CFCEval: Evaluating Security Aspects in Code Generated by Large Language Models

Cheng Cheng
Department of Computer Science and
Software Engineering, Concordia University,
Montreal, Canada
cheng.cheng.20171@mail.concordia.ca

Jinqiu Yang
Department of Computer Science and
Software Engineering, Concordia University,
Montreal, Canada
jinqiu.yang@concordia.ca

***Abstract*—Code-focused Large Language Models (LLMs), such as CodeX and Star-Coder, have demonstrated remarkable capabilities in enhancing developer productivity through context-aware code generation. However, evaluating the quality and security of LLM-generated code remains a significant challenge. Existing evaluation protocols for Code LLMs lack both methodological rigor and comprehensive scope. A key limitation is dataset bias, which arises from unintentional overlap between training and testing data. Furthermore, while CodeBLEU, a BLEU-based metric, is widely used to assess code similarity, it suffers from critical shortcomings, including imprecise tokenization, structural limitations, and low reference diversity. To address these challenges, we introduce CFCEval, a novel framework for evaluating the quality and security of code generated by LLMs. CFCEval mitigates dataset bias by creating a new benchmark, MLVBench, and incorporates ELRM, a new metric designed to assess the relevance between reference code and generated code. CFCEval evaluates generated code across four dimensions: programming quality, vulnerability-fixing capability, post-transformation fixing capability, and relevance. Our experiments show that CFCEval not only captures both quality and security aspects of generated code more effectively but also that its ELRM aligns more closely with human judgments than CodeBLEU, thus paving the way for future advancements in Code LLMs evaluation.**

***Index Terms*—large language model, security, vulnerability repair, generated code, metric**

I. Introduction

Recent advancements in large language models (LLMs) [1]–[3] have significantly improved performance across various natural language processing tasks, including text generation, machine translation, and question answering. Building on these successes, domain-specific LLMs, or code-focused LLMs, have emerged to address programming-related tasks. Models such as DeepSeek [1], CodeX [4], Code Llama [5], and StarCoder2 [6] excel at generating syntactically correct and semantically meaningful code, completing functions, and synthesizing full code blocks. Integrated into tools like GitHub Copilot and CodeGeeX, these models serve as intelligent coding assistants, enhancing developer efficiency, reducing repetitive tasks, and supporting rapid prototyping. Their widespread use marks a shift towards AI-augmented software engineering workflows.

Code-focused large language models (LLMs) are typically evaluated using benchmark datasets and metrics tailored to specific code-related tasks. To assess the functional correctness of generated code, datasets such as HumanEval [4] and DS-1000 [7] are widely adopted, where models are evaluated based on whether their outputs pass a predefined set of test cases. In addition, evaluation metrics such as Exact Match (EM) [8], perfect accuracy, BLEU [9], and CodeBLEU [10] are commonly used to quantify performance in tasks including text-to-code synthesis, code translation, and code change prediction [4]. These metrics primarily measure the degree of similarity between the generated code and the reference solution, typically at the token or n-gram level. Among them, CodeBLEU extends BLEU by incorporating additional structural features, such as abstract syntax tree (AST) match and data flow consistency, and has demonstrated improved alignment with semantic correctness in code-related evaluations.

Despite the widespread adoption of datasets and evaluation metrics, current assessment methodologies for code-focused large language models (LLMs) suffer from critical limitations in both methodological rigor and scope. Specifically, in code completion tasks, especially from a security perspective, two major challenges persist: dataset bias and deficiencies in metrics related to code structure, semantic correctness, and prediction diversity.

Training-testing dataset overlap, particularly in benchmarks derived from public repositories like GitHub, inflates performance estimates due to memorization rather than true generalization. Existing metrics, such as CodeBLEU, fail to capture crucial aspects of code quality and security. These limitations include imprecise tokenization (e.g., treating code without whitespace as a single token), structural issues (e.g., short code fragments not forming valid Abstract Syntax Trees or data-flow graphs), and low reference diversity (e.g., relying on a single ground-truth reference despite multiple valid repair predictions). Consequently, these shortcomings hinder the ability to fully assess the quality and security of code generated by LLMs. These limitations obscure the true capabilities of code LLMs and call for more principled, generalization-sensitive, and semantically aware evaluation frameworks.

To address these limitations, we introduce the Code Fix Capability Evaluation (CFCEval) framework, a novel methodology designed to assess both the quality and security of code LLMs. CFCEval mitigates training-testing data overlap bias by leveraging a new evaluation dataset, MLVBench, which introduces input perturbations through techniques such as variable renaming and structural refactoring. This enables the evaluation of model generalization on transformed inputs that remain semantically equivalent to the original ones. The CFCEval framework evaluates code LLMs across four key dimensions: Programming Language Quality, Fixing Capability, Post-Transformation Fixing Capability, and Element-Level Relevance. These dimensions are designed to provide a comprehensive assessment of code generation models, considering both correctness and security aspects. To assess Element-Level Relevance, we introduce the Element-Level Relevance Metric (ELRM), which quantifies the degree of similarity between the generated code and the secure reference code. ELRM evaluates the relevance by analyzing keyword and operator matches, as well as string literal similarity, allowing for a fine-grained comparison of how well the generated code aligns with the intended secure solution. This metric enables a deeper understanding of the models performance, going beyond surface-level syntactic comparisons to consider the semantic accuracy of the generated fixes.

We perform extensive experiments to assess the effectiveness of the CFCEval framework and the Element-Level Relevance Metric (ELRM), and their correlation with evaluation scores from intelligent code LLMs on generated code. These experiments are designed to evaluate both the quality and security aspects of the code produced by the models, as well as the overall performance of the CFCEval framework in assessing these dimensions, addressing the research questions outlined in Section III. The results demonstrate that CFCEval, through the use of ELRM, effectively differentiates the performance of code LLMs, capturing both fine-grained code quality and the security aspects of the generated fixes. Furthermore, ELRM shows a stronger alignment with human-assigned LLM quality scores, outperforming commonly used evaluation metrics such as CodeBLEU and BLEU. To further validate the robustness of CFCEval as an automatic evaluation benchmark, we also incorporate GPT-based metrics, providing additional support for the framework's ability to evaluate code generation in a comprehensive and reliable manner. In addition to these experiments, we present a pilot study in Section IV that further validates the CFCEval framework's effectiveness in real-world vulnerability repair tasks, demonstrating its ability to assess code generation models in a practical context.

The paper is organized as follows. Section II introduces the CFCEval framework, outlining its benchmark, evaluation dimensions, and the ELRM metric. Section III presents the experimental results, addressing the research questions and key findings. Section V discusses the threats to validity, including limitations in the experimental setup and potential biases. Section IV provides a pilot case study demonstrating the framework's application in a real-world context. Section VI reviews related work, highlighting differences between existing methods and the proposed approach. Finally, Section VII summarizes the main findings and suggests directions for future research.

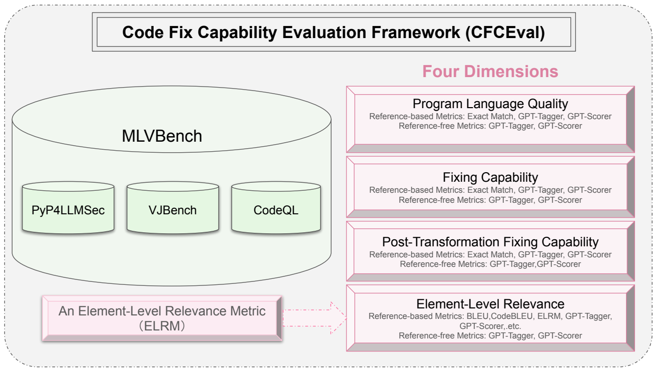

Code Fix Capability Evaluation Framework (CFCEval)
Four Dimensions
MLVBench
PyP4LLMSec
VJBench
CodeQL
An Element-Level Relevance Metric (ELRM)
Program Language Quality
Reference-based Metrics: Exact Match, GPT-Tagger, GPT-Scorer
Reference-free Metrics: GPT-Tagger, GPT-Scorer
Fixing Capability
Reference-based Metrics: Exact Match, GPT-Tagger, GPT-Scorer
Reference-free Metrics: GPT-Tagger, GPT-Scorer
Post-Transformation Fixing Capability
Reference-based Metrics: Exact Match, GPT-Tagger, GPT-Scorer
Reference-free Metrics: GPT-Tagger,GPT-Scorer
Element-Level Relevance
Reference-based Metrics: BLEU,CodeBLEU, ELRM, GPT-Tagger, GPT-Scorer,.etc.
Reference-free Metrics: GPT-Tagger, GPT-Scorer

Fig. 1: Overview of the CFCEval framework, highlighting the dataset, the four evaluation dimensions, and the corresponding metrics used to assess the code quality and security of Code LLMs.

II. Code Fix Capability Evaluation Framework

To address the limitations discussed in Section I, namely, training-testing data overlap and the lack of meaningful evaluation metrics, we propose the Code Fix Capability Evaluation (**CFCEval**) framework, a structured approach for evaluating the quality and security of Code LLMs.

CFCEval consists of three integral components: (1) A bias-mitig-ating **dataset** created through semantic-preserving code transformations; (2) A set of evaluation **dimensions** that collectively assess programming quality, vulnerability repairability, transformation robustness, and relevance; (3) An evaluation metric, **ELRM**, built upon CodeBLEU and BLEU, designed to evaluate the relevance dimension. Each component is introduced in turn in the following subsections.

A. Dataset Construction for Bias Mitigation

As discussed in the introduction, one critical limitation in existing evaluation protocols is the presence of dataset bias, which stems from unintentional overlap between training and testing data. This issue is particularly pronounced in code LLMs, which are typically pre-trained on large-scale open-source repositories such as GitHub. Many benchmark datasets are constructed from the same or similar codebases, often containing duplicated snippets, common templates, or boilerplate code structures. Consequently, models may achieve artificially high scores by memorizing seen patterns, rather than demonstrating true generalization. Such bias not only undermines the credibility of evaluation results but also obscures the model's ability to handle novel or obfuscated inputs-especially in security-critical settings. To address this issue, our evaluation framework introduces a dedicated dataset, MLVBench, designed with controlled code perturbations (e.g., variable renaming and structural refactoring) to produce vulnerability instances that are semantically equivalent but distributionally shifted from typical training data.

Dataset Construction. Several benchmarks have been developed for automated vulnerability repair, including Vul4J [11] for Java, Vul4C [12] for C/C++, and CVE-Bench [13], which covers multiple CVE-based samples. However, most of

TABLE I: Dataset Statistics

	PyP4LLMSec Python	VJBench Java	CodeQL C++	CodeQL Ruby
Selected Vulnerability	7	12	7	7
Original Vulnerability	7	12	7	7
Renamed Vulnerability	7	12	7	7
Restructured Vulnerability	5	12	2	1
Rename and Restructured Vulnerability	5	12	2	1

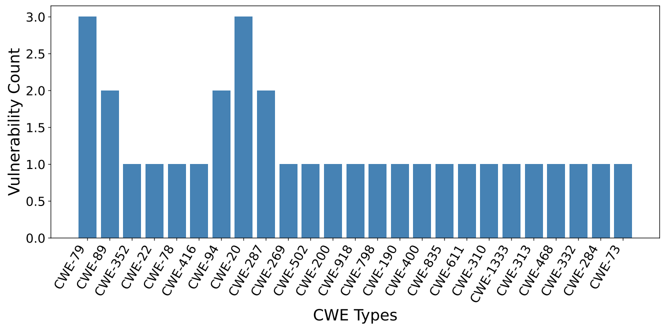


Fig. 2: Distribution of Vulnerabilities by CWE Types

these datasets are language-specific, limiting their ability to evaluate the cross-language generalization of repair models. To address this limitation, we construct a multi-language dataset, **MLVBench**, by unifying and adapting three public resources: PyP4LLMSec [14], VJBench [15], and CodeQL[1] security examples. Specifically, VJBench includes 35 transformed Java vulnerabilities derived from Vul4J by applying systematic transformation strategies, such as identifier renaming and structural modifications. The same transformation methodology is further applied to selected samples from PyP4LLMSec and CodeQL security examples, ensuring consistency in data representation across programming languages. PyP4LLMSec provides 156 vulnerabilities and 295 Python patches, covering 15 distinct CWE types [14]. CodeQL security examples offer a broad spectrum of vulnerabilities designed to demonstrate CodeQL query usage, spanning multiple CWE types across various programming languages. Inspired by the approach in prior work [16], we incorporate CodeQL into our benchmark to extract vulnerability patterns and facilitate dataset construction. The distribution details of MLVBench are presented in Table I.

To better align the dataset with real-world security evaluation scenarios, the selection of vulnerabilities from the three source benchmarks, PyP4LLMSec, VJBench, and CodeQL security examples, is guided by the 2024 CWE Top 25 Most Dangerous Software Weaknesses list[2]. However, not all of the Top 25 CWE types are covered in the original datasets. To ensure complete coverage, we supplement the dataset with additional vulnerabilities corresponding to other commonly occurring CWE types, resulting in 33 vulnerabilities spanning 25 distinct CWE categories. The distribution of the selected vulnerabilities and their associated CWE types is presented in Figure 2.

With the selected 33 vulnerabilities, we apply the transformation methodology introduced in VJBench [15] to generate structurally diverse yet semantically equivalent variants for evaluation. This methodology defines a set of transformation strategies applied to vulnerable functions, including: identifier renaming, if-condition flipping, loop transformation, conditional-statement rewriting, function chaining, function-argument modification, and code-order reordering. These transformations enhance structural diversity while preserving functional correctness, thereby enabling more robust and cross-language evaluation. Illustrative examples of the applied code transformations are provided in the Appendix, which is hosted on our GitHub repository[3].

Following the transformation process, our dataset includes 33 original function-level vulnerabilities along with 106 transformed variants, spanning four programming languages, Python, Java, Ruby, and C/C++, and providing a structurally diverse benchmark for evaluation.

B. Evaluation Dimensions

Before detailing the specific evaluation dimensions, we first define the standardized format of each evaluation instance used throughout CFCEval.

*1) **Evaluation Instance Format:*** Each instance for evaluating Code LLMs consists of four key components: the vulnerable function, the vulnerable code (referred to as the vulnerability) within that function, the generated code intended to fix the vulnerability, and the reference code that successfully addresses the vulnerability. These components are represented as a tuple (F, C_v, C_g, C_r) and are illustrated in Figure 3, where:

F denotes the **vulnerable function** that contains the vulnerable code, sourced from a selected security benchmark.

C_v denotes the **vulnerable code** located within F, identified by a commit message.

C_g denotes the **generated code** produced by the evaluated Code LLMs based on the prompt F.

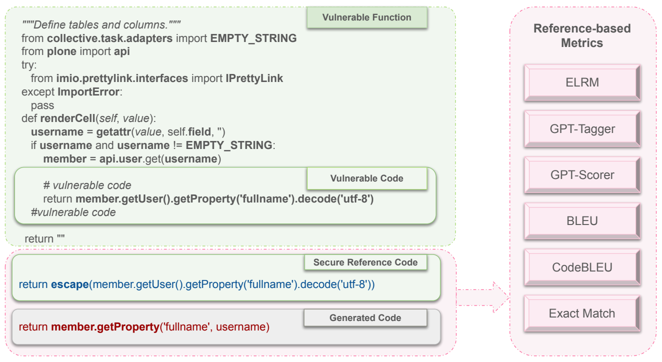
C_r denotes the secure **reference code** provided in the fix commit, which successfully repairs C_v in F.

Figure 3 illustrates the overall structure of the evaluation instance.

*2) **Overview of Evaluation Dimensions:*** CFCEval evaluates each generated output C_g across three complementary dimensions, focusing on both code quality and security relevance. An output is considered vulnerability-free only if it satisfies all four dimensions. Representative examples for each category in each dimension are provided in the Appendix of the GitHub repository[4].

Programming Language Quality (PLanQul.) This dimension identifies and eliminates poorly generated code, specifically

[1]https://github.com/github/codeql
[2]https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html
[3]https://github.com/Hahappyppy2024/CFCEval/blob/main/README.md
[4]https://github.com/Hahappyppy2024/CFCEval



```
"""Define tables and columns."""
from collective.task.adapters import EMPTY_STRING
from plone import api
try:
  from imio.prettylink.interfaces import IPrettyLink
except ImportError:
  pass
def renderCell(self, value):
  username = getattr(value, self.field, '')
  if username and username != EMPTY_STRING:
     member = api.user.get(username)

     # vulnerable code
     return member.getUser().getProperty('fullname').decode('utf-8')
  #vulnerable code

  return ""
```

```
return escape(member.getUser().getProperty('fullname').decode('utf-8'))
```

```
return member.getProperty('fullname', username)
```

Fig. 3: The application of the CFCEval framework in our experiments, including vulnerable functions, vulnerable code, secure reference code, and generated code, illustrating the evaluation process.

C_g with syntax errors such as unbalanced brackets or quotes, incorrect starting and ending characters, or invalid terminal characters. Each C_g is evaluated and classified as poor or good quality. Instances in which no code is generated are automatically marked as failing this dimension. This dimension can be measured by both reference-free metrics, such as GPT-Tagger and GPT-Scorer, and reference-based metrics, including Exact Match, GPT-Tagger and GPT-Scorer.

Fixing Capability (FixCap.) This dimension evaluates the ability of C_g to repair the identified C_v that appeared in F. It is a critical aspect of program repair in software engineering, commonly used to assess the effectiveness of automatic program repair (APR) tools [17], [18] and Code LLMs [19], [20]. We further classify the C_g into two distinct categories:

1) *Fixed:* The C_g produced by an individual Code LLM for prompt F successfully fix C_v.
2) *Not Fixed:* The C_g produced by an individual Code LLM for prompt F cannot fix C_v in F.

This dimension can also be measured by both reference-free metrics, such as GPT-Tagger and GPT-Scorer, and reference-based metrics, including Exact Match, GPT-Tagger, and GPT-Scorer.

Post-Transformation Fixing Capability (PTFixCap.) In evaluating C_g produced from the transformed function prompt, the original tuple (F, C_v, C_g, C_r) is updated to $(F_t, C_{vt}, C_g, C_{rt})$, where the subscript t denotes the transformed status. This dimension evaluates the ability of C_g to repair C_{vt} after applying code transformation rules to F. Code transformation is a strategy designed to reduce overlap between Code LLM training datasets and the evaluation dataset, as discussed in recent studies [15], [21]. The transformation rules, such as identifier renaming and structure modification, are detailed with illustrative examples in Appendix on GitHub. Similarly, C_g is classified into two distinct categories:

1) *Resolved:* The C_g produced by an individual Code LLM for prompt F_t successfully resolves C_{vt}.
2) *Unresolved:* The C_g produced by an individual Code LLM for prompt F_t fails to resolve C_{vt}.

This dimension can also be evaluated using both reference-free metrics, such as GPT-Tagger and GPT-Scorer, as well as reference-based metrics, including Exact Match, GPT-Tagger, and GPT-Scorer.

Element-Level Relevance (ELeRelv.) Element-Level Relevance is defined as the relevance between C_g and C_r or C_{rt}. This dimension measures these similarities and provides insight into the evolutionary process through which a standard Code LLM transitions into a secure Code LLM. Note that this dimension focuses on C_g, which fails to fix vulnerabilities in F or resolve the transformed vulnerabilities in F_t. We classify such $C_g$into the following two categories:

1) *Relevant:* The C_g produced by an individual Code LLM for prompt F or F_v, is relevant to C_r or C_{rt}.
2) *Irrelevant:* The C_g produced by an individual Code LLM for prompt F or F_v is not relevant to C_r or C_{rt}.

This dimension can be measured using both reference-free metrics, such as GPT-Tagger and GPT-Scorer, and reference-based metrics, including BLEU, CodeBLEU, GPT-Tagger, and GPT-Scorer. Additionally, this dimension can be evaluated using a novel reference-based metric, termed the Element-Level Relevance Metric (ELRM), which is introduced in the following section.

C. *ELRM: An Element-Level Relevance Metric*

During our experiments, we observe that only a small portion of both the original and transformed vulnerabilities could be accurately repaired by existing code LLMs. This raises a fundamental question: How relevant is the generated code to the actual repair patches? However, current evaluation metrics suffer from significant limitations, failing to capture essential dimensions of code quality and robustness. Widely used metrics such as BLEU and CodeBLEU primarily measure surface-level similarity based on n-gram or lexical overlap, offering limited insight into the structural and semantic correctness of the generated outputs. Specifically, these metrics fall short in the following areas: (1) Imprecise tokenization: Code lacking whitespace (e.g., x==1) is often treated as a single token, instead of being parsed into meaningful units such as identifiers and operators. (2) Structural limitations: Short code fragments often cannot be parsed into valid Abstract Syntax Trees (ASTs) or data-flow graphs, limiting the applicability of structure-aware metrics. (3) Low reference diversity: Most metrics rely on a single ground-truth reference, despite the fact that many repair tasks admit multiple semantically correct predictions. For example, both validators.length(max=256, message=_('Username too long.')) and validators.length(max=256, message=_('Username is too long.')) are semantically valid repairs, but existing metrics often fail to recognize this diversity.

These limitations substantially undermine the discriminative power of current metrics in real-world repair evaluation. We therefore introduce the Element-Level Relevance Metric (ELRM), which scores fine-grained alignment between a candidate patch and references across identifiers, operators, literals, API invocations, control keywords, and security cues, while allowing multiple valid fixes. We next formalize the metric.

1) Metric Formulation: The Element-Level Relevance Metric (ELRM) quantifies the lexical and semantic alignment between generated code and reference implementations through a weighted combination of complementary sub-metrics:

$$\text{ELRM} = \alpha \cdot \text{BLEU} + \beta \cdot \text{BLEU}_{\text{weight}} + \lambda \cdot \text{BLEU}_{\text{keywords\_ops}} + \mu \cdot \text{Similarity}_{\text{string\_literal}} \quad (1)$$

where BLEU is a standard metric for evaluating the quality of machine-generated code by comparing n-gram overlaps with reference code [9].

$\text{BLEU}_{\text{weight}}$ is a weighted variant of BLEU that assigns different importance to keywords and non-keywords in both the reference and the generated output.

$\text{BLEU}_{\text{keywords\_ops}}$ computes the BLEU score specifically over sequences of programming language keywords and operators.

$\text{Similarity}_{\text{string\_literal}}$ measures the lexical similarity of string literals between the reference and the generated code.

This formulation extends CodeBLEU by replacing its AST- and data-flow-based components with more lightweight yet effective lexical-level signals: $\text{BLEU}_{\text{keywords\_ops}}$ and $\text{Similarity}_{\text{String\_literal}}$, making it more suitable for evaluating short, structure-sparse code generations, where AST and data-flow information may be sparse or unreliable. The following items provide detailed descriptions of each component of the formulation.

1) **N-Gram Match and Weighted N-Gram Match**
The standard BLEU metric measures n-gram overlap between generated and reference code. However, its effectiveness in natural language tasks does not transfer well to programming languages, where strict syntax and limited keywords (e.g., *int*, *public*, =, +) make vanilla BLEU insensitive to the functional importance of language-specific tokens. To account for code-specific structure, **CodeBLEU** [10] employs a weighted n-gram scheme that emphasizes programming keywords by assigning them higher weights (e.g., 1). The weighted precision is computed as:

$$p_n = \frac{\sum_{C \in \text{Candidates}} \sum_{i=1}^{l} \mu_n^i \cdot \text{Count}_{\text{clip}}(C(i, i+n))}{\sum_{C' \in \text{Candidates}} \sum_{i=1}^{l} \mu_n^i \cdot \text{Count}(C'(i, i+n))} \quad (2)$$

where $C(i, i+n)$ denotes the n-gram starting at position i with length n, and μ_n^i is the assigned weight of the i-th n-gram as CodeBLEU [10] presents.
To penalize overly short candidates, CodeBLEU incorporates the standard BLEU brevity penalty [9]:

$$\text{BP} = \begin{cases} 1 & \text{if } c > r \\ e^{1-r/c} & \text{if } c \leq r \end{cases} \quad (3)$$

where c stands for the length of the candidate and r is the effective reference length as BLEU [9] presents.
By incorporating both weighted precision and the brevity penalty, the final weighted BLEU score is then calculated as:

$$\text{BLEU}_{\text{weight}} = \text{BP} \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right) \quad (4)$$

We briefly restate the core formulations from CodeBLEU [10] and BLEU [9] to clarify the underlying design of its weight-ed n-gram precision and brevity penalty. In our implementation, we further refine the tokenization procedure to address scenarios where the absence of whitespace causes entire lines of code to be interpreted as single tokens. This refinement ensures the correct separation and identification of key syntactic elements such as identifiers, keywords, and operators, thereby improving the fidelity of n-gram matching in code evaluation.

2) **Fine-Grained Tokenization**
To address the coarse-grained tokenization limitations in CodeBLEU, we introduce a refined tokenization strategy that more accurately reflects the syntactic granularity of source code. For example, CodeBLEU treats the expression *permission_classes=[permissions.IsAuthenticated, IsSuperUser]* as a single token, thereby failing to distinguish critical lexical components such as identifiers, keywords, operators, and delimiters. In contrast, our approach tokenizes the same expression into a list of fine-grained tokens: *[”permission_classes”, ”=”, ”[”, ”permissions”, ”.”, ”IsAuthenticated”, ”,”, ”IsSuperUser”, ”]”]* This fine-grained tokenization facilitates more accurate structural alignment between the generated and reference code, which is essential for evaluating semantic and syntactic relevance in program synthesis.

3) **Language Keywords and Operators Match**
Unlike CodeBLEU, which incorporates AST and data-flow comparisons, our method focuses exclusively on the sequences of programming language keywords and operators. This design choice stems from the observation that AST- and data-flow-based comparisons are often unreliable or infeasible when evaluating short code snippets, where structural information is limited or absent. Concretely, we extract the keyword and operator sequences (e.g., for, if, return, =, +) from both the candidate and reference implementations, and compute their n-gram overlap using the BLEU metric to assess lexical alignment.

4) **String Literal Similarity**
In many cases, two code snippets may be semantically and syntactically equivalent while differing only in their string literals, which refer to constant string values explicitly defined within the source code. Such differences, although functionally irrelevant, can significantly affect string-based similarity metrics, especially in short code

snippets. For example, the statement *setMessage("Invalid input format")* contains the string literal *"Invalid input format"*, which could be rewritten as *setMessage("Input format is invalid")*. Although the surrounding logic remains unchanged, such differences can affect the evaluations results, as illustrated in the example shown in Figure 5. To capture these subtle lexical variations, we compute the similarity between string literals using three complementary syntactic metrics : (1) **Levenshtein distance** [22], which quantifies character-level edit operations; (2) **SequenceMatcher**, which identifies the longest common subsequences; and (3) **Jaccard similarity** [23], which captures token overlap from a set-theoretic perspective. The final score is calculated as their average, leveraging the complementary strengths of all three for robust string-level alignment.

2) *Two Examples and Analysis:* In this section, we present two simple examples to demonstrate the calculation of ELRM. Additionally, we highlight the qualitative benefits of ELRM in comparison to the CodeBLEU metric.

```
[Reference]="paths = [ipython_dir]"
[Generated Code]="paths = [ipython_dir, get_ipython_package_dir()]"
```

Fig. 4: Example 1. CodeBLEU: 22.03 , ELRM: 13.71.

a) ***Example 1*** *:* Figure 4 illustrates a generated code snippet and its corresponding reference code. In this example, the key difference lies in the presence of an unnecessary segment,*get_ipython_package_dir()*, in the generated output (i.e., the candidate in BLEU evaluation), which is absent from the reference code. This deviation is expected to be effectively penalized by a robust evaluation metric.

To compute the ELRM score (normalized to a 0-100 scale), we follow four sequential steps: (1) **N-Gram Match:** The standard BLEU score is computed to quantify the n-gram overlap between the candidate and the reference, yielding a score of 25.27. (2) **Weighted N-Gram Match:** Since the code line contains no programming language keywords, each token is assigned an equal weight of 0.2. The resulting weighted BLEU score is 70.7. (3) **Keyword and Operator Match:** The keywords in the reference code are *["=", "[", "]"]*, while those in the generated code are *["=", "[", ",", "(", ")", "]"]*. Programming keywords and operators are extracted as ordered sequences and evaluated using the BLEU metric, resulting in a score of 9.55. (4) **String Literal Similarity:** As neither the reference nor the candidate contains string literals, the similarity score for this component is 0. By applying the combination weights $\alpha = 0.10$, $\beta = 0.05$, $\lambda = 0.80$, and $\mu = 0.05$, the final ELRM score is computed to be 13.71, which is substantially lower than the corresponding CodeBLEU score in this case.

```
[Reference]="if action == \"add\" and form.is_submitted():"
[Generated Code]="if action == 'add' and form.is_submitted():"
```

Fig. 5: Example 2. CodeBLEU: 62.87, ELRM: 94.95.

b) ***Example 2:*** To facilitate a comparative analysis between ELRM and CodeBLEU, Figure 5 presents a second example comprising a generated code snippet and its corresponding reference implementation. In this example, the only difference lies in the use of double quotes (i.e., ") versus single quotes (i.e., '). Despite this minor variation, the candidate and the reference are semantically and functionally equivalent. Consequently, the metric is expected to yield a high score in this case, reflecting the negligible discrepancy. To evaluate the ELRM score (on a normalized 0-100 scale), we execute the following four-step procedures: To evaluate the ELRM score (normalized to a 0-100 scale), we follow a four-step procedure: (1) **N-Gram Match:** We begin by computing the standard BLEU score to assess n-gram overlap between the generated output and its reference. In this example, the BLEU score is 64.07. (2) **Weighted N-Gram Match:** For weighted evaluation, programming keywords are assigned a weight of 1.0, while all other tokens receive a uniform weight of 0.2. This adjustment yields a weighted BLEU score of 70.89. (3) **Keyword and Operator Match:** We extract the following ordered list of programming-specific keywords and operators from both the candidate and the reference: ["if", "==", "and", ".", "(", ")", ":"]. The BLEU score computed over this sequence is 100.0. (4) **String Literal Similarity:** As both the candidate and reference contain the identical string literal *"add"*, this component contributes a similarity score of 100.0. Using the weighting configuration $\alpha = 0.10$, $\beta = 0.05$, $\lambda = 0.80$, and $\mu = 0.05$, the final ELRM score is computed to be 94.95, which is higher than the corresponding CodeBLEU score for this example.

c) ***Summary****:* The two examples demonstrate that ELRM enables finer-grained and more faithful evaluation than CodeBLEU. It effectively addresses the three core deficiencies of existing metrics: (1) by applying element-aware tokenization, it resolves the imprecise handling of compact code; (2) by operating on typed lexical units rather than full ASTs, it remains robust on short code fragments that cannot be structurally parsed; and (3) by supporting multiple reference variants, it mitigates low reference diversity and recognizes semantically equivalent fixes. Consequently, ELRM penalizes semantically redundant outputs (Example1) and rewards functionally equivalent variants with minor syntactic differences (Example2), capturing both syntactic structure and functional meaning. The next section presents empirical results validating its effectiveness and robustness across benchmarks.

III. Experimental Setup and Results

A. Experimental Setup

We evaluate four representative code LLMs, consistent with those used in the original CodeBLEU evaluation: **GitHub Copilot**[5], **Cursor**[6] (Gemini-2.5-flash), **DeepCoder**[7], and **CodeGeeX4**[8]. All models are assessed using their latest

[5]https://code.visualstudio.com/docs/copilot/overview
[6]https://cursor.com/cn/agents
[7]https://ollama.com/library/deepcoder:14b
[8]https://github.com/THUDM/CodeGeeX

available versions. Evaluations are conducted on **MLVulBench**, a multi-language vulnerability benchmark within CFCEval, featuring prompts in Python, Java, C++, and Ruby.

B. Results

Table II reports the ELRM, BLEU, and CodeBLEU scores, along with LLM-based evaluations, on 20 randomly sampled benchmark cases spanning all transformation types and four programming languages. Each input is paired with four generated outputs, resulting in 80 input-output prompt pairs for LLM-based scoring. The first three metrics are normalized to a 0-100 range, while LLM-based scores use a 5-point Likert scale (1=very poor, 5=excellent). Example prompts are provided in the Appendix of our GitHub repository [9]. Based on these results, we investigate the following four research questions.

TABLE II: The results of all baselines on the given vulnerabilities, evaluated by ELRM, BLEU, CodeBLEU, and LLM-based evaluation scores.

	ELRM	BLEU	CodeBLEU	LLMs
Cursor	24.72	30.33	29.98	2.38
GitHub Copilot	22.93	29.43	29.43	2.9
CodeGeeX4	29.19	36.22	30.75	2.4
DeepSeekCoder	18.65	21.19	24.16	1.92

*1) **RQ1**:* To what extent is ELRM a reliable and stable metric for differentiating code generation models, in terms of score significance and variance?

We compute the ELRM scores of the generated code from all selected Code LLMs, and calculate their means, variances, and paired *t*-statistics, which are reported in Table II and Table III. From Table II, we observe that Cursor, Copilot, and CodeGeeX exhibit relatively close ELRM scores (means ranging from 0.229 to 0.292), while DeepSeekCoder obtains a substantially lower mean of 0.186. This is consistent with our qualitative observation that DeepSeekCoder, as a locally deployed model, frequently produces outputs with syntactic and readability issues, whereas the other systems generate more consistent and well-formed code.

TABLE III: Mean, standard deviation, and paired *t*-statistics of ELRM across Code LLMs. Each *t*-statistic compares a system with the one above; the first compares Cursor with DeepSeek-Coder.

	Mean	StdDev	t
Cursor	24.72	23.75	1.86
GitHub Copilot	22.93	21.95	0.54
CodeGeeX4	29.20	25.87	1.66
DeepSeek-Coder	18.65	20.36	2.81

In Table III, each t-statistic compares a system with the one above it. For instance, the first t-statistic compares Cursor with DeepSeek-Coder. The paired t-statistic between CodeGeeX and DeepSeek-Coder reaches 2.81 ($p = 0.0056$), indicating a statistically significant difference. While other pairwise comparisons show less statistical significance (e.g., $p = 0.10$ and $p = 0.0651$), they are directionally consistent with the overall system quality and fall within the sensitivity threshold observed in LLM-based annotations. These results suggest that while the differences are not always statistically significant, they align with the general trends in model performance. The standard deviations (20.36-25.87) reflect expected variability across diverse prompts and model behaviors. Despite this variability, ELRM maintains consistent scoring trends, demonstrating its robustness to real-world input diversity. Notably, the pairwise t-test between CodeGeeX4 and DeepSeek-Coder yields a high statistic (t = 2.81), indicating a statistically meaningful performance gap that ELRM is able to capture. Even in cases with marginal significance (e.g., Cursor vs. Copilot, t = 0.54), the score differences align directionally with overall system quality. These results suggest that ELRM can effectively discriminate between higher- and lower-quality outputs. We therefore conclude that ELRM provides a reliable and valid basis for evaluating code generation performance. Additional ablation studies are available in the Appendix and on our GitHub repository[10].

*2) **RQ2**:* Does ELRM achieve higher correlation with LLMs judgments than CodeBLEU and BLEU in evaluating generated code?

TABLE IV: Comparison of the Pearson correlation coefficients between LLM-based evaluation scores and different metrics.

Metrics	Pearson's Correlation
BLEU (Orignal) [a]	-0.173
CodeBLEU	0.461
BLEU (ELRM) [b]	0.804
ELRM	0.816

BLEU[a] uses the CodeBLEU tokenizer; BLEU[b] uses the ELRM tokenizer.

Owing to their extensive pretraining on large-scale code corpora, large language models (LLMs) have demonstrated exceptional performance on function-level tasks, often outperforming human coders in terms of accuracy and efficiency [4], [24]. LLMs have shown to be highly proficient in generating function-level code with an understanding of both syntax and semantics, which enables them to produce reliable outputs for complex tasks. Furthermore, the labels generated by LLMs tend to align more closely with gold-standard annotations compared to those produced by human annotators, showcasing their ability to consistently reproduce high-quality outputs [25]. Consequently, we adopt labels generated by three distinct LLM versionsChatGPT-4o, ChatGPT-o3, and ChatGPT-4.1as the reference judgments for evaluating 20 selected vulnerabilities from the MLVulBench benchmark.

As demonstrated in Table IV, ELRM achieves the highest Pearson correlation with the LLM-generated annotations,

[9]https://github.com/Hahappyppy2024/CFCEval/blob/main/README.md

[10]https://github.com/Hahappyppy2024/CFCEval/blob/main/README.md

indicating that it most effectively aligns with the semantic and functional evaluations provided by the LLMs. This suggests that ELRM is particularly well-suited for capturing the nuanced judgments made by these models. In contrast, CodeBLEU frequently produces zero scores, which limits its ability to accurately capture semantic relevance and suppresses its correlation with LLM-generated ratings. This lack of sensitivity highlights the limitations of CodeBLEU, especially in tasks related to vulnerability detection and repair, where fine-grained semantic matching is crucial. These findings underscore ELRMs superiority as an evaluation metric, particularly in the context of code generation for vulnerability-related tasks, where the ability to capture both syntax and semantics is paramount.

*3) **RQ3**:* Does ELRM achieve higher correlation with human judgments than CodeBLEU and BLEU in evaluating generated code?

To evaluate the effectiveness of the Element-Level Relevance Metric (ELRM), we conduct a comprehensive assessment on the full set of function-level vulnerabilities within the MLVBench benchmark, utilizing four prominent code generation models: Cursor, Copilot, CodeGeeX, and DeepSeek-Coder. Each model is evaluated based on its ability to generate code patches for these vulnerabilities. To ensure consistency and reliability, two independent annotators score the generated patches on a 1-5 scale. Due to the varying performance of these models, sample sizes across models differ: Cursor and Copilot each produce results for 97 functions, CodeGeeX for 77 functions, and DeepSeek-Coder for 91 functions.

TABLE V: Correlations with human judgments on MLVBench (function level). Pearson's γ between *HumanAvg* (two 15 ratings; Cohens $\kappa = 0.7663$) and BLEU/CodeBLEU/ELRM per model (n shown). Bold = strongest per model.

Model	n	BLEU	CodeBLEU	ELRM
Cursor	97	0.3548	0.2253	**0.8281**
Copilot	97	0.3975	0.3096	**0.6681**
CodeGeeX	77	0.2931	0.2174	**0.8601**
DeepSeekCoder	91	0.0787	0.0772	**0.7991**

The inter-annotator agreement is robust, with a Cohens kappa value of 0.77, indicating substantial consistency between the raters. For each model, the average score, referred to as HumanAvg, is computed by averaging the two independent ratings. Subsequently, Pearsons correlation coefficient is calculated between HumanAvg and the evaluation scores derived from three different metrics: BLEU, CodeBLEU, and ELRM. This allows for a comparative analysis of the performance of each metric in capturing human judgment.

As shown in Table V, ELRM consistently demonstrates the strongest correlation with human evaluations, outperforming both BLEU and CodeBLEU. These results validate ELRM's ability to provide a more accurate and nuanced measure of code quality, especially in the context of function-level vulnerability repair.

*4) **RQ4**:* Can CFCEval serve as the first comprehensive benchmark for automatic evaluation metrics that assess code LLMs in terms of both quality and security?

To use CFCEval as an automatic benchmark, we construct reference-based prompts following the methodology of QUDeval [26]. We introduce two metrics, both based on ChatGPT-4.1mini: **GPT-Tagger**, which classifies the generated code according to CFCEval's dimensions, and **GPT-Scorer**, which rates code quality on a 5-point scale. We evaluate the code generated by GitHub Copilot and CodeGeeX for 20 MLVBench vulnerabilities. Example prompts are provided in the Appendix on GitHub repository [11].

TABLE VI: Reference-based assessment of code generated by GitHub Copilot and CodeGeeX using GPT-Tagger and GPT-Score across PLanQu., FixCap., PTFixCap., and EleReLv metrics.

		PLanQul. (%)		FixCap. (%)	
		Poor	Good	Not Fixed	Fixed
Copilot	GPT-Tagger	60.0	40.0	75.0	25.0
	GPT-Score	60.0	40.0	70.0	30.0
CodeGeeX	GPT-Tagger	55.0	45.0	85.0	15.0
	GPT-Score	80.0	20.0	90.0	10.0
		PTFixCap. (%)		**ELeReLv. (%)**	
		UnRes.	Res.	Irre.	Rel.
Copilot	GPT-Tagger	75.0	25.0	70.0	30.0
	GPT-Score	70.0	30.0	55.0	45.0
CodeGeeX	GPT-Tagger	80.0	20.0	80.0	20.0
	GPT-Score	90.0	10.0	90.0	10.0

As shown in Table VI, CFCEval evaluates four dimensions spanning code quality and security, using both quantitative metrics and LLM-based scoring prompts. The table reports LLM evaluation results based on reference-based prompts designed to assess each dimension. For instance, in the PLanQu. dimension, GitHub Copilot's GPT-Tagger scores 60% poor and 40% good, while CodeGeeX scores 55% poor and 45% good. In FixCap., Copilot's GPT-Score evaluates 75% as not fixed and 25% as fixed (out of 20 vulnerabilities), while CodeGeeX scores 85% and 15%, respectively. These examples illustrate CFCEval's ability to assess the code quality and security using reference-based prompts. While our current evaluation focuses on the reference-based setting, CFCEval also supports reference-free metrics, with corresponding prompts provided in the Appendix on GitHub repository.

IV. Pilot Study: Evaluation of Code LLMs for Vulnerability Repair

We conduct a pilot study comparing two prominent code generation LLMsCursor and Copiloton real-world benchmark instances, represented as a tuple (F, C_v, C_g, C_r). For each

[11] https://github.com/Hahappyppy2024/CFCEval/blob/main/README.md#gpt-based-metrics-reference-based-prompts

instance, C_v denotes the vulnerable code snippet, C_r is the secure reference patch, and C_g is the model-generated fix. Both models are evaluated under identical prompts and decoding settings, generating candidate patches for the given vulnerabilities. To assess robustness, we apply a single program transformation, identifier renaming, to obtain the transformed instance F_t, and re-evaluate the model-generated patch C_g and the reference patch C_{rt}.

In this pilot study, we use several metrics from the CFCEval framework to assess the quality of the generated fixes. FixCap and PTFixCap apply Exact Match with C_r and C_{rt}, respectively, to evaluate whether the generated patch successfully eliminates the vulnerability. FixCap checks if C_g exactly matches C_r, ensuring the correctness of the fix, while PTFixCap evaluates the robustness of the fix under distribution shift by comparing C_g against C_{rt}. Element-Level Relevance (ELeRelv) is measured using the Element-Level Relevance Metric (ELRM), which quantifies the fine-grained relevance between non-matching elements, providing a more detailed analysis of the semantic alignment of the generated patch with the reference patch.

This pilot study serves as a preliminary evaluation of the Code LLMs' performance in vulnerability repair tasks. It offers insights into their ability to generate accurate and robust fixes. Further details, including experimental setups and additional results, are available in the supplementary document provided at Supplementary Document.

V. Threats to Validity

Two human judges. Human evaluations are prone to subjectivity, expertise variance, and fatigue. To mitigate these, we standardize the evaluation process using a shared rubric, implement blinding and randomization, and apply double-annotation for each code sample. We measure inter-rater reliability using Cohen's κ and resolve disagreements through adjudication. Additionally, spot-checking with tests and static analysis ensures that the evaluations align with expected outcomes and maintain consistency. This approach helps minimize bias and ensures the robustness of human judgments in complex evaluation tasks.

LLM-as-judge (ChatGPTs). LLM-based evaluations can be affected by model biases and variations in prompts and versions. To address this, we standardize inputs, randomize A/B testing, and fix prompts to minimize bias. We log model IDs and dates to track changes and ensure transparency in the evaluation process. Multiple responses are aggregated to reduce outlier effects, reflecting the model's judgment rather than inconsistencies in prompt or version. This strategy ensures that results represent the model's capabilities while minimizing errors introduced by prompt-specific biases.

VI. Related Work

Recent research has explored the use of large language models (LLMs) for various code generation tasks, including program repair, debugging, and vulnerability detection. Early efforts in automatic program repair (APR) demonstrated the potential of LLMs to generate code fixes for common errors, but these methods were limited by small datasets and handcrafted rules [17], [18]. As LLMs have evolved, their application has expanded to security-related tasks, such as vulnerability detection and repair, though challenges remain in adapting general-purpose models to address the subtleties of security-critical tasks [19], [20], [27], [28]. These challenges stem from the complexity of security flaws, which often require nuanced understanding and precise fixes beyond simple syntactic corrections.

Common benchmarks like HumanEval [4], ReCode [29], and CoderEval [30] focus on functional correctness, offering insights into syntactic validity. However, specialized benchmarks such as LLMSecEval [31] and PyP4LLMSec [14] have emerged to assess LLMs on security-oriented tasks. Despite these efforts, security datasets like DS-1000 [7] still struggle to capture the full complexity of real-world vulnerabilities, particularly those that arise in dynamic software environments. Moreover, existing datasets often fail to account for evolving attack vectors and edge cases that are critical for robust security evaluation.

The evaluation of LLMs has expanded beyond functional correctness to include robustness under perturbations [32], additional quality metrics [33], and fairness [34]. While metrics like BLEU [9] and CodeBLEU [10] are widely used to assess syntactic and structural similarities, they fail to capture deeper semantic and security-related aspects critical for vulnerability repair. Metrics like Exact Match (EM) [8] focus on syntactic correctness but overlook finer-grained code qualities essential for secure and robust software generation. As a result, there is an increasing demand for new, more comprehensive metrics that address both the functional and security dimensions of code quality.

In summary, while LLMs have made significant strides in security-critical applications, there remains a need for more specialized benchmarks and evaluation methods to address the complex demands of vulnerability detection and repair. Developing semantic, robust, and security-focused metrics will be crucial for enhancing the reliability and security of LLM-generated code in real-world software systems.

VII. Conclusion

In conclusion, we present the Code Fix Capability Evaluation (CFCEval) framework, which addresses key evaluation limitations by leveraging the MLVBench dataset to mitigate training-testing data overlap bias. CFCEval evaluates code LLMs across four dimensions, including Element-Level Relevance, assessed using the Element-Level Relevance Metric (ELRM). Experimental results show that ELRM better aligns with LLM quality scores compared to existing metrics like CodeBLEU and BLEU. Additionally, GPT-based metrics confirm CFCEval's potential as an automatic evaluation benchmark. The pilot study presented in this paper demonstrates CFCEval's practical application and its ability to evaluate code generation models in real-world vulnerability repair tasks. In the future, we plan to incorporate additional vulnerabilities from a wider range of programming languages into the MLVBench dataset and conduct a comprehensive case study to further validate our framework and explore its applicability across diverse software ecosystems.

References

[1] "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," 2025. [Online]. Available: https://arxiv.org/abs/2501.12948

[2] S. et al., "Large language models are more persuasive than incentivized human persuaders," 2025. [Online]. Available: https://arxiv.org/abs/2505.09662

[3] OpenAI, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023. [Online]. Available: https://arxiv.org/abs/2303.08774

[4] C. et al., "Evaluating large language models trained on code," 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[5] B. R. et al., "Code llama: Open foundation models for code," 2024.

[6] L. et al., "Starcoder 2 and the stack v2: The next generation," 2024.

[7] B. Szalontai, G. Szalay, T. Márton, A. Sike, B. Pintér, and T. Gregorics, "Large language models for code summarization," 2024. [Online]. Available: https://arxiv.org/abs/2405.19032

[8] Y. Ding, Z. Wang, W. U. Ahmad, H. Ding, M. Tan, N. Jain, M. K. Ramanathan, R. Nallapati, P. Bhatia, D. Roth, and B. Xiang, "Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion," in *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2023. [Online]. Available: https://openreview.net/forum?id=wgDcbBMSfh

[9] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," ser. ACL '02. USA: Association for Computational Linguistics, 2002, pp. 311–318. [Online]. Available: https://doi-org.lib-ezproxy.concordia.ca/10.3115/1073083.1073135

[10] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," 2020.

[11] D. Bui, H. Pfohl, I. Scholz, T. Holz, and K. Rieck, "Vul4J: a dataset of reproducible java vulnerabilities geared towards the study of program repair techniques," in *19th IEEE/ACM International Conference on Mining Software Repositories (MSR 2022)*. IEEE/ACM, 2022, pp. 196–200.

[12] C. Liu, Y. Ye, Y. Zhou, and J. Yang, "Sok: Automated vulnerability repair," *arXiv preprint arXiv:2506.11697*, 2025. [Online]. Available: https://arxiv.org/abs/2506.11697

[13] P. Wang, X. Liu, and C. Xiao, "CVE-bench: Benchmarking LLM-based software engineering agent's ability to repair real-world CVE vulnerabilities," in *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, L. Chiruzzo, A. Ritter, and L. Wang, Eds. Albuquerque, New Mexico: Association for Computational Linguistics, Apr. 2025, pp. 4207–4224. [Online]. Available: https://aclanthology.org/2025.naacl-long.212/

[14] C. Cheng and J. Yang, "Benchmarking the security aspect of large language model-based code generation," in *LLM4Code 2024: Workshop on Large Language Models for Code (co-located with ICSE 2024)*, Lisbon, Portugal, Apr. 2024, position paper / talk; dataset: PyP4LLMSec. [Online]. Available: https://llm4code.github.io/assets/pdf/papers/42.pdf

[15] Y. Wu, N. Jiang, H. V. Pham, T. Lutellier, J. Davis, L. Tan, P. Babkin, and S. Shah, "How effective are neural networks for fixing security vulnerabilities," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, 2023, pp. 1282–1294. [Online]. Available: https://doi-org.lib-ezproxy.concordia.ca/10.1145/3597926.3598135

[16] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? assessing the security of github copilot's code contributions," 2021.

[17] Z. Lin, X. Jiang, D. Xu, B. Mao, and L. Xie, "Autopag: towards automated software patch generation with source code root cause identification and repair," in *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '07. New York, NY, USA: Association for Computing Machinery, 2007, pp. 329–340. [Online]. Available: https://doi-org.lib-ezproxy.concordia.ca/10.1145/1229285.1267001

[18] D. Drain, C. Clement, G. S. Castilla, and N. Sundaresan, "Deepdebug: Fixing python bugs using stack traces, backtranslation, and code skeletons," 2022. [Online]. Available: https://openreview.net/forum?id=9HXfisrWl1

[19] Y. Fu, P. Liang, A. Tahir, Z. Li, M. Shahin, J. Yu, and J. Chen, "Security weaknesses of copilot generated code in github," 2024.

[20] Z. Li, Z. Liu, W. K. Wong, P. Ma, and S. Wang, "Evaluating c/c++ vulnerability detectability of query-based static application security testing tools," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–18, 2024.

[21] P. Thongtanunam, C. Pornprasit, and C. Tantithamthavorn, "Autotransform: automated code transformation to support modern code review process," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 237–248. [Online]. Available: https://doi.org/10.1145/3510003.3510067

[22] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet Physics Doklady*, vol. 10, pp. 707–710, 1966.

[23] P. Jaccard, "The distribution of the flora in the alpine zone. 1," *New Phytologist*, vol. 11, no. 2, pp. 37–50, 1912.

[24] M. F. A. Khan, M. Ramsdell, E. Falor, and H. Karimi, "Assessing the promise and pitfalls of chatgpt for automated code generation," 2023. [Online]. Available: https://arxiv.org/abs/2311.02640

[25] A. Parfenova, A. Marfurt, J. Pfeffer, and A. Denzler, "Text annotation via inductive coding: Comparing human experts to LLMs in qualitative data analysis," in *Findings of the Association for Computational Linguistics: NAACL 2025*, L. Chiruzzo, A. Ritter, and L. Wang, Eds. Albuquerque, New Mexico: Association for Computational Linguistics, Apr. 2025, pp. 6456–6469. [Online]. Available: https://aclanthology.org/2025.findings-naacl.361/

[26] Y. Wu, R. Mangla, G. Durrett, and J. J. Li, "QUDeval: The evaluation of questions under discussion discourse parsing," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, H. Bouamor, J. Pino, and K. Bali, Eds. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 5344–5363. [Online]. Available: https://aclanthology.org/2023.emnlp-main.325

[27] J. Li, A. Sangalay, C. Cheng, Y. Tian, and J. Yang, "Fine tuning large language model for secure code generation," in *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering*, ser. FORGE '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 8690. [Online]. Available: https://doi.org/10.1145/3650105.3652299

[28] J. Li, F. Rabbi, C. Cheng, A. Sangalay, Y. Tian, and J. Yang, "An exploratory study on fine-tuning large language models for secure code generation," 2025. [Online]. Available: https://arxiv.org/abs/2408.09078

[29] W. et al., "ReCode: Robustness evaluation of code generation models," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, A. Rogers, J. Boyd-Graber, and N. Okazaki, Eds. Toronto, Canada: Association for Computational Linguistics, Jul. 2023, pp. 13 818–13 843. [Online]. Available: https://aclanthology.org/2023.acl-long.773

[30] H. Yu, B. Shen, D. Ran, J. Zhang, Q. Zhang, Y. Ma, G. Liang, Y. Li, Q. Wang, and T. Xie, "Codereval: A benchmark of pragmatic code generation with generative pre-trained models," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi-org.lib-ezproxy.concordia.ca/10.1145/3597503.3623316

[31] C. Tony, M. Mutas, N. E. D. Ferreyra, and R. Scandariato, "Llmseceval: A dataset of natural language prompts for security evaluations," 2023.

[32] F. Rabbi, Z. Ding, and J. Yang, "A multi-language perspective on the robustness of llm code generation," 2025. [Online]. Available: https://arxiv.org/abs/2504.19108

[33] F. Lin, D. J. Kim, Z. Li, J. Yang, Tse-Hsun, and Chen, "Robunfr: Evaluating the robustness of large language models on non-functional requirements aware code generation," 2025. [Online]. Available: https://arxiv.org/abs/2503.22851

[34] L. Ling, F. Rabbi, S. Wang, and J. Yang, "Bias unveiled: Investigating social bias in llm-generated code," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 39, no. 26, pp. 27 491–27 499, Apr. 2025. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/34961