Should We Evaluate LLM Based Security Analysis Approaches on Open Source Systems?

Kohei Dozono*, Jonas Engesser*, Benjamin Hummel†, Tobias Roehm† and Alexander Pretschner*

*Technical University of Munich
Munich, Germany
{kohei.dozono, jonas.engesser, alexander.pretschner}@tum.de

†CQSE GmbH
Munich, Germany
{hummel, roehm}@cqse.eu

***Abstract*—Existing research has demonstrated promising results when applying large language models (LLMs) to detect security vulnerabilities in source code. However, these studies have been exclusively evaluated on benchmarks from open-source systems, using publicly known vulnerabilities that are likely part of the LLMs' training data. This raises concerns that reported performance metrics may be inflated due to data contamination, providing a misleading view of the models' actual capabilities.**

In this paper, we quantify this effect with a case study that evaluates five frontier LLMs on two carefully curated datasets: CWE-Bench-Java (an open-source dataset) and TS-Vuls (based on a closed-source commercial codebase). To provide a second angle, we also split CWE-Bench-Java by CVE record date to explore temporal contamination based on LLM knowledge cutoff dates.

Our results reveal that the average F1 score dropped by approximately 20 percentage points when comparing the open-source to the closed-source dataset. Additionally, the precision drops from 56% to 34% on average, which is statistically significant ($p < 0.05$) for four of five models. This declining trend is consistent across all tested LLMs and metrics. In contrast, the results for the temporal split on the open-source data are inconclusive, suggesting that using a knowledge cutoff may reduce but does not ensure the elimination of contamination effects.

Although our study is based on a single closed-source system and thus not generalizable, these findings provide the first empirical evidence that evaluating LLM-based vulnerability detection on open-source benchmarks may lead to overly optimistic results. This motivates the inclusion of closed-source datasets in future LLM evaluations.

***Index Terms*—security, large language models, data contamination, static application security testing, security vulnerabilities**

I. Introduction

Advances in language and code processing enabled by progress in large language models (LLMs) have significantly shaped the landscape of software engineering research and enabled new approaches that have not been possible before [1]. Still, while we understand LLMs in terms of data structures, model architectures, training data and algorithms, we currently have no deep understanding of the capabilities and limitations of those models. The sheer size, the stochastic evaluation strategy, and variations in model training lead to a situation where it is impossible to formally deduce if a model can address a certain task. Instead, evaluation based on benchmarks has become the de-facto standard to determine the fitness of a model for a certain task.

A. Static Security Analysis

Static security analysis is one of the areas where LLMs have been applied recently [2]–[6]. With the constant rise of cybercrime [7] and a growing number of known software vulnerabilities, as indicated by the rise of CVE entries [8], static application security testing (SAST) becomes an increasingly important ingredient for maintaining and improving system security during software development. Additionally, many compliance frameworks, including NIS2 (Network and Information Security Directive 2 [9]) or DORA (Digital Operational Resilience Act [10]), effectively mandate the inclusion of SAST tools in the software development process.

The shift from traditional analysis based on program parsing and analysis, data-flow graph analysis, and similar techniques to LLM-based approaches is driven by the goal to cover a wider range of vulnerability patterns and improve precision and recall [3]. While existing work reports positive results, those approaches have been primarily evaluated on benchmark data distilled from open-source systems (see Section VII).

B. Threats of Evaluation on Open-Source Projects

Due to the availability of large amounts of data, empirical research and evaluations in software engineering are often based on open-source software systems. This can be a potential threat to validity, as open-source systems might be developed differently from commercial ones and hence exhibit different properties [11].

However, when evaluating LLM-based approaches on open-source projects, there is another effect that may impact the transferability of research results, namely the training bias towards open-source systems. Open-source projects are well-known to be a huge part of the training data used for the initial training of LLMs [12], while closed-source projects (due to their nature of being closed) are not used for training of frontier LLMs. Additionally, meta information outside of the core project (such as third-party blog posts, vulnerability reports, tutorials, etc.) is available on the Internet and thus potentially used for LLM training.

This training bias can affect the output of LLMs, as observed by [13], where LLMs were found to have a strong output preference for programming languages and libraries used in an open-source context, even when applied in a closed-source setting, despite the availability of more appropriate alternatives.

Similarly, an evaluation of LLM-based approaches using samples from open-source systems can lead to a situation where the same samples that were used for training the LLM are then used for the evaluation. This situation, where the training data already contains full or partial answers to the exact questions passed to the LLM, is often referred to as data contamination [14], [15].

This might give LLM-based approaches a head start when evaluated on open-source systems, as the relevant patterns could be partially encoded in the LLM's model weights. This is different from closed-source/commercial development, where typically the code and also accompanying information are not provided publicly, but only to internal teams or (partially) to customers, e. g. security information.

Due to this effect, approaches based on LLMs may overperform when evaluated on open-source systems, while showing degraded performance on closed-source codebases.

C. Contribution and Results

This paper studies and quantifies the effect of one specific task, namely the identification of security vulnerabilities in source code. We perform a case study to get insights into the differences in performance of LLMs in security analysis between open-source and a closed-source system. We demonstrate how state-of-the-art LLMs exhibit statistically significant differences in precision, recall, and F1 score when evaluated on samples from open-source systems (which are likely in the LLM's training data) compared to an evaluation on samples from a commercial closed-source software system (which was not part of the training data).

Our main result is that, on average, the used LLMs solve the selected vulnerability detection task on the open-source benchmark with a precision of 56%, while for the closed-source system, precision drops to 34%. This drop in result quality is consistent across all LLMs and metrics (precision, recall, F1 score). This also does not change when focusing on individual vulnerability classes in most of the chosen LLMs. Additionally, we observe a slight drop in performance metrics on vulnerabilities in the open-source benchmark that were fixed and recorded after their knowledge cutoff date, compared to the performance on those recorded and patched before this date (and hence, most of the solutions to remediate vulnerabilities and fix-related reports cannot be part of the training data).

We acknowledge that a case study based on one single closed-source system does not generalize to *all* closed-source systems, as the studied system could be an outlier. Yet, we are convinced that with more studies like ours, the community will get a better feeling for the capabilities of LLMs in industrial contexts.

II. Methodology

A. Dataset

To investigate the impact of data contamination on LLM-based vulnerability detection, we used two datasets representing different exposure levels to public training data.

1) Dataset Source:

a) CWE-Bench-Java: CWE-Bench-Java [16] is an open-source benchmark that consists of 120 open-source Java projects. The sizes of these projects span from 511 to 7.7 million Source Lines of Code (SLOC), with an average size of approximately 280 thousand SLOC. Each project contains a Common Vulnerabilities and Exposures (CVE)-recorded vulnerability and detailed information on how this vulnerability was fixed [16]. In this dataset, the featured vulnerability classes are CWE-22 (Path Traversal), CWE-78 (OS Command Injection), CWE-79 (Cross-Site Scripting), and CWE-94 (Code Injection). Furthermore, we excluded 20 projects due to technical problems, such as LLM context window limitation (if a project features extraordinarily big files) or missing access to the full commit history of a project. In the end, 100 CWE-Bench-Java samples remained for our study.

b) TS-Vuls: To create a basis for comparison that is free from potential data contamination, we constructed a closed-source dataset called *TS-Vuls*. This dataset is derived from the source code of Teamscale, a commercial continuous software quality analysis tool developed by CQSE GmbH. The system has a history of more than 10 years, consists of over 1.7 million lines of Java and TypeScript code, and is developed by a team of more than 30 people. It monitors code quality, detects test gaps, and provides feedback to developers [17]. CQSE uses the issue tracking tool Jira to report bugs in the Teamscale source code as tickets. Each ticket concerning a security bug contains detailed information, including the corresponding fix commits and records of the subsequent review processes. Due to Teamscale's closed-source nature, these security issues are not publicly recorded as CVEs. Therefore, these vulnerabilities are not assigned specific CWE identifiers internally, which makes it necessary to implement the broader classification scheme discussed in the II-A3 section. However, every vulnerability included in the TS-Vuls dataset has been validated by either internal or external security experts and fixed by experienced software engineers.

2) Dataset Design: We refer to each documented vulnerability in our datasets as a sample. Each sample includes the necessary metadata to reproduce and evaluate one specific vulnerability within a single project as follows:

- `git_repository` – The full git repository where the vulnerability exists, including its complete commit history and codebase.
- `vul_commit` – A specific commit hash that points to a state of the repository where the vulnerability is present and observable. Our data sources, such as the

TABLE I
CWE-BENCH-JAVA AND TS-VULS

Vulnerability Type	CWE-Bench-Java	TS-Vuls
Access Control	46	25
Injection	54	10
Total	100	35

TABLE II
CVE RECORD PERIOD BREAKDOWN FOR CWE-BENCH-JAVA

Group	CVE reported before 2023	CVE reported 2023 onwards
Access Control	37	9
Injection	44	10
Total	81	19

original CWE-Bench-Java dataset and Teamscale vulnerability reports, do not provide information about the exact commit that introduced a vulnerability for the first time. Therefore, it is not feasible to determine the exact commit that introduced the vulnerability. Consequently, the `vul_commit` is not necessarily the one that introduced the security flaw, but can be any commit between its introduction and the fix.

- `fix_commits` – The set of one or more commits that contain the expert-provided fix for the vulnerability.
- `fix_methods` - The set of all methods that were modified as part of the fix_commits. Each method is uniquely identified by its name, class, and file path.
- `fix_files` – The set of all source code files that contain at least one of the fix methods.
- `cve_year` – The year in which the CVE record associated with the vulnerability was officially published.

This design ensures that every sample is fully reproducible and that the code locations relevant for the vulnerability and its repair are unambiguously defined.

3) Vulnerability Types: We categorize all vulnerabilities into two high-level types: *Access Control* and *Injection*. This grouping is necessary because the closed-source dataset TS-Vuls does not contain a sufficient number of samples for the specific CWE types represented in CWE-Bench-Java. This limitation arose from the difficulty of assigning precise CWE identifiers to internal vulnerabilities, which lack public CVE records.

To ensure categorical comparability between the two datasets, we adopted this higher-level classification inspired by the CWE Mapping and Navigation Guidance [18], which organizes individual CWEs into a comprehensive hierarchical structure. Following this hierarchy, we mapped the specific vulnerabilities from CWE-Bench-Java to our broader categories. CWE-22 was assigned to the Access Control class because it deals with the improper limitation of a pathname to a restricted directory. The remaining three, CWE-78, CWE-79, and CWE-94, are all classic examples of Injection flaws; hence, they were categorized as the Injection class. The vulnerabilities in TS-Vuls were similarly classified into these two categories, creating a consistent basis for comparison. The resulting dataset configurations are detailed in Table I.

4) Temporal Split in Open-Source Dataset: Furthermore, in order to add a second data point for our analysis, we partition the CWE-Bench-Java dataset into two temporal subsets. This partition is based on the publication year of the CVE record associated with the vulnerability in each sample, which we refer to as `cve_year`. The first subset (CVEs before 2023) includes vulnerabilities whose CVE records were published before 2023 (`cve_year < 2023`). The second subset (CVEs from 2023 onwards) contains vulnerabilities with CVE records published in 2023 or later (`cve_year >= 2023`).

This division aligns with the knowledge cutoff dates of our chosen LLMs (see Table III). In particular, vulnerabilities, their corresponding fixed code, and such CVE records published before 2023 may have been part of the training data of those LLMs because a public CVE record is usually disclosed after public references (vulnerability details and fix commits) become available [19]. In contrast, the 2023 onwards subset represents more recent CVE records and corresponding fixed code, potentially not seen by the LLM. Moreover, this subset contains only one vulnerability with a CVE record in 2024, which LLMs with 2023 knowledge cutoffs would not have seen during training, while it potentially falls within the training data of the Claude models, whose cutoff dates are in 2024. Nevertheless, we retain the sample to ensure the full CWE-Bench-Java dataset is used consistently in all analyses, thus avoiding discrepancies in sample size between different parts of our study.

With this consideration in mind, the temporal split allows us to assess whether LLMs perform better on vulnerabilities whose CVE reports and patched code were likely present in their pre-training data, compared to more recent vulnerabilities (2023 onwards). Such recent vulnerabilities have expert-provided fixes and associated CVE reports that were published after the models' knowledge cutoff date, meaning these details were not available during training. The final counts for this configuration are presented in Table II.

B. Vulnerability Detection with LLMs

The primary task for the LLM is to perform fine-grained vulnerability localization at the method level. For each vulnerability sample, we provide the LLM with the content of all source code files that were eventually modified by the expert fix (`fix_files`), extracted from the state of the repository where the vulnerability is present (`vul_commit`) (see Figure 1 for the overview of our approach). The LLM aims to analyze these files and identify the precise set of methods that must be modified to remedy the vulnerability. We define the ground truth for this task as the set of expert-modified methods (i. e., `fix_methods`).

For example, Listing 1 shows the vulnerability reported in CVE-2018-1002201, which is one of the samples from the CWE-Bench-Java dataset. In this sample, the expert's fix is located only in a single method. Therefore, the ground truth for this sample is the `process` method.

```java
// File path: zeroturnaround/src/main/java/org/
     zeroturnaround/zip/ZipUtil.java
// ...
public final class ZipUtil {
  // ...
  public void process(InputStream in, ZipEntry zipEntry)
       throws IOException {
      String root = getRootName(zipEntry.getName());
      if (rootDir == null) {
        rootDir = root;
      }
      else if (!rootDir.equals(root)) {
        throw new ZipException("Unwrapping␣with␣multiple␣
             roots␣is␣not␣supported,␣roots:␣" + rootDir + ",
             ␣" + root);
      }

      String name = mapper.map(getUnrootedName(root,
           zipEntry.getName()));
      if (name != null) {
        File file = new File(outputDir, name);

        // START VULNERABILITY FIX
        if (name.indexOf("..") != -1 && !file.
             getCanonicalPath().startsWith(outputDir.
             getCanonicalPath())) {
          throw new ZipException("The␣file␣"+name+"␣is␣
               trying␣to␣leave␣the␣target␣output␣directory␣
               of␣"+outputDir+".␣Ignoring␣this␣file.");
        }
        // END VULNERABILITY FIX

        if (zipEntry.isDirectory()) {
          FileUtils.forceMkdir(file);
        }
        else {
          FileUtils.forceMkdir(file.getParentFile());

          if (log.isDebugEnabled() && file.exists()) {
            log.debug("Overwriting␣file␣'{}'.", zipEntry.
                 getName());
          }

          FileUtils.copy(in, file);
        }
      }
    }
    // ...
}
```

Listing 1. The Zip Slip vulnerability is fixed by disallowing file names containing the relative traversal string `".."`. Since this fix spans only one method, the sample's set of `fix_methods` contains exactly that method.

This approach was designed as a pragmatic solution to two key challenges. First, an ideal scenario would involve providing the entire code repository to an LLM for analysis. However, this is prevented by the context window limitations of current LLMs. Second, many security vulnerabilities are not confined to a single function but result from complex interactions across multiple methods or files [20]. Our approach approximates a realistic detection scenario by providing the necessary context for identifying such vulnerabilities, framing the task as a focused analysis within a pre-scoped set of relevant files. It is also important to note that the location of a fix may not always be unique. For instance, a vulnerability in a method A that is called by method B could be remediated in either A or B. Since our ground truth is defined by the specific human-implemented patch, our evaluation measures the LLM's ability to identify that particular solution. We acknowledge, however, that other valid and semantically different fixes may exist.

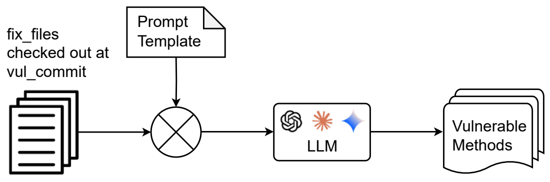

Fig. 1. High-level overview of the vulnerability detection pipeline

C. *Pipeline*

Figure 1 illustrates our automated pipeline for evaluating LLM-based vulnerability detection. The pipeline processes each vulnerability sample by executing a sequence of steps, from data preparation to the final calculation of performance metrics.

a) Input Data Preparation: For each sample, the pipeline first checks out the specific commit (`vul_commit`) from the git repository to ensure the code is in its vulnerable state. It then reads the full content of all source files listed in `fix_files`.

b) Prompting: This phase embeds all retrieved `fix_files` within a standardized prompt that instructs the LLM on its task and required output format.

c) API Interaction: This step sends the complete prompt as a single request to the respective LLM vendor's API. We used commercial APIs from vendors that are under contract with CQSE GmbH.

d) Response Processing: Upon receiving the response, this module parses the structured JSON output to extract the list of methods the LLM identified as vulnerable. It then compares this list against the ground-truth fix_methods. This comparison is used to calculate the number of true Positives (TPs) and False Positives (FPs) for that sample.

e) Metrics Calculation and Export: This phase aggregates the total counts of TPs and FPs across all samples. These aggregated values are then used to compute the overall precision, recall, and F1 score for the model's performance on the dataset.

D. *Prompts*

To ensure a consistent evaluation across all LLMs, we used a single, standardized prompt that combines role-setting instructions, the specific analytical task, and a strict JSON output format (see Listing 2). This is because not all LLM vendor APIs support a distinct system prompt.

The prompt instructs the LLM to act as a security expert and analyze a given set of Java files for vulnerabilities that may span multiple methods. It requires the model to return its findings in a raw JSON format. However, Gemini 1.5 Pro had issues returning a well-formatted JSON string when given the standard prompt used for the other models. To resolve this, we added detailed instructions about the JSON format to ensure the model would comply with the expected output format for our evaluation framework. Lastly, the full code of the chosen `fix_files` for each sample was appended to this prompt in each single request.

```
system_prompt:
You are a security expert specialized in analyzing Java
     code for vulnerabilities. Your task is to identify
     security vulnerabilities that may span across multiple
     methods and files. For each vulnerability you find, you
      must specify all methods that are part of the
     vulnerability pattern, using their exact names as they
     appear in the code. Note that you may return 0 or more
     vulnerabilities in your response. Be accurate and avoid
      false positives, while also ensuring you do not miss
     any vulnerabilities.

analysis_instruction:
Analyze the following Java files for security
     vulnerabilities. A single vulnerability may involve
     multiple methods that need to be fixed together. Return
      your response as raw JSON without any markdown
     formatting. The response must follow this exact format:
{
  "llm_vuls": [
    {
      "llm_fix_methods": [
        {
          "method_name": "exactMethodNameWithoutSignature",
          "class_name": "fully.qualified.ClassName",
          "file_path": "exact/path/to/File.java"
        },
        {
          "method_name": "anotherMethod",
          "class_name": "another.qualified.ClassName",
          "file_path": "another/path/to/File.java"
        }
      ]
    },
    {
      ...
    }
  ]
}
```

Listing 2. Input prompt template

III. Evaluation Setup

This section details the experimental design used to evaluate the impact of data contamination in LLM vulnerability detection. We outline the research questions that guide our study, the metrics used for performance evaluation, the statistical methods for analysis, and the selection and configuration of the LLMs.

A. Research Questions

It is a significant challenge for security teams in industry to determine how reliable LLMs are at detecting vulnerabilities in a closed-source project compared to their widely reported performance on open-source benchmarks. This uncertainty arises from a methodological limitation in prior research — its reliance on open-source benchmarks for evaluation, which come with the risk of data contamination.

To systematically investigate this issue, we present a comparative analysis. As our evaluation of proprietary code is based on a single industrial system, our study can be viewed as a case study. This case study aims to provide insight for future research on the aspect of data contamination in LLMs. Our study is guided by the following research questions:

1) **RQ1 (Dataset-level data contamination impact):** How large is the difference in the effectiveness when the same state-of-the-art LLMs are evaluated on (i) a public open-source benchmark (100 open-source systems) and (ii) a specific proprietary industrial codebase (1.7 million LOC)?
2) **RQ2 (Class-level data contamination impact):** How large is the performance difference between open-source and closed-source datasets for each vulnerability class?
3) **RQ3 (Temporal data contamination impact):** How do LLMs' vulnerability detection capabilities vary based on whether vulnerabilities are fixed and reported before or after their training cutoff dates on the open-source dataset? The rationale here is that better LLM performance on data before the cutoff date ("seen" data) compared to those after this date ("unseen" data) can further support the data contamination hypothesis.

B. Evaluation

1) Metrics: To evaluate the performance of each LLM, we adopt the standard metrics of recall, precision, and the F1 score. These metrics are computed based on true positives and false positives at the sample level.

For each sample i, let r_i denote the repository at commit $\texttt{vul\_commit}_i$, and define the set of Java methods as

$$M_i = \{m \mid m \text{ is a Java method in } r_i\}.$$

Let F_C^i denote the set of fix commits ($\texttt{fix\_commits}_i$), and define the ground truth set of fixed methods as

$$F_M^i = \{m \in M_i \mid m \text{ is modified in any } c \in F_C^i\}.$$

An LLM's response for sample i consists of zero or more predicted vulnerabilities. When evaluating a specific LLM $\mathcal{L}$, we denote its response by

$$R_i^{(\mathcal{L})} = \{V_{i,1}^{(\mathcal{L})}, \dots, V_{i,n_i}^{(\mathcal{L})}\}, \quad V_{i,j}^{(\mathcal{L})} \subseteq M_i,$$

where j is the index for each distinct vulnerability predicted by the LLM for that sample, and n_i represents the total number of such predictions for sample i. Each $V_{i,j}^{(\mathcal{L})}$ denotes a predicted vulnerable method set. Each sample contains one vulnerability, but the LLM is not limited to returning only one vulnerability; hence, it may return multiple vulnerabilities with their methods that are related to such vulnerabilities (i. e., the methods have to be fixed to patch the vulnerability).

True Positives (TP): We define the number of true positives for sample i with respect to $\mathcal{L}$ as follows:

$$\text{TP}(i) = \begin{cases} 1, & \text{if } \exists\, V_{i,j}^{(\mathcal{L})} \cap F_M^i \neq \emptyset, \\ 0, & \text{otherwise.} \end{cases}$$

In other words, if the LLM returns at least one predicted vulnerable method that is included in the ground truth $\texttt{fix\_methods}_i$ for sample i, it is counted as a true positive. We acknowledge that this creates a less strict success criterion, making the task easier since it requires only one overlapping method instead of an exact match of all methods involved in the fix.

TABLE III
LLMs SELECTED FOR THE EXPERIMENT

Model	Vendor	Knowledge Cutoff	Context Window
GPT-4o	OpenAI	October 2023	128,000 tokens
o3-mini	OpenAI	October 2023	128,000 tokens
Claude 3.5 Sonnet	Anthropic	April 2024	200,000 tokens
Claude 3.7 Sonnet	Anthropic	October 2024	200,000 tokens
Gemini 1.5 Pro	Google	November 2023	2,000,000 tokens

False Positives (FP): The number of false positives for sample i with respect to $\mathcal{L}$ is defined as the count of predicted vulnerabilities that do not overlap with the ground truth:

$$\text{FP}(i) = \left|\{V_{i,j}^{(\mathcal{L})} \mid V_{i,j}^{(\mathcal{L})} \cap F_M^i = \emptyset\}\right|.$$

That is, every predicted vulnerability for sample i that fails to include any of the ground truth $\texttt{fix\_methods}_i$ is counted as a false positive. Unlike true positives, there may be more than one false positive per sample.

Assuming n total samples, the evaluation metrics are defined as:

Recall: Recall quantifies the ratio of successfully detected vulnerabilities:

$$\text{Recall}^{(\mathcal{L})} = \frac{1}{n}\sum_{i=1}^{n}\text{TP}(i)$$

Precision: Precision measures the accuracy of the flagged vulnerabilities, defined as the ratio of true positives to the sum of true positives and false positives:

$$\text{Precision}^{(\mathcal{L})} = \frac{\sum_{i=1}^{n}\text{TP}(i)}{\sum_{i=1}^{n}(\text{TP}(i)+\text{FP}(i))}$$

F1 Score: The F1 score combines precision and recall into a single performance measure:

$$\text{F1}^{(\mathcal{L})} = 2\cdot\frac{\text{Precision}^{(\mathcal{L})}\cdot\text{Recall}^{(\mathcal{L})}}{\text{Precision}^{(\mathcal{L})}+\text{Recall}^{(\mathcal{L})}}$$

This balanced measure in the F1 score reflects both the LLM's ability to detect vulnerabilities and its precision in flagging them.

2) Statistical Analysis: To determine if the performance differences investigated in our research questions are statistically significant, we employ the Wilcoxon-Mann-Whitney U test. It is designed to examine if the distribution of a random variable A differs from the distribution of another random variable B by chance. In our case, a random variable $A^{\mathcal{L}}$ is either of our three performance metrics (precision, recall, and F1 score) scored by an LLM $\mathcal{L}$ using a certain set of samples, for example, the CWE-Bench-Java dataset. $B^{\mathcal{L}}$ is analogous, but based on a different set of samples, for example, TS-Vuls.

This non-parametric test is appropriate for our study as it does not assume that the performance data (recall, precision, and F1 scores) follow a normal distribution. It is designed to compare two independent groups, which in our case are the LLM performance scores on CWE-Bench-Java (open-source dataset) versus the scores on the proprietary dataset TS-Vuls.

A crucial aspect of our statistical analysis is that we do not aim to test whether LLMs perform better on open-source software versus proprietary software in general. Such a claim would require a representative sample of many proprietary systems. Instead, our statistical test is focused on the specific datasets we constructed, for instance, multiple open-source systems in CWE-Bench-Java vs. one closed-source system (Teamscale) in the TS-Vuls dataset.

We formulate distinct null hypotheses for RQ1 and RQ3 to align with the specific comparisons in each.

- For RQ1, which compares performance on public versus private datasets, the null hypothesis is $H_{0,RQ1}$: The distributions of LLM performance scores are the same for CWE-Bench-Java (open-source dataset) and TS-Vuls (closed-source dataset).
- For RQ3, which examines temporal data contamination within a single public dataset, the null hypothesis is $H_{0,RQ3}$: The distributions of LLM performance scores are the same for vulnerabilities published as CVE records before 2023 and those published from 2023 onward within the CWE-Bench-Java dataset.

For both, we will reject the null hypothesis if the resulting p-value is less than our chosen significance level of 0.05. A p-value below this threshold would indicate data contamination effects in our case study. Thus, our observed experimental results are unlikely under H_0, thus, we reject it. Conversely, if a p-value is above the threshold ($\mathbf{p \geq 0.05}$), it means that we do not have sufficient evidence to reject the null hypothesis H_0. We apply this test to the results of each LLM for RQ1 and RQ3 by calculating separate p-values for precision (p_P), recall (p_R), and the F1 score (p_{F1}) to analyze the significance for each metric individually.

C. LLM Selection and Setup

Different LLMs differ in architecture and training data, and thus may vary in performance. Therefore, we evaluate more than one LLM with our problem statement. The chosen models are GPT-4o, o3-mini, Claude 3.5 Sonnet, Claude 3.7 Sonnet, and Gemini 1.5 Pro. Furthermore, our LLM selection was guided by two key properties. First, the *context window size* had to be large enough to accommodate our prompts and multiple Java files. Second, to answer RQ3, our selection was driven by the need for models with specific *knowledge cutoff dates* in 2023 and 2024, rather than simply choosing the latest available model.

Access to the LLMs was provided through the official APIs from the model providers. To minimize randomness and improve reproducibility across all models, we set the temperature parameter to 0.

IV. RESULTS

A. RQ1: Dataset-Level Data Contamination Impact

As detailed in Table IV, there was a significant performance drop on the closed-source dataset. The F1 score was approximately 20 percentage points lower on the private dataset across all models on average. This decline was not isolated to a single model but was a consistent trend observed across all five evaluated LLMs. The most significant decrease was observed in Claude-3.5's recall, which dropped from 0.75 on the public benchmark to 0.40 on the private dataset, a relative decline of approximately 47%.

A Wilcoxon-Mann-Whitney U test was used to assess the statistical significance of these performance differences against our null hypothesis $H_{0,RQ1}$, using a significance level of 0.05. For Claude-3.5 and o3-mini, the drops in precision, recall, and F1 score were all statistically significant ($p < 0.05$, thus rejecting the null hypothesis). Although the change in recall was not significant in Claude-3.7, the differences in precision and F1 score were statistically significant ($p_P = 0.0209$ and $p_{F1} = 0.0209$, respectively). For GPT-4o, precision ($p_P = 0.0380$) and recall ($p_R = 0.0351$) differences were significant, while the F1 score difference was not. Conversely, none of the p-values were below the 0.05 threshold for Gemini-1.5 in all metrics. We therefore did not reject the null hypothesis for this model, meaning that none of the performance differences for Gemini-1.5 were statistically significant.

B. RQ2: Class-Level Data Contamination Impact

We analyzed performance separately for Access Control and Injection classes to understand if data contamination effects varied by the type of vulnerability. The detailed results for this comparison are presented in Table V.

For the Access Control class, all evaluated LLMs demonstrated a significant and consistent performance degradation on the closed-source TS-Vuls dataset. The most substantial drop was seen in GPT-4o, whose F1 score fell from 0.80 to 0.45, a 44% relative decrease. Similarly, Claude-3.5's recall dropped by approximately 47% (from 0.76 to 0.40).

In contrast, the results for injection vulnerabilities were more varied. While Claude-3.5, Gemini-1.5, and o3-mini experienced a performance drop on the closed-source dataset, GPT-4o demonstrated a slight improvement across precision, recall, and F1 score on the TS-Vuls samples.

In summary, the performance decline was considerably greater for the Access Control than for the Injection class. The average drop in F1 score for Access Control was 27 points, whereas it was approximately 16 points for Injection. The gap in recall was largest for Access Control, which decreased by an average of 28.2 points, compared to only 14.8 points for Injection.

C. RQ3: Temporal Data Contamination Impact

To analyze the temporal dimension of data contamination, we partitioned the CWE-Bench-Java dataset into two subsets based on the CVE reported year (before 2023 vs. 2023 onwards). As shown in Table VI, all tested LLMs demonstrated a consistent decline in performance when detecting vulnerabilities recorded from 2023 onwards.

The most noticeable performance drops were observed in o3-mini and GPT-4o. The F1 score in o3-mini fell by 26 percentage points (from 0.70 to 0.44), and that difference was statistically significant ($p_{F1} = 0.0288$). Similarly, its recall decreased significantly by 25 points ($p_R = 0.0335$). In addition, GPT-4o exhibited a statistically significant decline in recall (ΔR=0.22, $p_R = 0.0416$). While the negative deltas suggested a downward trend in Claude models, the evidence was insufficient to reject the null hypothesis that performance remains unchanged across the two temporal partitions. Furthermore, the recall in Gemini-1.5 Pro remained identical across both temporal datasets, and none of the changes in its other metrics were statistically significant.

In summary, while the negative deltas suggested a downward trend, only GPT-4o and o3-mini exhibited statistically significant degradation. This outcome contrasts with the findings in RQ1, where performance degradation on the private dataset was statistically significant for most of the evaluated models on at least one metric. In other words, that temporal data contamination affected fewer models than the dataset-level contamination observed in RQ1.

V. Discussion

A. Performance Inflation Through Data Contaminations in Open-Source and Closed-Source Datasets

Our case study results from RQ1 and RQ2 provide strong evidence that data contamination significantly inflates the performance of LLMs in detecting vulnerabilities. We observed a statistically significant decrease in detection effectiveness across four evaluated LLMs when tested on the private, closed-source TS-Vuls dataset, compared to the public CWE-Bench-Java benchmark. Although our case study is limited to the comparison between multiple open-source systems and our closed-source system (Teamscale), this finding suggests that assessing LLMs in vulnerability detection using open-source benchmarks could be problematic by design and may yield overly optimistic results.

B. The Limits of Temporal Cutoffs and Indirect Data Contamination

Detection effectiveness was slightly better for CVEs reported before 2023 than for those reported in 2023 and later across our tested LLMs. Although there were a few differences between models, those differences were marginal. Three of the five evaluated LLMs showed no statistically significant difference across this cutoff date boundary, whereas GPT-4o and o3-mini each exhibited a small but statistically significant post-cutoff decline in specific metrics. When considering together, these outcomes present a mixed picture: a temporal cutoff appears sufficient for some models, yet only partially effective for others. This divergence likely stems from two factors as follows.

First, the vulnerable code for some post-2023 CVE records may have already existed in repositories before the knowledge cutoff date, even though the CVE itself had not yet been reported. The model may have been trained on these patterns of vulnerable code, enabling it to detect the flaw even if it has never seen the actual CVE record. Unfortunately, there was no information available in terms of when a vulnerability was exactly introduced in each sample in the CWE-Bench-Java

TABLE IV
DETAILED LLM PERFORMANCE COMPARISON: CWE-BENCH-JAVA (N=100) VS. TS-VULS (N=35).

	CWE-Bench-Java			TS-Vuls			Delta			p-value		
LLM	P	R	F1	P	R	F1	ΔP	ΔR	ΔF1	p_P	p_R	p_{F1}
Claude-3.5	0.46	0.75	0.57	0.22	0.40	0.28	0.24↓	0.35↓	0.29↓	0.0002	0.0002	0.0002
Claude-3.7	0.39	0.74	0.51	0.26	0.57	0.35	0.13↓	0.17↓	0.15↓	0.0209	0.0634	0.0209
GPT-4o	0.67	0.76	0.71	0.45	0.57	0.51	0.21↓	0.19↓	0.20↓	0.0380	0.0351	0.0851
Gemini-1.5	0.65	0.42	0.51	0.43	0.26	0.32	0.22↓	0.16↓	0.19↓	0.3242	0.0889	0.0957
o3-mini	0.62	0.68	0.65	0.36	0.46	0.40	0.27↓	0.22↓	0.25↓	0.0002	0.0199	0.0043

TABLE V
CWE-BENCH-JAVA VS. TS-VULS PER VULNERABILITY CLASS.

Access Control

	CWE-Bench-Java			TS-Vuls			Delta		
LLM	P	R	F1	P	R	F1	ΔP	ΔR	ΔF1
Claude-3.5	0.47	0.76	0.58	0.22	0.40	0.28	0.25↓	0.36↓	0.30↓
Claude-3.7	0.40	0.80	0.54	0.25	0.52	0.33	0.16↓	0.28↓	0.20↓
GPT-4o	0.76	0.85	0.80	0.39	0.52	0.45	0.37↓	0.33↓	0.36↓
Gemini-1.5	0.64	0.50	0.56	0.44	0.32	0.37	0.19↓	0.18↓	0.19↓
o3-mini	0.68	0.70	0.69	0.34	0.44	0.39	0.34↓	0.26↓	0.30↓

Injection

	CWE-Bench-Java			TS-Vuls			Delta		
LLM	P	R	F1	P	R	F1	ΔP	ΔR	ΔF1
Claude-3.5	0.46	0.74	0.57	0.22	0.40	0.29	0.24↓	0.34↓	0.28↓
Claude-3.7	0.37	0.69	0.48	0.28	0.70	0.40	0.09↓	0.01↓	0.08↓
GPT-4o	0.59	0.69	0.63	0.64	0.70	0.67	0.05↑	0.01↑	0.03↑
Gemini-1.5	0.66	0.35	0.46	0.33	0.10	0.15	0.32↓	0.25↓	0.30↓
o3-mini	0.58	0.67	0.62	0.38	0.50	0.43	0.20↓	0.17↓	0.19↓

dataset. Consequently, we cannot rule out that vulnerabilities reported in 2023 or later were already present in the pre-2023 code used for the models' training. This means some temporal contamination may persist. More specifically, while the CVE reports and their corresponding fixed code for the 2023-onward group mostly appear after the models' cutoff dates, the vulnerable code itself may predate those cutoffs. This is difficult to verify when the vulnerability-introducing commit is unknown or the vulnerability emerged gradually over time.

In addition, another plausible explanation is that a significant portion of the code created after 2023 may already be influenced by content generated by LLMs, making it more aligned with the training data of these models. The widespread adoption of AI programming assistants like GitHub Copilot, which was released in 2021, and ChatGPT, launched in late 2022, has likely accelerated this trend. Consequently, code committed in 2023 and beyond may align more closely with the training data distribution of these models, which could obscure the true impact of using the knowledge cutoff date as a strategy for mitigating data contamination in model evaluation. The increasing popularity of these tools may worsen the risks of data contamination, indicating that simply relying on more recent data is not a guaranteed way to ensure a contamination-free evaluation. Our findings also reinforce the conclusions of previous studies in this field [21], [22].

C. *Varying Susceptibility Across Vulnerability Types*

Our analysis reveals that the impact of data contamination is not uniform across different types of vulnerabilities. GPT-4o, the best-performing model on the public dataset with an F1 score of 0.71, experienced a significant 20-point drop on the private dataset. A closer examination of the class-level results indicates that this decline was primarily due to a considerable drop in the F1 score, which fell by 35 percentage points (from 0.80 to 0.45) in Access Control vulnerabilities. Conversely, the precision, recall, and F1 score for GPT-4o concerning injection vulnerabilities showed a slight improvement on the closed-source dataset. These differences suggest that data contamination's effects may be complex and vary between vulnerability types, even for the same model.

VI. THREATS TO VALIDITY

Like with any empirical study, while we attempted to design the experiment in a way that controls a lot of variables, there are still multiple potential threats to validity.

A. *Internal Threats to Validity*

A major threat is in the selection and design of the task we analyzed. Due to the construction of LLMs, there is no guarantee of continuity across tasks and prompts. We could potentially get significantly different results for other tasks from the software engineering domain or the same task stated with a different prompt. Additionally, our prompt only includes a subset of all files in the repository. Thus, while we include all files in the prompt that were changed when fixing the vulnerability, other potential files that are required to detect the vulnerability could be missing. As the same task and prompt construction have been used for both datasets, the specific results remain valid. However, we cannot prove transferability to other tasks from the same or similar domains.

B. *External Threats to Validity*

Most external threats are in the design of the dataset. The main threat in dataset design is the selection of the software systems from which the vulnerabilities have been picked. While the open-source dataset covers a wider range of repositories, the other dataset was assembled from a single commercial system. In addition, while the commercial system is large (over 1.7 million lines of code), the development team and coding style are relatively consistent across the entire codebase. Additionally, even though their development process follows common best practices, like code reviews, usage of

TABLE VI
DETAILED LLM PERFORMANCE COMPARISON BETWEEN CVEs RECORDED BEFORE 2023 (N=81) AND FROM 2023 (N=19) ONWARDS IN CWE-BENCH-JAVA DATASET.

	CVEs Pre-2023			CVEs 2023-Onwards			Delta			p-value		
LLM	P	R	F1	P	R	F1	ΔP	ΔR	ΔF1	p_P	p_R	p_{F1}
Claude-3.5	0.48	0.78	0.59	0.39	0.63	0.48	0.09↓	0.15↓	0.11↓	0.1352	0.1895	0.1352
Claude-3.7	0.39	0.75	0.52	0.35	0.68	0.46	0.04↓	0.07↓	0.05↓	0.3713	0.5438	0.3713
GPT-4o	0.69	0.80	0.74	0.55	0.58	0.56	0.14↓	0.22↓	0.18↓	0.2503	0.0416	0.0512
Gemini-1.5	0.68	0.42	0.52	0.53	0.42	0.47	0.15↓	0.00 –	0.05↓	1	0.9959	0.9879
o3-mini	0.68	0.73	0.70	0.41	0.47	0.44	0.27↓	0.25↓	0.26↓	0.0898	0.0335	0.0288

SAST tools, and CI/CD, the commercial system could be an outlier for completely different reasons. For example, the contained vulnerabilities could be unusually complex, or the system's domain could be especially challenging for LLMs.

Consequently, the closed-source dataset cannot be assumed to be representative of commercial development in general. However, the results at least raise doubts about evaluations that are based on samples only extracted from public sources (like open-source systems). We cannot mitigate this threat, but instead label our research as a *case study* and take care to not overly generalize our findings.

Another threat is the choice of programming language (namely Java). While the programming language for both datasets was the same, results for other languages might differ, especially if they are less strongly represented in the training data of the LLMs.

Furthermore, a similar problem occurs in the distribution across CWE classes, which is slightly different between the two datasets. To mitigate this, we also report results for the datasets split by CWE classes.

Finally, we included only a limited set of LLMs in our evaluation and limited the experiment to a single prompt template. While we used the same selection of LLMs and prompts for all datasets, results could look different for other LLMs (including future ones) or prompts. However, as we included all of the so-called "frontier models" at the time of the experiment, we expect results to only get worse when used with other existing LLMs. The results might look different when using an LLM that is specifically trained for security analysis. However, our case study is explicitly about the usage of general-purpose LLMs for vulnerability detection, as a lot of work currently suggests the application of those to a wide class of software engineering problems.

VII. RELATED WORK

A. LLM-Based Security Analysis

As mentioned in the introduction, using LLMs for the detection of security vulnerabilities has been studied in multiple papers [2]–[6], [30]. Those approaches rely on existing LLMs, either from commercial providers or open-source/open-weights LLMs, and use a detection pipeline similar to the one in this paper. While those papers differ in the details, e. g. some use additional fine-tuning on the models, there is one commonality in the evaluation part. As can be seen in the datasets used in those papers (which are listed in Table VII), all evaluations have been done purely on data from open-source systems. Additionally, all but two of the datasets are publicly available and could potentially be part of current or future versions of LLM training data. Consequently, those papers neglect the threat of data contamination (and most of them do not even mention this threat). Contrarily, this paper focuses on data contamination and introduces a closed-source dataset, which is guaranteed to not be included in LLM training data. This allows us to study and quantify the effect of assumed data contamination when evaluating LLM-based security analysis.

B. The Effect of Data Contamination

Data contamination refers to test data having been part of an LLM's training dataset, which can lead to misleading performance metrics due to overfitting. Additionally, if only patterns from the training data are used for testing (due to data contamination), we can not learn how an LLM will answer for new patterns outside of the training data.

Many of the most powerful models have been trained by commercial companies, with training data not publicly disclosed. This makes it hard to estimate the amount of data contamination when evaluating those models on different tasks across research disciplines. Several publications have raised concerns about the integrity of LLM evaluation approaches in view of potential data contamination [31]–[33]

To understand the true extent of the problem, researchers started measuring data contamination effects quantitatively [21], [34]. For instance, it was found that LLMs perform significantly better on publicly known coding assignments than on new ones guaranteed not to have been part of their training [35]. Similarly, there is proof that widespread coding benchmarks such as MBPP or HumanEval have been part of most LLMs training datasets [36].

A similar study [37] compared the tasks of code completion and code summarization between open-source and closed-source code examples. In this study, no significant difference was found for samples in the C# programming language, while for C++, a statistically significant decline in performance for the closed-source dataset was observed.

All the mentioned studies either explore tasks that are not related to software engineering or focus only on source code generation and summarization tasks. However, the extent and effect of data contamination for analytical software engineering tasks and specifically vulnerability detection is largely unexplored.

TABLE VII
DATASETS USED IN EXISTING RESEARCH

Name of dataset	Reference	Public	Based on Open Source	Used in
OWASP benchmark	[23]	yes	yes	[2]
SARD Juliet	[24]	yes	yes	[2]
CVEfixes	[25]	yes	yes	[2]–[4]
CWE-snippets	[3]	no	yes	[3]
JVD	[26]	no	yes	[3]
Ponta et al.	[27]	yes	yes	[4]
VCMatch	[28]	yes	yes	[4]
Vul4J	[29]	yes	yes	[5]
PrimeVul	[6]	yes	yes	[6]
Real-Vul	[30]	yes	yes	[30]
Datasets for this paper				
CWE-Bench-Java	[16]	**yes**	**yes**	
TS-Vuls		**no**	**no**	

This case study, in contrast, explicitly explores the effect of data contamination on security vulnerability detection, as a specific case of an analytical software engineering task. To do this, we curated an industrial closed-source evaluation dataset that is guaranteed to not have been part of the evaluated LLMs' training data. Furthermore, we compare these results with performance achieved on open-source CVEs to further contribute to the understanding of LLMs' generalization capabilities in vulnerability detection.

VIII. CONCLUSION

This paper provides a case study on the performance of LLMs for the task of security vulnerability detection. We compared the results between two datasets, based on open-source systems and a closed-source commercial system, respectively. The outcome reveals a statistically significant difference in the performance of the approach between those datasets across four tested LLMs and observed metrics (precision, recall, F1 score). While the extent of degradation varies between the models and also between vulnerability classes, we see a clear indication that performance on the selected closed-source dataset is significantly worse, with a 0.2 drop in the F1 score on average.

Obviously, comparing against a single system does not allow generalization of these findings. Instead, we see this as a first step for the community to get a better feeling for the capabilities of LLMs in industrial contexts. Extending the dataset with a more diverse set of closed-source systems and re-evaluating with this extended dataset is an important next step that we plan to pursue in future work.

To provide a second angle, we evaluated the performance when filtering for vulnerabilities in the open-source system, which were not known at the time the training data for the LLM was assembled (cutoff date). The samples in this dataset span more than 100 projects, and the same date was used for all systems. Hence, the main difference in the dataset before and after the cutoff date is the potential inclusion in the LLMs' training data. In our results, we observe a decline in result quality for the samples published after the cutoff date. Two out of five LLMs' results were found to be statistically significant.

Our temporal-split analysis based on the LLMs' knowledge cutoff date produced mixed results: some LLMs showed statistically significant performance declines in specific metrics, while others exhibited no significant difference; thus, the findings remain inconclusive. These results motivate a deeper investigation into how much open-source-only benchmarks might overstate real-world performance. The ultimate answer, however, can only be given with more studies similar to ours.

DATASET AVAILABILITY

The open-source dataset we used is already available publicly [16]. Contrary to common practice, we decided not to publicly publish our closed-source dataset on the Internet. The reason is not only that this dataset contains intellectual property of CQSE GmbH, but also that by making the data publicly available, we can no longer guarantee its exclusion from LLM training data, which renders the dataset worthless when it comes to measuring data contamination. However, to allow reproduction and comparative studies, we will make the dataset available to individual research groups. Please get in touch with the author from CQSE to obtain the dataset.

ACKNOWLEDGMENT

Many thanks to Stephan Lipp for reviewing early versions of this paper and to Lara Nadine Brummel for her meticulous proofreading.

REFERENCES

[1] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 8, pp. 220:1–220:79, 2024.

[2] A. Khare, S. Dutta, Z. Li, A. Solko-Breslin, R. Alur, and M. Naik, "Understanding the effectiveness of large language models in detecting security vulnerabilities," in *IEEE Conference on Software Testing, Verification and Validation, ICST 2025*. IEEE, 2025, pp. 103–114.

[3] K. Dozono, T. E. Gasiba, and A. Stocco, "Large language models for secure code assessment: A multi-language empirical study," *CoRR*, vol. abs/2408.06428, 2024.

[4] A. Shestov, R. Levichev, R. Mussabayev, E. Maslov, P. Zadorozhny, A. Cheshkov, R. Mussabayev, A. Toleu, G. Tolegen, and A. Krassovitskiy, "Finetuning large language models for vulnerability detection," *IEEE Access*, vol. 13, pp. 38 889–38 900, 2025.

[5] J. Lin and D. Mohaisen, "Evaluating large language models in vulnerability detection under variable context windows," in *Proceedings of the 22nd IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE Press, 2024, pp. 1131–1134.

[6] Y. Ding, Y. Fu, O. Ibrahim, C. Sitawarin, X. Chen, B. Alomair, D. Wagner, B. Ray, and Y. Chen, "Vulnerability Detection with Code Language Models: How Far Are We?" in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, 2025.

[7] Federal Bureau of Investigation, "Internet Crime Report 2024," https://www.ic3.gov/AnnualReport/Reports/2024_IC3Report.pdf, 2024.

[8] Common Vulnerabilities and Exposures (CVE), "Cve statistics," https://www.cvedetails.com/browse-by-date.php, 2025.

[9] "Regulation (EU) 2022/2555 of the European Parliament and of the Council of 14 December 2022 on measures for a high common level of cybersecurity across the Union, amending Regulation (EU) No 910/2014 and Directive (EU) 2018/1972, and repealing Directive (EU) 2016/1148 (NIS 2 Directive)," https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32022L2555.

[10] "Regulation (EU) 2022/2554 of the European Parliament and of the Council of 14 December 2022 on digital operational resilience for the financial sector and amending Regulations (EC) No 1060/2009, (EU) No 648/2012, (EU) No 600/2014, (EU) No 909/2014 and (EU) 2016/1011," https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32022R2554.

[11] J. W. Paulson, G. Succi, and A. Eberlein, "An empirical study of open-source and closed-source software products," *IEEE Trans. Software Eng.*, vol. 30, no. 4, pp. 246–256, 2004.

[12] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima, S. Presser, and C. Leahy, "The pile: An 800gb dataset of diverse text for language modeling," *CoRR*, vol. abs/2101.00027, 2021.

[13] L. Twist, J. M. Zhang, M. Harman, D. Syme, J. Noppen, H. Yannakoudakis, and D. Nauck, "A study of llms' preferences for libraries and programming languages," 2025. [Online]. Available: https://arxiv.org/abs/2503.17181

[14] Y. Li, Y. Guo, F. Guerin, and C. Lin, "An open-source data contamination report for large language models," in *Findings of the Association for Computational Linguistics: EMNLP 2024*, 2024.

[15] Y. Oren, N. Meister, N. S. Chatterji, F. Ladhak, and T. Hashimoto, "Proving test set contamination in black-box language models," in *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*, 2024.

[16] Z. Li, S. Dutta, and M. Naik, "IRIS: LLM-assisted static analysis for detecting security vulnerabilities," in *The Thirteenth International Conference on Learning Representations*, 2025.

[17] L. Heinemann, B. Hummel, and D. Steidl, "Teamscale: software quality control in real-time," in *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE)*, 2014.

[18] MITRE, "Cwe mapping & navigation guidance," https://cwe.mitre.org/documents/cwe_usage/mapping_navigation.html.

[19] Y. W. Heng, Z. Ma, H. Zhang, Z. Li, Tse-Hsun, and Chen, "Discovery of timeline and crowd reaction of software vulnerability disclosures," 2024. [Online]. Available: https://arxiv.org/abs/2411.07480

[20] N. Risse and M. Böhme, "Top score on the wrong exam: On benchmarking in machine learning for vulnerability detection," *CoRR*, vol. abs/2408.12986, 2024.

[21] J. Cao, W. Zhang, and S. Cheung, "Concerned with data contamination? assessing countermeasures in code language model," *CoRR*, vol. abs/2403.16898, 2024.

[22] S. Balloccu, P. Schmidtová, M. Lango, and O. Dusek, "Leak, cheat, repeat: Data contamination and evaluation malpractices in closed-source LLMs," in *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics*, 2024.

[23] OWASP Foundation, "OWASP Benchmark Project," https://owasp.org/www-project-benchmark/.

[24] NIST, "Software Assurance Reference Dataset," https://samate.nist.gov/SARD/.

[25] G. P. Bhandari, A. Naseer, and L. Moonen, "Cvefixes: automated collection of vulnerabilities and their fixes from open-source software," in *PROMISE '21: 17th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2021.

[26] R. Ferenc, P. Hegedüs, P. Gyimesi, G. Antal, D. Bán, and T. Gyimóthy, "Challenging machine learning algorithms in predicting vulnerable javascript functions," *CoRR*, vol. abs/2405.07213, 2024.

[27] S. E. Ponta, H. Plate, A. Sabetta, M. Bezzi, and C. Dangremont, "A manually-curated dataset of fixes to vulnerabilities of open-source software," in *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019*, 2019.

[28] S. Wang, Y. Zhang, L. Bao, X. Xia, and M. Wu, "Vcmatch: A ranking-based approach for automatic security patches localization for OSS vulnerabilities," in *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022*, 2022.

[29] Q. Bui, R. Scandariato, and N. E. D. Ferreyra, "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*, 2022.

[30] P. Chakraborty, K. K. Arumugam, M. Alfadel, M. Nagappan, and S. McIntosh, "Revisiting the performance of deep learning-based vulnerability detection on realistic datasets," *IEEE Transactions on Software Engineering*, 2024.

[31] I. Magar and R. Schwartz, "Data contamination: From memorization to exploitation," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 2022.

[32] O. Sainz, J. Campos, I. García-Ferrero, J. Etxaniz, O. Lopez de Lacalle, and E. Agirre, "NLP evaluation in trouble: On the need to measure LLM data contamination for each benchmark," in *Findings of the Association for Computational Linguistics: EMNLP 2023*, 2023.

[33] J. Dekoninck, M. N. Müller, and M. Vechev, "Constat: Performance-based contamination detection in large language models," in *Advances in Neural Information Processing Systems*, 2024, pp. 92 420–92 464.

[34] S. Golchin and M. Surdeanu, "Time travel in LLMs: Tracing data contamination in large language models," in *Proceedings of the 12th International Conference on Learning Representations (ICLR)*, 2024.

[35] M. Riddell, A. Ni, and A. Cohan, "Quantifying contamination in evaluating code generation capabilities of language models," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 2024.

[36] A. Matton, T. Sherborne, D. Aumiller, E. Tommasone, M. Alizadeh, J. He, R. Ma, M. Voisin, E. Gilsenan-McMahon, and M. Gallé, "On leakage of code generation evaluation datasets," in *Findings of the Association for Computational Linguistics: EMNLP 2024*, 2024.

[37] T. Ahmed, C. Bird, P. T. Devanbu, and S. Chakraborty, "Studying LLM performance on closed- and open-source data," *CoRR*, vol. abs/2402.15100, 2024.