learning-Based Code Generation Techniques. In *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security* (Singapore, Singapore) *(MSR4P&S 2022)*. Association for Computing Machinery, New York, NY, USA, 29–33. https://doi.org/10.1145/3549035.3561184

[31] Mohammed Latif Siddiq, Joanna C. S. Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinicius Carvalho Lopes. 2023. Exploring the Effectiveness of Large Language Models in Generating Unit Tests. arXiv:2305.00418 [cs.SE]

[32] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL]

[33] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

[34] Valentin Wüstholz, Oswaldo Olivo, Marijn JH Heule, and Isil Dillig. 2017. Static detection of DoS vulnerabilities in programs that use regular expressions. In *Tools and Algorithms for the Construction and Analysis of Systems: 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II 23*. Springer, 3–20.

[35] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Tao Xie, and Qianxiang Wang. 2023. CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pre-trained Models. *arXiv preprint arXiv:2302.00288* (2023).

[36] Shuai Zhang, Xiaodong Gu, Yuting Chen, and Beijun Shen. 2023. InfeRE: Step-by-Step Regex Generation via Chain of Inference.

[37] Zexuan Zhong, Jiaqi Guo, Wei Yang, Jian Peng, Tao Xie, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2018. SemRegex: A Semantics-Based Approach for Generating Regular Expressions from Natural Language Specifications. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Brussels, Belgium, 1608–1618.

[38] Zexuan Zhong, Jiaqi Guo, Wei Yang, Tao Xie, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2018. Generating regular expressions from natural language specifications: Are we there yet?. In *AAAI Workshops*. 791–794.

[39] Albert Ziegler, Eirini Kalliamvakou, X. Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. 2022. Productivity Assessment of Neural Code Completion. In *Proc. of the 6th ACM SIGPLAN Int'l Symposium on Machine Programming* (San Diego, CA, USA) *(MAPS 2022)*. ACM, New York, NY, USA, 21–29. https://doi.org/10.1145/3520312.3534864