Understanding Regular Expression Denial of Service (ReDoS): Insights from LLM-Generated Regexes and Developer Forums

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

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

ABSTRACT

Regular expression Denial of Service (ReDoS) represents an algorithmic complexity attack that exploits the processing of regular expressions (regexes) to produce a denial-of-service attack. This attack manifests when regex evaluation time scales polynomially or exponentially with input length, posing sporadic yet significant challenges for software developers. The advent of Large Language Models (LLMs) has revolutionized the generation of regexes from natural language prompts, but not without its risks. Prior works showed that LLMs can generate code with vulnerabilities and security smells. In this paper, we synthesized a vast collection of regex patterns from a comprehensive dataset, assessing their correctness and ReDoS vulnerability. We investigated the characteristics of these vulnerable regexes, categorizing them into equivalence classes to unravel their weaknesses. Our study also examined ReDoS patterns in actual software projects, aligning them with corresponding regex classes. Moreover, we analyzed developer dialogues on GitHub and StackOverflow, constructing a taxonomy to investigate their experiences and perspectives on ReDoS. In this study, we found that GPT-3.5 was the best LLM to generate regexes that are both correct and secure. We observed that LLM-generated regexes mainly have polynomial ReDoS vulnerability patterns, and it is consistent with the real-world data. We also found that developers' main concern is related to mitigation strategies to remove vulnerable regexes.

CCS CONCEPTS

• **Software and its engineering** → *State based definitions*; • **Security and privacy** → **Denial-of-service attacks**; • **Computing methodologies** → Multi-task learning.

KEYWORDS

ReDoS, DoS Attack, large language models, regex generation

ICPC 2024, April 2024, Lisbon, Portugal

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:

Mohammed Latif Siddiq*, Jiahao Zhang*, and Joanna C. S. Santos. 2024. Understanding Regular Expression Denial of Service (ReDoS): Insights from LLM-Generated Regexes and Developer Forums. In *Proceedings of 32nd International Conference on Program Comprehension (ICPC 2024).* ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Regular expressions ("*regexes*") feature in 30-40% of software projects for tasks like pattern matching [16, 22]. Although regexes are useful for various tasks, they can be a source of denial-of-service attacks (ReDoS) caused by *catastrophic backtracking* [22]. This vulnerability is caused by a regex engine taking too long to process an input string, making the software system unresponsive [20, 57]. For example, the exploitation of a ReDoS vulnerability caused service disruption at StackOverflow in 2016 [10].

While a non-vulnerable regex can be hard to maintain and have readability issues [16, 25, 36, 56], ReDoS-vulnerable regexes can be more problematic to understand, and that makes it hard to detect and repair them by the developers [33]. The complexity of ReDoS-vulnerable regexes further exacerbates this issue, as many developers are not fully aware of the associated security risks. In fact, a recent study showed that only 38% of all surveyed developers knew about regular expression denial of service [36].

Although prior works focused on studying [22, 36], detecting [31, 50], or repairing [32] regular expression denial-of-service (ReDoS) vulnerabilities, we are at the verge of a **new threat**: with the recent release of GitHub Copilot [1] and ChatGPT [2], the reliance on AI assistants for software development is concerning, particularly because they can generate *insecure* code [42, 44, 52, 54, 64, 65]. 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 [49]. Part of this fast widespread adoption is due to the increased productivity perceived by developers [43, 72].

As code generation tools become widely used during software development, there is a risk that developers will blindly trust the output of the AI assistants, which may inadvertently introduce ReDoS-vulnerable regexes in a software system. Since developers are unaware of the security risks of regexes [36], they may focus on checking whether the regex is *functionally correct* (*i.e.*, passes

*Both authors contributed equally to this work

the test cases) but be oblivious to the fact the generated regex is vulnerable to denial-of-service attacks.

In light of this threat, in our work, we investigate to what extent LLMs are able to generate regexes that are both *functionally correct* and *secure*. We also study how ReDoS-vulnerable regexes are hard to understand, how similar LLM-generated regexes are to real-world vulnerabilities, and what types of concerns & questions developers have related to ReDoS. To our knowledge, this is the first study to focus on ReDoS comprehension from a dual perspective: the efficacy of LLMs in generating regex patterns and their susceptibility to ReDoS attacks, alongside the developers' understanding and discussions of these vulnerabilities.

The contributions of this paper are:

- We investigated **274,320** regexes[1] generated using a combination of prompt types, model temperatures, and samples, evaluating them for correctness and proneness to ReDoS attacks (**RQ1**).
- We studied the pattern classification of the ReDoS-vulnerable regexes detected by Li *et al.* [31] and manually analyzed them, mapping each to its corresponding regex class, as defined in Chapman *et al.* [17], for better understanding (**RQ2**).
- We studied ReDoS from real-world software systems and analyzed their corresponding ReDoS patterns and regex classes (**RQ3**) in order to contrast whether LLM-generated vulnerable regexes have similar characteristics to developer-written ones.
- We mined around 500 GitHub Pull Requests and StackOverflow posts about ReDoS to understand what concerns developers have related to regular expressions. We created a taxonomy based on the developers' discussion about ReDoS in the open forum (**RQ4**).
- A replication package with all the scripts used to gather the data and compile all the results[2].

2 BACKGROUND

2.1 Insecure Regular Expressions

Some regex patterns combined with specific input strings can cause what the literature named as ***catastrophic backtracking*** [22]. This issue arises when the underlying engine performing the string matching takes a long time to determine whether the string will match the patterns specified in the regex. The root cause for the execution delay is the extensive permutations and combinations the regex engine must check to determine a match.

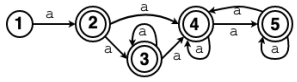

Figure 1: NFA for the regex `^(a+)+$`

For example, consider the regex `^(a+)+$`, whose Nondeterministic Finite Automaton (NFA) is shown in Figure 1. For the input `'aaaaX'`, the NFA has 16 potential pathways [66]. However, for the input `'aaaaaaaaaaaaaaaX'`, the number of potential pathways is 65,536 (it doubles for each additional `'a'`). When an input does not match, the engine retreats to earlier junctures at which it could opt for an alternate route. The engine persistently attempts this until it has exhausted all possible pathways. This causes very high CPU usage, leading to denial of service [4].

ReDoS Patterns. There are five common patterns that represent a different way in which a regex can be exploited to cause service disruption [31]. These patterns are: **Nested Quantifiers (NQ)** (*i.e.*, patterns with quantifiers placed within other quantifiers, *e.g.*, `^(a+)+$`), **Exponential Overlapping Disjunction (EOD)** (*i.e.*, patterns with alternatives in a regex that share common substrings, *e.g.*, `^(a+|a?)+$`), **Exponential Overlapping Adjacency (EOA)** (*i.e.*, patterns with adjacent elements that can be matched in multiple overlapping ways due to a shared quantifier, *e.g.*, `^(a+)+b$`), **Polynomial Overlapping Adjacency (POA)** (*i.e.*, patterns where adjacent elements share an optional quantifier, *e.g.*, `^(a?)+a{100}$`), and **Starting with Large Quantifier (SLQ)** (*i.e.*, patterns beginning with a large quantifier that forces the regex engine to try many starting positions in the input string, *e.g.*, `^a{100,}b`).

2.2 Regex Equivalence Class Groups

Regex Equivalence Class Groups provide a systematic approach to categorize regexes based on their behavior [16]. These groups are formed by classifying behaviorally equivalent regexes into the same class. This behavioral equivalence means that while the regexes may have different syntax, they fulfill the same function in terms of pattern matching. As shown in Table 1, regexes can be classified into five main equivalence class groups based on their functional characteristics: **Custom Character Class Group (CCC)**, **Double-Bounded Group (DBB)**, **Literal Group (LIT)**, **Lower-Bounded Group (LWB)**, and **Single-Bounded Group (SNG)**. Understanding these equivalence classes is not merely academic; it has practical implications in writing and optimizing regexes. By recognizing that certain complex patterns can be simplified into more straightforward, equivalent forms, developers can write more readable and maintainable code. Furthermore, identifying commonly used patterns through equivalence classes can inform better practices, as these are likely to be more recognizable to other programmers and, thus, more easily understood [17].

Table 1: Regex Equivalence Classes [17]

CCC	**C1**	Patterns with a range in a custom character class (*e.g.*, `[2-6]`).
	C2	Patterns with custom character classes without shorthand notation, *e.g.*, `[abc]`.
	C3	Patterns employing negation within custom character classes, *e.g.*, `[^eiu]`.
	C4	Patterns that use default character classes within custom classes, *e.g.*, `[\w\s]`.
	C5	Patterns that uses *OR* sequences as custom classes, converting `(x|y|z)` into `[xyz]`.
DBB	**D1**	Patterns that define a non-equal range of repetitions, *e.g.*, `h{2,5}`.
	D2	Patterns that use ? to denote an optional character, which is effectively a double-bound of zero or one occurrence (*e.g.*, `https?`).
	D3	Patterns that express bounded repetitions through ORs, *e.g.*, `(b|bb|bbb)`.
LIT	**T1**	Patterns featuring at least one literal character, *e.g.*, `abc`.
	T2	Patterns that use hexadecimal tokens, *e.g.*, `\x41\x42\x43` instead of "ABC".
	T3	Patterns that use literal characters within brackets to obviate the need for escape characters, *e.g.*, `[\.]`.
	T4	Patterns using octal tokens, *e.g.*, `\101\102\103` to match "ABC"
LWB	**L1**	Patterns using the curly brace notation for specifying only a lower bound, *e.g.*, `z{3,}`.
	L2	Patterns that include the Kleene star (*e.g.*, `9*`), which denotes zero or more repetitions.
	L3	Patterns that use the plus sign to indicate a minimum of one occurrence, *e.g.*, `1+` to match a string of '1's.
SNG	**S1**	Patterns with a singular repetition boundary, *e.g.*, `d{4}`.
	S2	Patterns that are explicitly repeated, transformable to `xy{3}` from `xyxyxy`.
	S3	Patterns with identical upper and lower bounds, with `a{2,2}` reducing to `'aa'`.

[1] 2 Prompt types × 6 Temperature settings × 3 LLMs × 762 Prompts × 10 Outputs
[2] https://anonymous.4open.science/r/redos-study

3 METHODOLOGY

In this study, we answer four research questions:

RQ1 ***To what extent LLMs can generate non-vulnerable and correct regexes?***

While prior studies investigated the capabilities of code generation models in terms of functional correctness and security [37, 39, 42, 54, 55], these studies do not investigate the generation of (insecure) regular expressions. Therefore, to fill in this research gap, this work includes an empirical investigation of three code generation models (GPT3.5, fine-tuned T5 and Phi-1.5) to verify how well they can generate secure and correct regexes.

RQ2 ***What are the characteristics of LLM-generated ReDoS vulnerabilities?***

Chapman *et al.* [17] developed a taxonomy of *regex classes* (Table 1) and studied their understandability by developers. We extend this study for LLM-generated RegExes and, more specifically, for the ReDoS vulnerable regexes. If regular expressions are more understandable [17], then developers might have a better comprehension of potential ReDoS vulnerabilities. In RQ2, we investigate the characteristics of the generated vulnerable regexes with respect to their equivalence class and ReDoS pattern type.

RQ3 ***What are the characteristics of real-world ReDoS vulnerabilities that were fixed?***

In this question, we investigate whether the distribution of features is similar between LLM-generated regexes and developer-written vulnerable regexes that were fixed.

RQ4 ***How are ReDoS discussed by developers?***

While understanding the intrinsic factors of LLMs that may lead to ReDoS vulnerabilities is crucial, it is equally important to comprehend the external perspective: how the wider developer community perceives, understands, and discusses ReDoS issues [59]. Thus, we investigate the most common questions or concerns raised by developers related to ReDoS posted in community forums. Specifically, we analyzed questions posted on StackOverflow[3], a popular Q&A platform, and on pull requests on GitHub repositories.

3.1 RQ1: ReDoS generation

During *inference* (*i.e.*, code generation), an LLM is provided with inputs (*prompts*) and configured with inference parameters (*e.g.*, *temperature*). Each of these can be a *contributing factor* to (insecure) regex generation. Thus, this work conducts an ablation study to relate correct/insecure regexes to *prompts* and *inference parameters*. In this study, each factor is investigated in *isolation* using three LLMs: **GPT-3.5-Turbo** [5], **T5** [45], and **Phi-1.5** [30]. We chose these LLMs because each of them has been used by several prior works [27, 41, 51, 61, 71] and is representative of *decoder-only* (or GPT-style) and *encoder-decoder* (or Seq2Seq) models. To conduct this study, we first created a dataset of prompts (§ 3.1.1), used three LLMs to generate regexes with different inference parameters (§ 3.1.2), and computed evaluation metrics to correlate prompts and inference parameters to insecure regex generation (§ 3.1.3).

[3]https://stackoverflow.com/

3.1.1 Creating a dataset of prompts. We needed a *dataset of prompts* since existing benchmarks [18, 28, 35] do not evaluate the correctness and security of regular expressions within the generated code. To create this dataset, we retrieved *all* the **4,128** regular expressions available on the RegExLib website [8], along with their *unique identifier*, *description*, and *test cases* (*i.e.*, a list of strings that are expected to *match* the regex, and strings that are *not* expected to match). We use this library because it contains user-contributed regular expressions, and it has been used by prior works [29, 70]. Subsequently, we perform a manual validation of each collected sample to **(1)** filter out *incorrect* regexes, **(2)** create more test cases (*i.e.*, matching and non-matching string examples), and **(3)** create *refined* problem descriptions (*i.e.*, *prompts*).

(1) Filtering Regex Samples. We disregarded any retrieved sample that matched one or more of these conditions: **(i)** it was missing any metadata *i.e.*, description, and/or list of expected *matches* and *non-matches*; **(ii)** its description is not 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 had **1,001** regex samples.

(2) Creating New Test Cases. 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 created additional test cases to ensure that each prompt had at least 13 matching and 12 non-matching string tests[4]. We aimed to have a total of 25 input/output pair examples (13 positive, 12 negative), where we could use 5 of them to be part of the prompt. This way, we increased the average number of examples from 4 to 25, significantly contributing to the test cases' robustness. After creating these additional test strings, we evaluated the regex with the new set of test cases again and excluded the failed regex samples, obtaining a total of **762** samples in our final dataset.

(3) Refined Prompt Creation. We observed that some samples lacked a more detailed explanation (*e.g.*, ID#84: *"SQL date format tester."*) or had unrelated information for generating 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.

These steps were conducted by two of the authors (2-3 years of experience), and peer-reviewed by a third author (over 10 years of experience). Less than 2% prompts had disagreements and were adjusted through discussion. Figure 2 shows an example of a prompt in our dataset.

3.1.2 Regex Generation. We used three models to answer the RQs. The **Text-to-Text transformer (T5)** [45] is an attention-based encoder-decoder transformer model [63]. In our work, we had a T5-base model (220 million parameters) fine-tuned with a popular synthetic dataset for regex generation (KB13 [28]). The **Pre-trained**

[4]We could not create at least 13 matches for problems with a strict matching pattern (*e.g.*, a regex that matches only a vowel character that will have only 5 match examples)

```
{"id": 161,
 "expression":"[A-Z][a-z]+",
 "original_prompt": "This expression was developed to match the Title cased words within
 a Camel cased variable name. So it will match 'First' and 'Name' within 'strFirstName'.",
 "refined_prompt": "This regular expression matches two or more consecutive letters in a string,
 where the first letter is uppercase (A-Z) and the subsequent letters are lowercase (a-z).
 - Match examples: \"strFirstName\", \"intAgeInYears\", \"Where the Wild Things Are\"
 - Non-match examples: \"123\", \"abc\"",
 "matches":["strFirstName", "intAgeInYears", "Where the Wild Things Are", ...],
 "non_matches":["123", "abc", ...] }
```


Figure 2: Example of a regex prompt from our dataset

Phi-1.5 [30] is a language model with 1.3 billion parameters, primarily trained on a highly refined synthetic "textbook-quality" dataset. The **Generative Pre-trained Model (GPT-3)** [15] is a task-agnostic model capable of both understanding and generating natural language. We used the ***GPT-3.5-Turbo*** released on June 2023, which is tuned for chat-style conversation and powers a popular chat-based question-answering tool, ChatGPT [2]. Unlike the T5 and Phi-1.5 models, this model is closed source (*i.e.*, data source, model weight *etc.* are unknown).

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. Since the ***phi-1.5 model*** is expecting either a Q&A-style, chat-style, or code-style prompt and *GPT-3 models* are task-agnostic, we added an instruction *after* the prompt to make it clear that we want the model to generate a regex for the described problem. This instruction was "*<prompt description here>. Generate a regex for this description:*". In addition to this, for the ***phi-1.5 model***, we added, at the end of the previous instruction text "*Answer:*", to make it a question-answering prompt. For the Phi-1.5 billion model, we also had to clear the data after generating the regex because this model generated an explanation after the regex (delimited by `\n\n`). Hence, we removed any text after getting the `\n\n` delimiter.

We run each model with two prompt types (*i.e.*, the *original* description collected from RegExLib and the *refined* version). In this ablation study, we vary the temperature from 0 to 1, in 0.2 increments (*i.e.*, 0.0, 0.2, ..., 1.0). We kept the other parameters with their default values and instructed all the models to generate **10** regular expressions with a maximum of **128** tokens for each of the 762 problems in our dataset. We used the Huggingface interface to generate regexes using the fine-tuned T5 & pre-trained Phi-1.5 and the OpenAI API for the GPT-3 model.

Managing LLM's instability: LLMs are non-deterministic by design [15, 30]. However, the output variability can be controlled using the *temperature* inference hyperparameter [5, 15]. Lower temperatures yield more consistent responses, while higher temperatures increase the diversity of the outputs. To mitigate the threats introduced by this intrinsic non-determinism in LLMs, we followed the guidelines outlined by Sallou *et al.* [47]: **(1)** we conducted a thorough analysis across a spectrum of temperature settings, ranging from `0` to `1` in increments of `0.2`; **(2)** At each temperature level, we generate multiple instances (10 regex patterns) to identify patterns of consistent output, and **(3)** we make all prompts and corresponding outputs publicly accessible in our replication package.

3.1.3 Evaluation Metrics. We compile and test all the generated regexes by using the test cases from our dataset. Consistent with prior works [31, 69], we used a 60-second timeout when testing the regexes. Subsequently, we calculated the following metrics to measure the ***functional correctness*** of the generated regexes:

- **`Exact Match (EM)`**: It measures how many generated regexes are equal to the solution in our dataset. For example, if 15 out 20 regexes are equal to the reference solution, then **EM**=75%.
- **`DFA-EQ@k`**: Two regexes are semantically equivalent if they have the same minimal Deterministic Finite Automaton (DFA) [26]. 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. For instance, consider that we have 10 problems, and a model produces 10 regexes for each. 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%. To compute the DFA-EQ@k we used an implementation from Park *et al.* [40].
- **`pass@k`**: It measures the success rate of finding the correct regex within the top k options [18]. A regex is considered *correct* if the generated regex passes *all* the test cases in the dataset.

While the above metrics measure whether a regex solves the problem described in the prompt, they do not evaluate whether the models produce regexes prone to ReDoS attacks. Thus, we compute the `vulnerable@k` metric to quantify the security of the generated regexes [54], which is defined as follows:

– **`vulnerable@k`**: It measures the probability of finding an insecure regex within the top k results [18]. We calculated the `vulnerable@k` using the source code from Siddiq *et al.* [54].

In our experiment, we use $k = 1, 3$ and 10 to compute the **`DFA-EQ@k`**, **`pass@k`**, and **`vulnerable@k`**.

3.2 RQ2: ReDoS Characteristics

In this question, we study the characteristics of the vulnerable regexes that were generated. We study these from a ***computational complexity*** perspective and a ***regex comprehension*** perspective.

3.2.1 ReDoS Computational Complexity. As explained in Section 2, insecure regexes lead to denial-of-service due to catastrophic backtracking. The time it takes for the engine to complete depends on the computational complexity of the vulnerable regex. For example, *nested quantifiers* (*e.g.*, `(a*)*`) and quantifying a disjunction (*e.g.*, `(a|a)*`) can be ***exponentially*** dangerous, while *concatenated quantifiers* (*e.g.*, `abc.*def.*`) can be ***polynomially*** dangerous. Thus, we investigate the computational complexity of the generated vulnerable regexes. To perform this analysis, we run ReDoSHunter [31] (with default configuration settings) for all the generated regexes for the prompts and inference parameters settings in RQ1. ReDoSHunter not only detects whether a regex is prone to ReDoS attacks, but it also reports the computational complexity type of the regex which are: *Nested Quantifiers* (NQ), *Exponential Overlapping Disjunction* (EOD), *Exponential Overlapping Adjacency* (EOA), *Polynomial Overlapping Adjacency* (POA), and *Starting with Large Quantifier* (SLQ). We report the computational complexity type distribution per model, temperature, and prompt setting.

3.2.2 Regex Classification. To explore how these regular expressions include patterns that are easier/more difficult for developers to understand, we classify each vulnerable regex according to the classification scheme described by Chapman *et al.* [17]. This taxonomy identifies regex patterns that are easier/more difficult for developers to understand. It has five major categories consisting of sub-categories, as described in Table 1.

To investigate the understandability of the generated regexes, we perform a stratified random sampling of the generated vulnerable regexes as follows. We first partition the generated insecure regexes into two sets: those regexes that are derived from the *original* prompts (S1) and those that are from the *refined* prompts (S2). Next, we randomly select a statistically significant sample size for each partition (95% confidence level and 5% margin of error). When randomly sampling, we keep the proportion to the total number of insecure regexes for a specific model and temperature combination. For example, if GPT-3.5-Turbo with temperature setting 0.6 has 100 insecure regexes, whereas GPT-3.5-Turbo with temperature setting 0.4 has 50 insecure regexes, we select twice as many samples from temperature setting 0.6 than temperature setting 0.4.

For each selected sample, we manually classify them based on the equivalent (sub-)classes described by Chapman *et al.* [17]. This classification is performed by one of the authors, who has over three years of professional software development experience. We presented the result from three perspectives: (i) prompt type (*original* and *refined*), (ii) computational complexity type (*exponential* and *polynomial*), and (iii) model (T5, Phi-1.5, and GPT-3.5).

3.3 RQ3: Analyzing real-world ReDoS

We used the dataset from Li *et al.* [32] that contains **448** vulnerabilities caused by using an insecure regular expression. The dataset consists of two sources: SOLA-DA is from Staicu, and Pradel [57], which has 34 samples, and another one is from 70 real-world vulnerabilities (CVEs [3]). One CVE can contain multiple vulnerable regexes. Li *et al.* [32] extracted 414 ReDoS-vulnerable regexes. Similar to RQ2, in this question, we first run ReDoSHunter [31] to find the ReDoS Computational Complexity. Then, we manually analyzed them to classify them based on the equivalent (sub-) classes described by Chapman *et al.* [17].

After classifying them to the corresponding ReDoS patterns and regex classes, we did the Mann–Whitney U test [38, 46] to test for the following hypotheses: (1) the Null hypothesis (H_0) is that the two populations are equal and (2) the alternative hypothesis (H_1) is that the two populations are not equal. We used these non-parametric tests for significance testing as data from this and the previous RQ consists of counts (non-normally distributed) for different categories (*i.e.*, ReDoS patterns and Regex classes), and the sample sizes differ between the two sources. We pair-wise compared the distribution of ReDoS patterns from different models, prompt type, and temperature combination with the dataset from Li *et al.* [32]. For regex classes, we compared the distribution of vulnerability types from both datasets as only this perspective is consistent in both results. We considered a two-tailed test (*i.e.*, the critical area of a distribution is two-sided and tests whether a sample is greater than or less than a certain range of value) and significant level, $\alpha = 0.05$ for the Mann–Whitney U test.

3.4 RQ4: Developers' Concerns

To answer RQ4, we used the Stack Overflow API [9] to search for all posts that contained the words "regex" and "ReDoS" (case-insensitive search). This search resulted in **151** posts. Similarly, we used the GitHub API [9] to retrieve all the pull requests that match all the following conditions **(a)** it contains the strings "regex" and "redos" (case insensitive); **(b)** the pull request has been *closed* and **(c)** it is not done by a bot[11]. With this search, we obtained a total of **1729** pull requests from GitHub. For each retrieved Stack Overflow post and GitHub pull request, we conducted a manual analysis to identify posts that are indeed discussing regex-related security risks by developers (true positives). Next, we obtained a total of **48** and **426** true positives from Stack Overflow posts and GitHub pull requests, respectively.

Subsequently, we performed open coding for all the **474** samples. During this open coding process, we analyzed each post and pull request in order to annotate them with concepts (codes). This coding was performed by the authors of this paper, whose software development experience level varied from 3-10 years. After reviewing the information in the post/pull request, we collaboratively highlighted the key points. We constantly refined the concepts throughout the open coding process, leading to the emergence of categories, which group these concepts based on common themes. The outcome of this question is a taxonomy of developer concerns with respect to the security risks of regular expressions.

4 RESULTS

4.1 RQ1: Regex Generation

We evaluated the generated regexes with respect to its ***functional correctness*** (§ 4.1.1) and ***security*** (§ 4.1.2).

4.1.1 Functional Correctness.

Exact Match (EM). Figure 3 shows the exact match (EM) distribution for different temperatures, prompt types, and LLMs. We found that no regex generated from the FINE-TUNED T5 has an exact match with the ground truth. GPT-3.5 has the highest exact match (an average of **24%** and **35.8%** for the original and refined prompts, respectively). The PHI-1.5 model had its EM ranging from **0.4%** to **2.8%** (**1.9%** average) and **0.4%** to **5.2%** (**2.6%** average) for the original and refined prompts, respectively. On average, the refined prompts lead to more incidence of exact matches for PHI-1.5 and GPT-3.5. With respect to temperature performance, there is no clear best performer. For PHI-1.5, temperatures 0 and 0.4 were the best performers for the refined and original prompt, respectively. For GPT-3.5, the best performers for the refined and original prompt, respectively, were temperatures 0.4 and 0.8.

DFA-EQ@k. Figure 4 shows the DFA-EQ@k for k equals 1, 3, and 10. GPT-3.5 performs better than the other LLMs (an average of **11%** and **13.8%** for the original and refined prompts, respectively) and the refined prompts have slightly better performance than the original ones. The PHI-1.5 model is the second best performing one, with a DFA-EQ@k with an average **3.3%** and **2.9%** for the original

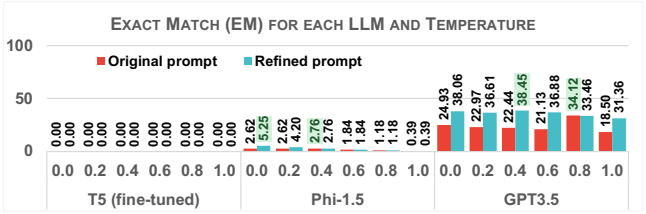

Figure 3: Exact Match (EM) for each LLM and temperature.

and refined prompts, respectively. While temperatures 1.0 and 0.8 were the best performing ones for the original prompt for the FINE-TUNED T5 and GPT-3.5 models, the same trend was not observed for the refined prompts.

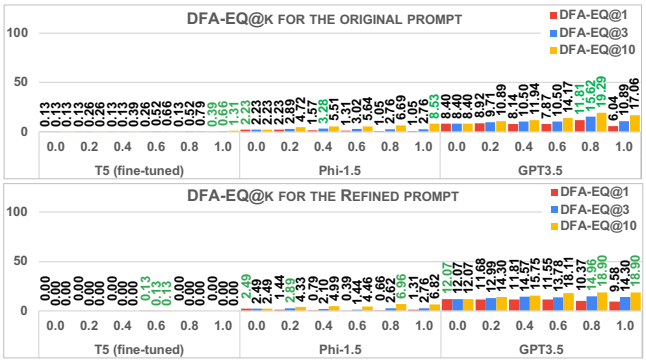

Figure 4: `DFA-EQ@k` for the original and refined prompts

pass@k. Figure 5 shows the `pass@1`, `pass@5`, and `pass@10` for all LLMs. Once again, GPT-3.5 is the best-performing LLM. Its performance is better with the temperature increase and the refined prompts perform better than the original prompts (average of 32% *vs.* 41%). The FINE-TUNED T5 is the worst performing model, with an average of 0.1% `pass@k` for the original and refined prompts.

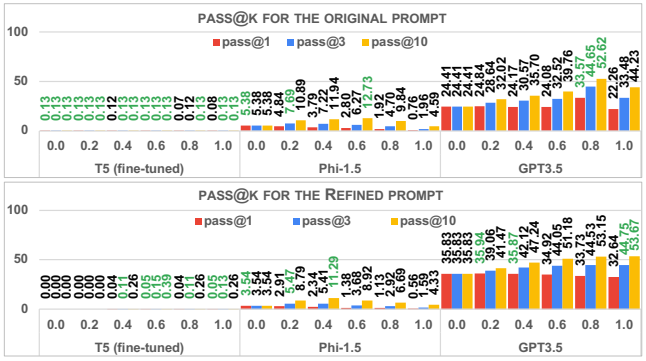

Figure 5: `pass@k` for the original and refined prompts

4.1.2 ReDoS Generation. Figure 6 shows the `vulnerable@k` (for k equals to 1, 3, and 10) for each LLM in different temperature settings. Recall that, for this metric, the best model is one that has the *lowest* `vulnerable@k`. We found that the FINE-TUNED T5 model generates *more* vulnerable regexes than the other two models, whereas GPT-3.5 generates *less* vulnerable regexes. When we increase the temperature, the percentage of the ReDoS vulnerable regex increases for the top-3 and top-10 results (`vulnerable@1` and `vulnerable@10`). For GPT-3.5, refined prompts lead to *less* vulnerable regexes than the original prompts. We can see the same phenomena for PHI-1.5 but the opposite in the FINE-TUNED T5 model.

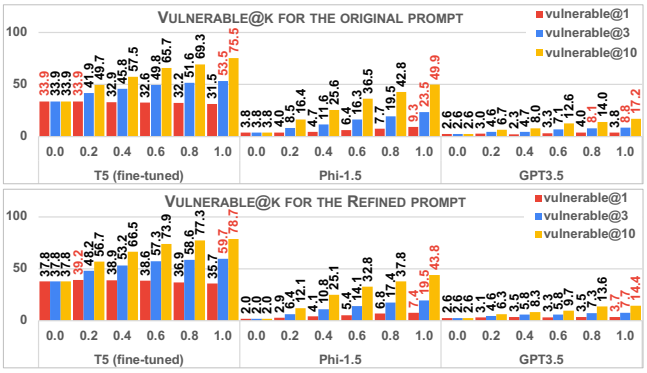

Figure 6: `vulnerable@k` for the original and refined prompts

Correctness vs. Security. To better understand whether LLMs are able to generate regexes that are both *correct and non-vulnerable* we computed the number of regexes which are correct (*i.e.*, passed all the prompt's test cases) and has a ReDoS vulnerability detected by ReDoSHunter [31]. As shown in Figure 7, although GPT-3.5 and PHI-1.5 generate vulnerable regexes, the majority of their correct regexes are both correct and secure (*i.e.*, ReDoSHunter did not detect any vulnerability). On the contrary, the FINE-TUNED T5 model did not generate any regex that was both correct and secure.

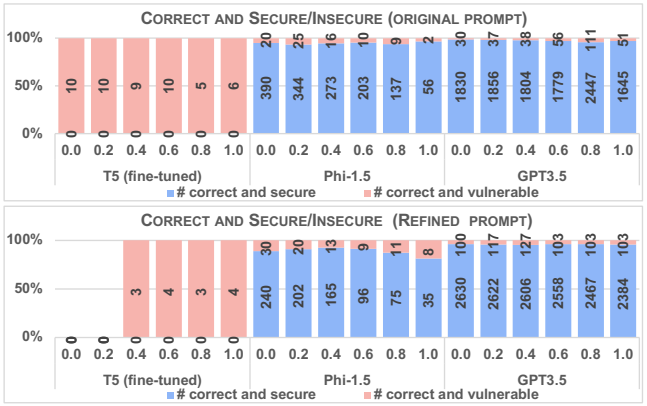

Figure 7: Number of correct and vulnerable regexes.

4.2 RQ2: ReDoS Characteristics

4.2.1 ReDoS Computational Complexity. Table 2 shows the ReDoS computational complexity class detected by ReDoSHunter [31] for **28,086** vulnerable regexes. Most of the vulnerable regexes have a polynomial computational complexity (*i.e.*, Polynomial Overlapping Adjacency - POA, and Starting with Large Quantifier - SLQ). This is consistent with a previous study of ReDoS in large software ecosystems [22]. Except for the PHI-1.5 model, the models usually generate more POA than SLQ types of ReDoS. As the temperature increases, the models tend to have more variation in the output,

which leads to more vulnerable regexes. The increment of polynomially vulnerable ReDoS for the T5 model is statistically significant with the increment of the temperature with the other two models (*i.e.*, for the original prompts, p-value for T5 and Phi-1.5 is 0.04 and for T5 and GPT-3.5 is 0.004. The significant level is $\alpha = 0.05$ for the Mann–Whitney U test). However, for Phi-1.5 and GPT-3.5-Turbo, the increment of polynomially vulnerable ReDoS with the temperature are not statistically significant, *i.e.*, p-value is $0.132 > \alpha$. For the exponential regexes, the most frequent complexities are: Exponential Overlapping Adjacency (EOA), Nested Quantifiers (NQ), and Exponential Overlapping Disjunction (EOD).

Table 2: Vulnerable regexes' computational complexity.

		T5 (Fine-Tuned)		Phi-1.5		GPT-3.5	
		Original	Refined	Original	Refined	Original	Refined
0.0	NQ	0	0	0	0	2	1
	EOD	0	0	1	0	0	0
	EOA	0	0	0	0	2	1
	POA	270	367	6	13	16	7
	SLQ	7	0	23	8	11	15
0.2	NQ	0	0	0	0	9	7
	EOD	0	0	4	0	1	0
	EOA	1	4	3	4	19	12
	POA	766	1,249	122	70	68	46
	SLQ	38	4	101	88	35	46
0.4	NQ	1	0	6	1	4	10
	EOD	0	0	6	4	1	3
	EOA	2	2	17	0	4	16
	POA	1,235	2,079	206	145	42	80
	SLQ	58	4	173	170	15	62
0.6	NQ	0	1	18	2	29	17
	EOD	2	0	10	2	4	0
	EOA	2	5	14	8	31	22
	POA	1,567	2,611	278	194	156	107
	SLQ	83	6	258	251	70	66
0.8	NQ	2	4	27	10	20	18
	EOD	2	2	17	10	4	3
	EOA	14	8	24	9	26	28
	POA	1,917	2,915	346	267	139	144
	SLQ	67	7	307	312	106	79
1.0	NQ	2	1	27	16	31	25
	EOD	6	4	20	8	8	7
	EOA	13	14	51	22	27	21
	POA	2,100	3,112	441	274	212	154
	SLQ	101	15	385	360	91	97

4.2.2 Regex Equivalence Classification. We manually analyzed a statistically significant subset (95% confidence level and 5% margin of error) of **336** ReDoS generated from the original prompt and **378** ReDoS generated from the refined prompt to classify them according to the semantically equivalent categories described by Chapman *et al.* [17]. Table 3 shows the results from three perspectives[5]: ***(i)*** prompt type, ***(ii)*** computational complexity type, and ***(iii)*** LLM. All five sub-categories have been identified from the Custom Character Class (CCC) group for both prompt types. The **C1** sub-category (*i.e.*, pattern that contains a non-negative custom character class with a range feature) is the most common from this group. For the Double-Bounded (DBB) group, we only observed patterns that use the curly brace repetition with a lower and upper bound (**D1**), and the questionable (*i.e.*, ?) modifier, which implies a lower bound of zero and an upper bound of one (**D2**).

[5]Due to space constraints, we omit the classes without any occurrence. The numbers in this table do not add up to 714 because a regex can be in multiple categories (*e.g.*, **C1** and **L2**)

There are four sub-categories for the Literal (LIT) group, but we only observed patterns that do not use any hex, wrapped, or octal characters but use at least one literal character (**T1**), which is the most common from this group, and patterns with a literal character wrapped in square brackets (**T3**). The Lower-Bounded (LWB) group is mainly responsible for the polynomial vulnerable regexes. Hence, all the analyzed pattern has at least one of the sub-categories of the class: pattern using this curly braces-style lower-bounded repetition (**L1**), pattern using the Kleene star (**L2**), which is the most common from this group, and pattern using the additional repetition (**L3**). In most cases, the **L2** sub-category dominates other sub-categories from this group. For the Single-Bounded (SNG) group, we have mainly a pattern with a single repetition boundary in curly braces (**S1**). Both prompt types have a similar distribution of regex classes.

Out of the **714** ReDoS we analyzed, **19** and **695** of them are exponentially and polynomially vulnerable, respectively. Exponentially vulnerable regexes do not have any sample from the SNG group, the **L1** sub-category from the LWB group, and **T3** sub-category from the LIT group in addition to the other absent sub-categories mentioned in the previous paragraph. It is noticeable that there are a significant number of regexes from the **T1** sub-categories.

Table 3: Regex Classification

	Prompt Type		Complexity Type		Model		
Class	Original	Refined	Exponential	Polynomial	T5	Phi-1.5	GPT-3.5
C1	189	187	11	365	289	55	32
C2	15	6	2	19	5	15	1
C3	12	15	2	25	0	16	11
C4	61	40	8	93	6	66	29
C5	23	4	3	24	15	8	4
D1	24	4	2	26	12	11	5
D2	54	33	9	78	0	50	37
L1	12	8	0	20	16	4	0
L2	309	345	15	639	521	86	47
L3	87	89	15	161	54	75	47
S1	56	54	0	110	95	10	5
S2	1	0	0	1	1	0	0
T1	301	354	16	639	519	91	45
T3	17	7	0	24	11	8	5

Chapman *et al.* [16] made a pair-wise comparison between regex classes about their understandablity by the developers. According to the study, T1 over T3, D1 over D2, and S1 over S2 are preferred (*i.e.*, more understandable by developers). While C1 is favored in comparison to C2, C4, and C5, none are significant. They also checked the open-source projects about their presence. C1, D2, T1, L2, and S2 are more frequently used from their corresponding classes. We can see the same phenomena in our result in Table 3, except for the Single-Bounded (SNG) group.

4.3 RQ3: Analyzing real ReDoS

4.3.1 ReDoS Computational Complexity. Table 4 presents the distribution of computation complexity patterns for the vulnerable regexes. The distribution indicates that most of the ReDoS have polynomially vulnerable patterns (*i.e.*, POA and SLQ), which is consistent with the result from our previous research question. However, the order for the exponentially vulnerable ReDoS patterns differs from the result of RQ2. We noticed more Exponential

Overlapping Disjunction (EOD) patterns in the real-world dataset compared to the other two patterns.

Table 4: Computational complexity classification

Pattern	Name	SOLA-DA	CVE
NQ	Nested Quantifiers	1	10
EOD	Exponential Overlapping Disjunction	0	29
EOA	Exponential Overlapping Adjacency	0	17
POA	Polynomial Overlapping Adjacency	10	181
SLQ	Starting with Large Quantifier	21	154

4.3.2 Regex Equivalence Classification. LLM-generated vulnerable regexes have more samples from the **C1** sub-category (Table 5). However, in the real-world dataset, most of the regexes have **C4** classes rather than **C1** classes. Except that results for other categories are consistent with the LLM-generated ReDoS-vulnerable regexes. It also seems that real-world datasets have more varieties of complex regexes, as we can find patterns that have a repetition with a lower and upper bound expressed using ORs (**D3**) and patterns using a hex token (**T2**).

Table 5: Regex classification for each dataset source.

	Source		Vulnerable Type	
Class	**SOLA-DA**	**CVE**	**Exponential**	**Polynomial**
C1	4	67	14	52
C2	10	122	25	93
C3	6	162	33	117
C4	29	347	48	308
C5	0	1	1	0
D1	1	15	6	10
D2	19	251	48	202
D3	1	2	1	2
L1	1	12	3	8
L2	17	337	53	282
L3	20	309	38	274
S1	0	24	3	21
T1	20	376	48	327
T2	0	2	2	0
T3	6	348	47	288

Table 6 shows a pair-wise comparison of the distribution of ReDoS patterns from different models, prompt type, and temperature combination with the dataset from Li *et al.* [32] by doing the Mann–Whitney U test. We can see that, for the distribution from temperature 0.0 of the Phi-1.5 and GPT-3.5 model, we can reject the Null hypothesis, and the result is statistically significant for them. For other cases, we can not reject the Null hypothesis, but the result is not statistically significant.

Table 6: p-values from the Mann–Whitney U test Result for ReDoS Patterns across different temperatures (temp.).

	Fine-Tuned T5		Phi-1.5		GPT-3.5	
Temp.	**Original**	**Refined**	**Original**	**Refined**	**Original**	**Refined**
0.0	0.139	0.131	0.036	0.020	0.027	0.021
0.2	0.402	0.141	0.222	0.209	0.421	0.295
0.4	0.421	0.143	0.599	0.222	0.094	0.310
0.6	0.402	0.151	1.000	0.675	0.917	0.600
0.8	0.530	0.151	0.600	0.675	0.691	0.691
1.0	0.548	0.310	0.421	1.000	1.000	0.691

For the regex classes, the distributions with exponent ReDoS patterns reject the Null hypothesis. The result is statistically significant (*i.e.*, $p - value = 0.015$), while regexes distributions with the polynomial ReDoS patterns can not reject the Null hypothesis, and the result is not statistically significant $p - value = 0.970$).

4.4 RQ4: Developers' Concerns

Figure 8 summarizes the developers' concerns as a taxonomy.

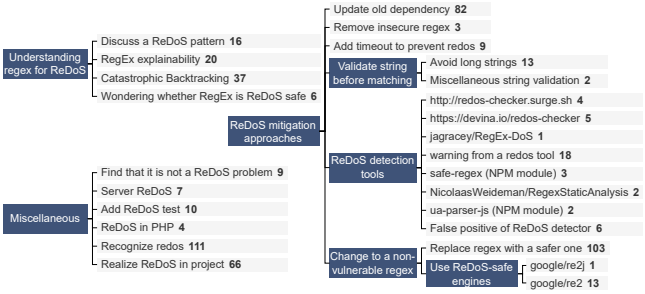

Figure 8: Taxonomy of developers' concerns

4.4.1 ReDoS Mitigation Approaches. The most discussed topic was on the different mitigation strategies for Regular Expression Denial of Service (ReDoS). The most discussed approach (48% of the discussions), is *"Change to a Non-vulnerable RegEx"*. Notably, 103 out of 117 instances in this category involve participants opting to change to a non-vulnerable regular expression independently. At the same time, the remainder refers to using ReDoS safe engines (*e.g.*, *re2* from Google [6]) to mitigate ReDoS.

The second most prominent approach *"Update old dependencies"* comprises 33.6% of the discussions, indicating a significant emphasis on keeping dependencies updated to enhance resilience against ReDoS attacks. *"Utilize ReDoS Detection Tools"* accounts for 7.4% of the mitigation approaches. The most popular tool is the ReDoS checker [7]. Moreover, 13 out of 15 discussions in the *"Valid String before Matching to Avoid ReDoS"* category focus on avoiding long strings as a strategy to mitigate ReDoS attacks. *"Add Timeout"* constitute smaller segments of the discussion at 3.7%. While such an approach may help mitigate the problem through a "bandaid" solution, they do not address the root cause of the problem. As there are only 3 data points that directly remove regex to avoid ReDoS, the analysis suggests that the community predominantly leans towards proactive strategies like modifying regular expressions and updating dependencies to mitigate the risk of ReDoS attacks.

4.4.2 Understanding RegEx for ReDoS Prevention. A significant portion of conversations is dedicated to identifying and discussing ReDoS patterns. Specifically, there are 16 instances where these patterns are scrutinized meticulously, illustrating the technical community's diligence in recognizing and addressing such vulnerabilities. *RegEx Explainability* has also surfaced as a significant area of interest, accounting for 20 related pull requests or posts. This topic encompasses approximately 29.1% of all ReDoS discussions on Stack Overflow, underscoring a prevalent demand among developers for clarity and a comprehensive understanding of Regex

functionalities in the context of ReDoS. Another pivotal concern highlighted is *Catastrophic Backtracking*, which has garnered 37 mentions. Furthermore, some developers struggled with whether a regex is ReDoS safe, with 6 instances and predominantly on Stack Overflow. This demonstrates a proactive approach within the developer community, where there is a concerted effort to ensure code resilience against ReDoS vulnerabilities through preemptive measures and active participation in forum discussions.

4.4.3 Miscellaneous. While the most vigorous discussions often revolve around direct approaches to fixing ReDoS and the understanding of regex to combat ReDoS, other less prominent but significant aspects also permeate conversations on developers' platforms, GitHub and StackOverflow. Server ReDoS discussions tally up to 7, indicating a keen interest in how ReDoS may affect server stability and performance. There are 4 conversations about ReDoS within the scope of PHP programming, suggesting a niche but vital concern for developers working with this server-side web development language. Furthermore, a proactive movement, as evidenced by 10 posts, towards integrating ReDoS tests in development workflows, demonstrating a commitment to preemptive defense measures against potential vulnerabilities.

There's a cluster of 9 instances where discussions initially flagged as being about ReDoS turned out to be unrelated, highlighting a potential gap in the community's understanding or identification of the issue. These topics, while not as dominant as the more general discussions on ReDoS, signify the multifaceted nature of the issue and the community's diverse approach to tackling it.

5 DISCUSSION

Usage of LLMs for Regex Generation. The RQ1 results show that LLMs can effectively generate regular expressions, especially GPT-3.5 [5]. A carefully crafted prompt, *i.e.*, a refined prompt, can produce the correct regex within the top 10 generations 53% of the time. However, they can still be vulnerable to ReDoS attacks. In 14.4% of cases, the GPT-3.5 model [5] can generate at least one vulnerable regex. LLMs tend to generate ReDoS-vulnerable regexes with polynomial patterns. However, they also significantly generate regexes with Nested Quantifiers (NQ), a type known for exponential vulnerability. LLMs can be utilized for generating regexes from natural language descriptions, but they should be vetted for ReDoS vulnerability before being used in production.

ReDoS generation. Our results in RQ2 and RQ3 indicate that the presence of Kleen star (∗) or pattern using additional repetition (+) is mainly responsible for introducing ReDoS attacks, as they can easily be exploited with long input. It is also noticeable that most of the ReDoS follow polynomial patterns rather than exponential patterns. There is also the presence of a significant number of custom character classes. From the analysis in RQ4, we found that developers have limited knowledge about regex and ReDoS, which can result in vulnerable regexes being deployed into production.

ReDoS Discussion in GitHub PR and StackOverflow. In the collaborative space of GitHub PRs and Stack Overflow discussions, developers deliberate the nuances of ReDoS vulnerabilities and their mitigation. However, discussion in GitHub pull requests and Stack Overflow happens differently. In GitHub PR, the discussion focuses on a particular project, but in Stack Overflow, it can be about any specific matter. If we dive deep into the taxonomy described in RQ4, developers discussed more about understanding a regex than their mitigation approaches. For instance, developers inquire about the explainability of a regex and question its safety. They caution against "Catastrophic Backtracking" in regexes provided on Stack Overflow. On GitHub, the discussion primarily revolves around mitigation strategies, such as transitioning to safer libraries and employing timeout mechanisms and detection tools.

Implication for the developers. The key takeaway for developers is that LLMs can be effective for RegEx generation, but are not free of ReDoS vulnerabilities (RQ1). As such, developers would benefit from vetting them using a ReDoS detection tool (such as ReDoSHunter [31]). The classification in RQ2 indicates that LLMs generate ReDoS vulnerable regexes patterns that can be less understandable by the developers, according to the study from Chapman *et al.* [17]. This takeaway is also confirmed in RQ4, which showed that developers need to be made aware of ReDoS or understand their effect. These findings also suggest that the developers can employ various strategies, such as implementing tests for ReDoS vulnerabilities, imposing execution time limits, or validating user inputs to mitigate ReDoS attacks. Utilizing regex engines known for their safety and regularly updating libraries prone to vulnerabilities are also advisable practices. While comparing the generated LLM-generated and real-world ReDoS vulnerable regexes in RQ3, most of the cases, the distribution may be the same as we can not reject the null hypothesis. Thus, developers should be aware that LLM-generated regexes may have the ReDoS with similar patterns from the real world ReDoS-vulnerable regexes. In short, our results showed that while LLMs can aid in regex generation, their effectiveness might require further refinement, as they are susceptible to ReDoS attacks. Consequently, rigorous testing for correctness and security is paramount before integrating LLM-generated regexes into production.

Implication for Researchers. Our findings highlight a persistent necessity for the explainability of a regex, with understandability being a key factor in mitigating ReDoS attacks. This study illuminates the aspect of ReDoS comprehension for the first time. Yet, there is ample scope for future research to delve deeper into the impact of mitigation techniques applied in real-world software to facilitate the transition away from libraries vulnerable to ReDoS and to enhance the understandability of regex as a preventive measure against ReDoS attacks. With the increasing prevalence of LLMs, there is a significant opportunity for researchers to develop methods that augment the efficacy and security of regex generation, particularly in the context of ReDoS threats.

6 THREATS TO VALIDITY

We collected original prompts from RegexLib [8] and crafted the refined prompts and additional test cases ourselves. The crafted refined prompts can be subjective and introduce an internal validity threat. We mitigate this threat by having two authors creating the refined prompts and having the additional tests checked using the ground truth from RegexLib [8]. The data collection step was

checked by the senior author, and less than 2% of the prompts had disagreements and were adjusted by them. Another internal validity threat is to manually assign a regex class to each regex and open coding for GitHub and Stack Overflow posts. However, the authors carry significant experience with software development, which helped mitigate these issues.

We used ReDoSHunter [31] to find a ReDoS vulnerability, which can introduce external validity threats. However, we chose this tool because it is a state-of-the-art technique with a precision of over 95% for the dataset used in RQ3 and that has been shown to find vulnerabilities in real software systems. Another external validity threat is to include only models based on the transformer [63] architecture. However, this is the most popular architecture for current state-of-the-art code generation models. Our work also includes a fine-tuned model, an open-source pre-trained model, and a closed-source model generalized for different tasks, which is in-line with the guidelines for empirical SE research using LLMs [47].

LLMs are prone to generate unstable output, but they can be controlled with its *temperature* parameter [15]. While the lower temperature provides a more predictable output, the higher temperature provides more variation in the output. As discussed in the methodology section, we provided comprehensive results by generating 10 regexes and varying temperatures from 0 to 1 with 0.2 increments. We also tackled data contamination threats by not only using the *original* prompts retrieved from RegExLib [8], but also *refined* prompts, where we manually re-worded the problem descriptions. This is a prompting technique called *metamorphic testing* [47], where the inputs are transformations of the original inputs that are different but maintain semantic equivalence to the original input.

7 RELATED WORK

Regular expressions and DoS attacks. The study of regex comprehension began with Chapman *et al.* [17], who were among the first to explore how different regex features impact their understandability. Michael *et al.* [36] identified multiple difficulties developers encounter with regexes, revealing that less than 40% of developers were aware of the security vulnerabilities associated with regex usage. Hassan *et al.* [25] developed a theory of regex infinite ambiguity to characterize regexes vulnerable to ReDoS, proposing a set of anti-patterns and fix strategies to enhance developer understanding. Unlike these studies, our work focuses on ReDoS comprehension mapping with the RegEx comprehension for both the ReDoS vulnerable LLM-generated and real-world RegExes.

The empirical study of ReDoS vulnerabilities has revealed significant security risks across various platforms and programming languages [20, 21]. Barlas *et al.* [13] shed light on the inherent dangers present in web services through their pioneering black-box study of live services, highlighting how client-side regex sanitization logic can render ReDoS vulnerabilities. Complementing this, Turoňová *et al.* [58] delve into the weaknesses of nonbacktracking regex matches to ReDoS attacks. Furthermore, the prevalence of ReDoS issues has been extensively documented, indicating vulnerabilities on dozens of major websites and within hundreds of JavaScript projects [23, 57], as well as thousands of codebases in Python and Java [22, 68]. These studies collectively underscore the critical need for heightened awareness and robust countermeasures against ReDoS attacks in software development.

Vulnerable regexes detection. Berglund *et al.* [14] presented an automata model to systematically analyze catastrophic backtracking. Wustholz *et al.* [68] introduces Rexploiter to identify vulnerable regexes and confirm their exploitability with crafted input strings. Building on the concept of attack pattern generation, Shen *et al.* [50] introduced ReScue, a three-phase gray-box technique that generates strings that trigger ReDoS. Further advancing the methodology, Liu *et al.* [34] proposed an integrated approach (*ReDoSHunter*) that combines both static and dynamic analyses to efficiently detect ReDoS vulnerabilities. In our work, we utilized the advanced capabilities of ReDoSHunter [31] for detecting and characterizing ReDoS-vulnerable regexes.

Vulnerable regexes repair. Van *et al.* [62] explored how to transform exploitable regexes into efficient, ambiguity-free alternatives. Chida *et al.* [19] pioneered a Programming by Example method [67] to repair regexes. Li *et al.* [32] introduced RegexScalpel, an automatic regex repair framework that fixes vulnerabilities while ensuring semantic equivalence to the original regexes. We used the evaluation datasets from Li *et al.* [32] to look in the comprehension of the real-world RegExes prone to ReDoS attack.

Empirical studies on LLMs. As the field progresses, the advent of code-generating Large Language Models like GitHub Copilot and OpenAI's Codex has spurred research into their efficacy and security implications. Nguyen and Nadi [39] and Finnie-Ansley et al. [24] have evaluated these tools' performance on coding problems and student assessments. In contrast, Vaithilingam et al. [60] looked into user interaction with Copilot, particularly in error recognition and task completion times. Security analyses of LLM-generated code from multiple works [12, 42, 44, 48] reveal varying incidence rates of security bugs and the influence of LLMs on secure coding practices. Sobania et al. [55] and Dakhel et al. [37] provide comparative studies of Copilot's solutions against traditional benchmarks, addressing correctness, complexity, and diversity. Furthermore, Siddiq et al. [52, 53] expand the discussion to the prevalence of code smells and the ability of LLMs to generate unit tests.

The collective body of research underscores the necessity of a dual perspective approach that considers both the developers' comprehension and technical tools for detecting and mitigating ReDoS vulnerabilities. Our work bridges this gap by offering insights into developers' practices and the efficacy of LLM-generated regex patterns in enhancing software resilience against ReDoS attacks.

8 CONCLUSION

Our investigation provides a dual perspective on Regular Expression Denial of Service (ReDoS) comprehension, emphasizing the interplay between developer discussions and LLM-generated regexes. We have conducted an extensive analysis, investigating 274,320 generated regexes to benchmark the performance and security of LLM outputs. Our study also categorized ReDoS-vulnerable regexes, connecting them to established equivalence classes and dissecting real-world software instances to elucidate prevalent patterns and

classifications. We also presented a novel taxonomy derived from developer interactions on GitHub and Stack Overflow, highlighting the community's approach to addressing ReDoS concerns. The insights gained from this study not only shed light on the current state of ReDoS comprehension but also pave the way for future developments in creating more secure regexes through the assistance of LLMs, bridging the gap between theoretical understanding and practical application in the realm of software security.

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. Common Vulnerabilities and Exposures (CVE). https://cve.mitre.org/index.html [Online; accessed 11. Sep. 2023].

[4] 2023. CWE - CWE-1333: Inefficient Regular Expression Complexity (4.13). https://cwe.mitre.org/data/definitions/1333.html [Online; accessed 6. Nov. 2023].

[5] 2023. GPT-3.5. https://platform.openai.com/docs/models/gpt-3-5 [Online; accessed 11. Sep. 2023].

[6] 2023. re2. https://github.com/google/re2 [Online; accessed 5. Nov. 2023].

[7] 2023. ReDoS checker | Devina.io. https://devina.io/redos-checker [Online; accessed 5. Nov. 2023].

[8] 2023. Regular Expression Library. https://regexlib.com [Online; accessed 11. Sep. 2023].

[9] 2023. Stack Exchange API. https://api.stackexchange.com [Online; accessed 11. Sep. 2023].

[10] 2024. *Outage Postmortem - July 20, 2016.* https://stackstatus.tumblr.com/post/147710624694/outage-postmortem-july-20-2016 [Online; accessed 23. Jan. 2024].

[11] Ahmad Abdellatif, Mairieli Wessel, Igor Steinmacher, Marco A. Gerosa, and Emad Shihab. 2022. BotHunter: An Approach to Detect Software Bots in GitHub. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. 6–17. https://doi.org/10.1145/3524842.3527959

[12] Owura Asare, Meiyappan Nagappan, and N Asokan. 2022. Is GitHub's copilot as bad as humans at introducing vulnerabilities in code? *arXiv preprint arXiv:2204.04741* (2022).

[13] Efe Barlas, Xin Du, and James C Davis. 2022. Exploiting input sanitization for regex denial of service. In *Proceedings of the 44th International Conference on Software Engineering*. 883–895.

[14] Martin Berglund, Frank Drewes, and Brink Van Der Merwe. 2014. Analyzing catastrophic backtracking behavior in practical regular expression matching. *arXiv preprint arXiv:1405.5599* (2014).

[15] 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.

[16] Carl Chapman and Kathryn T. Stolee. 2016. Exploring Regular Expression Usage and Context in Python. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 282–293. https://doi.org/10.1145/2931037.2931073

[17] Carl Chapman, Peipei Wang, and Kathryn T Stolee. 2017. Exploring regular expression comprehension. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 405–416.

[18] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[19] Nariyoshi Chida and Tachio Terauchi. 2022. Repairing dos vulnerability of real-world regexes. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2060–2077.

[20] Scott Crosby. 2003. Denial of service through regular expressions. (2003).

[21] Scott A. Crosby and Dan S. Wallach. 2003. Denial of Service via Algorithmic Complexity Attacks. In *12th USENIX Security Symposium (USENIX Security 03)*. USENIX Association, Washington, D.C. https://www.usenix.org/conference/12th-usenix-security-symposium/denial-service-algorithmic-complexity-attacks

[22] 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*. 246–256.

[23] James C. Davis, Eric R. Williamson, and Dongyoon Lee. 2018. A Sense of Time for JavaScript and Node.js: First-Class Timeouts as a Cure for Event Handler Poisoning. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 343–359. https://www.usenix.org/conference/usenixsecurity18/presentation/davis

[24] James Finnie-Ansley, Paul Denny, Brett A Becker, Andrew Luxton-Reilly, and James Prather. 2022. The robots are coming: Exploring the implications of openai codex on introductory programming. In *Proceedings of the 24th Australasian Computing Education Conference*. 10–19.

[25] Sk Adnan Hassan, Zainab Aamir, Dongyoon Lee, James C Davis, and Francisco Servant. 2023. Improving Developers' Understanding of Regex Denial of Service Tools through Anti-Patterns and Fix Strategies. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1238–1255.

[26] 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.

[27] Kelvin Jiang, Ronak Pradeep, and Jimmy Lin. 2021. Exploring listwise evidence reasoning with t5 for fact verification. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*. 402–410.

[28] 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).

[29] Eric Larson and Anna Kirk. 2016. Generating evil test strings for regular expressions. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 309–319.

[30] 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]

[31] 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)*. 3847–3864.

[32] Yeting Li, Yecheng Sun, Zhiwu Xu, Jialun Cao, Yuekang Li, Rongchen Li, Haiming Chen, Shing-Chi Cheung, Yang Liu, and Yang Xiao. 2022. {RegexScalpel}: Regular Expression Denial of Service ({{{{{ReDoS}}}}}) Defense by {Localize-and-Fix}. In *31st USENIX Security Symposium (USENIX Security 22)*. 4183–4200.

[33] Yeting Li, Zhiwu Xu, Jialun Cao, Haiming Chen, Tingjian Ge, Shing-Chi Cheung, and Haoren Zhao. 2020. Flashregex: Deducing anti-redos regexes from examples. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 659–671.

[34] Yinxi Liu, Mingxue Zhang, and Wei Meng. 2021. Revealer: Detecting and exploiting regular expression denial-of-service vulnerabilities. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1468–1484.

[35] 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*, Jian Su, Kevin Duh, and Xavier Carreras (Eds.). Association for Computational Linguistics, Austin, Texas, 1918–1923. https://doi.org/10.18653/v1/D16-1197

[36] 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)*. IEEE, 415–426.

[37] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Zhen Ming (Jack) Jiang. 2023. GitHub Copilot AI Pair Programmer: Asset or Liability? *J. Syst. Softw.* 203, C (sep 2023), 23 pages. https://doi.org/10.1016/j.jss.2023.111734

[38] Nadim Nachar et al. 2008. The Mann-Whitney U: A test for assessing whether two independent samples come from the same distribution. *Tutorials in quantitative Methods for Psychology* 4, 1 (2008), 13–20.

[39] Nhan Nguyen and Sarah Nadi. 2022. An Empirical Evaluation of GitHub Copilot's Code Suggestions. In *Proceedings of the 19th International Conference on Mining Software Repositories* (Pittsburgh, Pennsylvania) *(MSR '22)*. Association for Computing Machinery, New York, NY, USA, 1–5. https://doi.org/10.1145/3524842.3528470

[40] Jun-U Park, Sang-Ki Ko, Marco Cognetta, and Yo-Sub Han. 2019. Softregex: Generating regex from natural language descriptions using softened regex equivalence. In *Proceedings of the 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing (EMNLP-IJCNLP)*. 6425–6431.

[41] Rishov Paul, Md. Mohib Hossain, Mohammed Latif Siddiq, Masum Hasan, Anindya Iqbal, and Joanna C. S. Santos. 2023. Enhancing Automated Program Repair through Fine-tuning and Prompt Engineering. arXiv:2304.07840 [cs.LG]

[42] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In *2022 IEEE Symposium on Security and Privacy*

(SP). 754–768. https://doi.org/10.1109/SP46214.2022.9833571

[43] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. 2023. The impact of AI on developer productivity: Evidence from GitHub copilot. *arXiv preprint arXiv:2302.06590* (2023).

[44] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. 2022. Do Users Write More Insecure Code with AI Assistants? *arXiv preprint arXiv:2211.03622* (2022).

[45] 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

[46] Denise Rey and Markus Neuhäuser. 2011. Wilcoxon-Signed-Rank Test. In *International Encyclopedia of Statistical Science*. https://api.semanticscholar.org/CorpusID:30088448

[47] June Sallou, Thomas Durieux, and Annibale Panichella. 2024. Breaking the Silence: the Threats of Using LLMs in Software Engineering. In *ACM/IEEE 46th International Conference on Software Engineering - New Ideas and Emerging Results*. ACM/IEEE.

[48] Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. 2022. Security Implications of Large Language Model Code Assistants: A User Study. *arXiv preprint arXiv:2208.09727* (2022).

[49] 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

[50] 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*. ACM, Montpellier France, 225–235. https://doi.org/10.1145/3238147.3238159

[51] Mohammed Latif Siddiq, Beatrice Casey, and Joanna Santos. 2023. A Lightweight Framework for High-Quality Code Generation. *arXiv preprint arXiv:2307.08220* (2023).

[52] Mohammed Latif Siddiq, Shafayat H Majumder, Maisha R Mim, Sourov Jajodia, and Joanna CS Santos. 2022. An Empirical Study of Code Smells in Transformer-based Code Generation Techniques. In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 71–82.

[53] Mohammed Latif Siddiq, Joanna 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 preprint arXiv:2305.00418* (2023).

[54] Mohammed Latif Siddiq and Joanna C. S. Santos. 2023. Generate and Pray: Using SALLMS to Evaluate the Security of LLM Generated Code. arXiv:2311.00889 [cs.SE]

[55] Dominik Sobania, Martin Briesch, and Franz Rothlauf. 2022. Choose Your Programming Copilot: A Comparison of the Program Synthesis Performance of Github Copilot and Genetic Programming. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Boston, Massachusetts) *(GECCO '22)*. Association for Computing Machinery, New York, NY, USA, 1019–1027. https://doi.org/10.1145/3512290.3528700

[56] Eric Spishak, Werner Dietl, and Michael D. Ernst. 2012. A Type System for Regular Expressions. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs* (Beijing, China) *(FTfJP '12)*. Association for Computing Machinery, New York, NY, USA, 20–26. https://doi.org/10.1145/2318202.2318207

[57] Cristian-Alexandru Staicu and Michael Pradel. 2018. Freezing the Web: A Study of {ReDoS} Vulnerabilities in {JavaScript-based} Web Servers. In *27th USENIX Security Symposium (USENIX Security 18)*. 361–376.

[58] Lenka Turoňová, Lukáš Holík, Ivan Homoliak, Ondřej Lengál, Margus Veanes, and Tomáš Vojnar. 2022. Counting in Regexes Considered Harmful: Exposing {ReDoS} Vulnerability of Nonbacktracking Matchers. In *31st USENIX Security Symposium (USENIX Security 22)*. 4165–4182.

[59] Sri Lakshmi Vadlamani and Olga Baysal. 2020. Studying Software Developer Expertise and Contributions in Stack Overflow and GitHub. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 312–323. https://doi.org/10.1109/ICSME46990.2020.00038

[60] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 2022. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Chi conference on human factors in computing systems extended abstracts*. 1–7.

[61] Tim Valicenti, Justice Vidal, and Ritik Patnaik. 2023. Mini-GPTs: Efficient Large Language Models through Contextual Pruning. *arXiv preprint arXiv:2312.12682* (2023).

[62] Brink Van Der Merwe, Nicolaas Weideman, and Martin Berglund. 2017. Turning evil regexes harmless. In *Proceedings of the South African Institute of Computer Scientists and Information Technologists*. 1–10.

[63] 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.

[64] Shangwen Wang, Mingyang Geng, Bo Lin, Zhensu Sun, Ming Wen, Yepang Liu, Li Li, Tegawendé F. Bissyandé, and Xiaoguang Mao. 2023. Natural Language to Code: How Far are We?. In *Proceedings of the 31st ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM. https://doi.org/10.1145/3611643.3616323

[65] Cody Watson, Nathan Cooper, David Nader Palacio, Kevin Moran, and Denys Poshyvanyk. 2021. A Systematic Literature Review on the Use of Deep Learning in Software Engineering Research. arXiv:2009.06520 [cs.SE]

[66] Adar Weidman. 2022. Regular expression denial of service - redos. https://owasp.org/www-community/attacks/Regular_expression_Denial_of_Service_-_ReDoS

[67] Jiarong Wu, Lili Wei, Yanyan Jiang, Shing-Chi Cheung, Luyao Ren, and Chang Xu. 2023. Programming by Example Made Easy. *ACM Transactions on Software Engineering and Methodology* (jul 2023). https://doi.org/10.1145/3607185

[68] 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.

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

[70] 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.

[71] Honglei Zhuang, Zhen Qin, Rolf Jagerman, Kai Hui, Ji Ma, Jing Lu, Jianmo Ni, Xuanhui Wang, and Michael Bendersky. 2023. Rankt5: Fine-tuning T5 for text ranking with ranking losses. In *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 2308–2313.

[72] 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 *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming* (San Diego, CA, USA) *(MAPS 2022)*. Association for Computing Machinery, New York, NY, USA, 21–29. https://doi.org/10.1145/3520312.3534864