LLM-based Vulnerability Discovery through the Lens of Code Metrics

Felix Weissberg*
BIFOLD & TU Berlin
Berlin, Germany

Lukas Pirch*
BIFOLD & TU Berlin
Berlin, Germany

Erik Imgrund
BIFOLD & TU Berlin
Berlin, Germany

Jonas Möller
BIFOLD & TU Berlin
Berlin, Germany

Thorsten Eisenhofer
BIFOLD & TU Berlin
Berlin, Germany

Konrad Rieck
BIFOLD & TU Berlin
Berlin, Germany

Abstract

Large language models (LLMs) excel in many tasks of software engineering, yet progress in leveraging them for vulnerability discovery has stalled in recent years. To understand this phenomenon, we investigate LLMs through the lens of classic code metrics. Surprisingly, we find that a classifier trained solely on these metrics performs on par with state-of-the-art LLMs for vulnerability discovery. A root-cause analysis reveals a strong correlation and a causal effect between LLMs and code metrics: When the value of a metric is changed, LLM predictions tend to shift by a corresponding magnitude. This dependency suggests that LLMs operate at a similarly shallow level as code metrics, limiting their ability to grasp complex patterns and fully realize their potential in vulnerability discovery. Based on these findings, we derive recommendations on how research should more effectively address this challenge.

CCS Concepts

• **Security and privacy → Vulnerability scanners**.

Keywords

Vulnerability Discovery, Large Language Models, Code Metrics

ACM Reference Format:
Felix Weissberg, Lukas Pirch, Erik Imgrund, Jonas Möller, Thorsten Eisenhofer, and Konrad Rieck . 2026. LLM-based Vulnerability Discovery through the Lens of Code Metrics. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3764574

1 Introduction

Machine learning models are on the verge of becoming an integral part of software development, supporting essential tasks like code completion, refactoring, and auditing. A particularly crucial aspect in this context is the identification of security vulnerabilities early during the development process. Over the past decade, several methods have been proposed to detect security flaws using machine learning, ranging from simple classifiers [25, 48, 54, 69] to large language models (LLMs) for code [18, 26, 39, 63]. However, despite increasingly larger models and broader training data, progress in vulnerability discovery has recently begun to stall, indicating challenges in unlocking the full potential of machine learning for this task [8, 14].

In this paper, we take a step back and ask: After a decade of research, what progress have we made compared to minimal baselines? To this end, we contrast LLMs with a classic tool of software engineering: *code metrics*. Originating from the early days of programming, these metrics quantify basic properties of software, such as the lines of code, the nesting level of loops, or the number of goto statements [27, 29, 40, 57]. In particular, we focus on a generalized family of *syntactic code metrics*, encompassing both historical and current metrics derived from syntax alone [16, 37, 41, 58]. Although these metrics lack the sophistication of LLMs, their simplicity offers an interesting lens for evaluating progress—a viewpoint that has not been explored so far.

For this retrospective investigation, we consider five state-of-the-art LLMs for vulnerability discovery and generative code tasks: PDBERT [38], UniXcoder [26], LineVul [23], CodeGen 2.5 [43], and StarCoder 2 [39]. Among these, UniXcoder is currently regarded as the most effective approach for learning-based vulnerability discovery, following a comprehensive evaluation of LLMs for this task [14]. We compare these models against a set of 23 syntactic code metrics through a series of experiments on PrimeVul [14], the latest benchmark dataset for vulnerability discovery approaches, consisting of 220,000 labeled C/C++ functions from more than 750 open-source projects.

Our findings challenge the prevailing paradigm in prior work that large and sophisticated learning models excel over simple baselines in vulnerability discovery. On the contrary, we find that a classifier trained solely on syntactic code metrics performs on par with the best current LLMs. In other words, state-of-the-art performance can be achieved using simple statistical features of code and just 6% of the model parameters, as shown in Table 1. Surprisingly, even a classifier trained only on a single metric achieves almost the same effectiveness, reaching over 90% of the LLMs' performance. Neither the complex mechanisms of the transformer architecture nor the vast number of model parameters can be fully exploited to significantly outperform the performance of this simple metric, indicating a notable discrepancy to prior work on source-code vulnerability discovery.

ICSE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/26/04
https://doi.org/10.1145/3744916.3764574

*Authors contributed equally.

To gain insights into these unexpected results, we conduct a root-cause analysis and examine causal dependencies between LLMs and code metrics within the framework of Pearl [47]. First, we demonstrate that LLMs can reproduce code metrics and hence have access to the same underlying information. Second, we show that combining LLMs with code metrics does not lead to any improvement, suggesting that they rely on overlapping rather than complementary information. Finally, we demonstrate that the predictions of the best LLMs are not only strongly correlated with code metrics but also causally dependent: When the value of a metric is changed, the LLM predictions tend to shift by a proportional magnitude. This dependency indicates that both approaches rely on equivalent—though not necessarily identical—information for their decision-making process.

While we cannot fully open the black box of LLMs, our analysis reveals that these models do not access information fundamentally different from counting lines or loops in code. The models operate at a surface level of the code, focusing on basic statistical properties rather than uncovering genuine patterns of vulnerabilities. This outcome is both unexpected and disappointing. It raises questions about the considerable resources required for training and deploying LLMs and underscores the need for a deeper understanding of their capabilities in code analysis.

Based on this new perspective, we derive recommendations for the research community: First, code metrics need to be adopted as simple baselines, serving as a sanity check for analyzing performance improvements. To foster this development, we make our implementation and experimental framework publicly available[1]. Second, we should strive for a more balanced perspective when evaluating learning models for vulnerability discovery, ensuring that model complexity is always assessed in relation to detection results. Finally, we must aim to develop learning models that move beyond surface-level statistics, likely necessitating a rethinking of the entire discovery process.

In summary, we make the following contributions:

- *Retrospective evaluation.* We systematically evaluate state-of-the-art LLMs for vulnerability discovery against code metrics, uncovering a surprising similarity in their detection performance (→ Section 4).
- *Unified code metrics.* As basis for our investigation, we introduce a generalized family of code metrics for vulnerability discovery, unifying past and recent work on measuring properties of syntax (→ Section 3).
- *Root-cause analysis.* We analyze the decision making of recent language models through the lens of code metrics and derive recommendations for improving current research (→ Sections 5 and 6).

Note that our work is not intended as a critique of prior research efforts. Rather, we reveal that current approaches have deviated from their intended goals, hindering progress in detecting vulnerable code using machine learning. With our findings and recommendations, we are optimistic that this new perspective will help to better gauge progress and inspire more effective methods for vulnerability discovery.

[1] https://github.com/mlsec-group/cheetah

Table 1: Performance of different vulnerability discovery approaches on the PrimeVul dataset. Factors are relative to UniXcoder, the best performing model in our experiments.

Model	F1 score ↑	Factor	#param ↓	Factor
UniXcoder [26]	20.69 ± 1.43	1.00	125M	1.00
CodeGen 2.5 [43]	18.57 ± 0.54	0.90	7B	53
GPT-4o	5.31 ± 0.35	0.26	≈ 1T	8000
Code metrics	20.32 ± 0.59	0.98	7M	0.06

2 LLM-based Vulnerability Discovery

The discovery of vulnerabilities through static program analysis is a long-standing and notoriously difficult problem in software engineering. Since a universal detection approach is generally unattainable [50], research initially focused on specific defect types, such as buffer overflows [15, 36, 62], integer issues [12, 13, 66], or taint-style vulnerabilities [4, 68]. Recent advances in machine learning, however, have sparked optimism that broader detection methods can be developed by automatically inferring patterns of insecure code from past vulnerabilities. This optimism has been further fueled by the impressive capabilities of LLMs, which have excelled across various application domains. In the following, we briefly review this line of research, discussing relevant approaches and benchmark datasets.

2.1 Discovery Approaches

Learning to detect vulnerabilities is a challenging problem that has evolved significantly over time. Early approaches have relied on classic concepts from text mining and information retrieval to learn from vulnerabilities. For instance, Gruska et al. [25], Yamaguchi et al. [69], and Scandariato et al. [54] use simple bag-of-words and abstract syntax tree representations to identify insecure or anomalous code. However, due to the complexity of many vulnerability types, these initial methods lack the conceptual depth required for broader detection. Hence, over the past decade, numerous studies have addressed these limitations, advancing both data representations and learning models.

Graph-based models. One branch of this research has specifically focused on enhancing data representations by working with code graphs [e.g., 7, 9, 24, 70]. These graphs capture the syntax, control flow, and data flow of code in a unified representation, providing a rich basis for inferring discriminative patterns. Two prominent examples of this approach are *Devign* [70] and *ReVeal* [7], both of which use gated graph neural networks to embed code graphs and identify vulnerabilities on a function level. Although these models showed state-of-the-art performance at the time when they were introduced, more recent work shifted towards language models for this task. Compared to GNN-based approaches, they require significantly less pre-processing of the data, since accurate parsing and control or data flow analyses are generally not required. At the same time, they have been shown to outperform the detection performance of graph neural networks [8].

Large language models. More recent methods for vulnerability discovery therefore leverage pre-trained language models for code and augment them with a classification head [18, 26, 39, 43, 63, 64]. In this work, we consider five state-of-the-art models of this research branch: *LineVul* [23], *PDBERT* [38], *UniXcoder* [26], *CodeGen 2.5* [43], and *StarCoder 2* [39]. The first three are based on the RoBERTa architecture, which has shown the best performance for vulnerability discovery in a recent comparison of language models [14]. We use all three of them as encoder-only models. The last two follow the trend of resorting to larger models and a transformer decoder-only architecture. Besides their architecture and size, the models differ in terms of their training tasks. For example, LineVul uses masked language modeling and denoising tasks for both code and natural language inputs and UniXcoder augments this with code completion as an additional training objective. StarCoder 2 and CodeGen 2.5 take this a step further relying solely on code completion for pre-training.

An alternative to fine-tuning specialized language models is the use of large, general-purpose models. For example, Ding et al. [14] leverage models from the GPT family, such as GPT-4 from OpenAI, combined with chain-of-thought prompting to detect vulnerable code samples. We consider these general approaches as an additional baseline in our comparative evaluation (Section 4.2).

2.2 Benchmark Datasets

As with any application of machine learning, evaluating its efficacy requires representative data of high quality and quantity. Consequently, datasets for vulnerability discovery have been actively developed and improved in recent years. While initial evaluations relied on synthetic datasets like SARD [45] and code labeled by static analyzers [53], recent work has shifted toward constructing benchmark data from publicly available sources of real-world code, such as vulnerability reports and security patches [5, 7, 17, 44, 65, 70]. The resulting datasets comprise thousands of functions from open-source projects both before and after security patches were applied, effectively capturing the differences between vulnerable and non-vulnerable code samples.

PrimeVul. In this work, we employ the most recent benchmark dataset for vulnerability discovery *PrimeVul* [14] (ICSE 2025). It combines several previous datasets, including *BigVul* [17], *CrossVul* [44], *CVEfixes* [5] and the largest available dataset *DiverseVul* [8]. These datasets have been merged and refined through deduplication and commit-filtering techniques to ensure high-quality labeling of secure and vulnerable code. The resulting dataset comprises 224,533 functions from 755 open-source projects, where 6,062 functions contain vulnerabilities. Unlike previous datasets, it enables a more realistic estimation of classifier performance by using a temporal split of commits between training and testing.

3 Metrics for Code

As a counterpoint to LLMs for vulnerability discovery, we consider a classic tool of software engineering: *code metrics*. Originating over four decades ago [27, 40], these metrics offer quantitative insights into various aspects of software design, implementation, and deployment. For example, they can encompass dynamic measures, such as execution time and test suite coverage, as well as static measures, like complexity or coupling and cohesion between components. Moreover, these quantities can be measured at different levels of granularity, ranging from individual functions in a program to entire software modules.

For our investigation, we concentrate on code metrics that analyze the syntactic structure at the function level, consistent with common approaches for learning-based vulnerability discovery. Numerous metrics fall into this category, including historic complexity measures [27, 40], as well as recent approaches designed to indicate insecure code [16, 37, 41, 58]. Interestingly, although these metrics capture a broad spectrum of properties, they differ only in the syntactic elements they analyze and how they aggregate the collected information. Based on this observation, we introduce a generalized family of *syntactic code metrics (SCM)*, which serves as the primary tool for our study.

3.1 A Family of Metrics

To unify the calculation of different code metrics, we propose to express them using the *filter-map-reduce* paradigm from functional programming. Given a syntax tree, a syntactic code metric first *filters* relevant subtrees, *maps* them to numerical values, and then *reduces* the results into a single quantity. Note that specific metrics may require either abstract or concrete syntax trees for their calculation, both of which are supported by our framework and are therefore not explicitly differentiated in the following.

Formally, this calculation can be defined as a function μ that assigns a numerical score $s \in \mathbb{R}$ to a piece of code $c \in C$, based on its syntax tree $t \in T$ as follows

$$\mu : T \mapsto \mathbb{R}, \quad \mu(t) = (R \circ M \circ F)(t),$$

where F represents a filter function over subtrees, M a map function, and R the final reduction to the code metric.

Filter function. The function F traverses a syntax tree and returns all subtrees that satisfy a specified predicate. For example, the function may return specific statements, function calls, or control structures. To provide a unified interface for filtering, we introduce a query language to define predicates, similar to how regular expressions identify patterns in text. Specifically, we define a variant of *S-expressions* for querying syntax trees in our framework. A detailed description of this process and the underlying expressions is provided in Appendix A.

Map function. After filtering, each selected subtree is assigned a value using the function M. This mapping produces a numerical quantity that reflects a specific property of each subtree, such as its presence, depth, size, or complexity. For example, when counting the number of goto statements in a piece of code, the filter F first collects all instances of these statements, while the map M simply returns 1 for each occurrence.

Reduce function. Finally, the values returned from the mapping step are combined into a single score using the function R. This reduction iteratively applies a binary operator to aggregate the values into the final metric output. Common reduction functions include the maximum, sum, and average of the mapped values. In our example of counting goto statements, the reduction R simply corresponds to the sum.

Although the filtering step is essential for isolating specific code constructs, the importance of the map and reduction steps becomes evident when computing more complex aggregates. Several advanced code metrics, such as cyclomatic complexity, maximum loop nesting depth, the number of heap allocations, or the number of pointer dereferences, can be expressed within our framework through a combination of filters, maps, and reductions, as shown in Appendix A.

3.2 Implementation of Metrics

We continue to present the specific code metrics implemented in our framework. Our goal is to capture a diverse range of characteristics that are relevant to understanding and identifying software vulnerabilities. To achieve this, we analyze, unify, and extend several code metrics proposed in prior research [16, 37, 41, 58]. Through this process, we define 23 distinct syntactic code metrics, organized into four categories.

S: Code smell metrics (5). The first category includes code metrics for code smells—patterns in code that indicate underlying design or implementation issues. In particular, we implement metrics for the number of magic numbers (S1), goto statements (S2), and function pointers (S3). Additionally, we include a metric for function calls with unused return values (S4), which may indicate overlooked errors, as well as a metric counting if-statements without corresponding else branches (S5), which may be indicative of incomplete program logic.

C: Complexity metrics (12). Second, we consider metrics that directly measure code complexity [16]. Specifically, we build implementations for the cyclomatic complexity (C1), which quantifies the number of linearly independent control flow paths, as well as metrics for the number and nesting level of loops (C2–C4). Furthermore, we utilize metrics for the number of function parameters (C5), the complexity of control structures (C6–C8), the number of return statements (C9), type casts (C10), local variables (C11), and the maximum number of operands in an expression (C12).

M: Memory metrics (3). Third, we design code metrics that measure the frequency and nature of memory operations. Specifically, we implement metrics for the number of heap allocations (M1), pointer dereferences (M2), and pointer arithmetic operations (M3). These metrics are particularly relevant to vulnerability discovery, as memory-related operations are a common source of security flaws. For example, frequent heap allocations may suggest potential memory management issues, such as leaks or improper deallocations.

T: Syntax tree metrics (3). Finally, we consider metrics characterizing the syntax tree itself, providing structural insights into the underlying code. Specifically, we implement metrics for the number of tree nodes (T1), the maximum height of the tree (T2), and the average number of children per node (T3).

In the appendix, we provide a detailed explanation of each metric, including how they can be represented within our filter-map-reduce framework and the query language based on S-expressions.

Table 2: Performance of different vulnerability prediction approaches on the PrimeVul [14] dataset.

Predictor	Parameters	F1 ↑	AUPRC ↑
Code Metrics			
SCM	6.78×10^6	**20.32** ± 0.59	**13.80** ± 0.82
Code Language Models			
UniXcoder [26]	1.25×10^8	**20.69** ± 1.43	13.32 ± 1.22
PDBERT [38]	1.25×10^8	19.50 ± 1.17	**13.38** ± 0.68
CodeGen 2.5 [43]	6.69×10^9	18.57 ± 0.54	13.22 ± 0.30
LineVul [23]	1.25×10^8	18.48 ± 1.23	11.68 ± 0.82
StarCoder 2 [39]	7.17×10^9	17.09 ± 0.41	12.46 ± 0.40
General-purpose LMs			
GPT-3.5 Turbo	$\approx 10^{10}$	5.80 ± 0.83	2.56 ± 0.18
GPT-4o	$\approx 10^{12}$	5.31 ± 0.35	2.50 ± 0.10
Random	-	4.42 ± 0.19	2.33 ± 0.01

3.3 Learning with Code Metrics

While individual code metrics provide valuable insights into a given piece of code, integrating them within a classifier allows us to consider various combinations and weightings, further enhancing our analysis. To this end, we compile the 23 selected metrics into a numerical feature vector,

$$(\mu_1, \ldots, \mu_{23}) \in \mathbb{R}^{23}$$

For each function in the PrimeVul dataset, we then construct a labeled feature vector, providing the necessary setup for training and evaluating a supervised learning model.

In contrast to the token sequences used in LLMs, this feature representation is straightforward to process with most learning algorithms, accommodating both traditional and modern classifiers. We use the Auto-ML framework [19, 20], which automatically constructs an effective classifier for tabular data by optimizing both the choice of learning models and their hyperparameters. Furthermore, the framework automatically applies data scaling and constructs ensembles of models where necessary.

Specifically, the classifier is constructed through an automated search over a set of fifteen traditional learning models, including fully connected neural networks, random forests, support vector machines, logistic regression, and others. The resulting model uses a mixed ensemble of these models and reaches a total of 7 million parameters when all components are aggregated. Although this model is relatively large considering the limited input space, it remains minuscule compared to the LLMs in our evaluation, which contain up to billions (10^9) or even trillions (10^{12}) of parameters.

4 Retrospective Evaluation

Equipped with a unified representation of code metrics and a resulting classifier, we are ready to put LLMs to the test by comparing their performance to this simple baseline. Unlike typical evaluations that measure relative differences between recent approaches, this retrospective view enables us to assess progress through the lens of long-standing code features and identify where and how improvements have occurred over the last decades.

4.1 Experimental Setup

As a basis for this evaluation, we fine-tune PDBERT, LineVul, CodeGen 2.5, StarCoder 2, and UniXcoder for 10 epochs and select the best performing models based on the validation loss. The code metric model is trained with a 10-minute time budget, and its best candidate is selected based on validation performance as well. For the general-purpose language models, we consider GPT-4o and GPT-3.5 Turbo and reproduce the chain-of-thought experiment for vulnerability classification from Ding et al. [14]. To limit the costs associated with this experiment, we conduct it on a stratified random subset of 2000 samples from PrimeVul's test split.

To ensure robustness of the evaluation and account for randomness, we repeat this process 10 times for each model, employing non-exhaustive cross-validation. The training of all models is performed on a computing cluster using 20 GPUs (NVIDIA A100 class) and 700 CPU cores. Each training run utilizes one GPU and four CPU cores for the LLM-based approaches, while the Auto-ML models are trained using four CPU cores without GPU acceleration.

4.2 Comparative Analysis

The results of our comparative analysis on PrimeVul are presented in Table 2. We report the F1 and AUPRC (area under precision-recall curve) scores, along with the corresponding 90% confidence interval. We select these measures over alternatives, such as true-positive rate and false-positive rate, because they are better suited for analyzing imbalanced datasets [3], as is the case in our study. Moreover, we provide the number of parameters as a reference for the complexity of the learning models.

As a first observation, we find that all models face significant challenges in effectively solving the task of vulnerability discovery. Even the best-performing models achieve F1 scores of only around 20%, highlighting substantial room for improvement. However, when compared to the performance of random guessing with an F1 score of about 4%, the models still show a clear improvement, indicating that several vulnerabilities can be identified successfully.

Upon closer examination, we find that LLMs do not demonstrate statistically significant improvements over our simple classifier based on code metrics. In the experiment, the fine-tuned UniXcoder model achieves the highest F1 score of 20.69%, but its mean performance is surprisingly close to that of the code metrics classifier with an F1 score of 20.32%. LineVul, CodeGen 2.5, StarCoder 2, and PDBERT fall short of this performance. With respect to AUPRC, the SCM-based classifier as well as PDBERT, UniXcoder and CodeGen 2.5 perform almost identically with a maximum difference in mean performance of 0.58 between the first and fourth best model. Considering the 90% confidence interval, the results of the best approaches are hardly distinguishable.

This result is counterintuitive, considering the significantly larger parameter sizes of state-of-the-art models compared to the classifier based on code metrics. Consistent with this finding, we also observe that general-purpose language models are outperformed by all other approaches. Specifically, GPT-4o and GPT-3.5 Turbo exhibit the weakest performance. While this may also seem unexpected given their sheer size, these models were not tuned for this task and therefore cannot compete with specialized LLMs.

Our analysis uncovers an unexpected phenomenon: Despite their fundamentally different data representations and model architectures, vulnerability discovery approaches based on language models and code metrics provide comparable performance on a state-of-the-art benchmark for vulnerability discovery. In other words, 23 input dimensions and an ensemble of basic classifiers give rise to a viable competitor to a pre-trained transformer model with 125 million parameters and a vocabulary of 60 thousand tokens.

4.3 Analysis of Code Metrics

Our findings challenge a prevailing theme in prior research, which focuses on increasing model size and complexity. To better understand these results, we conduct a detailed analysis of the predictive performance of the syntactic code metrics.

Individual performance. As the first experiment, we investigate the performance of individual code metrics on the PrimeVul dataset. To this end, we re-train our classifier on each metric separately and measure their predictive power in isolation. The results of this experiment are presented in Figure 1.

We find that the number of local variables (C11) performs best. This single metric alone achieves over 90% of the F1 score obtained by a model using all code metrics. Notably, it is not the only metric performing well on its own; a total of eight code metrics reach more than 75% of the combined metrics' performance. We conclude that even basic statistical properties, such as the number of local variables (C11), the number of tree nodes (T1), and the depth of nested control structures (C6), enable achieving performance levels close to state-of-the-art methods for vulnerability discovery.

Leave-one-out performance. Second, we conduct a leave-one-out experiment, where each metric is excluded once during classifier training. This allows us to assess the significance of each metric and its interplay within the full set. The results of this experiment are also shown in Figure 1.

The experiment reveals that missing one individual metric does not cause a significant drop in model performance, indicating that the remaining metrics can compensate for the lost information. As a consequence, none of the code metrics, even those that show high discriminative value in isolation, are crucial for a strong classifier. This result suggests that the observed performance of code metrics is not simply due to a few lucky metrics. Instead, it demonstrates the reliability of the overall discovery approach.

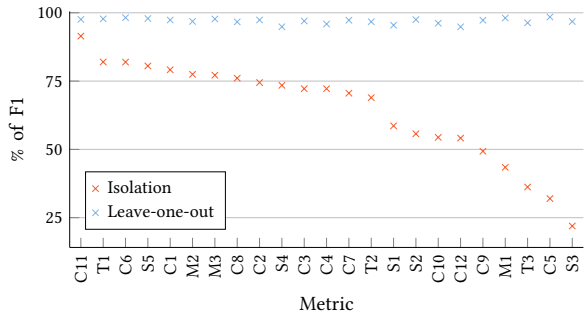

Figure 1: Analysis of the performance impact when trained solely on one code metric (red ●), and when trained on all but one code metric (blue ●).

4.4 Alternative Approaches

For comparison, we consider static application security testing (SAST) tools and graph neural networks (GNNs) as an additional reference point in our evaluation.

Graph neural networks. We additionally evaluate Devign [70] and ReVeal [7], two well-established GNNs for function-level vulnerability discovery. Both models are trained for 100 epochs, with early stopping triggered after 10 epochs of no improvement in validation loss. The underlying graph representations are extracted using the tool *Joern* [34]. If the tool fails to parse the code, we return an inconclusive prediction with a confidence score of 0.5. Consistent with the language model experiments, we repeat this experiment 10 times and investigate mean values.

Overall, we find that the GNNs demonstrate significantly lower F1 scores on the PrimeVul dataset with a maximum of 17.8% (±3.85). While their performance is still much better than random guessing, it is only about as good as the language model with the worst performance in our study. Access to the rich representation of code graphs does not prove beneficial in our experiments. Instead, the simple numerical quantities provided through code metrics offer a more effective representation to identify vulnerable code.

Static application security testing. For the SAST tools, we utilize *Rats* [22], one of the earliest freely available examples in this category, and *SemGrep* [56], a recent representative. For both, we calculate a weighted sum of warnings and errors for each code sample and apply a threshold to classify them. With this approach, the maximum F1 score achieved is 14.2%, placing SAST tools slightly behind GNN-based approaches but still well above general-purpose language models in terms of detection performance.

5 Root-Cause Analysis

Why do LLMs for vulnerability discovery fail to outperform simple code metrics? Theoretically, these models have access to a far broader range of information than code metrics could ever provide. LLMs can analyze variable names, inspect structured data types, and track control flow within code, providing a wealth of insights for separating secure from vulnerable code. However, we observe no significant performance difference between a language model and the classifier trained on code metrics.

We hypothesize that LLMs do not unlock a deeper level of code analysis, despite the information being available. Instead, they rely on basic statistical properties that are equivalent, though not necessarily identical, to those calculated by syntactic code metrics. To support our hypothesis, we identify four essential preconditions and conduct a series of experiments to confirm them:

- P1 *Information access.* LLMs and code metrics can only rely on the same information if both have access to it. In our first experiment, we explore whether equivalent information is present by reconstructing code metrics from the learning models' embeddings.
- P2 *No cross-information gain.* If LLMs and code metrics use complementary information in their decisions, combining them should enhance performance and invalidate our hypothesis. Therefore, we test whether their combination leads to a measurable performance gain.
- P3 *Prediction correlation.* Even if LLMs and code metrics have access to the same information, they may not use it in the same way during inference. In our third experiment, we therefore measure the correlation between code metrics and the predictions of LLMs.
- P4 *Causal dependence.* Correlation only indicates a relationship between the information and the prediction, but not the type of relationship. Thus, to measure the direction of the relationship, we determine the causal dependency between code metrics and language models.

This testing procedure is inspired by the causal hierarchy framework of Pearl [47], encompassing both correlation validation and causal effect inference. While this approach cannot reveal the complex inner workings of LLMs, it helps rule out alternative explanations for their performance similarity to syntactic code metrics. If all preconditions hold, we must conclude that the information LLMs process for vulnerability discovery is not fundamentally different from that provided by code metrics.

5.1 Information Access (P1)

In our first experiment, we aim to determine whether LLMs have access to the same information as the code metrics. Recall that the code language models in our evaluation comprise two components: an embedding model and a classification head. Any information used for prediction must be encoded in the embedding before being passed to the classification head. Therefore, if the models have access to data equivalent to code metrics, that information must be present in the embedding. To test this hypothesis, we train linear regression models to predict the code metrics from the language models' embeddings.

This approach aligns with recent research on the interpretability of large language models. For instance, Park et al. [46] advocate for the *linear representation hypothesis*, which states that semantic information is represented linearly as directions in the models' embedding space. This hypothesis supports the use of linear probes to extract concept-level information from the hidden representations of the models [1].

Experimental setup. We consider all language models for vulnerability discovery from the previous section trained on PrimeVul and fit linear regressors on the penultimate layer of these models using the training split. For evaluation, we use the test split, ensuring a temporal difference between the training and evaluation data.

A hindrance to this training is that most code metrics represent counts of pattern occurrences, so their values typically follow a Poisson distribution rather than a Gaussian distribution. To address this, we train the regression models to predict the logarithm of the metric values, as the log-transformed values better approximate a normal distribution [42]. Furthermore, the metrics are computed only for the respective context of the LLM instead of the complete function. This is because the models lack information beyond their context, making predictions outside of them impossible.

Results. The results of this experiment are summarized in Figure 2, which presents the coefficient of determination for each learned metric. A higher coefficient indicates a better ability to predict the corresponding code metric. The data indicates that most

metrics can be effectively learned. Simpler metrics, such as the number of local variables (C11), the number of if statements without else clauses (S5), and the tree height (T2), are among the best learned. Notably, the metric with the highest performance is the number of nodes (T1), with a coefficient of determination of 0.91 for UniXcoder, 0.86 for PDBERT, and 0.8 for CodeGen 2.5. Additionally, some complex features, including the control structure complexity (C8) and the count of pointer arithmetic operations (M3), are also learned with high accuracy, demonstrating the models' capability to capture intricate patterns.

5.2 No Cross-Information Gain (P2)

In our second experiment, we aim to demonstrate that the information processed by LLMs is not complementary to code metrics. If it were, combining both should improve the performance of vulnerability discovery and invalidate our hypothesis about their relationship. Conversely, if incorporating code metrics into an LLM does not lead to performance gains, this suggests that the relevant information is already present in identical or equivalent form.

Experimental setup. We concatenate the feature vector of the 23 syntactic metrics with the model embeddings from the penultimate layer and retrain the classification heads for all language models using the PrimeVul dataset. We then measure the performance difference between models augmented with code metrics and those without across 10 runs.

Results. We find that the average difference in F1-scores is less than 0.14 percentage points, which is substantially smaller than the variance observed between two independently trained instances of the same model. This result indicates that the code metrics do not provide additional information beyond what is already captured by the language model, suggesting that the available information overlaps rather than being complementary.

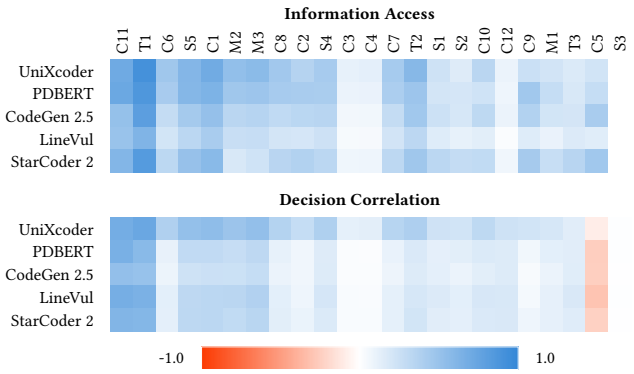

Figure 2: The coefficient of determination for the prediction and the correlation coefficient to the predicted label for each feature. The features are sorted by the mutual information between the feature and the ground truth label with more predictive features on the left.

5.3 Prediction Correlation (P3)

We proceed to investigate whether LLMs and code metrics not only have access to the same information but also rely on it for identifying vulnerabilities. To this end, we analyze the correlation between individual code metrics and the predictions of LLMs. In the framework of Pearl [47], this correlation is an essential prerequisite for demonstrating causal dependence.

Experimental setup. Specifically, we compute the Pearson correlation coefficient between each syntactic code metric and the predictions of the LLMs on the PrimeVul dataset, averaging the coefficients over ten experimental runs.

Results. The results are presented in the lower half of Figure 2. We observe that all models exhibit correlation with the code metrics, where the strength depends on the accessibility of the information. A weaker correlation is observed when the information is inaccessible, whereas accessible information leads to a strong correlation. This suggests that the information available to the models is also used for predictions. Interestingly, the metric C5 (number of parameters) shows a negative correlation, indicating that the models treat it as a negative predictor of vulnerabilities.

5.4 Causal Dependence (P4)

So far, we have confirmed that LLMs have access to information equivalent to code metrics and that their predictions are strongly correlated with them. However, these conclusions are solely based on passive observations. Establishing a causal relationship requires demonstrating the direct impact of code metrics on predictions through an *interventional approach*. This is analogous to a dose-response analysis in epidemiological studies [31]. For instance, if cigarette consumption not only correlates with the development of lung cancer, but also an increase in smoking leads to a higher risk of the disease, a causal effect is likely.

Experimental setup. For our investigation, we formulate causal dependency as follows: If an LLM depends on code metrics for its decisions, then changes in the metrics should systematically influence its predictions. To test this condition, we require a method for modifying code along with its associated metrics and a measure to quantify the influence of this change on LLMs.

(a) Intervening metrics. Ideally, causal dependency is measured by modifying only a single variable, such as a specific metric. However, since code metrics are inherently dependent on the underlying code, it is impossible to change one metric in isolation without affecting others. Additionally, artificially altering code risks creating unnatural samples that fall outside the distribution learned by the models. To ensure that code modifications remain both natural and minimal, we retrieve the previous and subsequent commits for each code sample in PrimeVul from their respective Git repositories. We then apply the corresponding patches to each function, thereby intervening on the code metrics in a realistic manner.

(b) Measuring dependency. Given a piece of code c and its modifications, we can compute two quantities: the difference in predictions of an LLM, denoted as $\Delta y \in \mathbb{R}$, and the difference in code metrics, $\Delta\mu \in \mathbb{R}^{23}$. Since these quantities operate on entirely different scales, we aggregate the metric differences $\Delta\mu$ into a single

value $\Delta s \in \mathbb{R}$ by training a linear regression model to predict Δy from $\Delta \mu$. We then compare the aggregation Δs with Δy using the *coefficient of determination* R^2, a standard measure in statistics for dependence [67]. In our case, the coefficient can be defined as

$$R^2 = 1 - \frac{\sum_c (\Delta y_c - \Delta s_c)^2}{\sum_c (\Delta y_c - \Delta \bar{y})^2}$$

where $\Delta \bar{y}$ is the average of prediction differences. Intuitively, R^2 represents the proportion of variance in the prediction differences that can be explained solely by code metrics. If changes in the code lead to identical variations in both the LLM predictions and the metrics, then $R^2 = 1$. Conversely, if the changes are indistinguishable from the average of all differences, then $R^2 = 0$.

Results. The code metrics achieve an R^2 of 0.42 for UniXcoder, 0.38 for PDBERT, and 0.25 for LineVul, indicating a notable causal dependency. In line with our hypothesis, the strongest dependency is observed for the LLM that achieves the best performance. While the LLMs do not exclusively rely on code metrics for finding vulnerabilities, the metrics account for between 20% and 40% of information in their predictions. Note that these results likely underestimate the true dependency, as the aggregation Δs is based on a linear regression, whereas the actual relationship between the metrics and the predictions is likely more complex.

For the larger models, CodeGen 2.5 and StarCoder 2, we measure R^2 close to 0, indicating no significant dependence. This suggests that either the larger models perform more sophisticated analyses or that our set of 23 metrics misses simple features used by the models. The evaluation results of the larger models support the latter explanation, as more sophisticated analysis should lead to significantly better performance in the vulnerability discovery task. Furthermore, recent studies have shown that models with higher dimensionality exhibit more differentiated and specific features, which may be responsible for their predictions [6].

5.5 Summary

Our root-cause analysis, summarized in Table 3, reveals that all evaluated models have access to information provided by code metrics, exhibit correlation with them, and do not benefit from cross-information. These findings indicate that LLMs predominantly rely on information equivalent but not necessarily identical to the considered syntactic code metrics. For medium-sized LLMs, we identify a clear causal dependency, quantifying the extent to which their predictions are influenced by code metrics. However, this causal indicator is absent in larger models, suggesting that they correlate with code metrics but may rely on alternative information for making their decisions.

Table 3: Summary of the results of the root-cause analysis.

Model	Parameters	P1	P2	P3	P4
UniXcoder	1.25×10^8	✓	✓	✓	✓
PDBERT	1.25×10^8	✓	✓	✓	✓
LineVul	1.25×10^8	✓	✓	✓	✓
CodeGen 2.5	6.69×10^9	✓	✓	✓	✗
StarCoder 2	7.17×10^9	✓	✓	✓	✗

6 Discussion & Recommendations

Our investigation leads to a disappointing outcome: despite the impressive capabilities of language models in other domains, their performance in vulnerability discovery is not significantly different from that of a simple baseline. The substantial resources required to train these models, along with the considerable effort in curating high-quality training datasets, do not yield a substantial advantage over simple techniques developed decades ago.

These findings are not entirely unexpected. Previous studies have highlighted limitations of machine learning for the task of vulnerability discovery, pointing to issues such as inappropriate benchmarks and low predictive performance [3, 7, 14, 33, 51]. Furthermore, the recent study by Ding et al. [14] indicates that scaling language models further up is unlikely to address this issue, as larger models do not automatically perform better on this task. Unfortunately, this generally creates a rather pessimistic outlook. Current research appears to have reached a plateau, and simply expanding the amount of training data, scaling up learning models, or making incremental adjustments does not seem to offer a particularly promising path forward at this stage.

Therefore, we suggest taking a step back to leverage the insights from our analysis in developing new directions unlocking more promising vulnerability discovery approaches. In the following, we summarize these insights into actionable recommendations:

R1: Code metrics are relevant baselines. Despite their simplicity, classifiers using code metrics should be employed as baselines for vulnerability discovery. If we expect modern learning models to uncover complex patterns in code, it is essential to contrast them with simple approaches. Hence, we recommend using classifiers based on code metrics as a sanity check in empirical evaluations. It is worth noting that our approach is automatically optimized and trained within 10 minutes on PrimeVul, adding minimal overhead as a baseline in experimentation.

R2: Occam's razor matters. At first glance, any improvement in vulnerability discovery appears beneficial. However, when comparing different approaches, it is crucial to balance model complexity against detection capabilities [30]. For instance, in our experiments in Section 4, UniXcoder–if at all–only marginally outperforms the classifier based on code metrics. Yet, our model uses 94% fewer parameters than the language model and runs without specialized hardware, greatly enhancing efficiency and facilitating integration into development workflows. Therefore, we recommend to evaluate model performance relative to the model size. Ideally, a model's capabilities should be plotted as a function of its number of parameters or a similarly meaningful measure that considers both model complexity and performance.

R3: Rethinking learning-based discovery. Current research largely approaches the task of identifying vulnerable code as a black-box problem on input samples with limited context information, where learning models are expected to achieve high performance from scratch. The success of general pre-training has further reinforced this perspective, making it challenging to discern whether a model is truly learning patterns of vulnerabilities or merely replicating simple metrics.

To address this issue, we suggest rethinking the application of machine learning in vulnerability discovery:

(1) *Better code representations:* It is unclear whether current representations, such as token sequences and code graphs (for GNNs), provide a suitable basis for learning patterns of vulnerabilities. Based on our findings, there is no evidence that these representations on their own unlock a deeper level for code inspection within the learning models.

(2) *Improvements that matter:* The sole reliance on performance-driven loss functions may limit a model's ability to improve beyond current methods. Ultimately, we are interested in spotting those kinds of defects that remain undetected so far. One strategy could involve making the employed loss function aware of this objective [11] by incorporating feedback from other approaches for vulnerability discovery.

(3) *Steps rather than leaps:* There is increasing evidence that vulnerability discovery cannot be approached as an end-to-end learning task [14]. New intermediate representations and learning steps are likely essential for improving performance and enforcing the required deeper insights into the code that existing approaches as well as code metrics cannot offer.

(4) *Better vulnerability benchmarks:* Lastly, we need to move beyond the artificial setting of detecting vulnerabilities from individual functions alone. As Risse et al. [52] point out, many vulnerabilities used in benchmarks cannot be reliably identified without additional context. That is, code metrics also perform well because essential context, beneficial for more complex models, is simply not available.

To foster this development and further advancements in the field of vulnerability discovery, we make our implementation and experimental framework publicly available[2].

7 Related Work

Critical reflections have a long tradition in security-related research, especially when machine learning techniques are used to address challenging problems. Notable examples of this line of research include critical reviews of machine learning in network intrusion detection [21, 59], website fingerprinting [10, 35], and security applications in general [3]. We continue this line of work by offering a critical reflection on vulnerability discovery through the lens of code metrics, building on prior research that examines data quality and detection capabilities.

Reflections on data quality. Several previous studies have focused on understanding and improving data quality in learning-based approaches to vulnerability discovery. As one of the first, Jimenez et al. [33] propose an improved labeling scheme that accounts for temporal dependencies between vulnerabilities, helping to prevent potential data leaks from other splitting approaches. Building on this, Chakraborty et al. [7] analyze the quality of training datasets, focusing on the sources of labeling and the naturalness of source code samples. They contribute an additional, more realistic dataset and propose a representation learning and sampling technique to fight the strong class imbalance.

Follow-up work by Sejfia et al. [55] and Risse et al. [52] investigate the quality of code samples, revealing that many vulnerabilities cannot be detected from these samples alone. This research provides evidence that detecting a vulnerability often requires assessing multiple parts of a software in combination and consequently, is often infeasible without further context information. Constructing suitable and realistic datasets not only faces the issue of sample quality but a fundamental trade-off between incorporating relevant context and restricting the input size to a manageable amount. Most recently, Ding et al. [14] further enhance the label accuracy of existing benchmark datasets on function-level granularity, resulting in the curation of the PrimeVul dataset that we used in our study.

Reflections on detection capabilities. Another direction of prior work has focused on investigating the capabilities and limitations of learning-based approaches to vulnerability discovery. For example, it has been shown that language models tend to overfit to code variants that are present in the training data, prompting the development of augmentation techniques to mitigate this overfitting by applying code transformations during training that do not semantically alter the code samples [32, 51]. Additionally, Ullah et al. [61] demonstrate that language models produce non-deterministic responses and often provide incorrect or unfaithful reasoning over vulnerable code. Similarly, studies on limitations of graph neural networks describe effects like over-squashing [2], over-smoothing [28] and low performances on heterophilic graphs [49], such as those used in vulnerability discovery.

In contrast to this previous work, our study aims to better understand the low performance of current learning models. By analyzing them through the lens of code metrics, we reveal their tendency to focus on basic statistical properties rather than code analysis. This finding aligns with observations of limited robustness and unclear reasoning from prior studies, adding a new facet to our understanding of why learning models continue to struggle with identifying vulnerabilities.

8 Limitations

Our retrospective analysis of LLMs for vulnerability discovery inherently carries limitations due to its empirical nature. Potential threats to validity arise from the evaluation data as well as the choice of syntactic code metrics used in our experiments and the general possibility of confounding variables in machine learning, which we discuss in the following.

Dataset quality. The quantity and quality of data for machine learning is a critical factor in vulnerability discovery and remains an active research area [3, 7, 8, 14]. Our findings are tied to the characteristics of the employed datasets. While LLMs could potentially outperform code metrics if considerably larger and more representative datasets were available for evaluation, with PrimeVul we are working with the currently largest and most refined dataset in this domain. PrimeVul is the result of substantial efforts within the research community, and there is no indication that significantly larger datasets will become available in the near future due to the difficulty of labeling security defects automatically.

[2] https://github.com/mlsec-group/cheetah

Selection of metrics. For our experiments, we consider a set of 23 syntactic code metrics. These metrics include re-implementations from prior work on identifying vulnerable code regions, as well as newly designed metrics aimed at capturing common characteristics of security flaws. We made a best effort to compile a comprehensive and representative set of metrics derived from syntax. However, this selection naturally does not encompass the full range of metrics available in software engineering, which also includes dynamic and operational measures. Some of these additional metrics may introduce new information and could correlate even more strongly with recent LLMs than the ones we considered. Nevertheless, this does not invalidate our findings; rather, it demonstrates that our results establish a lower bound on the predictive power that can be explained through the lens of code metrics.

Hidden variables. As in most machine learning research, hidden variables that influence predictions cannot be fully ruled out. For instance, a code change may alter semantics without affecting code metrics, in which case the metrics cannot fully account for the change in model prediction. This could stem from hidden variables exploited by the models or from further code metrics not included in our assessment. Likewise, a confounding hidden variable might affect both the metrics and the prediction outcome. In the first case, our evaluation likely underestimates the impact of code metrics, while in the second it overestimates their influence.

9 Conclusion

Our study uncovers an unexpected behavior of current LLMs for vulnerability discovery. While these models are widely believed to offer sophisticated analysis, we find that they largely capture basic statistical properties rather than deeper, structural insights of code. Simple code metrics can measure the very same properties using just a fraction of the computing resources. In combination with a traditional classifier, they currently perform on par with the best vulnerability discovery models. Our root-cause analysis confirms that the models closely align with code metrics, drawing on equivalent information for their decisions. The best-performing LLMs even exhibit a causal dependency on them.

While code metrics are valuable for identifying potential issues, they are, by design, insufficient for accurately pinpointing vulnerabilities. This limitation is evident from the consistently low performance of all methods in our experiments. Consequently, we do not advocate code metrics as a promising direction for advancing vulnerability discovery. Instead, we propose to take a step back and use code metrics as a reality check for rethinking the process of vulnerability discovery. We hope that the recommendations outlined in this work contribute to the development of more effective methods capable of obtaining deeper insights into code when learning and identifying security flaws.

Acknowledgments

This work was supported by the European Research Council (ERC) under the consolidator grant MALFOY (101043410), the German Federal Ministry of Research, Technology and Space under the grant AIgenCY (16KIS2012), and the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC 2092 CASA (390781972).

References

[1] Guillaume Alain and Yoshua Bengio. 2017. Understanding intermediate layers using linear classifier probes. In *Proc. of International Conference on Learning Representations (ICLR)*.

[2] Uri Alon and Eran Yahav. 2021. On the Bottleneck of Graph Neural Networks and Its Practical Implications. In *Proc. of the International Conference on Learning Representations (ICLR)*.

[3] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. 2022. Dos and Don'ts of Machine Learning in Computer Security. In *Proc. of the USENIX Security Symposium*.

[4] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. 2017. Efficient and Flexible Discovery of PHP Application Vulnerabilities. In *Proc. of the IEEE European Symposium on Security and Privacy (EuroS&P)*.

[5] Guru Prasad Bhandari, Amara Naseer, and Leon Moonen. 2021. CVEfixes: Automated collection of vulnerabilities and their fixes from open-source software. In *Proc. of the International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*.

[6] Trenton Bricken, Adly Templeton, Joshua Batson, Brian Chen, Adam Jermyn, Tom Conerly, Nick Turner, Cem Anil, Carson Denison, Amanda Askell, Robert Lasenby, Yifan Wu, Shauna Kravec, Nicholas Schiefer, Tim Maxwell, Nicholas Joseph, Zac Hatfield-Dodds, Alex Tamkin, Karina Nguyen, Brayden McLean, Josiah E Burke, Tristan Hume, Shan Carter, Tom Henighan, and Christopher Olah. 2023. Towards Monosemanticity: Decomposing Language Models With Dictionary Learning. *Transformer Circuits Thread* (2023).

[7] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2022. Deep Learning Based Vulnerability Detection: Are We There Yet? *IEEE Transactions Software Engineering* 48, 9 (2022), 3280–3296.

[8] Yizheng Chen, Zhoujie Ding, Lamya Alowain, Xinyun Chen, and David Wagner. 2023. DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection. In *Proc. of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. 654–668.

[9] Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. 2021. DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network. *ACM Transactions on Software Engineering and Methodology* 30, 3 (2021).

[10] Giovanni Cherubin, Rob Jansen, and Carmela Troncoso. 2022. Online Website Fingerprinting: Evaluating Website Fingerprinting Attacks on Tor in the Real World. In *Proc. of the USENIX Security Symposium*. 753–770.

[11] Christopher Clark, Mark Yatskar, and Luke Zettlemoyer. 2019. Don't Take the Easy Way Out: Ensemble Based Methods for Avoiding Known Dataset Biases. In *Proc. of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

[12] Zack Coker and Munawar Hafiz. 2013. Program transformations to fix C integers. In *Proc. of the International Conference on Software Engineering (ICSE)*. 792–801.

[13] Will Dietz, Peng Li, John Regehr, and Vikram S. Adve. 2012. Understanding integer overflow in C/C++. In *Proc. of the International Conference on Software Engineering (ICSE)*.

[14] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David A. Wagner, Baishakhi Ray, and Yizheng Chen. 2025. Vulnerability Detection with Code Language Models: How Far Are We?. In *Proc. of the International Conference on Software Engineering (ICSE)*.

[15] Nurit Dor, Michael Rodeh, and Shmuel Sagiv. 2003. CSSV: towards a realistic tool for statically detecting all buffer overflows in C. In *Proc. of the ACM Conference on Programming Language Design and Implementation (PLDI)*. 155–167.

[16] Xiaoning Du, Bihuan Chen, Yuekang Li, Jianmin Guo, Yaqin Zhou, Yang Liu, and Yu Jiang. 2019. Leopard: Identifying Vulnerable Code for Vulnerability Assessment through Program Metrics. In *Proc. of the International Conference on Software Engineering (ICSE)*. 60–71.

[17] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In *Proc. of the International Conference on Mining Software Repositories (MSR)*.

[18] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Proc. of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

[19] Matthias Feurer, Katharina Eggensperger, Stefan Falkner, Marius Lindauer, and Frank Hutter. 2022. Auto-Sklearn 2.0: Hands-free AutoML via Meta-Learning. *Journal of Machine Learning Research* 23 (2022), 261:1–261:61.

[20] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Tobias Springenberg, Manuel Blum, and Frank Hutter. 2015. Efficient and Robust Automated Machine Learning. In *Proc. of Advances in Neural Information Processing Systems (NeurIPS)*.

[21] Robert Flood, Gints Engelen, David Aspinall, and Lieven Desmet. 2024. Bad Design Smells in Benchmark NIDS Datasets. In *Proc. of the IEEE European Symposium on Security and Privacy (EuroS&P)*. 658–675.

[22] Fortify. 2013. RATS on Google Code. https://code.google.com/archive/p/rough-auditing-tool-for-security/issues.

[23] Michael Fu and Chakkrit Tantithamthavorn. 2022. LineVul: A Transformer-based Line-Level Vulnerability Prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories (MSR)*. 608–620.

[24] Tom Ganz, Erik Imgrund, Martin Härterich, and Konrad Rieck. 2023. PAVUDI: Patch-based Vulnerability Discovery using Machine Learning. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*. 704–717.

[25] Natalie Gruska, Andrzej Wasylkowski, and Andreas Zeller. 2010. Learning from 6,000 Projects: Lightweight Cross-Project Anomaly Detection. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*.

[26] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Proc. of the Annual Meeting of the Association for Computational Linguistics (ACL)*.

[27] Maurice Howard Halstead. 1977. *Elements of software science*. Elsevier.

[28] William L. Hamilton. 2020. *Graph Representation Learning*. Morgan & Claypool Publishers.

[29] Warren Harrison, Kenneth Magel, Raymond Kluczny, and Arlan DeKock. 1982. Applying Software Complexity Metrics to Program Maintenance. *Computer* 15, 9 (1982), 65–79.

[30] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. 2009. *The Elements of Statistical Learning: Data Mining, Inference and Prediction* (second ed.). Springer.

[31] Austin Bradford Hill. 1965. The Environment and Disease: Association or Causation? *Proceedings of the Royal Society of Medicine* (1965).

[32] Erik Imgrund, Tom Ganz, Martin Härterich, Lukas Pirch, Niklas Risse, and Konrad Rieck. 2023. Broken Promises: Measuring Confounding Effects in Learning-based Vulnerability Discovery. In *Proc. of the ACM Workshop on Artificial Intelligence and Security (AISec)*. 149–160.

[33] Matthieu Jimenez, Renaud Rwemalika, Mike Papadakis, Federica Sarro, Yves Le Traon, and Mark Harman. 2019. The importance of accounting for real-world labelling when predicting software vulnerabilities. In *Proc. of the ACM Joint Meeting on European Software Engineering Conference (ESEC)*.

[34] Joern Developer Community 2024. Joern: The Bug Hunter's Workbench. https://joern.io

[35] Marc Juarez, Sadia Afroz, Gunes Acar, Claudia Díaz, and Rachel Greenstadt. 2014. A Critical Evaluation of Website Fingerprinting Attacks. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. 263–274.

[36] David Larochelle and David Evans. 2001. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *Proc. of the USENIX Security Symposium*.

[37] Bingchang Liu, Guozhu Meng, Wei Zou, Qi Gong, Feng Li, Min Lin, Dandan Sun, Wei Huo, and Chao Zhang. 2020. A large-scale empirical study on vulnerability distribution within projects and the lessons learned. In *Proc. of the International Conference on Software Engineering (ICSE)*. 1547–1559.

[38] Zhongxin Liu, Zhijie Tang, Junwei Zhang, Xin Xia, and Xiaohu Yang. 2024. Pre-training by Predicting Program Dependencies for Vulnerability Analysis Tasks. In *Proc. of the International Conference on Software Engineering (ICSE)*.

[39] 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, et al. 2024. StarCoder 2 and The Stack v2: The Next Generation. *Computing Research Repository (CoRR)* (2024).

[40] Thomas J. McCabe. 1976. A Complexity Measure. In *Proc. of the International Conference on Software Engineering (ICSE)*. 407.

[41] Dongyu Meng, Michele Guerriero, Aravind Machiry, Hojjat Aghakhani, Priyanka Bose, Andrea Continella, Christopher Kruegel, and Giovanni Vigna. 2021. Bran: Reduce Vulnerability Search Space in Large Open Source Repositories by Learning Bug Symptoms.. In *Proc. of the ACM Asia Conference on Computer and Communications Security (AsiaCCS)*. 731–743.

[42] J. A. Nelder. 1974. Log Linear Models for Contingency Tables: A Generalization of Classical Least Squares. *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 23, 3 (1974), 323–329.

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

[44] Georgios Nikitopoulos, Konstantina Dritsa, Panos Louridas, and Dimitris Mitropoulos. 2021. CrossVul: A Cross-Language Vulnerability Dataset with Commit Data. In *Proc. of the ACM Joint European Software Engineering Conference (ESEC)*.

[45] National Institute of Standards and Technology (NIST). 2024. Software Assurance Reference Dataset (SARD). https://samate.nist.gov/SARD

[46] Kiho Park, Yo Joong Choe, and Victor Veitch. 2024. The Linear Representation Hypothesis and the Geometry of Large Language Models. In *Proc. of the International Conference on Machine Learning (ICML)*.

[47] Judea Pearl. 2009. *Causality: Models, Reasoning and Inference* (2nd ed.). Cambridge University Press.

[48] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. 2015. VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. 426–437.

[49] Oleg Platonov, Denis Kuznedelev, Michael Diskin, Artem Babenko, and Liudmila Prokhorenkova. 2023. A critical look at the evaluation of GNNs under heterophily: Are we really making progress?. In *Proc. of the International Conference on Learning Representations (ICLR)*.

[50] Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.* 74 (1953), 358–366.

[51] Niklas Risse and Marcel Böhme. 2024. Uncovering the Limits of Machine Learning for Automatic Vulnerability Detection. In *Proc. of the USENIX Security Symposium*.

[52] Niklas Risse, Jing Liu, and Marcel Böhme. 2025. Top Score on the Wrong Exam: On Benchmarking in Machine Learning for Vulnerability Detection. *Proc. ACM Softw. Eng.* 2, ISSTA (2025), 388–410.

[53] Rebecca L. Russell, Louis Y. Kim, Lei H. Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul M. Ellingwood, and Marc W. McConley. 2018. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. In *Proc. of the IEEE International Conference on Machine Learning and Applications (ICMLA)*.

[54] Riccardo Scandariato, James Walden, Aram Hovsepyan, and Wouter Joosen. 2014. Predicting Vulnerable Software Components via Text Mining. *IEEE Transactions on Software Engineering* 40, 10 (2014), 993–1006.

[55] Adriana Sejfia, Satyaki Das, Saad Shafiq, and Nenad Medvidovic. 2024. Toward Improved Deep Learning-based Vulnerability Detection. In *Proc. of the International Conference on Software Engineering (ICSE)*.

[56] Semgrep, Inc. 2025. *Semgrep*. https://semgrep.dev

[57] Yonghee Shin and Laurie Williams. 2008. An Empirical Model to Predict Security Vulnerabilities using Code Complexity Metrics. In *Proc. of the International Symposium on Empirical Software Engineering and Measurement*. 315–317.

[58] Yonghee Shin and Laurie Williams. 2013. Can traditional fault prediction models be used for vulnerability prediction? *Empirical Software Engineering* (2013).

[59] Robin Sommer and Vern Paxson. 2010. Outside the Closed World: On Using Machine Learning for Network Intrusion Detection. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*. 305–316.

[60] Tree-sitter 2024. *Tree-Sitter: Parser Generator Tool*. https://tree-sitter.github.io/tree-sitter/

[61] Saad Ullah, Mingji Han, Saurabh Pujar, Hammond Pearce, Ayse K. Coskun, and Gianluca Stringhini. 2024. LLMs Cannot Reliably Identify and Reason About Security Vulnerabilities (Yet?): A Comprehensive Evaluation, Framework, and Benchmarks.. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*. 862–880.

[62] David A. Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. 2000. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*.

[63] Yue Wang, Hung Le, Akhilesh Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. In *Proc. of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

[64] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proc. of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

[65] Felix Weissberg, Jonas Möller, Tom Ganz, Erik Imgrund, Lukas Pirch, Lukas Seidel, Moritz Schloegel, Thorsten Eisenhofer, and Konrad Rieck. 2024. SoK: Where to Fuzz? Assessing Target Selection Methods in Directed Fuzzing. In *Proc. of the ACM Asia Conference on Computer and Communications Security (AsiaCCS)*.

[66] Christian Wressnegger, Fabian Yamaguchi, Alwin Maier, and Konrad Rieck. 2016. Twice the Bits, Twice the Trouble: Vulnerabilities Induced by Migrating to 64-Bit Platforms. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. 541–552.

[67] Sewall Wright. 1921. Correlation and causation. *Journal of agricultural research* 20, 7 (1921), 557.

[68] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*. 590–604.

[69] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. 2012. Generalized vulnerability extrapolation using abstract syntax trees. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*. 359–368.

[70] Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In *Proc. of the Conference on Neural Information Processing Systems (NeurIPS)*.

A Structural Code Metrics

In our study, we analyze and implement 23 code metrics within the framework introduced in Section 3. For each metric, we provide a formal definition of the filter function F, mapping function M, and reduction function R in Table 4. We use $\mathbb{1}$ as the indicator function. Some metrics use auxiliary measures, which are denoted with an additional index, such as M1.1.

For the filtering step, we traverse the syntax tree to identify and extract all subtrees that match specific code metric patterns. These patterns are defined using a tree query language capturing node types, their relationships, and apply quantifiers to these relationships. We build on the Tree-sitter query language [60], extending it to provide the necessary expressiveness. Conceptually, our language is based on *S-expressions*, which are textual representations of tree structures originating from the LISP programming language. Just as regular expressions used for pattern matching in text, this query language enables pattern matching within tree structures. Table 5 lists the building blocks of our query language.

The core unit of our language is a node, which can be matched by a specific type, a wildcard specifier `(_)`, an alternative of types `((a) | (b))`, or a type negation `(!a)`. Each node can be connected by three types of relationships: parent-child, siblings, and descendants. The parent-child relationship is specified by placing the child node within the parentheses of the parent, as in `(a (b))`.

Table 5: Building structures for our tree query language.

Rule	Notation	Description
Node	`(a)`	Matches each node of type 'a'
Parent-Child	`(a (b))`	…with one child of type 'b'
Siblings	`(a (b) (c))`	…with exactly two children of type 'b' and 'c'
Sibling Quantifier	`(a (b)*)`	…with arbitrarily many children of type 'b'
Descendant Quantifier	`(a (b)ˆ*)`	…with a path graph of nodes of type 'b'
Alternative	`((a) | (b))`	Matches each node of type 'a' or 'b'
Negation	`(!a)`	Matches each node whose type is not 'a'
Wildcard	`(_)`	Matches any node
Annotation	`(a) @x`	Assigns the name 'x' to node of type 'a'

Adding more child nodes implies a sibling relationship, as in `(a (b) (c))`, where `b` and `c` are siblings. To indicate an arbitrary number of children, we use sibling quantifiers `*` (zero or more matches) and `+` (one or more matches) on a child node. Similarly, the descendant quantifiers `ˆ*` and `ˆ+` allow expressing depth-traversal of the tree. Lastly, to access attributes of specific nodes in the mapping function, they can be annotated with `(a) @node`. The following map function can use this annotation to calculate various numerical values. For example, $|t.\text{node}|$ corresponds to the cardinality of the node type in the subtree t.

Table 4: Overview over all 23 structural code metrics.

	Description	Filter	Map	Reduce
		Code Smell Metrics		
S1	# magic numbers	`(number_literal) @num`	$t \mapsto \mathbb{1}[t.\text{num} \notin \{-1, 0, 1\}]$	sum
S2	# goto	`(goto_stmt)`	$t \mapsto 1$	sum
S3	# function pointers	`((declaration (init_declarator (function_declarator)))` `(parameter_declaration (function_declarator)))`	$t \mapsto 1$	sum
S4	# function calls with unused returns	`(expr_stmt (call_expr))`	$t \mapsto 1$	sum
S5	# if without else	`(if_stmt !alternative)`	$t \mapsto 1$	sum
		Complexity Metrics		
C1	cyclomatic complexity	`(cond_stmt (_))`	$t \mapsto 1 + \text{c1.1}(t)$	sum
C1.1	# logical operators	`(binary_expr (operator) @op)`	$t \mapsto \mathbb{1}[t.\text{op} \in \{\text{'\&\&'}, \text{'\|\|'}\}]$	sum
C2	# loops	`(loop_stmt)`	$t \mapsto 1$	sum
C3	# nested loops	`(loop_stmt ((!loop_stmt)ˆ* (loop_stmt)))`	$t \mapsto 1$	sum
C4	max nesting level of loops	`(loop_stmt ((!loop_stmt)ˆ* (loop_stmt))ˆ*)`	$t \mapsto \text{C2}(t) + 1$	max
C5	# parameters	`(parameter_declaration)`	$t \mapsto 1$	max
C6	# nested control structures	`(ctrl_stmt ((!ctrl_stmt)ˆ* (ctrl_stmt)))`	$t \mapsto 1$	sum
C7	max nesting level of control structure	`(ctrl_stmt ((!ctrl_stmt)ˆ* (ctrl_stmt))ˆ*)`	$t \mapsto \text{C6}(t) + 1$	max
C8	max # control structures in a control structure	`(ctrl_stmt ((!ctrl_stmt)ˆ+ (ctrl_stmt)ˆ*)`	$t \mapsto \text{C6}(t)$	max
C9	# return stmts	`(return_stmt)`	$t \mapsto 1$	sum
C10	# casts	`(cast_expr)`	$t \mapsto 1$	sum
C11	# local variables	`(declaration)`	$t \mapsto 1$	sum
C12	max # operands	`(binary_expr)`	$t \mapsto \mu_{(\text{identifier}\|\text{literal}),\text{sum}}(t)$	max
		Memory Metrics		
M1	# heap allocations	`(_)`	$t \mapsto \text{M1.1}(t) + \text{M1.2}(t)$	sum
M1.1	# new allocations	`(new_expr)`	$t \mapsto 1$	sum
M1.2	# call allocations	`(call_expr function: (identifier) @name)`	$t \mapsto \mathbb{1}[\text{'alloc' in } t.\text{name}]$	sum
M2	# pointer dereferences	`((pointer_expr) | (subscript_expr) | (field_expr))`	$t \mapsto 1$	sum
M3	# pointer arithmetic	`((binary_expr) | (unary_expr))`	$t \mapsto \mathbb{1}[\text{M3.1}(t) > 0]$	sum
M3.1	# pointer variables	`({type: 'pointer'})`	$t \mapsto 1$	sum
		Syntax Tree Metrics		
T1	# AST nodes	`(_)`	$t \mapsto 1$	sum
T2	max height of AST	`(_ (_)ˆ*)`	$t \mapsto \|t\|$	max
T3	average # of children	`(_ (_)+ @children)`	$t \mapsto \|t.\text{children}\|$	avg

Table 6: Performance of different vulnerability prediction approaches on the PrimeVul [14] dataset.

Predictor	Efficiency	F1 ↑	AUPRC ↑	MCC ↑	BAcc ↑	VD-S ↓
Code Metrics						
SCM	2.354	**20.32** ± 0.59	**13.80** ± 0.82	18.90 ± 0.55	62.22 ± 0.64	**90.20** ± 0.70
Code Language Models						
UniXcoder [26]	0.130	**20.69** ± 1.43	13.32 ± 1.22	**21.36** ± 1.06	67.23 ± 2.38	92.84 ± 0.79
PDBERT [38]	0.121	19.50 ± 1.17	**13.38** ± 0.68	**20.67** ± 0.40	**69.19** ± 1.86	**92.40** ± 1.23
CodeGen 2.5 [43]	0.002	18.57 ± 0.54	13.22 ± 0.30	17.07 ± 0.41	63.23 ± 1.19	92.50 ± 0.49
LineVul [23]	0.112	18.48 ± 1.23	11.68 ± 0.82	19.34 ± 0.93	68.36 ± 2.06	92.66 ± 1.35
StarCoder 2 [39]	0.002	17.09 ± 0.41	12.46 ± 0.40	19.73 ± 0.48	**72.25** ± 1.41	94.17 ± 0.44
General-purpose LMs						
GPT-3.5 Turbo	1.3×10^{-4}	5.80 ± 0.83	2.56 ± 0.18	3.34 ± 1.01	52.96 ± 1.09	100.00 ± 0.00
GPT-4o	8.90×10^{-7}	5.31 ± 0.35	2.50 ± 0.10	2.72 ± 0.90	54.17 ± 1.47	100.00 ± 0.00
Random	-	4.42 ± 0.19	2.33 ± 0.01	0.05 ± 0.36	50.15 ± 0.45	99.60 ± 0.18

B Additional Results

In addition to the main results presented in the paper, we aim to provide further insight into the model configurations and a more fine-grained view of our results.

Model configuration. The transformer-based models were trained for 10 epochs using a learning rate of 2×10^{-5}, a batch size of 8 with 8 gradient accumulation steps, and early stopping with a patience of 10 and no minimal improvement requirement. ReVeal and Devign used larger batch sizes of 256 with 50 gradient accumulation steps to stabilize optimization, a higher learning rate of 1×10^{-4} and were trained for 100 epochs with the same early stopping criterion. The code metrics-based model was trained for 10 minutes, requiring no further hyper parameter selection.

Model performance. In addition to the performance metrics reported as part of our evaluation in Table 2, we further report in Table 6 the Matthews correlation coefficient (MCC), the balanced accuracy (BAcc), and the vulnerability detection score (VD-S)—the false negative rate at a fixed false positive rate of 0.05%—introduced alongside the PrimeVul dataset [14].

Most importantly, we find that no single model achieves the best performance across all metrics. However, three models stand out: UniXcoder ranks highest in two of the five metrics but never ranks second; PDBERT ranks second-best in four of the five metrics but never ranks highest; and the code metrics-based model achieves the best score in two metrics and the second-best score in one.

This finding suggests that analyses of vulnerability discovery models should always report results across a variety of metrics, as the assessment of which model performs best can vary depending on the chosen metric. In our experiments, UniXcoder, PDBERT, and the code metrics-based model perform comparably overall considering different metrics.

In addition to the model performance, we report the parameter efficiency by calculating the improvement in percentage points over the random baseline per million parameters. This metric addresses the intuitive question: How much improvement over the baseline is gained by adding one million parameters? We find that the code metrics-based model shows the greatest parameter efficiency at two orders of magnitude higher than those of the next best, UniXCoder, which shows the greatest efficiency among the LLM-based approaches. This metric can guide the selection of models that offer the best trade-off between performance and size.