Vulnerability Detection with Code Language Models: How Far Are We?

Yangruibo Ding[†], Yanjun Fu[◇], Omniyyah Ibrahim[♮], Chawin Sitawarin[‡], Xinyun Chen[∇]
Basel Alomair [♮,*], David Wagner [‡], Baishakhi Ray[†], Yizheng Chen[◇]

[†]*Columbia University*
[*]*University of Washington*
[♮]*King Abdulaziz City for Science and Technology*
[∇]*Google DeepMind*
[‡]*UC Berkeley*
[◇]*University of Maryland*

***Abstract*—In the context of the rising interest in code language models (code LMs) and vulnerability detection, we study the effectiveness of code LMs for detecting vulnerabilities. Our analysis reveals significant shortcomings in existing vulnerability datasets, including poor data quality, low label accuracy, and high duplication rates, leading to unreliable model performance in realistic vulnerability detection scenarios. Additionally, the evaluation methods used with these datasets are not representative of real-world vulnerability detection.**

To address these challenges, we introduce PRIMEVUL, a new dataset for training and evaluating code LMs for vulnerability detection. PRIMEVUL incorporates a novel set of data labeling techniques that achieve comparable label accuracy to human-verified benchmarks while significantly expanding the dataset. It also implements a rigorous data de-duplication and chronological data splitting strategy to mitigate data leakage issues, alongside introducing more realistic evaluation metrics and settings. This comprehensive approach aims to provide a more accurate assessment of code LMs' performance in real-world conditions.

Evaluating code LMs on PRIMEVUL reveals that existing benchmarks significantly overestimate the performance of these models. For instance, a state-of-the-art 7B model scored 68.26% F1 on BigVul but only 3.09% F1 on PRIMEVUL. Attempts to improve performance through advanced training techniques and larger models like GPT-3.5 and GPT-4 were unsuccessful, with results akin to random guessing in the most stringent settings. These findings underscore the considerable gap between current capabilities and the practical requirements for deploying code LMs in security roles, highlighting the need for more innovative research in this domain.

I. INTRODUCTION

In the evolving landscape of software development, code language models (code LMs) have become pivotal in automating various software engineering tasks, fundamentally altering developers' coding approaches [1]. Leading examples like GitHub Copilot [2] and Amazon CodeWhisperer [3] have already integrated into real-world software development, significantly assisting developers in their daily work. Consequently, LM-based vulnerability detection (VD) has gained traction, with researchers utilizing code LMs' expanding capabilities to autonomously identify security vulnerabilities in codebases [4–7].

In this paper, we aim to evaluate whether code LMs are able to detect security vulnerabilities in real code, in settings representative of that needed for real-world use. This exploration is anchored in the belief that while code LMs possess remarkable potential, realizing their full capability to enhance software security necessitates a rigorous validation of their *training* and *evaluation* frameworks against the challenges of real-world software development. In particular, we scrutinize the datasets used to train code LMs and the benchmarks and metrics used to evaluate their effectiveness at vulnerability detection.

Limitation of existing datasets and benchmarks. First, we meticulously analyze existing VD benchmarks [5, 8–11], examining data collection methods, label accuracy, and prevalence of data duplication. Our investigation reveals critical data quality problems, that impact their effectiveness for training and their suitability for evaluating code LMs.

Noisy labels: In VD literature, researchers typically label datasets either automatically or manually. Most large datasets [4, 5, 9] use automatic labeling because manual labeling is too expensive. However, automatic labeling can introduce significant label noise. For instance, datasets like BigVul [9] curate hundreds of thousands of functions from the real world and rely on vulnerability-fixing commits for labeling. However, they suffer from a flawed assumption that each function modified by such a commit corresponds to a (separate) vulnerability. In practice, vulnerability-fixing commits often fix one vulnerability but also make other changes to surrounding code, and existing automatic labeling methods wrongly label that surrounding code as vulnerable. In contrast, manual labeling offers higher accuracy, but its cost means it can only be applied to smaller datasets. For instance, the most accurate prior dataset, SVEN [12], which was manually labeled, covers only 9 Common Weakness Enumerations (CWEs) and comprises only 1.6k samples. This dichotomy presents a challenge: how to acquire high-quality labeled VD data at scale for training code LMs to detect security vulnerabilities.

Data duplication: Furthermore, data duplication is prevalent in these datasets. Our analysis identified significant levels of exact copies and cloned vulnerabilities within the datasets. This is particularly problematic when one copy appears in the training set and another copy in the testing set, as performance metrics become unrepresentative of real-world performance and misrepresent the model's ability to generalize to unseen data. We found that up to 18.9% test samples are leaked from the train set in some benchmarks.

Limitation in existing evaluation metrics. Besides the data quality issue, the evaluation metrics used by current benchmarks fail to capture the practical utility of models at VD.

Accuracy: Many benchmarks report accuracy scores, but accuracy is not an appropriate metric for vulnerability detection, because of the base rate problem (vulnerabilities are rare in practice; most code is not vulnerable) and because of mismatches in class balance (the proportion of vulnerable samples in most research datasets does not match the ratio of vulnerable code in real life). For instance, because most samples in realistic benchmarks are not vulnerable, it is possible to achieve high accuracy by always predicting "not vulnerable". A high accuracy score does not necessarily signify effective detection of security vulnerabilities; it may simply reflect the accurate identification of non-vulnerable cases, or a bias towards predicting "not vulnerable".

F1: While the F1 score is widely perceived to be a better metric for assessing classification performance on imbalanced datasets, we argue that it is not appropriate in reality either. The F1 score (the harmonic mean of precision and recall) reflects both false positives and false negatives by combining them into a single penalty. Yet, for VD tools in practice, the overwhelming majority of code is not vulnerable, so a critical challenge is preventing excessive false alarms. The F1 score fails to reflect this asymmetry, so tools with a high F1 score may be useless in practice.

Proposed solution. To address the above limitations, we propose a new VD dataset and novel evaluation guidelines.

New dataset: We introduce PRIMEVUL to tackle the limitations of existing datasets through rigorous data collection, data normalization, and data filtering process. We further introduce two rigorous labeling techniques: (i) PRIMEVUL-NVDCHECK uses expert analysis from CVE entries, and (ii) PRIMEVUL-ONEFUNC utilizes unique changes within commits, ensuring high label accuracy. Consequently, PRIMEVUL not only ensures better label accuracy but also significantly reduces the possibility of data duplication, thereby offering a more realistic and less noisy VD benchmark. To this end, PRIMEVUL contains 6,968 vulnerable and 228,800 benign functions, covering 140 CWEs while maintaining similar accuracy as SVEN, marking a substantial advancement in both scale and accuracy compared to previous datasets.

Novel evaluation guidelines: We propose novel evaluation guidelines, to ensure evaluation results will be predictive of the real-world performance of these tools. First, we suggest splitting samples chronologically to reflect the evolution of both vulnerabilities and coding patterns. Chronological splitting also reduces the risks of data leakage. Second, we introduce the Vulnerability Detection Score (VD-S), a novel metric designed to measure how well vulnerability detectors will perform in practice. VD-S measures the false negative rate, after the detector has been tuned to ensure the false positive rate is below a fixed threshold (e.g., 0.5%). Finally, we introduce a pair-wise evaluation method to assess the model's ability to distinguish between a vulnerable code sample and its benign (fixed) counterpart, offering deeper insights into models' vulnerability understanding.

Effectiveness of Code LMs in realistic vulnerability detection. We evaluate seven code LMs with varied sizes, including the state-of-the-art open-source models like StarCoder2 [13] and the proprietary models from OpenAI [14], on PRIMEVUL. Our findings paint a sobering picture of the current state of code LMs in vulnerability detection. Across various models and experimental setups, code LMs consistently performed poorly, as measured on the PRIMEVUL benchmark. This is in sharp contrast to prior evaluations on prior benchmarks, which reported seemingly good results. For example, StarCoder2, which previously showed promising performance with a 68.26% F1 score on BigVul, drastically underperformed in our assessment, achieving only a 3.09% F1 score on PRIMEVUL.

Our findings highlight the ineffectiveness of existing models in vulnerability detection and the misleading nature of previous benchmarks. This underscores the significance of PRIMEVUL in offering a more challenging and realistic evaluation environment, exposing the limitations of current models when confronted with real-world vulnerabilities. Additionally, the introduction of the Vulnerability Detection Score (VD-S) and pairwise evaluations further elucidated the challenges faced by the models.

Our attempts at enhancing model performance through advanced training techniques, such as class weights and contrastive learning, resulted in marginal improvements at best. We also evaluated state-of-the-art LLMs, including GPT-3.5 and GPT-4, where we employed zero-shot prompting and fine-tuning, hoping to leverage their massive pre-training and model capacity for enhanced performance. However, even these models cannot distinguish vulnerabilities from their similar yet benign counterparts. In our pair-wise evaluation, GPT-4 with the chain-of-thought reasoning could not even outperform the random guessing. This indicates that we are a long way from being able to usefully detect security vulnerabilities with code language models, and fundamentally new approaches may be needed. These findings also prompt a reassessment of the benchmarks used to gauge progress in the field, emphasizing the role of PRIMEVUL in advancing toward more realistic and rigorous evaluations.

To summarize, our contributions in this paper are as below:

- We conducted an in-depth analysis of existing datasets, uncovering significant flaws, including poor data quality, low label accuracy, and a high incidence of data duplication.

- We developed PrimeVul, a new vulnerability dataset with high-quality, accurately labeled data and stringent de-duplication, to offer realistic training and testing data for code LMs.
- We introduced new evaluation guidelines, including the Vulnerability Detection Score (VD-S) and a pair-wise evaluation method, advancing the rigor of model assessment for vulnerability detection.
- We evaluated a range of code LMs with PrimeVul. Our evaluation demonstrates that, despite various attempts at optimization, the performance of current models significantly falls short of the requirements for real-world deployment, highlighting a pressing need for innovative approaches in the training of code LMs for vulnerability detection. We release our artifact at: https://github.com/DLVulDet/PrimeVul

II. Background & Challenges

In this section, we will revisit the existing code-LM-based vulnerability detection models and study the core problems that prevent them from being promising and reliable for realistic deployment.

A. Existing Data Collection Methods

A vast amount of high-quality data is the key to successfully train deep neural networks. Earlier studies [15, 16] train deep-learning models using synthetic datasets such as SATE IV Juliet [17] and SARD [18]. However, synthetic datasets do not adequately capture the complex and nuanced nature of vulnerabilities in real-world code [4]. To address this, a series of benchmarks collect vulnerabilities from real-world, open-source software repositories [4, 5, 8, 9, 19]. Existing vulnerability datasets suffer from one of the two following issues: (1) automated labeling is cheap but too coarse-grained, and (2) manual labeling is reliable but labor-intensive. We elaborate on this trade-off below.

Automated Labeling. The first strategy uses heuristics to automatically label all the samples so that the size of the dataset can be large enough for training deep neural networks. For example, BigVul [9], ReVeal [4], CrossVul [11], CVEfixes [10], and DiverseVul [5] follow this strategy. First, they collect vulnerability-fixing commits from open-source resources, such as the NVD database, Bugzilla, and Debian security trackers. Then, they label the before-commit version of changed functions as vulnerable, and their after-commit version and unchanged functions as nonvulnerable. This strategy assumes that the vulnerability-fixing commits only change vulnerable functions to fix the security flaw. In our label noise analysis (Section II-B), we find that this is often not the case, which introduces wrong labels for vulnerable functions.

Manual Labeling. To improve the quality of the data labels, the second strategy is to involve human experts to verify the commits or functions manually. For example, the authors of SVEN [12] manually labeled 1,606 functions (half of them being vulnerable), from noisy datasets like BigVul and CrossVul. Unfortunately, even for the most experienced security experts, manually verifying the vulnerability labels of code samples is challenging and time-consuming. Therefore, the manually verified datasets, though having higher label accuracy, are not ideal for training the deep neural networks due to the limited diversity (e.g., SVEN only covers 9 CWEs) and a limited number of samples (e.g., SVEN only has 1.6k samples).

B. Label Accuracy

In this section, we quantify the label accuracy of the existing benchmarks.

1) Experiments: We analyze the label accuracy by randomly sampling 50 vulnerable functions from each benchmark and manually analyzing whether the function indeed contains security vulnerabilities. The manual analysis was performed by three of the authors, including two researchers with several years of experience in computer security and one senior security expert.

Human Judgement. We follow the same method from Chen et al. [5] to analyze whether each function is vulnerable and categorize nonvulnerable functions accordingly. Our human annotators comprehensively check the commit message that changed the sampled function, the function before and after the commit, the affiliated CVE, the NVD description, as well as the discussions among the developers in security issue trackers if available. Using these information, our human annotators confirm 1) whether the commit is related to fixing a security vulnerability, and 2) whether the sampled function is indeed vulnerable.

For nonvulnerable functions, we categorize them into 1) vulnerability spread across multiple functions, 2) other changes to make the fix consistent (relevant, but not vulnerable), and 3) irrelevant changes. In the first category, common scenarios where we cannot tell whether a single function is vulnerable include examples such as race condition, denial of service, or command injection that exploits multiple functions, etc. In the second category, we often observe changes to nonvulnerable functions as a side-effect of fixing vulnerable functions, e.g., reporting more errors outside the patched function, or changing the ways of calling the patched function. In the last category, the most common cases are changing spacing and functionality while fixing a different vulnerable function.

Majority Vote. As an improvement over Chen et al. [5], we use three human annotators with majority vote labeling for all datasets except SVEN. We label a function as vulnerable if at least two out of the three human annotators agree that the function is vulnerable. If any one of the three labelers is unsure about a function, the security expert will lead a discussion among annotators to achieve an agreement. Then, the final decision will be made through a majority vote.

2) Results: Table I summarizes our analysis results. We conduct accuracy analysis over three benchmarks that were originally constructed with some kind of manual labeling from prior work: CodeXGLUE, SVEN, and VulnPatchPairs. For benchmarks that use automated labeling, specifically, BigVul,

TABLE I: The label accuracy across existing vulnerability benchmarks and their comparison with two PRIMEVUL automated labeling techniques. Results show that our new labeling techniques achieve a high accuracy on par with SVEN which requires manual labeling. Moreover, PRIMEVUL contains 16.7 × as many vulnerable C/C++ functions as in SVEN.

Benchmark	Manual	Correct (%)
SVEN [12]	✓	94.0
CodeXGLUE [8, 19]	✓	24.0
VulnPatchPairs [20]	✓†	36.0
BigVul [9]	✗	25.0*
CrossVul [11]	✗	47.8*
CVEFixes [10]	✗	51.7*
DiverseVul [5]	✗	60.0*
PRIMEVUL-ONEFUNC	✗	**86.0**
PRIMEVUL-NVDCHECK	✗	**92.0**

† The VulnPatchPairs dataset takes pairs of functions from CodeXGLUE. The dataset does not involve further manual verification beyond its data resource, CodeXGLUE.
* Refers to label accuracy numbers in Chen et al. [5].

CrossVul, CVEFixes, and DiverseVul, we refer to label accuracy numbers in Chen et al. [5]. We will publicly release our label accuracy analysis results as part of our artifacts.

As shown in Table I, the benchmarks without manual verification have very low accuracy between 25% and 60%, for vulnerable functions. On the other hand, only one out of the three prior datasets that used manual labeling method has a high accuracy: SVEN has a 94% label accuracy for vulnerable functions. The other two datasets, CodeXGLUE and VulnPatchPairs have low accuracy.

Surprisingly, the widely used dataset CodeXGLUE [8] (a.k.a. Devign [19] dataset) has a label accuracy of only 24% for vulnerable functions, even though Zhou et al. [19] had recruited human annotators to label security-related commits. We find that their human annotation on security-related commits is highly inaccurate, such that many commits in CodeXGLUE do not fix security vulnerabilities. Moreover, they adopt the same automated labeling policy as BigVul [9] to label all changed functions as vulnerable. The VulnPatchPairs [20] dataset is derived from CodeXGLUE, and as a result, it is also inaccurate, with a label accuracy of only 36%. Our labeling policy is more stringent than VulnPatchPairs. Similar to CodeXGLUE, we find that 24% of functions from VulnPatchPairs are changed by some security-relevant commits, but the functions are not vulnerable. The vulnerability is either spread across multiple functions or the function is changed to make the fix consistent.

The most accurate dataset from prior work is SVEN, which has only 803 vulnerable functions (total ∼1.6k of vulnerable and non-vulnerable functions) across nine CWEs. In comparison, the other noisy datasets from Table I are much larger. Previous works have used different noisy datasets to fine-tune code LMs. Code LMs trained with these noisy datasets cannot be trusted for realistic deployment. When nearly half of the vulnerable samples in the training data have a wrong label, the quality of the trained model is rather concerning [21].

Due to the space limit, we provide a more detailed label error analysis in the Supplementary Material I.A.

C. Data Leakage

Data leakage has been identified as a significant issue in the area of machine learning for code. We consider two types of leakage: code copy and time travel. Realistic evaluation of vulnerability detection models requires no data leakage to ensure their performances are reasonably measured. We study the data leakage issue of four most frequently used vulnerability benchmarks: BigVul, CVEFixes, CodeXGLUE, and DiverseVul, and they have been used by more than ten code-LM-based vulnerability detection models as training and evaluation material [5–7, 19, 22–27].

Data Splits. To study the data leakage issue specifically in the setting of fine-tuning for vulnerability detection, we need to create the train/validation/test splits. For CodeXGLUE, we directly take the original split from the benchmark [28]. For BigVul, we take the public split [29] used by [6, 7, 27]. For CVEFixes and DiverseVul, we follow the methodology introduced in [5] to randomly split the data with 80%/10%/10%.

1) Code Copy: One main reason for leakage is data duplication [30] since code data is highly repetitive [31, 32], and LMs are known to be good at memorizing the code text [33]. Specifically, leaking exact copies across the training and evaluation set will inevitably inflate the evaluation performance.

Experiments. We study the exact copy of vulnerable functions. To identify the code copy, we exhaustively compare the vulnerable functions in the test set to all training samples. We normalize the formatting characters in the code samples to eliminate its noisy effect and then identify the exact copy if two strings are identical after the normalization.

TABLE II: The statistics of data duplication in existing vulnerability detection benchmarks.

Benchmark	De-dup	Copy (%)
BigVul [9]	✗	12.7
CVEFixes [10]	✗	18.9
CodeXGLUE [8]	✗	0.6
DiverseVul [5]	✓	3.3
PRIMEVUL (Ours)	✓	**0.0**

Results. Table II reveals that existing benchmarks suffer from significant exact copy, up to 18.9% of samples being duplicated. Unfortunately, this simple step is overlooked by prior work. Interestingly, we notice that, with hash-based deduplication, DiverseVul still has 3.3% copies. This is mainly because they did not normalize formatting characters, and the same code with varied spacing will be mapped to distinct MD5 hashes, failing to be identified as copies. Further, we manually check the root cause of such duplication, but we realize the reasons are varied. For example, in CodeXGLUE, we notice two different fixing commits targeting the same

CVE [1] [2]. Consequently, this vulnerability gets sampled twice while one is in the training set and the other is in the test set. Differently, in BigVul, we identified exactly the same functions that being sampled twice before and after commits [3] [4], having contradicting labels. Such noises will confuse the model during training and hurt the model's performance as a result.

2) Time Travel: Existing datasets also have the issue of time travel since they randomly separate functions into train, validation, and test sets. Consequently, it is possible to train on future data and test on past data. It is also possible to have the fixed nonvulnerable function in the training set, and the older vulnerable function in the test set.

In addition, after manual analysis, we find that many code samples from the same commit (example[2]) were randomly separated into train, validation, and test sets. This commit fixes multiple occurrences of the same CVE in different functions, in the same way. In general, developers tend to fix similar issues altogether before merging the changes into the main branch. However, training and testing with samples from the same commit is unrealistic and leaks information from the test time to the training time. In a realistic setting, the models are trained on the historical data to predict future samples.

D. Evaluation in Practical Settings

Nearly all code-LM-based vulnerability detection models use accuracy and F1 score for evaluation. Accuracy measures the proportion of correctly predicted samples (both vulnerable and benign) out of all samples, capturing overall model performance. The F1 score, on the other hand, is a harmonic mean of precision and recall, offering a balanced measure of the model's ability to identify vulnerable samples without excessively misclassifying benign samples. We argue that accuracy and F1 fail to capture properties that developers care about in practice. Rather, developers are generally concerned about (1) the detection error tradeoffs and (2) discriminative power over textually similar code samples across vulnerable and fixed patches.

1) Detection Error Tradeoffs: Balance between false positives and false negatives is critical when deploying vulnerability detection tools. Developers rely on these tools not only to catch as many real vulnerabilities as possible (minimizing false negatives) but also to do so without overwhelming them with false alarms (minimizing false positives). This dual expectation reflects the practical tradeoffs in software development: missed vulnerabilities can lead to security breaches, while too many false alarms can lead to alert fatigue, potentially causing real issues to be ignored. F1 and accuracy offer little insight into the tradeoff between false negatives and positives.

2) Textually Similar Pairs of Vulnerable & Corresponding Patch: Security fixes could involve only minor modifications to the code in many cases, such as adjusting buffer sizes, fixing data types, or adding security checks. These changes often result in a patched version of a function that is textually similar to its vulnerable counterpart. Code LMs, which primarily analyze the textual representation of code, struggle to differentiate between such closely related versions [20, 34, 35]. Only by evaluating models on these paired data, we can expose this weakness of code LMs.

[1] https://github.com/qemu/qemu/commit/71d0770
[2] https://github.com/qemu/qemu/commit/902b27d
[3] https://github.com/php/php-src/commit/3798eb6
[4] https://github.com/torvalds/linux/commit/6062a8d

III. Our Benchmark: PrimeVul

As demonstrated in Section II-B, existing vulnerability benchmarks suffer from two major data quality issues: data duplication and low label accuracy. To address these shortcomings, we introduce a new vulnerability benchmark, PrimeVul. We propose a new automated data collection pipeline to produce high-quality samples with accurate labels.

A. Data Merging and Thorough Data De-duplication

To build a large database, we start by merging all security-related commits and functions changed by them from BigVul [9], CrossVul [11], CVEfixes [10], and DiverseVul [5]. We exclude data from Devign/CodeXGLUE [8, 19] because we discover that a large portion of its commits is unrelated to security issues during our annotation process.

We de-deduplicate code copies as well as functions with only formatting differences. For each commit, we first normalize the changed functions before and after commits by stripping away characters such as spaces, tabs ("\t"), newline characters ("\n"), and carriage return characters ("\r"). Then we compute the MD5 hash of both the pre- and post-commit versions of the changed function. If the pre- and post-commit versions of a function result in identical hash values, we regard this function as unchanged and discard it. Finally, we combine all remaining functions, and further de-duplicate the whole set by the MD5 hashes of the normalized functions. During the de-duplication, we maintain a unique set of hashes. If the hash of a normalized function is already in the set, we exclude it from further processing.

As a result, PrimeVul's thorough data de-duplication ensures that no vulnerable function from the training set can be leaked to the test set (Table II).

B. More Accurate Data Labeling

We propose two new labeling techniques: PrimeVul-OneFunc and PrimeVul-NVDCheck.

PrimeVul-OneFunc: We notice that the previous labeling method has errors particularly when dealing with commits that modify multiple functions. Therefore, PrimeVul-OneFunc regards a function as vulnerable if it's the *only* function changed by a security-related commit.

PrimeVul-NVDCheck: Since human experts have analyzed the CVEs in the NVD database, the vulnerability description in each CVE entry is a reliable reference to label vulnerable functions. We develop PrimeVul-NVDCheck as the following. First, we link security-related commits to their CVE numbers and the vulnerability description in the NVD database. We label a function as vulnerable if it satisfies one of the two following criteria: (1) NVD description explicitly mentions its name, or (2) NVD description mentions its *file* name, and it is the only function changed by the security-related commit in that file.

After applying our two labeling techniques, we obtain two sets of vulnerable functions. Next, we merge the sets and de-duplicate the functions again. We normalize the formatting characters in the functions, and compute their MD5 hashes to identify and remove duplicates. Subsequently, we label the post-commit versions of these identified vulnerable functions, as well as all other unchanged functions within the same commits, as benign. Only a subset of commits from the merged database mentioned in Section III-A, meet the criteria for labeling by our techniques. Commits without any function that matches these criteria are excluded from our dataset.

Our pipeline results in a collection of 6,968 vulnerable and 228,800 benign functions across 755 projects and 6,827 commits. To assess our labeling accuracy, we conducted a manual review following the same process used in Section II-B, with our results presented in Table I. The most accurate prior dataset SVEN has only 417 vulnerable functions in C/C++, and 386 vulnerable functions in Python. Our PRIMEVUL dataset not only matches the label accuracy of SVEN but also significantly expands the collection of vulnerable C/C++ functions by 16.7× compared to SVEN. PRIMEVUL is diverse, containing 140 CWEs (15.6× of SVEN).

TABLE III: Statistics of PRIMEVUL

Split	**All**		**Paired**	
	# Vuln.	**# Benign**	**# Vuln.**	**# Benign**
Train	5,574	178,853	4,354	4,354
Dev	699	24,731	562	562
Test	695	25,216	564	564

IV. NEW EVALUATION GUIDELINES

As discussed in Section II-C, we need new methods to properly evaluate vulnerability detection models in deployment settings. This section proposes new evaluation guidelines.

A. Temporal Splits

To minimize the data leakage issue and formulate a realistic train-evaluate setup for vulnerability detection, we split the train/validation/test set of PRIMEVUL according to the commit date of the samples. Concretely, we find the original commit for each sample and collect the time of that commit, tying it with the sample. Then, we sort the samples according to the commit, where the oldest 80% will be the train set, 10% in the middle will be the validation set, and the most recent 10% will be the test set. We also make sure that the samples from the same commit will not be split into different sets. This ensures that the vulnerability detection model is trained using past data and tested over future data.

B. More Realistic and Challenging Evaluation

1) Vulnerability Detection Score: The primary goal in vulnerability detection is to catch as many real vulnerabilities as possible (can be measured by False Negative Rate, or FNR, where we expect low FNR). Meanwhile, from a practical perspective, a certain level of false positives can be manageable within the development workflow, without causing alert fatigue (typically captured by False Positive Rate, or FPR, where a lower rate is better). Therefore, a metric that focuses on minimizing the false negative rate within a tolerable level of false positives is essential.

To this end, we propose Vulnerability Detection Score (VD-S), that evaluates the False Negative Rate of a vulnerability detector within an acceptable False Positive Rate, i.e., FNR @ $(FPR \leq r)$, where $r \in [0\%, 100\%]$ is a configurable parameter. In this paper, we choose a tolerance rate $r = 0.5\%$ to perform the evaluation in Section V.

2) Paired Functions and Pair-wise Evaluation: As discussed in Section II-D2, evaluating the models on paired functions—vulnerable and benign versions of code—could potentially reveal whether a model merely relies on superficial text patterns to make predictions without grasping the underlying security implications, indicating areas where the model needs improvement to reduce the false positives and false negatives.

We collected 5,480 such pairs in PRIMEVUL, significantly larger than existing paired datasets [12, 20]. Concretely, we match the vulnerable functions with their patches in PRIMEVUL to construct such pairs. As we show in Table III, the paired vulnerable functions are fewer than all vulnerable functions, since not all vulnerable functions have a patch (e.g., a patch could delete the vulnerable function), and we only include those challenging pairs sharing at least 80% of the string between the vulnerable and benign version.

Accordingly, we also propose a pair-wise evaluation method. The core idea is to evaluate the model's predictions on the entire pair as a single entity, emphasizing the importance of correctly identifying both the presence and absence of vulnerabilities in a textually similar context, while recording the model's concrete predicting behaviors.

We define four outcomes of the pair-wise prediction:

- *Pair-wise Correct Prediction (P-C)*: The model *correctly* predicts the ground-truth labels for both elements of a pair.
- *Pair-wise Vulnerable Prediction (P-V)*: The model *incorrectly* predicts both elements of the pair as *vulnerable*.
- *Pair-wise Benign Prediction (P-B)*: The model *incorrectly* predicts both elements of the pair as *benign*.
- *Pair-wise Reversed Prediction (P-R)*: The model *incorrectly* and inversely predicts the labels for the pair.

V. EXPERIMENTAL RESULTS

Considering the better data labeling and closer alignment with the real-world data distribution in PRIMEVUL, coupled with the introduction of new evaluation techniques, we re-assess code LM's performance using PRIMEVUL to gauge their performance in a more realistic setting. To this end, we evaluate the following three Research Questions:

- **RQ1.** How do open-source code LMs perform on PRIMEVUL? (Section V-B)

- **RQ2.** Can employing more advanced training techniques enhance the performance of code LMs in detecting vulnerabilities? (Section V-C)
- **RQ3.** Can larger language models (LLMs) improve vulnerability detection performance? (Section V-D)

A. Study Subject

Datasets. We mainly use PRIMEVUL to conduct our experiments. In RQ1, we additionally use BigVul [9] as a case study of existing benchmarks due to its popularity [5–7, 27, 29]. By comparing it to PRIMEVUL, we illustrate the impact of its limitation on training and evaluating code-LM-based VD models. For RQ2 and RQ3, we will focus on PRIMEVUL to improve code LMs performances for VD.

TABLE IV: The code LMs we will study in this section.

Model	Parameters	Arch	Methods
CODET5 [25]	60 M	Enc-Dec	Fine-tune
CODEBERT [22]	125 M	Encoder	Fine-tune
UNIXCODER [24]	125 M	Encoder	Fine-tune
STARCODER2 [13]	7 B	Decoder	Fine-tune
CODEGEN2.5 [36]	7 B	Decoder	Fine-tune
GPT-3.5 [37]	> 100B	Decoder	Fine-tune Few-shot Prompt Chain-of-thought
GPT-4 [14]	> 100B	Decoder	Few-shot Prompt Chain-of-thought

Models. We will study seven code LMs with varied sizes regarding their capabilities for VD, as shown in Table IV. Specifically, we will fine-tune all open-source models in RQ1 and study the advanced training techniques using UnixCoder in RQ2. In RQ3, we prompt GPT-3.5 and GPT-4 with two settings: (1) two-shot examples (2) and chain-of-thoughts [38] reasoning, and we also fine-tune GPT-3.5 on a subset of PRIMEVUL using the OpenAI API.

Experimental Settings. For all experiments with open-source models, we implement our fine-tuning framework following the existing benchmarks [5, 8], where we will use the learning rate of 2×10^{-5} for all the fine-tuning. For smaller models with less than 7B parameters, we will fine-tune the models for 10 epochs, and for 7B models, we fine-tune for four epochs. For GPT-3.5, we just fine-tune for one epoch, due to the limited budget. We load the model weights from Hugging Face Models[5]. All training tasks are conducted on a cluster with NVIDIA A100 GPUs (80 GB). Experiments with OpenAI models are performed through API using greedy decoding.

B. RQ1: Performance of Open-Source Code LMs on PRIMEVUL

In this RQ, we fine-tune open-source code LMs and evaluate their performances on PRIMEVUL. To illustrate the limitations of existing VD benchmarks on the model training and evaluation, we reproduce the code LMs' performances on BigVul [9], using which as a direct comparison to PRIMEVUL. Specifically, we additionally fine-tune code LMs on BigVul, and evaluate them on both BigVul and PRIMEVUL.

Results. The empirical results are shown in Table V. The rows of "Train=PV" and "Test=PV" in each section are results for code LMs fine-tuned and evaluated on PRIMEVUL, and rows of "Train=BV" are the comparative results for code LMs fine-tuned with BigVul.

TABLE V: Performance of code-LM-based vulnerability detection models in different settings.

Model	Train	Test	Acc ↑	F1 ↑	VD-S ↓	P-C ↑	P-V ↓	P-B ↓	P-R ↓
CT5	BV	BV	95.67	64.93	77.30	24.98	50.90	22.79	1.33
		PV	97.00	5.82	95.97	0.18	3.01	96.10	0.71
	PV	PV	96.67	19.7	89.93	1.06	12.94	84.75	1.24
CB	BV	BV	95.57	62.88	81.77	22.60	48.34	27.83	1.23
		PV	97.04	4.49	95.54	0.35	1.95	96.99	0.71
	PV	PV	96.87	20.86	88.78	1.77	11.35	86.17	0.71
UC	BV	BV	96.46	65.46	62.30	39.60	23.74	33.24	3.42
		PV	97.27	1.94	95.11	0.35	0.35	98.76	0.53
	PV	PV	96.86	21.43	89.21	1.60	12.06	85.11	1.24
SC2	BV	BV	96.20	68.26	69.14	35.23	41.98	20.61	2.18
		PV	97.09	3.09	96.83	0.89	0.89	97.70	0.53
	PV	PV	97.02	18.05	89.64	2.30	8.16	88.30	1.24
CG2.5	BV	BV	96.57	67.30	61.73	40.84	26.02	29.63	3.51
		PV	97.23	1.91	95.68	1.24	0.00	98.76	0.00
	PV	PV	96.65	19.61	91.51	3.01	10.82	84.22	1.95

CT5: CodeT5, CB: CodeBERT, UC:UnixCoder, SC2: StarCoder2, CG2.5: CodeGen2.5. BV: BigVul, PV: PRIMEVUL. VD-S is the Vulnerability Detection Score defined as false negative rate (FNR) @ false positive rate (FPR) $\leq$ 0.5% in Section IV-B1. P-C, P-V, P-B, and P-R are the metrics defined in Section IV-B2 to evaluate the models on paired functions.

Finding-RQ1.1: Code LMs' performance is overestimated on a prior benchmark and perform poorly on PRIMEVUL*:* The comparative performance evaluation of code LMs between PRIMEVUL and BigVul lays bare a startling truth: the proficiency of these models is greatly overestimated by benchmarks like BigVul, which fail to mimic the complexity of real-world vulnerabilities. For example, while StarCoder2 shows a commendable F1 score of 68.26% on BigVul, it plummets to a paltry 3.09% on PRIMEVUL. This precipitous drop is not an isolated case but a trend observed across all models, exemplified by the observed false negative rates.

In addition, even when code LMs are fine-tuned on PRIMEVUL, they fail to achieve the same level of performance as on BigVul. For instance, when trained on PRIMEVUL, StarCoder2's performance shows only a modest improvement in F1 score from 3.09% to 18.05%, which is still markedly lower than the 68.26% F1 score achieved on BigVul. This persistent underperformance, even after fine-tuning, suggests that the models cannot effectively learn from the more complex and realistic distribution of vulnerabilities in PRIMEVUL. This stark discrepancy confirms that code LMs trained on existing benchmarks may develop a false sense of security due to the

[5]https://huggingface.co/models

benchmarks' failure to capture the intricate and diverse nature of vulnerabilities found in the wild.

Finding-RQ1.2: Vulnerability Detection Score (VD-S) offers a more concrete sense of realistic performance: The VD-S emerges as a pivotal metric, capturing the essence of a model's capability in real-world settings, where balancing the FNR and FPR is crucial. For example, CodeBERT reports a high accuracy of 96.86% and a satisfactory F1 of 62.88% on BigVul. Relying on these metrics, CodeBERT seems to be an acceptable candidate for detecting vulnerabilities. However, its astonishing false negative rate of 81.77% (as reported by VD-S) far exceeds the realistic tolerable limits, revealing that it is actually useless in practice, overturning the conclusions drawn from the accuracy and F1. Code LMs' VD-S on PRIMEVUL is even more concerning than on BigVul, highlighting the models' limitations in accurately identifying true vulnerabilities, a critical factor for real-world applications where missing a single vulnerability can have serious repercussions.

In addition, we realize that VD-S is not necessarily correlated with accuracy or F1. The detachment of VD-S from traditional metrics also signals a shift in how we should evaluate code LMs for VD.

Finding-RQ1.3: Code LMs are weak at differentiating vulnerabilities from their similar but benign counterparts: As we have discussed in Section II-D2 and IV-B2, pair-wise evaluation not only offers a lens into the precision of code LMs but also serves as a stress test for their real-world application viability. These metrics crucially reveal whether a model has truly learned to identify security vulnerabilities or is merely recognizing patterns without comprehending the implications—a distinction that's vital for the deployment of code LMs as a reliable security tool.

However, our pair-wise evaluation uncovers a significant deficiency in this aspect. Our results reveal that code LMs frequently misclassify both functions in a pair as vulnerable (P-V) or benign (P-B), indicating an overreliance on textual patterns rather than a substantive understanding of the code's security context. For instance, StarCoder2 reports only 2.30% cases to correctly label both elements in paired functions while misclassifying 4× more cases as both vulnerable and 88.30% pairs as both benign, demonstrating their difficulty in recognizing the vulnerable patterns and subtle distinctions that a patch can introduce. Such unreasonable behaviors undermine the trustworthiness of these models in realistic deployment. This insight is also crucial as it emphasizes the need for models that go beyond surface-level text comprehension to grasp deeper semantic implications of code changes for reliable vulnerability detection.

Result-1: Code LMs' significant underperformance on PRIMEVUL highlights their limitations when faced with realistic, diverse, and challenging vulnerabilities.

C. RQ2: Exploring to Improve the Performance on PRIMEVUL

Given code LMs' poor performance on PRIMEVUL, we decided to delve deeper into the training process, trying to figure out whether more advanced training techniques could help code LMs achieve promising performance on PRIMEVUL.

To this end, we perform analysis to inspect both the challenging samples within PRIMEVUL and monitor the models' behaviors during the inference time, carefully studying the failing predictions from these models. After these analyses and a comprehensive literature review, we decided to explore two advanced training techniques that have shown effectiveness in helping binary classifications.

1) Exploration-1: Class Weights: When we delve deeper into analyzing the experimental results, one of the notable differences between BigVul and PRIMEVUL is the ratio of vulnerable samples. As we have mentioned in Section II-B, a lot of samples are mislabeled as vulnerable in BigVul and other resources that constitute PRIMEVUL. After applying our novel labelers, (Section III-B), the ratio of vulnerabilities significantly decreases. Therefore, we suspect whether the significantly more imbalanced ratio hinders the learning process [39].

To verify our assumptions, we implement the weighted loss similar to Chen et al. [5] and integrate it into our fine-tuning framework. The general idea is to give a higher penalty when models make mistakes on the rare class (i.e., the vulnerable samples) by up-weighting the loss value for this class in the cross entropy loss. With a higher weight on the rarer class, the imbalance ratio will be less harmful, since the model pays comparable attention to both classes when optimizing the loss. To find an optimal weight, we tried several different values: besides the standard binary classification with equal weights (weight of vulnerable class = 1), we further explored upweighting the loss for the vulnerable class for 5, 20, 30, and 40 times. Note that the vulnerable to benign ratio in PRIMEVUL is roughly 1:32.

TABLE VI: The impact of class weights during training.

Weight	Acc↑	F1↑	VD-S↓	P-C↑	P-V↓	P-B↓	P-R↓
1	96.86	21.43	89.21	1.60	12.06	85.11	1.24
5	96.24	25.29	90.65	1.77	18.97	78.55	0.71
20	95.28	24.26	88.92	0.89	25.71	72.16	1.24
30	96.14	24.49	90.07	2.13	18.09	78.01	1.77
40	95.99	26.28	88.49	1.42	22.52	74.82	1.24

Findings-RQ2.1: Class weights do not fundamentally improve code LMs' performance on PRIMEVUL*:* Similar to Chen et al. [5], we observed an increased F1 when applying class weights, though not as significant as theirs due to the difficulty of PRIMEVUL. However, VD-S scores warn us that such a marginal improvement is far from promising in the realistic scenario: FNR is oscillating around 90% across different weights, and such models could not be trusted to detect security flaws. These results, though disappointing, help us exclude the potential impact of the class imbalance, providing convincing evidence regarding the difficulty of PRIMEVUL and the struggle of code LMs in detecting realistic vulnerabilities.

2) Exploration-2: Contrastive Learning: Contrastive learning has been proven effective at learning better-quality representations of text and code [24, 34, 40] since they are able to decrease the cosine similarity among semantically different samples, and consequently help to improve the models' performance in downstream classification tasks. Therefore, we hope to study whether contrastive learning could help code LMs to achieve promising performance on PRIMEVUL.

Different from the existing works [24, 34] which apply contrastive learning at the pre-training phase, we enforce such "contrasting" signals together with the classification objective. Specifically, code LMs will be fine-tuned to both classify vulnerabilities and contrast representations with distinct semantics. We implement the objective according to Gao et al. [40] (referred to as CLR), where the model is trained to maximize the representation similarity between each sample and the perturbed version of itself and minimize the similarity between two randomly chosen samples. The perturbation is directly applied to the code representation through dropouts [41] in the Transformer model.

TABLE VII: The impact of contrastive learning during fine-tuning UnixCoder for vulnerability detection.

Finetune	Acc↑	F1↑	VD-S↓	P-C↑	P-V↓	P-B↓	P-R↓
CLS	96.86	21.43	89.21	1.60	12.06	85.11	1.24
+ CLR	96.83	21.73	90.07	1.77	11.35	85.82	1.06
+ CA-CLR	96.64	24.46	90.50	2.30	15.96	81.38	0.35

Findings-RQ2.2: Contrastive learning fails to significantly improve Code LMs' performance on PRIMEVUL*:* Unfortunately, as shown in Table VII, we could not see a significant difference by adding the CLR objective. We further analyze the results to see what might go wrong. One notable misalignment we notice is that, since CLR from Gao et al. [40] is not crafted for classification tasks, it will distinguish any two samples regardless of whether their labels are the same. Therefore, we further improve CLR to be a second approach, called Class-aware Contrastive Learning (CA-CLR), which will only minimize the similarity between samples with different labels.

This time, we see a more notable improvement over F1 and P-C by applying CA-CLR (Table VII). However, the performance change is still marginal. This result empirically demonstrates code LMs' inherent incapability to detect vulnerabilities. It is not only because the representations are too similar to draw the classification boundary, but these models fail to identify the vulnerable patterns, so that, even if enlarging the cosine distance among representations through contrastive learning, code LMs still fail to draw the classification boundary correctly.

Result-2: Advanced training techniques for binary classification, such as contrastive learning and class weights, could not particularly improve code LMs' performance on PRIMEVUL, highlighting its realistic difficulty.

D. RQ3: Larger Code LMs on PRIMEVUL

After getting unsatisfying results with advanced training techniques, we started to question whether the models we tried so far have too few parameters to solve such a challenging benchmark. Therefore, we decided to explore the state-of-the-art large language models (LLM) to see whether significantly more parameters could bring a performance.

We perform experiments using OpenAI GPT models: GPT-3.5 and GPT-4. Considering the cost, we only evaluate these models on the paired functions of PRIMEVUL, since it has much fewer samples than the full set while representing the more challenging scenarios in reality. For GPT-3.5, we experiment with three settings: two-shot prompting, chain-of-thought prompting [38], and fine-tuning. We fine-tune GPT-3.5 for one epoch on all vulnerabilities plus three times more randomly sampled benign samples from the train split of PRIMEVUL, as fine-tuning on the full training set will run out of the budget. For GPT-4, we only consider two-shot and chain-of-thought prompting since its fine-tuning API has not been released yet. More details about the prompt template will be discussed in Supplementary Material I.B.

TABLE VIII: Results of OpenAI GPT models on PRIMEVUL paired functions.

Model	Method	P-C ↑	P-V ↓	P-B ↓	P-R ↓
GPT-3.5	Two-shot	5.67	13.83	77.84	2.66
	CoT	6.21	4.79	83.51	5.50
	Fine-tune	1.24	5.32	90.96	2.48
GPT-4	Two-shot	5.14	71.63	21.45	**1.77**
	CoT	12.94	54.26	24.47	8.33
RANDOM GUESS	-	22.70	26.24	26.42	24.65

Results are shown in Table VIII. In general, GPT-3.5 and GPT-4 outperform open-source models for the pair-wise evaluation even in the basic two-shot prompting setting, and chain-of-thought reasoning further pushes the performance boundary. However, we realize that such performance is actually no better than a random guess since the majority of the pairs in PRIMEVUL still cannot be distinguished by these large SOTA code LMs, which might indicate the fundamental weaknesses of code LMs to differentiate subtle vulnerabilities from their benign versions. Furthermore, when we fine-tune GPT-3.5, we notice that the model is strongly biased by the 1:3 vulnerable to benign ratio and reports even lower performance than the prompting approaches, showing a red flag that even such a large LM (LLM) still fails to capture the vulnerable patterns but take shortcuts from data instead.

Result-3: Even the state-of-the-art OpenAI models could not achieve a reliable performance on PRIMEVUL, calling for fundamentally novel approaches to improve the task.

VI. DISCUSSIONS & THREATS TO VALIDITY

Discussion. Our study of code LMs in the realm of vulnerability detection (VD) reveals that they do not perform well enough for real-world applications. Prior evaluations looked promising, but our work reveals their subtle issues: data quality problems, misleading metrics, and methodology that poorly matches the way in which these models would be used in practice. We highlight below several key areas where current code LMs fall short.

a) Need for More Context: Prior work formulates the problem as: given the code of a single function, determine whether that function contains a security vulnerability. However, this may be asking an impossible question. Determining whether code is vulnerable generally depends on information about other components of the system as well, such as whether inputs to the function have already been sanitized, how outputs will be used, or what invariants are established by the rest of the system. This focus on function-level analysis without the consideration of other contexts (such as interprocedural data flows) would make it difficult for even a human to detect vulnerabilities, let alone a model. We recommend that the problem be reformulated so that the model also has access to a broader context. To enable such a process, PRIMEVUL maintains the metadata for the included commits, providing resources to extract relevant contexts.

b) Augmenting Security Awareness: Our empirical results, particularly the shortcomings of code LMs in pairwise evaluations, suggest that these models make decisions primarily based on textual similarity, without considering the underlying root causes or fixes of the vulnerabilities. We suggest researchers explore ways to teach code LMs about security concepts, such as pre-training methods inspired by how we teach human software developers about security, or ways to build hybrid systems that combine LMs with traditional security analysis or program analysis tools [42, 43].

c) Teaching the model to reason about VD: Finally, posing vulnerability detection as a binary classification problem and teaching the Code LMs accordingly might be too simplistic. This approach banks on the slim hope that a lone summary token or a condensed representation can embody all the intricacies of code vulnerabilities—such expectation might be overly optimistic. Instead, we should decompose the VD problem into digestible sub-problems and teach the model to reason about each step to reach a conclusion [44]. The slight yet encouraging progress observed with the chain of thought experiments in Table VIII shows some promise in this direction.

Threats to validity. Even with our stringent labeling methods, label accuracy is less than 100% (see Table I), so there is still a small portion of mislabeled data in PRIMEVUL. However, given the small percentage of mislabeling, we believe all the reported results will still hold good.

For the proposed evaluation metric VD-S, there is a configurable parameter r to control the maximum false positive rate. The practically acceptable value r might vary for different scenarios, changing the exact value of VD-S, but we expect our general conclusions to hold.

For experiments with OpenAI models, we only reported results with default settings, which could vary slightly by changing the hyperparameters. However, we do not expect hyperparameter tweaking would change the conclusion.

VII. RELATED WORK

Throughout the paper, we have reviewed many related works. Here, we provide a summary of the remaining ones and compare them with our contribution.

Code-LM-based Vulnerability Prediction: Two primary methods to use Code LMs for VD are fine-tuning and prompting. Fine-tuning adds a randomly initialized binary classification head to the language model and jointly optimizes all weights based on ground-truth labels. Various approaches, such as encoder-decoder Transformers [25, 45], encoder-only Transformers [22, 24], and decoder-only Transformers [13, 36, 46] have been used for fine-tuning. On the other hand, prompting-based methods [42–44, 47] rely on LLMs, typically proprietary ones like GPT-4. Previous studies yield mixed results: Khare et al. [42] showed that LLMs perform well on *synthetic datasets* but not promising on real-world datasets. Experimentation with different prompting strategies, notably variations of Chain-of-Thought (CoT) prompts, has further shown promising results [47]. Integrating LLMs into larger frameworks has shown promise in detecting specific vulnerabilities, such as Use Before Initialization vulnerabilities [43] and smart contract vulnerabilities [44]. We analyzed many of these Code LM-based VD methods in this paper and showed that in a very realistic setting none of them performs well.

Empirical Analysis of Deep-learning-based Vulnerability Prediction (DLVP): Several works [4, 6, 20, 27, 48] have pointed out that while DLVP models make correct predictions, they do so for the wrong reasons; they often rely on "spurious features" that are not the root cause of the vulnerabilities. Chen et al. [5] find that, through a large-scale evaluation involving 26k vulnerable functions across 300 projects and 150 CWEs, DLVP lacks generalization to unseen projects and is still far from being deployed in the industry. Some of the prior works [20, 49] focus on the lack of robustness of ML-based vulnerability detection algorithms against semantically preserving modifications. Another recent work [27] attempts to measure how much models pick up the bug semantics through interpretability techniques involving the attention mechanism and shows that extra annotation on the bug semantics also improves the model's performance. Our current work complements these lines of work by focusing more on benchmark creation and evaluation techniques.

VIII. Conclusions

In this paper, we uncover significant limitations in existing vulnerability detection datasets, such as poor data quality, low label accuracy, and high data duplication rates, as well as the limited practical utilities of current evaluation metrics.

In response to these concerns, we present PRIMEVUL, accompanied by updated evaluation criteria designed to more accurately gauge practical effectiveness of Code LM-based vulnerability detectors. Through a series of experiments on PRIMEVUL, we find that even with efforts to improve performance using sophisticated methods and expansive models, existing Code LMs consistently fail to meet the demands of effective vulnerability detection in practical settings. This underscores the urgent requirement for fundamentally innovative approaches in training Code LMs for security applications, while also establishing a new benchmark for evaluating their efficacy.

Acknowledgement

This material is based upon work supported by the National Science Foundation under grants 2229876, 2154873, 2221943, 2313055, 1845893, and 2107405, and by the Department of Homeland Security, IBM, the Center for AI Safety Compute Cluster, the Noyce Foundation, C3.ai DTI, and the KACST-UCB Center of Excellence for Secure Computing. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the sponsors.

References

[1] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review, 2024.

[2] GitHub. Github copilot: Your ai pair programmer. https://copilot.github.com/, 2021.

[3] Amazon. Amazon codewhisperer: Build applications faster and more securely with your ai coding companion. https://aws.amazon.com/codewhisperer/, 2023.

[4] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering*, 2021.

[5] Yizheng Chen, Zhoujie Ding, Lamya Alowain, Xinyun Chen, and David Wagner. Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID '23, page 654–668, New York, NY, USA, 2023. Association for Computing Machinery.

[6] Benjamin Steenhoek, Md Mahbubur Rahman, Richard Jiles, and Wei Le. An empirical study of deep learning models for vulnerability detection. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2237–2248, 2023.

[7] Michael Fu and Chakkrit Tantithamthavorn. Linevul: A transformer-based line-level vulnerability prediction. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, pages 608–620, 2022.

[8] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

[9] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. Ac/c++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 508–512, 2020.

[10] Guru Bhandari, Amara Naseer, and Leon Moonen. Cvefixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 30–39, 2021.

[11] Georgios Nikitopoulos, Konstantina Dritsa, Panos Louridas, and Dimitris Mitropoulos. Crossvul: a cross-language vulnerability dataset with commit data. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1565–1569, 2021.

[12] Jingxuan He and Martin Vechev. Large language models for code: Security hardening and adversarial testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 1865–1879, New York, NY, USA, 2023. Association for Computing Machinery.

[13] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder 2 and the stack v2: The next generation, 2024.

[14] OpenAI. Gpt-4 technical report, 2024.

[15] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.

[16] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 19(4):2244–2258, 2021.

[17] Vadim Okun, Aurelien Delaitre, Paul E Black, et al. Report on the static analysis tool exposition (sate) iv. *NIST Special Publication*, 500:297, 2013.

[18] National Institute of Standards and Technology. Nist software assurance reference dataset, Last accessed on March 19, 2023.

[19] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32, 2019.

[20] Niklas Risse and Marcel Böhme. Limits of machine learning for automatic vulnerability detection, 2023.

[21] Curtis Northcutt, Lu Jiang, and Isaac Chuang. Confident learning: Estimating uncertainty in dataset labels. *J. Artif. Int. Res.*, 70:1373–1411, may 2021.

[22] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin

Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[23] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

[24] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. UniXcoder: Unified cross-modal pre-training for code representation. In Smaranda Muresan, Preslav Nakov, and Aline Villavicencio, editors, *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7212–7225, Dublin, Ireland, May 2022. Association for Computational Linguistics.

[25] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

[26] Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. CodeT5+: Open code large language models for code understanding and generation. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 1069–1088, Singapore, December 2023. Association for Computational Linguistics.

[27] Benjamin Steenhoek, Md Mahbubur Rahman, Shaila Sharmin, and Wei Le. Do language models learn semantics of code? a case study in vulnerability detection, 2023.

[28] Microsoft. Codexglue – defect detection, 2019.

[29] Benjamin Steenhoek. Hugging face datasets, 2024.

[30] Miltiadis Allamanis. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2019, page 143–153, New York, NY, USA, 2019. Association for Computing Machinery.

[31] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 281–293, New York, NY, USA, 2014. Association for Computing Machinery.

[32] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Comput. Surv.*, 51(4), jul 2018.

[33] Md Rafiqul Islam Rabin, Aftab Hussain, Mohammad Amin Alipour, and Vincent J. Hellendoorn. Memorization and generalization in neural code intelligence models. *Information and Software Technology*, 153:107066, 2023.

[34] Yangruibo Ding, Saikat Chakraborty, Luca Buratti, Saurabh Pujar, Alessandro Morari, Gail Kaiser, and Baishakhi Ray. Concord: Clone-aware contrastive learning for source code. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 26–38, New York, NY, USA, 2023. Association for Computing Machinery.

[35] Alex Gu, Wen-Ding Li, Naman Jain, Theo X Olausson, Celine Lee, Koushik Sen, and Armando Solar-Lezama. The counterfeit conundrum: Can code language models grasp the nuances of their incorrect generations? *arXiv preprint arXiv:2402.19475*, 2024.

[36] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. Codegen2: Lessons for training llms on programming and natural languages, 2023.

[37] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback, 2022.

[38] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed H. Chi, Quoc V Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022.

[39] Justin M. Johnson and Taghi M. Khoshgoftaar. Survey on deep learning with class imbalance. *Journal of Big Data*, 2019.

[40] Tianyu Gao, Xingcheng Yao, and Danqi Chen. SimCSE: Simple contrastive learning of sentence embeddings. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2021.

[41] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, jan 2014.

[42] Avishree Khare, Saikat Dutta, Ziyang Li, Alaia Solko-Breslin, Rajeev Alur, and Mayur Naik. Understanding the effectiveness of large language models in detecting security vulnerabilities. *arXiv preprint arXiv:2311.16169*, 2023.

[43] HAONAN LI, YU HAO, YIZHUO ZHAI, and ZHIYUN QIAN. Enhancing static analysis for practical bug detection: An llm-integrated approach. In *Proceedings of Proceedings of the ACM on Programming Languages (PACMPL), Issue OOPSLA*, 2024.

[44] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Wei Ma, Lyuye Zhang, Miaolei Shi, and Yang Liu. Llm4vuln: A unified evaluation framework for decoupling and enhancing llms' vulnerability reasoning. *arXiv preprint arXiv:2401.16185*, 2024.

[45] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. In Kristina Toutanova, Anna Rumshisky, Luke Zettlemoyer, Dilek Hakkani-Tur, Iz Beltagy, Steven Bethard, Ryan Cotterell, Tanmoy Chakraborty, and Yichao Zhou, editors, *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668, Online, June 2021. Association for Computational Linguistics.

[46] Starcoder: may the source be with you! *Transactions on Machine Learning Research*, 2023. Reproducibility Certification.

[47] Saad Ullah, Mingji Han, Saurabh Pujar, Hammond Pearce, Ayse Coskun, and Gianluca Stringhini. Can large language models identify and reason about security vulnerabilities? not yet. *arXiv preprint arXiv:2312.12575*, 2023.

[48] Antonio Valerio Miceli Barone, Fazl Barez, Shay B. Cohen, and Ioannis Konstas. The larger they are, the harder they fail: Language models do not recognize identifier swaps in python. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Findings of the Association for Computational Linguistics: ACL 2023*, pages 272–292, Toronto, Canada, July 2023. Association for Computational Linguistics.

[49] Saad Ullah, Mingji Han, Saurabh Pujar, Hammond Pearce, Ayse Coskun, and Gianluca Stringhini. Can large language models identify and reason about security vulnerabilities? Not yet, December 2023.

[50] Roland Croft, M Ali Babar, and Mehdi Kholoosi. Data quality for software vulnerability datasets. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023.

Supplementary Material

A. *Detailed Label Error Analysis*

As demonstrated in Section II.B and Section III.B of the main paper, we assess 50 vulnerable functions randomly sampled from each dataset, including CodeXGLUE [8, 19], VulnPatchPairs [20] and SVEN [12], along with vulnerable functions identified by our two techniques, PrimeVul-OneFunc and PrimeVul-NVDCheck. The manual analysis results are shown in Table 1 of the main paper. In this section, we provide a more detailed analysis of the wrongly labeled functions.

Chen et al. [5] identify three primary sources of labeling errors. First, irrelevant functions—those subject to formatting, non-functional alterations not linked to security fixes, or contained within commits unrelated to security issues—were incorrectly tagged as vulnerable. Second, the vulnerability may spread across multiple functions, which does not match our goal of training neural networks to learn whether a single function is vulnerable. Third, benign functions undergoing modifications during vulnerability fixes, such as parameter list adjustments for consistency with altered vulnerable functions, were erroneously marked as vulnerable. Building on these three categories of labeling errors, we dissect the inaccuracies identified in our manual analysis, with the detailed breakdown presented in Table IX.

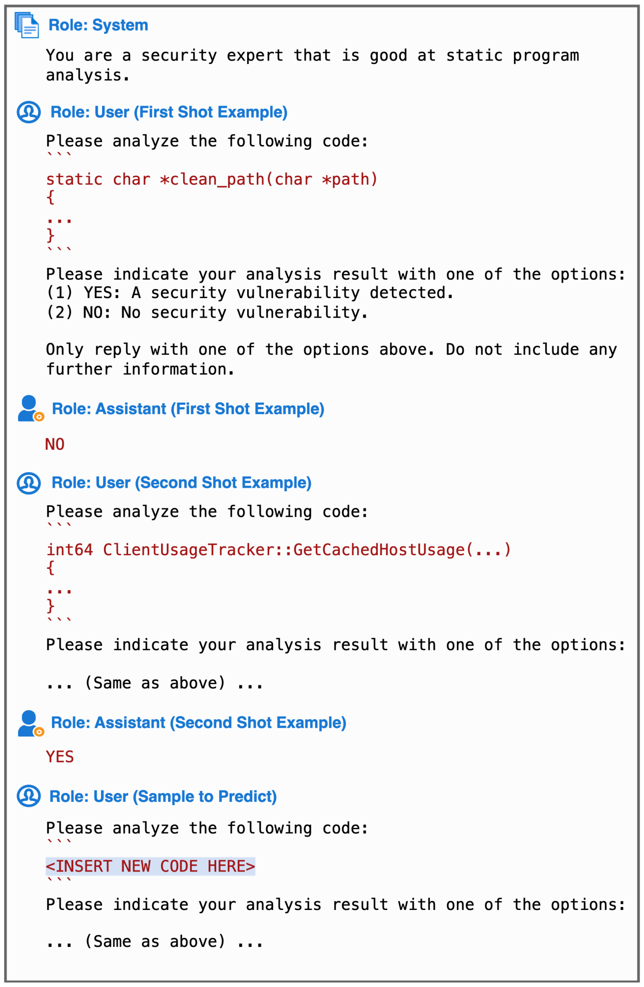
Our manual analysis reveals that only 24% of the 50 data points evaluated from CodeXGLUE and 36% from VulnPatchPairs are genuinely vulnerable. This is noteworthy considering CodeXGLUE's manual efforts in labeling security-related commits and VulnPatchPairs is built on CodeXGLUE. Contrasting our findings, Croft et al. [50] acknowledge a labeling accuracy issue with CodeXGLUE, yet their review finds 80% of their sampled vulnerable functions correctly labeled. This divergence largely stems from our more stringent criteria for identifying vulnerable functions.

First, after a more careful examination, we find that 58% of the samples from CodeXGLUE and 40% from VulnPatchPairs are irrelevant to security and most of them originate from commits unrelated to security issues. Examples include commits mentioning "instrument memory management"[6] or "remove ad-hoc leak checking code".[7] Previous human annotators may misinterpret these commits as security-related due to certain keywords in the commit messages, despite their irrelevance to actual security issues. Additionally, we discover examples like commits focused solely on code migration[8] or operating system compatibility improvements[9], which are not security-related at all.

Second, contrary to the approach taken by Croft et al. [50], our methodology labels a function as vulnerable only if it independently constitutes a security risk. For instance, addressing a race condition, as seen in certain commits[10], demands a comprehensive understanding of the system architecture, making it improper to assess a function's vulnerability in isolation. Previous annotators often mark all functions associated with such race conditions as vulnerable. Another case[11] involves a denial-of-service (DoS) vulnerability linked to the repetitive invocation of the `recvmsg` function. The `qio_channel_websock_encode` function, which merely shifts some values, does not directly lead to a DoS threat. Thus, we categorize `qio_channel_websock_encode` as non-vulnerable, diverging from prior analyses.

Third, Croft et al. [50] label the callers of vulnerable functions as vulnerable. Conversely, we classify such functions as benign because they are altered solely to align with security updates. One case[12] is the `wma_decode_init` function that invokes a vulnerable function `init_vlc`. In our analysis, the function `wma_decode_init` is benign.

[6]https://github.com/qemu/qemu/commit/cd245a1
[7]https://github.com/qemu/qemu/commit/7d1b009
[8]https://github.com/qemu/qemu/commit/60fe637
[9]https://github.com/qemu/qemu/commit/b981289
[10]https://github.com/qemu/qemu/commit/c5a49c6
[11]https://github.com/qemu/qemu/commit/eefa3d8



Role: System

```
You are a security expert that is good at static program
analysis.
```

Role: User (First Shot Example)

~~~
Please analyze the following code:
```
static char *clean_path(char *path)
{
...
}
```
Please indicate your analysis result with one of the options:
(1) YES: A security vulnerability detected.
(2) NO: No security vulnerability.

Only reply with one of the options above. Do not include any
further information.
~~~

Role: Assistant (First Shot Example)

```
NO
```

Role: User (Second Shot Example)

~~~
Please analyze the following code:
```
int64 ClientUsageTracker::GetCachedHostUsage(...)
{
...
}
```
Please indicate your analysis result with one of the options:

... (Same as above) ...
~~~

Role: Assistant (Second Shot Example)

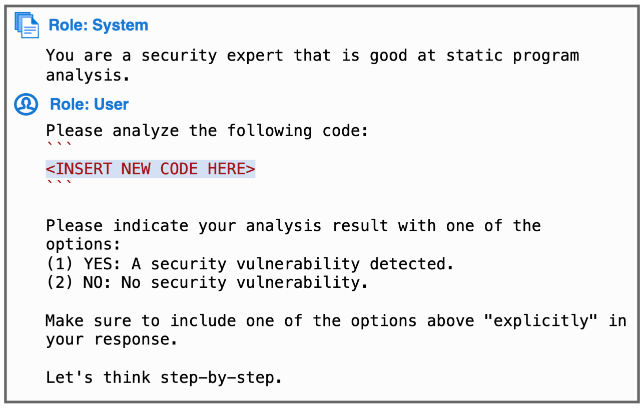
```
YES
```

Role: User (Sample to Predict)

~~~
Please analyze the following code:
```
<INSERT NEW CODE HERE>
```
Please indicate your analysis result with one of the options:

... (Same as above) ...
~~~

Fig. 1: The template for the two-shot prompt.



Role: System

```
You are a security expert that is good at static program
analysis.
```

Role: User

~~~
Please analyze the following code:
```
<INSERT NEW CODE HERE>
```

Please indicate your analysis result with one of the
options:
(1) YES: A security vulnerability detected.
(2) NO: No security vulnerability.

Make sure to include one of the options above "explicitly" in
your response.

Let's think step-by-step.
~~~

Fig. 2: The template for the chain-of-thought prompt

[12]https://github.com/FFmpeg/FFmpeg/commit/073c259

TABLE IX: Detailed breakdown for label error analysis. The error categories are adopted from Chen et al. [5].

Benchmark	**Correct Label**	**Wrong Label**		
		Vulnerability Spread Across Multiple Functions	Relevant Consistency	Irrelevant
SVEN [12]	94%	0%	0%	6%
CodeXGLUE [8, 19]	24%	18%	0%	58%
VulnPatchPairs [20]	36%	10%	14%	40%
PRIMEVUL-ONEFUNC	86%	4%	4%	6%
PRIMEVUL-NVDCHECK	92%	4%	2%	2%

B. Prompts for Open AI Models

In this section, we show the template we used for prompting the OpenAI models.

Figure 1 shows the template of two-shot prompting, where we start from a system prompt to warm up the model with the task it will deal with. Then it is followed by one benign example and one positive example. In the end, the new code to be predicted will be wrapped into the position `<INSERT_NEW_CODE_HERE>`.

Figure 2 shows the template of chain-of-though reasoning for detecting the vulnerability. Similarly, the new code to be predicted will be wrapped into the position `<INSERT_NEW_CODE_HERE>`.