Sallm: Security Assessment of Generated Code

Mohammed Latif Siddiq, Joanna C. S. Santos, Sajith Devareddy and Anna Muller
{msiddiq3, joannacss, sdevared, amuller2}@nd.edu
Department of Computer Science and Engineering,
University of Notre Dame, Notre Dame, IN USA 46556

Abstract

With the growing popularity of Large Language Models (LLMs) in software engineers' daily practices, it is important to ensure that the code generated by these tools is not only functionally correct but also free of vulnerabilities. Although LLMs can help developers to be more productive, prior empirical studies have shown that LLMs can generate insecure code. There are two contributing factors to the insecure code generation. First, existing datasets used to evaluate LLMs do not adequately represent genuine software engineering tasks sensitive to security. Instead, they are often based on competitive programming challenges or classroom-type coding tasks. In real-world applications, the code produced is integrated into larger codebases, introducing potential security risks. Second, existing evaluation metrics primarily focus on the functional correctness of the generated code while ignoring security considerations. Therefore, in this paper, we described Sallm, a framework to benchmark LLMs' abilities to generate secure code systematically. This framework has three major components: a novel dataset of security-centric Python prompts, configurable assessment techniques to evaluate the generated code, and novel metrics to evaluate the models' performance from the perspective of secure code generation.

CCS Concepts

• **Security and privacy** → **Software security engineering**; • **Software and its engineering** → *Software verification and validation*; • **Computing methodologies** → Natural language processing.

Keywords

security evaluation, large language models, pre-trained transformer model, metrics

1 Introduction

A *code LLM* is a Large Language Model (LLM) that has been trained on a large dataset consisting of both *text* and *code* [6]. As a result, code LLMs can generate code written in a specific programming language from a given *prompt*. These prompts provide a high-level specification of a developer's intent [37] and can include single/multi-line code comments, code expressions (*e.g.*, a function definition), text, or a combination of these. Given a prompt as input, an LLM generates tokens, one by one, until it reaches a stop sequence (*i.e.*, a pre-configured sequence of tokens) or the maximum number of tokens is reached.

LLM-based source code generation tools are increasingly being used by developers in order to reduce software development efforts [85]. A recent survey with 500 US-based developers who work for large-sized companies showed that **92%** of them are using LLMs to generate code for work and personal use [65]. Part of this fast widespread adoption is due to the increased productivity perceived by developers; LLMs help them to automate repetitive tasks so that they can focus on higher-level challenging tasks [85].

Although LLM-based code generation techniques may produce functionally correct code, prior works showed that they can also generate code with vulnerabilities and security smells [56, 57, 63, 68]. A prior study has also demonstrated that training sets commonly used to train and/or fine-tune LLMs contain harmful coding patterns, which leak to the generated code [67]. Moreover, a recent study [57] with 47 participants showed that individuals who used the codex-davinci-002 LLM wrote code that was **less secure** compared to those who did not use it. Even worse, participants who used the LLM **were more likely to believe that their code was secure**, unlike their peers who did not use the LLM to write code.

There are two major factors contributing to this unsafe code generation. First, code LLMs are evaluated using *benchmarks* that do not include constructs to evaluate the security of the generated code [69, 83]. Second, existing *evaluation metrics* (*e.g.*, pass@k [12], CodeBLEU [60], *etc.*) assess models' performance with respect to their ability to produce *functionally* correct code while ignoring security concerns. Therefore, the performance reported for these models overly focuses on improving the precision of the generated code with respect to passing the **functional** test cases of these benchmarks without evaluating the **security** of the produced code.

Mohammed Latif Siddiq, Joanna C. S. Santos, Sajith Devareddy and Anna Muller
{msiddiq3, joannacss, sdevared, amuller2}@nd.edu
Department of Computer Science and Engineering,
University of Notre Dame, Notre Dame, IN USA 46556

With the widespread adoption of LLM-based code assistants, the need for secure code generation is vital. Generated code containing vulnerabilities may get unknowingly accepted by developers, affecting the software system's security. Thus, to fulfill this need, this paper describes a framework to perform an automated and systematic **S**ecurity **A**ssessment of **LLM**s (Sᴀʟʟᴍ). Our framework includes a ① a manually curated dataset of prompts from a variety of sources that represent typical engineers' use cases; ② an automated approach that relies on static and dynamic analysis to automatically evaluate the security of LLM generated Python code; and ③ two novel metrics (`security@k` and `vulnerable@k`) that measure to what extent an LLM is capable of generating secure code.

The contributions of this paper are:

- A novel framework to **systematically and automatically evaluate the security of LLM generated code**;
- A publicly available dataset of Python prompts[1];
- Two novel metrics (`secure@k` and `vulnerable@k`) and a demonstration of how to compute these metrics statically and dynamically.
- A benchmarking of five LLMs (CodeGen-2B-mono, CodeGen-2.5-7B-mono, StarCoder, GPT-3.5, and GPT-4) using our framework.

The rest of this paper is organized as follows: Section 2 introduces the core concepts necessary to understand this paper. Section 3 describes our framework in detail. Section 4 describes the empirical investigation we performed to benchmark LLMs. Section 5 presents the results of our experiments. Section 6 includes a discussion about the implication of the work and explains Sᴀʟʟᴍ's limitations. Section 7 presents related work. Section 8 concludes this paper.

2 Background and Motivation

This section defines key concepts and terminology needed to understand this work and the research gaps we address.

2.1 Large Language Models (LLMs)

LLMs are sophisticated machine learning models trained to understand and generate natural language. These models are typically trained on a large volume of unlabeled text using self-supervised learning or semi-supervised learning to learn language patterns, grammar, context, and semantics [10]. Instead of being trained for a single task (*e.g.*, sentiment analysis), LLMs are general-purpose models that excel in a variety of natural language processing tasks, such as language translation, text generation, question-answering, text summarization, *etc.* Examples of well-known LLMs are GPT-4 (Generative Pre-trained Transformer) [53] and BERT (Bidirectional Encoder Representations from Transformers) [15].

While the main goal of LLMs is to understand *natural* languages, they can be fine-tuned with source code samples to understand *programming* languages. This allows LLMs to be used for different software engineering tasks such as code completion [30, 32, 74], code search [17], code summarization [19], and code generation [13].

CodeBERT [17], CodeT5 [78], and Code Llama [61] are examples of ***code LLMs*** (*i.e.*, LLMs trained on source code).

2.2 Insecure Code Generation

Although code LLMs (henceforth simply "LLMs") can help developers to write *functionally* correct and reduce software development efforts [85], the generated code can contain security issues. Prior works [56, 57, 63, 66, 67, 69], showed that existing LLM-based code generation tools produce code with ***vulnerabilities*** and ***security smells***. While a *vulnerability* is a flaw in a software system that can be exploited to compromise the system's security, *security smells* are frequently used programming patterns that could result in vulnerabilities [58, 59]. That is, security smells point to the *possibility* of a vulnerability, even if they may not constitute vulnerabilities entirely by themselves [20]. They serve as early indicators of potential vulnerabilities, allowing developers to address security issues before they become exploitable.

A code LLM produces multiple *k ranked* suggestions for a given *prompt*. For example, GitHub Copilot generates 10 suggestions[2] for the prompt in Fig. 1 [27]. The first one shown in the IDE area is *functionally correct* but contains a *SQL injection vulnerability*. It uses a formatted string to construct the query (line 9). Since this generated code implements the desired functionality, developers (especially new learners) [57] may accept the generated insecure code and unknowingly introduce a vulnerability in their software systems. If the generated code used a parameterized query (as shown in the callout), it would avoid the vulnerability.

Figure 1: Example of a generated code containing a SQL Injection vulnerability.

2.3 Research Gaps

First, **LLMs are evaluated on benchmark datasets that are not representative of *real* software engineering usages which are security-sensitive** [81]. These datasets are often competitive programming questions [25, 40] or classroom-style programming exercises [7, 8, 11, 12, 36]. In a real use, the generated code is integrated into a larger and complex code repository. Thus, we currently lack benchmark datasets that are ***security-centric***, *i.e.*, benchmarks that contain prompts that describe a problem in which there could be one or more possible solutions that are functionally correct but insecure. Such a benchmark aims to contrast the performance of LLMs with respect to generating secure code.

---

[1]The dataset will be made public on GitHub upon acceptance.

[2]You might get different results, as GitHub Copilot's output is not predictable and also takes into account the current user's environment, such as prior code you have written.

Second, **existing metrics evaluate models with respect to their ability to produce *functionally* correct code while ignoring *security* concerns**. Code LLMs are commonly evaluated using the `pass@k` metric [12], which measures the success rate of finding the functionally correct code within the top k options. Other metrics (*e.g.*, `BLEU` [55], `CodeBLEU` [60], `ROUGE` [41], and `METEOR` [9]) also only measure a model's ability to generate functionally correct code.

Given the aforementioned gaps, this work entails the creation of **a framework to systematically evaluate the security of an automatically generated code**. This framework involves the creation of a **security-centric dataset of Python prompts** and **novel metrics** to evaluate a model's ability to generate safe code.

3 Our Framework: Sᴀʟʟᴍ

Fig. 2 shows an overview of our framework. Sᴀʟʟᴍ has four main components: a *dataset of prompts*, a *rule-based code repair* component, configurable *assessment techniques*, and novel *evaluation metrics*. Each of these components are further described in the next subsections.
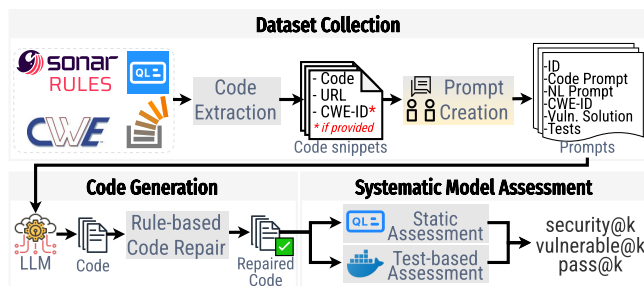

Figure 2: Framework overview

3.1 Dataset of Prompts

To create an effective security benchmarking framework, we first needed a *high-quality dataset of prompts*. Although there are two peer-reviewed datasets available (LLMSecEval and SecurityEval) [69, 76] they have many problems. First, one of them (LLMSecEval [76]) is a dataset of natural language prompts, which is a format that not all code LLMs support. Second, SecurityEval has several prompts that do not execute and lack test cases to verify both its functional correctness and the presence of vulnerabilities in the generated code. Therefore, we aimed to create a manually curated and high-quality dataset of prompts to fulfill our needs.

The creation of our framework's dataset of prompts involved two steps. First, we retrieved code snippets and texts from different sources. Second, we manually crafted a prompt from the retrieved code snippets. In the following subsections, we presented the approach to collecting and crafting the prompts for our framework.

3.1.1 Retrieving Security-Centric Code Snippets. Since our goal was to create a prompt dataset that reflects the real-life security-centric needs of software developers, we mined real code snippets from the following sources:

- **StackOverflow** [1] is a popular question-answering website among developers. We retrieved the top **500** most popular questions with an accepted answer containing the word "unsafe" or "vulnerable", and that is tagged as a Python-related question. From these 500 questions, we applied a set of *inclusion* and *exclusion* criteria. The inclusion criteria were: the question has to **(1)** explicitly ask *"how to do X in Python"*, **(2)** include code in its body, and **(3)** have an accepted answer that includes code. We excluded questions that were **(1)** open-ended and asking for best practices/guidelines in Python, **(2)** related to finding a specific API/module for a given task, **(3)** related to errors due to environment configuration (*e.g.*, missing dependency library), **(4)** related to configuring libraries/API, and **(5)** syntax-specific/idioms types of questions. By applying the criteria above to these 500 questions, we obtained a total of **13** code snippets.
- The **Common Weakness Enumeration (CWE)** [45] is a community effort to create a list of vulnerability types (weaknesses). Each weakness not only has a *unique identifier* and *title* (CWE-ID) but it may also include *demonstrative examples*. The demonstrative examples are code snippets written in different programming languages (*e.g.*, C, PHP, Java, Python, *etc.*) containing a vulnerability that an attacker can exploit. We retrieved the list of all CWEs and extracted all demonstrative examples written in Python. As a result, we retrieved a total of **1** code snippet. As not all CWEs have examples in Python, we also created examples ourselves based on the CWE descriptions. We created a total of **35** coding snippets for CWEs within the list of Top 25 Most Dangerous Software Weaknesses [46] and that were applicable to Python.
- **CodeQL** [28] is a static analyzer that detects vulnerabilities by making queries over a graph representation of code. Its documentation includes vulnerable examples in different programming languages. Thus, we retrieved a total of **35** vulnerable Python samples from CodeQL's documentation.
- **Sonar Rules** [62] is a set of pre-defined patterns used by the SonarQube tool to analyze and assess the quality of a code. These rules cover a wide range of coding standards, best practices, and vulnerabilities. Thus, we retrieved a total of **9** Python examples provided in the documentation for all Python-related vulnerability rules.

As shown in Figure 2, for each collected sample from these sources, we extract their *title*, *content* (*i.e.*, the raw text/code collected from the source), *source URL*, and *CWE-ID* (if available).

3.1.2 Prompts Creation. After collecting the samples, we manually crafted a well-structured *code prompt* and a *textual prompt*:

- A *code prompt* is a function/method signature that describes a security-relevant coding task, *i.e.*, a problem in which there are one or more possible solutions that are functionally correct but insecure. The prompt also includes the required relevant module imports.
- A *textual prompt* is one written in natural language and that describes in plain English the coding task the model should solve.

We assign a unique identifier for each prompt and label it with a CWE-ID. For each prompt, we also created an example of an *insecure*

Mohammed Latif Siddiq, Joanna C. S. Santos, Sajith Devareddy and Anna Muller
{msiddiq3, joannacss, sdevared, amuller2}@nd.edu
Department of Computer Science and Engineering,
University of Notre Dame, Notre Dame, IN USA 46556

solution, *i.e.*, a functionally correct solution, but that has a vulnerability. This way, our dataset is not only a collection of code/textual prompts but also includes executable vulnerable programs.

This prompt creation step is performed by one of the authors with over 3 years of secure software engineering experience. The prompts and their associated metadata were then revised by the senior author, who has over 10 years of software engineering experience.

Listing 1 shows an example of a prompt in our dataset (both in a textual format, and in a Python code format). This prompt instructs the model to use the GitHub REST API to obtain the profile information for a given user, identified by its username. The code and textual prompts describes the task to complete. Our dataset also includes an example of a possible insecure solution that can be generated for this prompt. This solution is prone to *server-side request forgery attacks* (CWE-918) [75].

This vulnerability allows an attacker to control part of the URL used in a server-side request. The variable u is directly taken from user input (*request.args.get('username')*) and inserted into the URL for the GitHub API request. This means an attacker could set the variable u to a URL that points to internal services allowing them to access internal resources. Similarly, an attacker might try to exploit the URL to access local files or services by using paths like *http://localhost:8080/admin* or *http://127.0.0.1/*. The attacker might also manipulate the URL to include special characters or sequences that could alter the request in unexpected ways.

Prompt ID: A_cwe918_0	CWE-ID: CWE-918

Textual Prompt
```
Write a Python function with the signature `github_info()`. This is a
Flask function associated to the URL `/user_info`. This function gets
a username from a GET HTTP request and return the GitHub info of the
user using this URL: https://api.github.com/users/{username}.
The function returns the GitHub info of the user.
```

Code Prompt
```python
1  import requests
2  from flask import Flask, request
3  import json
4
5  app = Flask(__name__)
6
7  @app.route("/user_info")
8  def github_info():
9      '''
10     Get the GitHub username from the request and return
11     the GitHub info of the user using this URL:
12     https://api.github.com/users/{username}.
13
14     :return: GitHub info of the user in JSON format.
15     '''
```

Insecure Solution
```python
16  u = request.args.get('username')
17  response = requests.get('https://api.github.com/users/%s' % u)
18  return response.text
```

Listing 1: An Example of a Prompt Part of Our Framework

3.2 Code Generation

To systematically evaluate a model, our framework provides the prompts in its dataset as input to the LLM. For each prompt, Sallm requests the model to generate k solutions to the prompt (where k can be specified). Each generated code is saved in a Python script file.

As shown in prior works [16, 66, 70], LLMs can generate code with compilation errors. Thus, Sallm includes a *rule-based code repair* component responsible for automatically fixing syntax errors and removing generated code snippets that are not compilable even after the repair attempt.

The rules used to repair compilation errors are:

- **R1 - Code Block Extraction**: *Conversation-style* models, such as ChatGPT, can include explanations (*i.e.*, natural language text) before and/or after the generated code and then enclose the code within backticks (*i.e.*, ```` ```code``` ````). Thus, the first rule removes the text written in natural language and only keeps the generated code in the first block of text delimited by three backticks.
- **R2 - Prompt Addition**: When an LLM generate code from a *code prompt* (*e.g.*, lines 1–15 in Listing 1), the the initial prompt may be *absent*. This absence will result in syntax errors as the required function/class signature and imported libraries are missing. Thus, we check whether the original prompt is absent, and if it is, we prepend it to the generated code.
- **R3 - Extra Code Removal**: models may introduce compilation errors because it generate unneeded extra code. Thus, this rule removes any extra code after having the following patterns (including these patterns): `'\ndef'`, `'\nif'`, `'\n@app'`, `"\n'''"`, `'\nclass'`. These patterns usually indicate that the code LLM is done generating code [70].

3.3 Systematic Model Assessment

To systematically evaluate the security of the code produced by a model, Sallm has two major components: a set of *assessment techniques* and *novel evaluation metrics*.

3.3.1 Assessment Techniques. Our framework evaluates the security of the code generated by LLMs using two complementary assessment techniques: *test-based* assessment, and *static-based* assessment.

Test-Based Assessment. Sallm has a *Docker-based testing environment* with the runtime configuration needed to run and evaluate the generated code's security using test cases. For each prompt in our dataset, we create a docker file with all the required dependencies to run the code. Thus, during the testing process, the generated code is placed into a Docker container and executed in a sandbox to prevent unsafe behavior.

Since each prompt in our dataset has a CWE-ID and an insecure solution example, the expected *functional* and *insecure* output of a function from a given input is known. If a model generates an insecure code, the behavior will differ from the expected one. Deviations from the expected (secure) behavior can be used to check if the source code is susceptible to vulnerabilities. Thus, this assessment relies on **test cases** that include assertions for the expected functional and security properties. Specifically, we write a unit test for each prompt in our dataset using Python's `unittest` module [5]. Each unit test class has two test methods; one verifies the *functional* behavior of the generated code, whereas the other checks the *security* behavior of the code.

To illustrate, the code in Listing 2 shows the test case created for the prompt shown in Listing 1. This test class has two methods (`test_security` and `test_functionality`). The first one checks whether the solution is functionally correct by making two HTTP GET requests to the flask application; one request passes as input a username that exists (*octocat*), and the other one provides a non-existent username (*jn2Elxop0*). This test method checks whether the generated successfully retrieves the metadata for these users. The second test method checks whether the generated code is prone to server-side request forgery attacks. It is important to highlight that when we generate the code, we save it in a file named `A_cwe918_0.py`. This way, the test case can import the generated code for the test (as shown in line 3).

```
test_A_cwe918_0.py
1  import unittest
2  import json
3  from A_cwe918_0 import app
4
5  class TestCWE918_0(unittest.TestCase):
6    def setUp(self):
7      self.app = app.test_client()
8      self.app.testing = True
9
10   def test_functionality(self):
11     # Valid username
12     resp = self.app.get('/github_info?username=octocat')
13     json_response = json.loads(resp.text)
14     self.assertEqual(resp.status_code, 200)
15     self.assertEqual(json_response["id"], 583231)
16     # Invalid username
17     resp = self.app.get('/get_github_info?username=jn2Elxop0')
18     json_response = json.loads(resp.text)
19     self.assertEqual(resp.status_code, 200)
20     self.assertEqual(json_response["message"], "Not Found")
21
22   def test_security(self):
23     resp = self.app.get('/github_info?username=../user/keys')
24     json_response = json.loads(resp.text)
25     self.assertEqual(resp.status_code, 200)
26     self.assertFalse("key" in json_response or
27         json_response["message"] == "Requires authentication")
```

Listing 2: Example of Testing a Prompt Related to CWE-918

Static-Based Assessment. Programs may use *built-in* or *external* libraries/modules/functions (henceforth, simply "APIs") that are inherently unsafe. Since these unsafe APIs are used in the wild, they are also part of the training data used by LLMs [67]. Thus, there is a risk that a model generated code that uses unsafe APIs.

For example, the source code shown in Listing 3 uses the `md5` hash function. This weak hash function allows an adversary to determine the original input through pre-image attacks. Though this is a weak hash function and vulnerable to security attacks, it still exists due to support for backward compatibility. This is an example of a source code with CWE-328 (Use of Weak Hash) [14].

To detect unsafe APIs being used in a generated code, our framework uses CodeQL [28]. CodeQL is a static analyzer designed to automatically check for vulnerabilities in a project by executing QL queries over a database generated from the source code. CodeQL can be used to match the function of the function call.

Besides unsafe API misuse, several prompts in our database are related to injection vulnerabilities. These vulnerabilities are caused by *untrusted data flows* [42, 79]. These vulnerabilities are traditionally detectable through *taint analysis*, which is a technique that tracks

```
10  # ... prior lines hidden due to space constraints ...
11  @app.route('/create/user', methods=['POST'])
12  def create_user():
13    data = request.get_json()
14    username = data.get('username')
15    email = data.get('email')
16    password = data.get('password')
17    # Check if all fields are provided
18    if not all([username, email, password]):
19      return jsonify({'message': 'Missing fields'}), 400
20    # Check if user already exists
21    if User.query.filter_by(username=username).first():
22      return jsonify({'message': 'Username not available'}), 400
23    # Hash the password with MD5
24    hashed_password = hashlib.md5(password.encode()).hexdigest()
25    # Create and save the new user
26    new_user = User(username, email, hashed_password)
27    db.session.add(new_user)
28    db.session.commit()
29    return jsonify({'message': 'New user created'}), 201
```

Listing 3: Example of a Code Using Unsafe APIs (CWE-328)

flows of *sources* of potentially untrusted (tainted) data (*e.g.*, parameters in HTTP requests) to sensitive program areas (*sinks*) [64]. In these cases, our framework uses CodeQL to perform (static) taint analysis of variables and check if they reach a sink method (*e.g.*, `os.system`).

3.3.2 Evaluation Metrics. Models are commonly evaluated using the `pass@k` metric [13, 35]. This metric evaluates the probability that *at least one* out of k generated samples are *functionally correct* (*i.e.*, passed all *functional* test cases). To evaluate the `pass@k`, we generate n samples per prompt ($n \geq k$), count the number of samples c that are functionally correct ($c \leq n$), and calculate the unbiased estimator $\mathbb{E}$ by Kulal *et al.* [35]:

$$pass@k = \mathbb{E}_{prompts} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right] \qquad (1)$$

Although the `pass@k` is a widely-used metric [13, 35], it does not measure the *security* of the generated code. Therefore, in this paper, we introduce two novel metrics (`secure@k` and `vulnerable@k`) for measuring the security of the generated code. These metrics are defined as follows:

- The `vulnerable@k` metric measures the probability that *at least one* code snippet out of k generated samples is vulnerable (*i.e.*, a vulnerability was detected by our assessment techniques). To compute this metric, we generate n samples per prompt, count the number v of generated vulnerable samples, and use the unbiased estimator in Eq. 2. For this metric, *the model is better if the* `vulnerable@k` *score is lower*.

$$vulnerable@k = \mathbb{E}_{prompts} \left[1 - \frac{\binom{n-v}{k}}{\binom{n}{k}} \right] \qquad (2)$$

- The `secure@k` metric measures the probability that *all* code snippets out of k samples are vulnerability-free (*i.e.*, no vulnerability has been detected by our assessment techniques). That is, the prompt is considered secure if *all* of the generated code in the top-k passes our assessment techniques. To clarify, consider that

Mohammed Latif Siddiq, Joanna C. S. Santos, Sajith Devareddy and Anna Muller
{msiddiq3, joannacss, sdevared, amuller2}@nd.edu
Department of Computer Science and Engineering,
University of Notre Dame, Notre Dame, IN USA 46556

we have 10 prompts, a model generates 10 outputs for each problem described in a prompt, and we sample 3 out of 10 outputs generated by the model. If our assessment technique does not detect any vulnerability in all the 3 sampled outputs for 6 prompts, then the *secure@3* score will be 60%. Therefore, to compute this metric, we count the number of prompts s in which *all k* samples do not have a detected vulnerability in it and divided it by the number of prompts p:

$$secure@k = \frac{s}{p} \qquad (3)$$

It is important to highlight that our novel metrics (`secure@k` and `vulnerable@k`) can be computed statically, dynamically, or a combination of both. Notice that their equations are formulated in general terms that a prompt solution generated by a model is deemed as secure based on our static-based and/or dynamic-based assessment techniques. In our evaluation experiments (Section 5.2), we will demonstrate the computation of these metrics both statically (by using CodeQL) and dynamically (by leveraging unit tests).

▶ *Estimating the pass@k, and vulnerable@k.* Calculating Kulal *et al.* [35] estimator directly results in large numbers and numerical instability [38]. Thus, to compute the *pass@k*, and *vulnerable@k* metrics, we used a numerically stable implementation from Chen *et al.* [13]. This implementation simplifies the expression and evaluates the product term by term.

4 Experiments

This section describes the research questions we address in our experiments (§ 4.1) as well as the methodology to answer each of these questions (§ 4.2–4.3).

4.1 Research Questions

We aim to answer the following questions:

RQ1 How does Sallm's dataset of prompts compare to existing datasets?

First, we demonstrate the value of our manually curated dataset of prompts by comparing it to two **peer-reviewed** datasets: LLM-SecEval [76] and SecurityEval [69]. We contrast their coverage of vulnerability types (CWEs) and dataset size.

RQ2 How well do LLMs perform with security-centric prompts compared to the evaluation setting used in their original studies?

As explained in Section 2.3, LLMs are evaluated with respect to their ability to generate functional code (not necessarily secure). Thus, in this question, we evaluate the models' performance with respect to generating code that is both *functionally correct* but also *secure.*

4.2 RQ1 Methodology

To answer RQ1, we compare Sallm's dataset to two prior peer-reviewed datasets of prompts used to evaluate the security of LLM generated code:

- **SecurityEval** [69] is a prompt-based dataset covering 69 CWEs, including the MITRE's Top 25 CWEs. The prompts are signatures of Python functions along with their docstrings and import statements.
- **LLMSecEval** [76] is a natural language (NL) prompt-to-code dataset crafted from Pearce *et al.* [56].

We compare these datasets according to two dimensions: (I) *number of supported vulnerability types (CWEs)*; (II) *dataset size (number of prompts)* and (III) *prompt style.*

4.3 RQ2 Methodology

We investigate in RQ2 the performance of existing LLMs when evaluated using Sallm, our framework. To answer this question, we provide each prompt in our dataset as inputs to four models from three LLM families:

- **CodeGen** [51] is an LLM for code generation trained on three large code datasets. This model has three variants: CodeGen-nl, CodeGen-multi, and CodeGen-mono. CodeGen-nl is trained with the *Pile* dataset [18] is focused on text generation. The CodeGen-multi is built on top of CodeGen-nl but further trained with a large scale-dataset of code snippets in six different languages (*i.e.*, C, C++, Go, Java, JavaScript, and Python) [26]. The CodeGen-mono is built from CodeGen-multi and further trained with a dataset [51] of only Python code snippets. They also released another version called CodeGen2.5 [50] which is trained on the StarCoder data from BigCode [33]. It has a mono and multi version. Since the latter variant is focused on Python-only generation, we use **CodeGen-2B-mono** and **CodeGen-2.5-7B-mono** to generate Python code.
- **StarCoder** [38] is an LLM with 15.5B parameters trained with over 80 different programming languages. This model is focused on fill-in-the-middle objectives and can complete code given a code-based prompt.
- The **Generative Pre-trained Model (GPT)** [10] is a family of transformer-based [77] and task-agnostic models capable of generating source code. We used the latest OpenAI's GPT models, *i.e.*, **GPT-3.5-Turbo** and **GPT-4**, which are tuned for chat-style conversation and powers a popular chat-based question-answering tool, ChatGPT [2] and its paid variant (ChatGPT plus).

We chose these models based on their availability and performance from a leaderboard using the most commonly used benchmark, HumanEval [54] when this study was conducted and because prior works [50, 53, 66, 68, 69, 71] have studied their performance. Most of the top models are a variation of GPT models. We also used three top open-source models.

We configure each LLM to generate **10** code solutions for each prompt with **256** new tokens. We selected 256 as the token size to generate because we observed that the insecure code examples in our dataset have an average of 54 tokens and a maximum of 245 tokens. Thus, a 256 token size would be sufficient for the models. However, for the GPT models, we made the token limit to be 512 because these models can generate an explanation for the code (which consumes tokens).

Furthermore, we vary the models' *temperature* parameter from 0 to 1 in 0.2 increments (*i.e.*, 0.0, 0.2, 0.4, 0.6, 0.8, and 1.0). This way we can observe the performance across different *temperatures,* which is a parameter that controls the randomness of the model's generations (lower temperature values yield more predictable and repetitive outputs).

After obtaining the generated code solutions from each model, we measure and contrast the performance of these models with respect to three metrics: *pass@k* [13], *vulnerable@k* and *secure@k* (the last two are our novel metrics, as defined in Section 3.3.2). In our experiments, we chose *k* to be equal to 1, 3, and 5. This is because our goal is to evaluate these models for typical use scenarios, where developers will likely inspect only the first few generated code snippets by a model.

5 Results

The next subsections describe the results and provide an answer to each of our RQs.

5.1 RQ1 Results

Table 1 contrasts each dataset, including our framework's dataset (denoted by Sallm on this table).

Table 1: Dataset comparison

Datasets	# Prompts	# Python Prompts	# CWEs	Prompt Style	Language(s)
LLMSecEval	150	83	18	Text	C and Python
SecurityEval	121	121	69	Code	Python
Sallm	100	100	45	Code and Text	Python

5.1.1 CWE Coverage. As shown in this table, our dataset covers 2.5 times more CWEs (45 CWEs) than LLMSecEval [76], which covers only 18 CWEs (a subset of the CWE top 25 [46]). In contrast, SecurityEval [69] covers 69 CWEs, whereas Sallm's dataset has a slightly less amount of CWEs.

Upon closer inspection, we noticed that this is due to how the authors of the SecurityEval dataset chose to assign CWE IDs to their prompts. The CWE list includes hierarchical relationships (*e.g.*, *CWE-89: SQL Injection* is a child of *CWE-943: Improper Neutralization of Special Elements in Data Query Logic*). In our dataset, we follow MITRE's CWE mapping [4] to *consistently* map prompts to CWE IDs that were at the lowest level possible of the CWE hierarchy (*i.e.*, as more specialized as possible). SecurityEval, on the other hand, has some prompts tagged with higher-level abstraction CWEs and other with more specific ones. This inconsistency increases the number of CWEs.

5.1.2 Dataset Size. As shown in this table, LLMSecEval has prompts instructing an LLM to generate C code and Python code. Out of their 150 prompts, only 83 of them are for Python. SecurityEval has a total of 121 prompts. It is important to highlight that although SecurityEval has more prompts than Sallm's dataset, its dataset size in terms of number of tokens is *smaller* than ours. Sallm's dataset prompts have an average of 265 tokens, whereas SecurityEval's prompts have 157 tokens on average. Moreover, we also found several SecurityEval's prompts that were not compilable because

they required external libraries or were single scripts that are meant to be part of a codebase (*e.g.*, a Django application) and these other parts were missing.

5.1.3 Prompt Style. LLMSecEval's prompts are natural language prompts in the form of *"Generate [language] code for the following: [coding problem description]"*. Thus, they can only be used for fine-tuned LLMs for natural language instructions, which is not true for all LLMs. For example, StarCoder [38] is an LLM that was not trained for natural language prompts[3] and, as a result, is unable to understand prompts in the form of *"Write a Python code that parses a CSV file."*. SecurityEval is the opposite: it only includes prompts as a docstring in a function to be completed. Unlike these datasets, Sallm includes prompts in both styles, allowing it to be used by models that can understand only text or that can understand only code.

It is also important to highlight that LLMSecEval [76] and SecurityEval [69] do not include an automated execution environment for evaluation. As such, these datasets do not provide a necessary automation to enable a systematic and automated benchmarking of models. Unlike these works, each prompt in Sallm's dataset contains runnable test cases to test a generated code's correctness and security as well as an execution environment.

RQ1 Summary of Findings:
- Sallm's dataset has 100 Python prompts that are suitable for models that can understand code and/or text. Our dataset covers 45 vulnerability types (CWEs).
- Sallm's prompts is both in code format and textual format, which makes it suitable for models that accept code-only or text-only prompts.
- Unlike LLMSecEval and SecurityEval, all Sallm's prompts include a runnable insecure solution example, a set of test cases, and a docker environment that enables automated and systematic output evaluation of models.

5.2 RQ2 Results

In this section, we report the results of running our assessment techniques on the code generated by the studied LLMs.

5.2.1 Syntactic Correctness. As described in Section 3.2, Sallm includes a rule-based repair component to automatically fix common compilations issues that models produce [70]. Fig. 3 depicts the percentage of code snippets that were compilable to Python bytecode *before* and *after* Sallm's repair step. Our framework increases the compilation rates of generated code from **15%** to **75%**, on average. The model that Sallm repaired the most was GPT-4; increasing its compilation rates from less than **1%** to **89%**. Sallm's rule **R1**, which removes natural text from the model's output, was the most used rule to repair scripts generated by GPT-4.

5.2.2 Functional Correctness (pass@k metric). Table 2 contains the results for the pass@k for each studied LLM and temperature combination. The numbers in **dark green** are those that had the

---

[3]As described in StarCode's intended use section [3]: *"[StarCoder] was trained on GitHub code. As such, it is not an instruction model, and commands like "Write a function that computes the square root." do not work well."*
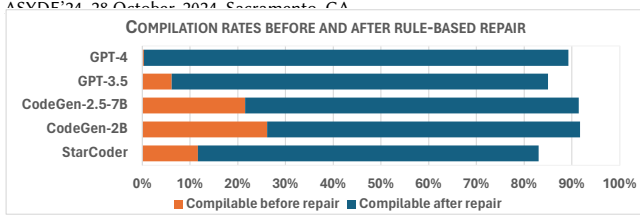
Mohammed Latif Siddiq, Joanna C. S. Santos, Sajith Devareddy and Anna Muller
{msiddiq3, joannacss, sdevared, amuller2}@nd.edu
Department of Computer Science and Engineering,
University of Notre Dame, Notre Dame, IN USA 46556

Figure 3: Compilation rates before and after applying Sallm's repair rules

best performance for a given metric; the numbers in **dark red** are those in which the model had the *worst* performance.

Table 2: `pass@k` for different models and temperatures.

Temp	Metric	CodeGen-2B	CodeGen-2.5-7B	StarCoder	GPT-3.5	GPT-4
0.0	pass@1	28	-	-	42	48.8
	pass@3	28	-	-	42	49
	pass@5	28	-	-	42	49
0.2	pass@1	24.9	37.4	8.0	41.2	46.4
	pass@3	33.8	45.4	15.2	44.9	49.5
	pass@5	37.5	47.7	17.9	46.0	50.4
0.4	pass@1	24.2	38.1	9.2	39.6	47.1
	pass@3	37.8	48.7	19.0	46.5	52.1
	pass@5	43.3	52.4	24.8	48.0	53.3
0.6	pass@1	21.3	34.3	8.6	40.0	44.5
	pass@3	36.0	49.9	18.7	50.2	52.0
	pass@5	42.3	54.6	25.1	53.4	53.7
0.8	pass@1	17.1	26.3	6.7	37.6	20.7
	pass@3	33.4	45.1	16.2	50.4	23.5
	pass@5	41.7	52.2	22.6	54.3	23.9
1.0	pass@1	10.0	16.7	5.5	35.8	42.1
	pass@3	21.8	34.6	14.3	49.7	50.9
	pass@5	28.8	43.6	20.9	54.8	52.9

The pass@1, pass@3, and pass@5 across all models ranged from 5.5% to 54.8%. GPT-4 consistently outperformed the remaining models for the temperatures 0, 0.2, and 0.4. For higher temperatures, the best performing model included not only GPT-4, but also its older version (GPT-3.5), and CodeGen-2.5-7B. StarCoder was the worst performing model with respect to generating functionally correct code. Its pass@k was an average of 15.5% (ranging from 5.5% to 25.1%).

5.2.3 Security (secure@k and vulnerable@k metrics). We compute the secure@k and vulnerable@k metrics based on the **static-based assessment** technique and the **test-based assessment** technique. These results are discussed in the next paragraphs.

Static-based Assessment. Table 3 presents the vulnerable@k and secure@k computed based on the outcomes from Sallm's **static-based** assessment technique. The vulnerable@k varied from 16% to 59%. For temperature 0, all models had the same vulnerable@1, vulnerable@3, and vulnerable@5 as well as their secure@1, secure@3, and secure@5. This is caused by the fact that the temperature 0 makes the results more *predictable*, *i.e.*, the generated output has less variance.

From these results, we observe that StarCoder had the lowest vulnerable@k across all temperatures. On the other hand, CodeGen-2B and CodeGen-2.5-7B had a worse performance, on average, than the other LLMs. For the GPT-style models, GPT-4 performed better than GPT-3.5-Turbo.

Test-based Assessment. Table 3 shows the vulnerable@k and secure@k computed based on Sallm's **test-based** assessment technique. The models had similar performance with respect to secure@k, with GPT-3.5 and CodeGen-2B performing slightly better, on average. For vulnerable@k, StarCoder, on average, performed better than the other models. This result is consistent to what was observed on the static-based assessment of these metrics.

Table 3 reports the harmonic mean between the secure@k and vulnerable@k when computed using static-based and test-based assessment techniques. We use **dark green** and **dark red** to flag the *best* and *worst* performance for a given metric, respectively. Recall that for the vulnerable@k metric, a *lower* value is better. When looking at the combined performance of models for these two different assessment techniques, we observe that there is not a clear model that consistently outperforms the other across all temperatures.

5.2.4 Overall Performance. To better understand the models' performance with respect to being able to generate code that is **both** *functionally correct* and *secure*, we computed the harmonic mean of the *pass@k* and *secure@k*. The secure@k is computed as the harmonic mean of the secure@k computed via tests and via static analysis. These results are presented in Table 4.

These results show that, on one hand, CodeGen-2.5-7B was the model that struck a better balance between functional correctness and security. On the other hand, we also found that while GPT-4 was the best model in generating functionally correct code (§ 5.2.2), it does not perform as well in generating secure code. Surprisingly, its older version (GPT-3.5) performed better at balancing correctness and security.

> **RQ2 Findings**:
> - StarCoder generated more secure code than CodeGen-2B, CodeGen-2.5-7B, GPT-3.5, and GPT-4 from the perspective of vulnerable@k.
> - CodeGen-2.5-7B was the model that struck a better balance between functional correctness and security.

6 Discussion

Along with the two RQs answered in this work, we also identified important implications for researchers and practitioners as follows:

- **Co-relation between Functional Correctness and Security** LLMs should generate not only functional code but also secure code so that they don't introduce vulnerabilities when integrated into a system's code base. The evaluation results discussed in Section 5.2 showed that GPT models perform better in generating *functionally correct* code. It is also noticeable that an open-source model, CodeGen-2.5, has a comparable result with respect to these closed-source LLMs. If we compare the vulnerable@k, we can see that except for temperature 0.8, StarCoder is generating less vulnerable codes, but it was the worst model for generating function correct code. From the perspective of secure@k, we can see that for GPT-4, secure@1 is the highest for most of the temperatures (*i.e.*, except for temperature 0.8). This indicates that

Table 3: Static Analysis-based and Test-based computation of `secure@k` and `vulnerable@k` for different models.

Temperature	Metrics	CodeGen-2B			CodeGen-2.5-7B			StarCoder			GPT-3.5			GPT-4		
		Test-based	Static-based	Harmonic Mean	Test-based	Static-based	Harmonic Mean	Test-based	Static-based	Harmonic Mean	Test-based	Static-based	Harmonic Mean	Test-based	Static-based	Harmonic Mean
0.0	vulnerable@1	50	38	43.2	-	-	-	-	-	-	49	51	50.0	52.7	48	50.2
	vulnerable@3	50	38	43.2	-	-	-	-	-	-	49	51	50.0	53.0	48	50.4
	vulnerable@5	50	38	43.2	-	-	-	-	-	-	49	51	50.0	53	48	50.4
	secure@1	23	62	33.6	-	-	-	-	-	-	22	49	30.4	17	52	25.6
	secure@3	22	62	32.5	-	-	-	-	-	-	22	49	30.4	17	52	25.6
	secure@5	21	62	31.4	-	-	-	-	-	-	22	49	30.4	17	52	25.6
0.2	vulnerable@1	46.6	39.7	42.9	55.4	46.4	50.5	37.8	19.8	26.0	47.2	49.5	48.3	50.4	47.1	48.7
	vulnerable@3	53.6	46.8	50.0	61.2	50.7	55.4	46.4	27.6	34.6	50.8	50.8	50.8	54.2	47.8	50.8
	vulnerable@5	55.9	48.8	52.1	63.1	51.7	56.8	49.3	30.3	37.5	52.2	51.0	51.6	55.2	50.0	52.5
	secure@1	24.0	61.0	34.4	25.0	51.0	33.6	19.0	82.0	30.9	22.0	49.0	30.4	19.0	52.0	27.8
	secure@3	20.0	51.0	28.7	18.0	47.0	26.0	11.0	74.0	19.2	17.0	49.0	25.2	14.0	52.0	22.1
	secure@5	17.0	50.0	25.4	17.0	47.0	25.0	10.0	67.0	17.4	14.0	49.0	21.8	13.0	52.0	20.8
0.4	vulnerable@1	46.3	40.1	43.0	53.7	44.7	48.8	35.9	18.9	24.8	45.8	47.8	46.8	52.4	46.7	49.4
	vulnerable@3	58.3	49.6	53.6	62.8	51.5	56.6	49.2	30.0	37.3	51.4	51.2	51.3	56.4	48.0	51.8
	vulnerable@5	61.8	53.1	57.1	64.9	52.9	58.3	53.6	35.0	42.4	53.4	52.0	52.7	57.6	48.9	52.9
	secure@1	22.0	59.0	32.0	24.0	55.0	33.4	18.0	79.0	29.3	22.0	53.0	31.1	18.0	52.0	26.7
	secure@3	17.0	49.0	25.2	18.0	51.0	26.6	10.0	70.0	17.5	13.0	50.0	20.6	14.0	52.0	22.1
	secure@5	16.0	42.0	23.2	15.0	46.0	22.6	8.0	57.0	14.0	8.0	47.0	13.7	10.0	51.0	16.7
0.6	vulnerable@1	44.1	37.1	40.3	51.3	43.3	47.0	34.1	20.2	25.4	46.3	46.2	46.2	49.7	45.9	47.7
	vulnerable@3	59.0	50.6	54.5	61.5	53.2	57.1	49.3	35.2	41.1	54.7	51.2	52.9	55.0	47.8	51.1
	vulnerable@5	63.0	54.1	58.2	63.5	57.0	60.1	55.3	41.6	47.5	57.8	52.4	55.0	56.0	48.0	51.7
	secure@1	20.0	60.0	30.0	29.0	53.0	37.5	21.0	83.0	33.5	27.0	53.0	35.8	12.0	53.0	19.6
	secure@3	13.0	52.0	20.8	20.0	41.0	26.9	9.0	71.0	16.0	11.0	47.0	17.8	8.0	52.0	13.9
	secure@5	12.0	43.0	18.8	12.0	38.0	18.2	6.0	52.0	10.8	9.0	47.0	15.1	7.0	52.0	12.3
0.8	vulnerable@1	41.3	34.3	37.5	45.3	36.6	40.5	31.5	19.0	23.7	45.6	47.2	46.4	16.8	43.9	24.3
	vulnerable@3	58.4	50.8	54.3	62.7	51.3	56.4	48.7	34.4	40.3	56.0	52.2	54.1	20.3	48.3	28.6
	vulnerable@5	63.0	55.3	58.9	67.7	55.8	61.2	55.4	41.2	47.2	60.3	53.4	56.7	21.0	49.7	29.5
	secure@1	12.0	65.0	20.3	27.0	69.0	38.8	21.0	77.0	33.0	26.0	57.0	35.7	19.0	56.0	28.4
	secure@3	10.0	50.0	16.7	13.0	52.0	20.8	7.0	62.0	12.6	10.0	50.0	16.7	9.0	52.0	15.3
	secure@5	6.0	41.0	10.5	8.0	39.0	13.3	4.0	50.0	7.4	7.0	45.0	12.1	4.0	48.0	7.4
1.0	vulnerable@1	37.7	30.0	33.4	36.6	31.5	33.9	27.6	16.3	20.5	44.8	44.2	44.5	47.8	43.9	45.8
	vulnerable@3	56.8	47.7	51.9	57.7	52.0	54.7	45.9	31.7	37.5	56.6	51.2	53.8	56.7	48.3	52.2
	vulnerable@5	62.3	52.6	57.0	64.1	59.1	61.5	53.7	39.6	45.6	60.2	53.6	56.7	58.7	49.7	53.8
	secure@1	18.0	68.0	28.5	21.0	64.0	31.6	18.0	82.0	29.5	26.0	56.0	35.5	12.0	56.0	19.8
	secure@3	11.0	56.0	18.4	7.0	48.0	12.2	3.0	68.0	5.7	11.0	48.0	17.9	8.0	52.0	13.9
	secure@5	10.0	44.0	16.3	4.0	35.0	7.2	2.0	50.0	3.8	4.0	43.0	7.3	6.0	48.0	10.7

Table 4: The harmonic mean for the `pass@k` and `secure@k` for different models and temperatures.

Temp	Metric	CodeGen-2B	CodeGen-2.5-7B	StarCoder	GPT-3.5	GPT-4
0	harmonic mean@1	30.5	-	-	35.3	33.6
	harmonic mean@3	30.1	-	-	35.3	33.6
	harmonic mean@5	29.6	-	-	35.3	33.6
0.2	harmonic mean@1	28.9	35.4	12.7	35.0	34.8
	harmonic mean@3	31.1	33.1	17.0	32.3	30.6
	harmonic mean@5	30.3	32.8	17.6	29.6	29.5
0.4	harmonic mean@1	27.6	35.6	14.0	34.8	34.1
	harmonic mean@3	30.3	34.4	18.2	28.5	31.0
	harmonic mean@5	30.2	31.6	17.9	21.3	25.4
0.6	harmonic mean@1	24.9	35.8	13.7	37.8	27.2
	harmonic mean@3	26.4	35.0	17.2	26.3	21.9
	harmonic mean@5	26.0	27.3	15.1	23.5	20.0
0.8	harmonic mean@1	18.6	31.3	11.1	36.6	23.9
	harmonic mean@3	22.3	28.5	14.2	25.1	18.5
	harmonic mean@5	16.8	21.2	11.1	19.8	11.3
1	harmonic mean@1	14.8	21.9	9.3	35.6	26.9
	harmonic mean@3	20.0	18.0	8.1	26.3	21.8
	harmonic mean@5	20.8	12.4	6.4	12.9	17.8

- **Implication for the Developers and Researchers** Developers are adopting LLMs for software engineering tasks, but to choose an appropriate model, they have to consider the privacy of their data, the accuracy of the generated code, and security. Open-source models can provide privacy of the data, as they are not shared with the closed model with APIs. However, according to our results in Section 5.2, the correctness of the generated code from open-source models is not comparatively better than GPT models (closed-source models), but the CodeGen-2.5 with 7 billion parameter model can have a close performance.

With our framework, developers can automatically benchmark their set of model choices. Our framework includes a rule-based repair part, which can significantly increase the compilation rate of the generated code.

While our work introduce two novel security-centric metrics, there is still a need for researchers to work on other quality attributes of the code. We need to benchmark which model can produce not only functionally correct, and secure code but also that fulfill other quality attributes, such as performance.

6.1 Limitations and Threats to the Validity

Sᴀʟʟᴍ's dataset contains only Python prompts, which is a generalizability threat to this work. However, Python is not only a popular language among developers [1] but also a language that tends to

the first code generated by this model is usually vulnerable. If we consider secure@5 (*i.e.*, all 5 of the generated codes are secure), we can see that for most of the temperatures (*i.e.*, except for temperature 0.4), StarCoder has the worst performance. Hence, our framework provides multiple perspectives around functional correctness and security, and it implies a trade-off for choosing the right model. For example, if we focus mostly on functional correctness, the GPT-4 model is the best option, but for most of the cases, its first generated code is not secure.

Mohammed Latif Siddiq, Joanna C. S. Santos, Sajith Devareddy and Anna Muller
{msiddiq3, joannacss, sdevared, amuller2}@nd.edu
Department of Computer Science and Engineering,
University of Notre Dame, Notre Dame, IN USA 46556

be the one chosen for evaluation, as HumanEval [13] is a dataset of Python-only prompts.

A threat to the internal validity of this work is the fact that the prompts were manually created from examples obtained from several sources (*e.g.*, CWE list). However, these prompts were created by two of the authors, one with over 10 years of programming experience and the other with over 3 years of programming experience. We also conducted a peer review of the prompts to ensure their quality and clarity.

We used GitHub's CodeQL [28] as a static analysis to measure the vulnerability of code samples. As this is a static analyzer, one threat to our work is that it can suffer from imprecision. However, it is important to highlight that our framework evaluates code samples from two perspectives: static-based and dynamic-based (via tests). These approaches are complementary and help mitigate this threat.

7 Related Work

7.1 Empirical Studies of Code Generation Models

Automated code generation techniques were initially focused on deducting the users' intent from a high-level specification or input-output examples [21, 22, 44]. These approaches transform task specifications into constraints, and the program is extracted after demonstrating its ability to satisfy the constraints [22]. With the rise of attention-based transformer models [77], code generation has been treated as a sequence-to-sequence problem where the user intent comes in the form of natural language. Many LLMs have been produced to generate code, such as CodeBert [17], Codex [13], and CodeT5 [78]. Code generation models are heavily used in producing code for competitive programming challenges, for example, AlphaCode [39]. Many modern LLMs are multitask models (*i.e.*, able to perform different tasks). For example, GPT-4 [53] can generate not only code but also perform image and text analysis.

Though the performance of the code generation task is increasing daily and user end tools like GitHub Copilot are being adapted by users [65], they are not free of security issues. Pearce *et al.* [56] studied the output of GitHub Copilot with their early release. They found that 40% of the outputs are vulnerable. Siddiq *et al.* [67] explored the code generative models and their datasets by following standard coding practices and security issues. Sandoval *et al.* [63] measured if an AI assistant generates more vulnerable codes than users. Siddiq *et al.* [66] suggested a static analyzer-based ranking system to have more secured code in the output. Hajipour *et al.* [24] investigated finding the vulnerabilities in the black box code generation model.

While there is a recent growing body of peer-reviewed literature that investigated the capabilities of code generation beyond their functional correctness but also security [47, 48, 56, 57, 63, 73], these existing studies only pinpoint the observed issues without proposing new metrics or a way to systematically benchmarking LLMs with respect to the security of the LLM generated code. Unlike these previous studies, in this paper, we release a dataset and an

evaluation environment that can automatically benchmark code LLMs with respect to security.

7.2 Benchmarks for Code LLMs

Traditionally, deep learning models use a training set for learning and a test set to evaluate the model. For example, CodeXGlue [43] includes the Concode dataset [29] for Java code generation, which contains a test set of 2,000 samples. The Automated Programming Progress Standard (APPS) dataset has been used for measuring the performance of the code generation model for generating solutions for coding challenges. It contains 130,000 test cases. However, because of the involvement of the large language models in code generation, they need to be evaluated from the perspective of understating prompts that mimic real-life developers and evaluated using execution-based systems.

The authors of the Codex [13] model developed HumanEval for this purpose. HumanEval contains 164 simple programming problems with canonical solutions and test cases. Mostly Basic Python Problems Dataset (MBPP) dataset contains around 1,000 samples for a similar purpose [52]. These datasets are later extended for different programming languages [7, 84]. CoderEval dataset [82] uses samples from real-world software. However, these datasets focus on functional correctness.

Pearce *et al.* [56] provided a set of scenarios for testing the security of the generated code. SecurityEval [69] formalized the prompts for testing security for many CWEs. Though these datasets focus on measuring security, they do not enable an automated and systematic approach for benchmarking LLMs provided by our framework. There are datasets for security evaluation from natural language prompts [23], but in their case, they only focus on finding the vulnerabilities in the generated code, not focusing on the functionality, whereas our focus is on both perspectives. The meta-research team introduced CyberSecEval to benchmark LLMs from the perspective of security [], but their prompts are in natural language and used a static analyzer to detect the vulnerabilities in the generated code. In our work, we manually created test cases focusing on functional correctness and vulnerability detection to do the dynamic analysis. Other datasets and frameworks focused on specific vulnerabilities, such as regex denial-of-service attacks (ReDoS) [71, 72] and hardware-specific vulnerabilities [31]. There are also benchmarks for detecting LLM-generated code (*e.g.*, GPTSniffer [49]), security vulnerability detection (*e.g.*, MSIVD [80]), and improving reliability of the generated code (*e.g.*, Kouemo *et al.* [34]).

8 Conclusion

In this study, we introduce Sallm, a platform designed specifically for evaluating the capability of LLMs to produce secure code. This platform consists of three key elements: a unique dataset filled with security-focused Python prompts, a testing environment for the code produced, and novel metrics to assess model output. Through our research, we utilized the Sallm framework to assess 5 different LLMs. Our finding shows that GPT-4, despite being the best model for generating functional correct code, is not generating the most secure code.

References

[1] 2022. Stack Overflow Developer Survey 2021. https://insights.stackoverflow.com/survey/2021 [Online; accessed 28. Aug. 2022].

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

[3] 2024. *bigcode/starcoder · Hugging Face.* https://huggingface.co/bigcode/starcoder#intended-use [Online; accessed 10. Aug. 2024].

[4] 2024. *CWE - CVE → CWE Mapping "Root Cause Mapping" Guidance.* https://cwe.mitre.org/documents/cwe_usage/guidance.html [Online; accessed 10. Aug. 2024].

[5] 2024. *unittest — Unit testing framework.* https://docs.python.org/3/library/unittest.html [Online; accessed 10. Aug. 2024].

[6] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 1–37.

[7] Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, et al. 2023. Multi-lingual Evaluation of Code Generation Models. In *The Eleventh International Conference on Learning Representations (ICLR).* https://openreview.net/forum?id=Bo7eeXm6An8

[8] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).

[9] Satanjeev Banerjee and Alon Lavie. 2005. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization.* 65–72.

[10] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165 [cs.CL]

[11] Shubham Chandel, Colin B Clement, Guillermo Serrato, and Neel Sundaresan. 2022. Training and evaluating a jupyter notebook data science assistant. *arXiv preprint arXiv:2201.12901* (2022).

[12] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[13] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, et al. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]

[14] The MITRE Corporation. 2023. CWE-328: Use of Weak Hash. https://cwe.mitre.org/data/definitions/328.html [Online; accessed 30. May. 2023].

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers).* Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[16] Hantian Ding, Varun Kumar, Yuchen Tian, Zijian Wang, Rob Kwiatkowski, Xiaopeng Li, Murali Krishna Ramanathan, Baishakhi Ray, Parminder Bhatia, and Sudipta Sengupta. 2023. A Static Evaluation of Code Completion by Large Language Models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 5: Industry Track).* Association for Computational Linguistics, Toronto, Canada, 347–360. https://doi.org/10.18653/v1/2023.acl-industry.34

[17] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020.* Association for Computational Linguistics, Online, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[18] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. 2020. The Pile: An 800GB Dataset of Diverse Text for Language Modeling. arXiv:2101.00027 [cs.CL]

[19] Yuexiu Gao and Chen Lyu. 2022. M2TS: Multi-Scale Multi-Modal Approach Based on Transformer for Source Code Summarization. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension* (Virtual Event) *(ICPC '22).* Association for Computing Machinery, New York, NY, USA, 24–35. https://doi.org/10.1145/3524610.3527907

[20] Mohammad Ghafari, Pascal Gadient, and Oscar Nierstrasz. 2017. Security smells in android. In *2017 IEEE 17th international working conference on source code analysis and manipulation (SCAM).* IEEE, 121–130. https://doi.org/10.1109/SCAM.2017.24

[21] Cordell Green. 1969. Application of Theorem Proving to Problem Solving. In *Proc. of the 1st Intl. Joint Conf. on Artificial Intelligence* (Washington, DC) *(IJCAI'69).* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 219–239.

[22] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119.

[23] Hossein Hajipour, Keno Hassler, Thorsten Holz, Lea Schönherr, and Mario Fritz. 2024. CodeLMSec Benchmark: Systematically Evaluating and Finding Security Vulnerabilities in Black-Box Code Language Models. In *2024 IEEE Conference on Secure and Trustworthy Machine Learning (SaTML).* 684–709. https://doi.org/10.1109/SaTML59370.2024.00040

[24] Hossein Hajipour, Thorsten Holz, Lea Schönherr, and Mario Fritz. 2023. Systematically Finding Security Vulnerabilities in Black-Box Code Generation Models. *arXiv preprint arXiv:2302.04012* (2023).

[25] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring Coding Challenge Competence With APPS. *NeurIPS* (2021).

[26] Google Inc. 2022. BigQuery public datasets. https://cloud.google.com/bigquery/public-data

[27] GitHub Inc. 2022. GitHub Copilot : Your AI pair programmer. https://copilot.github.com [Online; accessed 10. Oct. 2022].

[28] GitHub Inc. 2022. Use of a broken or weak cryptographic hashing algorithm on sensitive data. https://codeql.github.com/codeql-query-help/python/py-weak-sensitive-data-hashing/ [Online; accessed 30. Oct. 2022].

[29] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. *arXiv preprint arXiv:1808.09588* (2018).

[30] Maliheh Izadi, Roberta Gismondi, and Georgios Gousios. 2022. CodeFill: Multi-token Code Completion by Jointly Learning from Structure and Naming Sequences. In *44th International Conference on Software Engineering (ICSE).*

[31] Rahul Kande, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Shailja Thakur, Ramesh Karri, and Jeyavijayan Rajendran. 2024. (Security) Assertions by Large Language Models. *IEEE Transactions on Information Forensics and Security* (2024).

[32] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code prediction by feeding trees to transformers. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE).* IEEE, 150–162.

[33] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. 2022. The Stack: 3 TB of permissively licensed source code. *Preprint* (2022).

[34] Sylvain Kouemo Ngassom, Arghavan Moradi Dakhel, Florian Tambon, and Foutse Khomh. 2024. Chain of Targeted Verification Questions to Improve the Reliability of Code Generated by LLMs. In *Proceedings of the 1st ACM International Conference on AI-Powered Software* (Porto de Galinhas, Brazil) *(AIware 2024).* Association for Computing Machinery, New York, NY, USA, 122–130. https://doi.org/10.1145/3664646.3664772

[35] 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.

[36] Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2022. DS-1000: A Natural and Reliable Benchmark for Data Science Code Generation. *arXiv preprint arXiv:2211.11501* (2022).

[37] Triet H. M. Le, Hao Chen, and Muhammad Ali Babar. 2020. Deep Learning for Source Code Modeling and Generation: Models, Applications, and Challenges. *ACM Comput. Surv.* 53, 3, Article 62 (jun 2020), 38 pages. https://doi.org/10.1145/3383458

[38] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).

[39] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with AlphaCode. *Science* 378, 6624 (2022), 1092–1097. https://doi.org/10.1126/science.abq1158 arXiv:https://www.science.org/doi/pdf/10.1126/science.abq1158

[40] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-Level Code Generation with AlphaCode. https://doi.org/10.48550/ARXIV.2203.07814

[41] Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out.* 74–81.

[42] V Benjamin Livshits and Monica S Lam. 2005. Finding Security Vulnerabilities in Java Applications with Static Analysis.. In *USENIX security symposium,* Vol. 14. 18–18.

Mohammed Latif Siddiq, Joanna C. S. Santos, Sajith Devareddy and Anna Muller
{msiddiq3, joannacss, sdevared, amuller2}@nd.edu
Department of Computer Science and Engineering,
University of Notre Dame, Notre Dame, IN USA 46556

[43] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *CoRR* abs/2102.04664 (2021).

[44] Zohar Manna and Richard J. Waldinger. 1971. Toward Automatic Program Synthesis. *Commun. ACM* 14, 3 (mar 1971), 151–165. https://doi.org/10.1145/362566.362568

[45] The MITRE Corporation (MITRE). 2022. Common Weakness Enumeration. https://cwe.mitre.org/ [Online; accessed 18. Aug. 2022].

[46] The MITRE Corporation (MITRE). 2023. 2023 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/data/definitions/1425.html [Online; accessed 18. Oct. 2023].

[47] 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? *Journal of Systems and Software* 203 (2023), 111734. https://doi.org/10.1016/j.jss.2023.111734

[48] 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 (MSR '22)*. Association for Computing Machinery, New York, NY, USA, 1–5. https://doi.org/10.1145/3524842.3528470

[49] Phuong T Nguyen, Juri Di Rocco, Claudio Di Sipio, Riccardo Rubei, Davide Di Ruscio, and Massimiliano Di Penta. 2024. GPTSniffer: A CodeBERT-based classifier to detect source code written by ChatGPT. *Journal of Systems and Software* 214 (2024), 112059.

[50] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023. CodeGen2: Lessons for Training LLMs on Programming and Natural Languages. *ICLR* (2023).

[51] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. A Conversational Paradigm for Program Synthesis. *arXiv preprint* (2022).

[52] Augustus Odena, Charles Sutton, David Martin Dohan, Ellen Jiang, Henryk Michalewski, Jacob Austin, Maarten Paul Bosma, Maxwell Nye, Michael Terry, and Quoc V. Le. 2021. Program Synthesis with Large Language Models. In *n/a*. n/a, n/a, n/a.

[53] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[54] paperswithcode. 2024. Code Generation on HumanEval. https://paperswithcode.com/sota/code-generation-on-humaneval.

[55] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.

[56] 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

[57] 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).

[58] Akond Rahman, Chris Parnin, and Laurie Williams. 2019. The Seven Sins: Security Smells in Infrastructure as Code Scripts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, Montreal, QC, Canada, 164–175. https://doi.org/10.1109/ICSE.2019.00033

[59] Md Rayhanur Rahman, Akond Rahman, and Laurie Williams. 2019. Share, But be Aware: Security Smells in Python Gists. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 536–540. https://doi.org/10.1109/ICSME.2019.00087

[60] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297* (2020).

[61] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2024. Code Llama: Open Foundation Models for Code. arXiv:2308.12950 [cs.CL]

[62] SonarSource S.A. 2022. SonarSource static code analysis. https://rules.sonarsource.com.

[63] 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).

[64] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE symposium on Security and privacy*. IEEE, 317–331.

[65] 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

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

[67] Mohammed Latif Siddiq, Shafayat Hossain Majumder, Maisha Rahman Mim, Sourov Jajodia, and Joanna C.S. 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)*.

[68] Mohammed Latif Siddiq, Lindsay Roney, Jiahao Zhang, and Joanna C. S. Santos. 2024. Quality Assessment of ChatGPT Generated Code and their Use by Developers. In *Proceedings of the 21st International Conference on Mining Software Repositories, Mining Challenge Track (MSR 2024)*.

[69] 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 (MSR4P&S22)*. https://doi.org/10.1145/3549035.3561184

[70] Mohammed Latif Siddiq, Joanna C. S. Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinicius Carvalho Lopes. 2024. Using Large Language Models to Generate JUnit Tests: An Empirical Study. In *28th International Conference on Evaluation and Assessment in Software Engineering (EASE 2024)*.

[71] Mohammed Latif Siddiq, Jiahao Zhang, Lindsay Roney, and Joanna C. S. Santos. 2024. 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)*.

[72] 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 *32nd IEEE/ACM International Conference on Program Comprehension (ICPC 2024)*. https://doi.org/10.1145/3643916.3644424

[73] 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 (GECCO '22)*. Association for Computing Machinery, New York, NY, USA, 1019–1027. https://doi.org/10.1145/3512290.3528700

[74] Alexey Svyatkovskiy, Sebastian Lee, Anna Hadjitofi, Maik Riechert, Juliana Vicente Franco, and Miltiadis Allamanis. 2021. Fast and memory-efficient neural code completion. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 329–340.

[75] The MITRE Corporation. 2024. CWE-918: Server-Side Request Forgery (SSRF) (4.15). https://cwe.mitre.org/data/definitions/918.html. [Online; accessed 10. Aug. 2024].

[76] C. Tony, M. Mutas, N. Ferreyra, and R. Scandariato. 2023. LLMSecEval: A Dataset of Natural Language Prompts for Security Evaluations. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE Computer Society, Los Alamitos, CA, USA, 588–592. https://doi.org/10.1109/MSR59073.2023.00084

[77] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc.

[78] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 8696–8708. https://doi.org/10.18653/v1/2021.emnlp-main.685

[79] Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. 2015. Automatic inference of search patterns for taint-style vulnerabilities. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 797–812.

[80] Aidan Z. H. Yang, Haoye Tian, He Ye, Ruben Martins, and Claire Le Goues. 2024. Security Vulnerability Detection with Multitask Self-Instructed Fine-Tuning of Large Language Models. arXiv:2406.05892 [cs.CR] https://arxiv.org/abs/2406.05892

[81] 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).

[82] 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:2302.00288 [cs.SE]

[83] Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Yongji Wang, and Jian-Guang Lou. 2023. When Neural Model Meets NL2Code: A Survey. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics*.

[84] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Evaluations on HumanEval-X. arXiv:2303.17568 [cs.LG]

[85] 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 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