ANVIL: Anomaly-based Vulnerability Identification without Labelled Training Data

Weizhou Wang✉, Eric Liu, Xiangyu Guo, Xiao Hu, Ilya Grishchenko, David Lie

University of Toronto, Toronto, Canada
{weizhou.wang,ec.liu,xiangyu.guo,hx.hu}@mail.utoronto.ca
{ilya.grishchenko,david.lie}@utoronto.ca

Abstract. Supervised-learning-based vulnerability detectors often fall short due to limited labelled training data. In contrast, Large Language Models (LLMs) are trained on vast unlabelled code corpora, yet perform only marginally better than coin flips when directly prompted to detect vulnerabilities. In this paper, we introduce ANVIL, a line-level vulnerability localization approach that reframes vulnerability detection as anomaly detection, based on the premise that vulnerable code is rare and thus anomalous relative to patterns learned by LLMs. ANVIL performs a masked code reconstruction task: The LLM reconstructs a masked line of code, and deviations from the original are scored as anomalies. We propose a hybrid anomaly score that combines exact match, cross-entropy loss, prediction confidence, and structural complexity. We evaluate our approach across multiple LLM families, scoring methods, and context sizes, and against vulnerabilities after the LLM's training cut-off. On the PrimeVul dataset, ANVIL outperforms state-of-the-art supervised detectors – LineVul, LineVD, and LLMAO – achieving up to 2× higher Top-3 accuracy, 75% better Normalized MFR, and a significant improvement on ROC-AUC. Finally, by integrating ANVIL with fuzzers, we uncover two previously unknown vulnerabilities, demonstrating the practical utility of anomaly-guided detection.

Keywords: Vulnerability detection · Large language models.

1 Introduction

Probabilistic approaches to static vulnerability detection have garnered considerable research attention due to the promise of automatically detecting common bug patterns by training a model on historical data. Confidence in statistical modelling has only grown with recent advances in large language models (LLMs). One of the significant advantages of LLMs is that they can be trained in a self-supervised fashion without labelled data. This has enabled them to be trained on very large training sets, and led to impressive and surprising results in code understanding and generation [16,48].

Unfortunately, the same cannot be said about their ability to detect vulnerabilities. As shown in a recent study [47], even with advanced LLMs like

GPT-4 [42], vulnerability prediction accuracy on real-world code is little better than a coin-flip. A key reason for this limitation is that LLMs are trained to generate code that resembles their training data, without explicit supervision about what constitutes buggy or vulnerable code. As a result, they lack the domain-specific knowledge needed to reliably distinguish vulnerable code from benign code. Meanwhile, existing learning and LLM-based vulnerability detectors, while attempting to embed the notion of vulnerabilities, still suffer from the lack of a large and well-labelled dataset [10,14]. Efforts to compensate for the shortage of labelled data by selecting relevant code features (e.g., the granularity of the code context, commit history, control-flow, data-flow, etc.) and converting these features into a learnable representation (e.g., a vector or a graph) [31,30,11,18,23,37,44] have only yielded modest improvements. Similarly, recent attempts to generate new vulnerable training corpora [37,41,40] have not improved real-world performance significantly, as synthetically generated vulnerabilities still cannot capture the huge variety of ways that vulnerabilities can manifest themselves.

LLMs are effective because they are trained on massive unlabelled datasets, enabling them to generate code in the same distribution. However, this generative capability does not translate to an ability to recognize vulnerable code, as such recognition is a different objective requiring additional training, ideally with labelled vulnerable code. Out of the huge amounts of unlabelled code, only a minuscule portion contains vulnerabilities, and even less has labels.

Rather than trying to learn bug patterns from such sparse signals, we advocate for the inverse. Vulnerable code is more likely to fall outside the LLM's learned distribution, because vulnerabilities constitute only a small fraction of the code used in training. This allows us to recast bug detection as an *anomaly detection* task: By comparing the original line of code to the generated version in a masked code reconstruction task, we determine whether the line is anomalous, i.e., out-of-distribution for an LLM. This focus allows for isolating the vulnerability at a particular location, enabling line-level detection, while some vulnerability detectors only work at the function granularity [38,29,52,49,27].

There were attempts at modelling software defects as anomalies. For example, a study of code "naturalness" [43] leveraged statistical methods to model software bugs and their fixes, creating a purely statistical 10-gram model trained on 35 million LOC for next-token prediction. Then, given a line of code, the model calculated the cross-entropy based on probabilities in its internal statistics and achieved a bug detection accuracy competitive with static code analyzers.

We hypothesize that the 10-gram model's result can be improved by modern transformer-based LLMs since they have considerably greater representational power, being trained on 10s-100s of billions of LOC (a 4-5 order of magnitude increase). However, it is not straightforward to adapt this previous work on code defects to vulnerability detection with LLMs. We highlight the following challenges: First, the entropy measurement used by previous work does not sufficiently discriminate between small text-level differences, which can be significant for software correctness. For instance, just a one-character difference in an

if-statement's predicate can lead to an out-of-bound array access (e.g. $>$ vs $>=$). To address this challenge, this paper explores different methods for quantifying the anomaly level of predictions. Second, unlike 10-gram models, LLMs can use a context of hundreds to thousands of tokens to make predictions, raising the question of the appropriate context size to use, i.e., providing the LLMs with context that is both sufficient and yet not overly large so that it confuses the LLM. Third, the design space of LLMs includes diverse model architectures, training objectives and datasets. Thus, we also explore whether anomaly-based bug detection with LLMs generalizes to different model sizes and architectures. Finally, given that modern LLMs are trained on large datasets, there is a risk of "leakage," that is, a pre-trained LLM may have seen both the vulnerable and fixed version of the same code. Therefore, we evaluate our approach on a leakage-free dataset containing only vulnerabilities disclosed after the LLMs' training cutoff, demonstrating its ability to generalize to unseen vulnerabilities.

To summarize, the paper makes the following contributions:

- We present ANVIL, a line-level anomaly-based vulnerability detection approach that does not require labelled training data. We demonstrate that ANVIL, leveraging pre-trained LLMs, effectively identifies vulnerable code as anomalous through masked code reconstruction.
- We evaluate different design choices, namely anomaly scoring functions, model selections and prompt contexts, and find that hybrid scoring, larger models, and compound statement contexts yield the best performance.
- ANVIL outperforms the state-of-the-art supervised-learning-based line-level detectors, namely, LineVul, LineVD, and LLMAO, with 1.8×–2× higher Top-3 accuracy, 40%–75% better Normalized MFR, and a significant improvement on ROC-AUC.
- We augment fuzzers with ANVIL to guide seed selection, uncovering two previously unknown vulnerabilities, one of which has been assigned a CVE.

We make all code for our tool ANVIL available at https://github.com/joe-weizhou-wang/anvil-paper. All data and scripts used for running our evaluation are also made public for reproducibility.

2 Methodology

2.1 Overview

Similar to previous research [43] in code "naturalness," we consider a piece of code anomalous (i.e., unnatural) if a model considers it out-of-distribution. Figure 1 describes the workflow of our technique, ANVIL. For each line of code under analysis, we mask (remove) the original (ground truth) line and instruct the LLM to reconstruct it based on the surrounding context. The LLM-generated code is then compared with the ground truth to compute an *anomaly score*, which quantifies the divergence between them. LLMs are trained to predict the most probable token given a chain of previous tokens. Hence, a low reconstruction accuracy compared to the original code indicates that the original code is

anomalous, i.e., has deviated from the learned representation of the model. While our study focuses on C/C++ as a proof of concept, ANVIL is language-agnostic in design and can be applied to other programming languages supported by FIM–trained code LLMs.

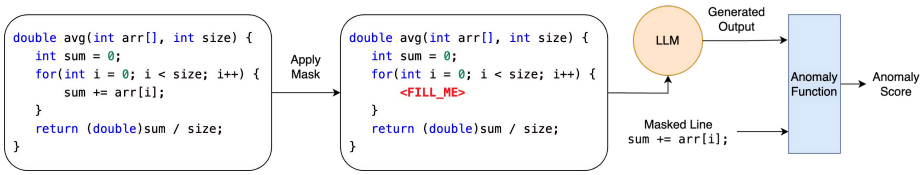

Fig. 1. Overview of the ANVIL's workflow

2.2 FIM & Context Selection

Code LLMs are trained for generative tasks such as code-completion, which predicts the next n tokens given a code prefix, and fill-in-the-middle (FIM, or infilling) [4], which generates tokens that fall between a given prefix and suffix. We focus on FIM for detecting anomalies. Although code completion is a more prevalent task, research by Bavarian et al. [4] highlights that FIM generally results in lower reconstruction loss due to additional constraints provided by the suffix. Suffix information allows FIM to capture downstream control and data flows, which are crucial for generating correct code for masked lines.

However, adapting FIM for anomaly detection introduces challenges. First, GPU memory and token limits make it infeasible to use entire source files as context. Moreover, while LLMs claim to support long contexts, recent studies [34,28,25] show that LLM performance often degrades when relevant information is buried within long contexts. As there is limited work on context selection for code reconstruction, let alone anomaly detection, we empirically evaluate two strategies: (1) a naive fixed-size approach and (2) a structure-aware approach.

For (1), we investigate the effect of fixing the context to a specific number of source-code lines, equally split between the prefix and suffix. For instance, a 500-line context uses 250 lines preceding and succeeding the line under analysis. Naturally, this naive context selection may fail to capture relevant features beyond its fixed limits (e.g., a 100-line window applied to a 200-line function). Conversely, large contexts may pull in unrelated code, introducing noise that could confuse the model.

Alternatively, (2) uses an Adaptive Context (AC) approach, which selects the maximum compound statement enclosing the masked line. This method ensures the model receives semantically relevant context without being distracted by irrelevant lines. In C/C++, a compound statement refers to a sequence of statements enclosed by curly braces. Given a line of source code, we define the AC as the largest compound statement encapsulating the line. Often, the AC

equates to the function body containing the line. However, if the AC is too large, exceeding token limits or GPU memory capacity, we resort to the next smaller compound statement that meets these constraints. We found that a context limit of 500 lines reached near-maximum peak usage of our GPU memory. Hence, our AC selection only considers compound statements that are 500 lines or fewer. Figure 2 demonstrates how we extract the compound statements associated with a masked line.

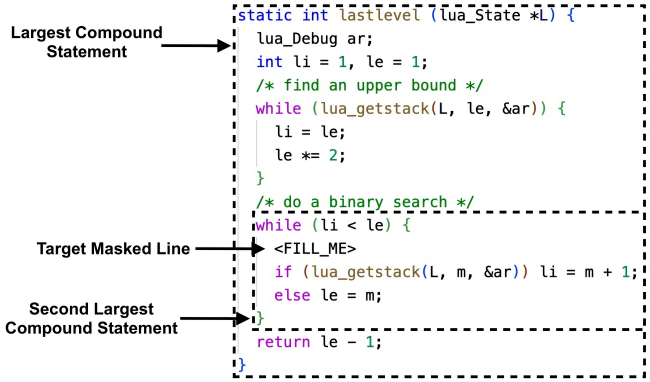

Fig. 2. An example from *lua* (a scripting language developed in C) showing the largest and second-largest compound statement for a line of interest

2.3 Anomaly Score

We assign an *anomaly score* to quantify the discrepancies between the generated line and the ground truth. We define the anomaly score as a function that takes the masked ground truth (p) and the generated line (q) as arguments and calculates a value $\delta(p, q)$, which estimates how "anomalous" the ground truth is, with higher values being more anomalous.

Previous work on code "naturalness" [43] uses cross-entropy loss for each token based on 10-gram frequencies counted from their corpus to compute such an anomaly score. For LLMs, we compute the anomaly score as the average cross-entropy loss on the masked line, given its full surrounding context. This yields an overall loss measure $\delta_{loss}(p) = loss(p)$ for the line.

However, loss values tend to be soft and diffuse, which could under-estimate anomaly severity: If a masked ground truth is "$if(a >= 0)$" and the model believes that the line is "$if(a > 0)$", the off-by-one-character "=" might result in a low average loss but represents a potentially significant change, such as becoming the root cause of an out-of-bounds array access. Thus, we propose adding an exact match indicator as another metric, which treats the masked ground truth and the generated line as complete sequences. This approach assigns a

value of 1 if the LLM's generated line matches the ground truth exactly, and 0 otherwise. An exact match implies that, during model inference for each token, the most probable token in each token distribution is perfectly aligned with the ground truth, and hence the line under analysis should not be considered anomalous. Thus, we can use the negation of exact match (EM) for anomaly scoring, $\delta_{EM}(p, q) = -\mathbb{1}_{exact_match}(p, q)$. Surprisingly, a crude measure like exact match has discriminative ability, as demonstrated by experiments in Section 3.2.

Beyond the improved scale and ability of LLMs over the n-gram models used by previous work, LLMs expose more information for inferring model confidence. Namely, LLMs output probability distributions, which can be used to estimate their confidence in a prediction. We hypothesize that while there are perhaps only a few ways that code can be correct, there can be many ways that code can be incorrect, which should manifest as low-confidence predictions[1]. To estimate confidence, we observe differences in the models' probability distributions for the first generated token when comparing vulnerable and benign lines. Specifically, we compute the variance among the top-10 predicted tokens at the first generation location, which reflects how concentrated or spread out the model's confidence is. Interestingly, as illustrated in Figure 3, vulnerable lines tend to cluster around moderate variance, while benign lines exhibit a wider spread, mostly with very high and a few very low cases. High variance reflects strong model confidence, typical of benign lines with clear expected completions. In low variance cases, we find that the model is not necessarily "confused," but is hedging between many valid completions, in syntactically simple code. This often arises in descriptive contexts (e.g., function or variable declarations), where there can be several plausible types or symbol names.

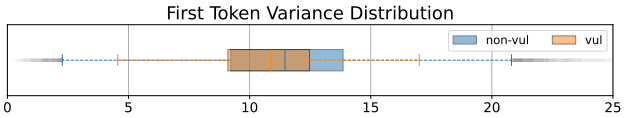

Fig. 3. First generated token distribution variance on Magma dataset

To better highlight such extremes, we transform the first-token variance into an anomaly score, as shown in Equation 1. Let v denote the observed variance. The term $\overline{V}$, representing the empirical mean variance, serves as a calibration constant, while C is a scaling constant. Both $\overline{V}$ and C are determined empirically from a vulnerability dataset. This transformation penalizes lines with variance near the average and favours those with more extreme values.

$$\delta_{var}(q) = -\exp(\frac{|v - \overline{V}|}{C}) \tag{1}$$

[1] We draw inspiration from Tolstoy who observed: "All happy families are alike; each unhappy family is unhappy in its own way."

As noted above, there are syntactically simple cases, such as simple function invocations, where there can be many plausible benign completions. However, uncertainty about code can also be correlated with complex lines, such as nested conditionals or long expressions, where intricate logic may cause developers to overlook edge cases. To differentiate these two cases, we introduce an additional component based on the number of nodes in the Abstract Syntax Tree (AST) of the originally masked line, denoted $\delta_{ast}(p)$. A higher node count reflects greater lexical and syntactic complexity, which may signal higher vulnerability risk.

Our final anomaly scoring function is outlined in Equation 2, aggregating all four components introduced above, each normalized. This formulation rewards exact matches while capturing subtle deviations, model uncertainty, and code complexity between vulnerable and benign code. As shown in Section 3.2 and Appendix C, our experiments demonstrate that this combined approach consistently yields the most reliable vulnerability detection performance, with all components contributing positively to the overall effectiveness.

$$\delta_{\text{hybrid}}(p, q) = \delta_{\text{loss}}(p) + \delta_{\text{EM}}(p, q) + \delta_{\text{var}}(q) + \delta_{\text{ast}}(p) \tag{2}$$

3 Evaluation

This section explores the effectiveness of ANVIL's anomaly-based vulnerability detection through four research questions:

- **RQ1:** How do different anomaly scoring methods affect ANVIL's vulnerability detection capability?
- **RQ2:** Does an adaptive context size enhance vulnerability detection performance compared to a fixed context size?
- **RQ3:** How does ANVIL perform relative to supervised-learning-based vulnerability detection approaches?
- **RQ4:** Can ANVIL's anomaly and vulnerability detection capabilities generalize to unseen data?

These questions investigate different aspects of ANVIL's design and how varying parameters affect the efficacy of vulnerability detection. They also encompass comparisons between ANVIL, which requires no labelled inputs, and supervised vulnerability detectors, as well as an analysis of potential LLM data leakage.

3.1 Experiment Setup

All experiments were conducted using LLMs and tokenizers from the Huggingface library [51], on a machine equipped with an Nvidia H100 GPU with 80GB of HBM. For all LLMs, we used a temperature value of zero during code generation to prevent randomization.

We used three datasets to evaluate ANVIL across different dimensions. First, RQ1 and RQ2 explore how ANVIL performs under varying configurations, such

as different scoring methods and context sizes. For these experiments, we use the Magma [21] benchmark due to its reliable vulnerability labelling. Specifically, Magma's labels are backed by a *Proof of Vulnerability* (PoV) for each bug, representing an input that triggers the vulnerability. These PoVs are extracted through manual reviews and complemented by extensive fuzzing campaigns, ensuring that all labelled lines are directly tied to actual vulnerabilities. Magma contains 138 real-world CVEs from nine widely-used C/C++ projects (libpng, libtiff, libsndfile, libxml2, poppler, openssl, sqlite3, php, and lua), totalling 240,000 LoC and 5,227 functions. It includes both vulnerable and patched versions, with 256 lines labelled as vulnerable. Due to space limitations, we use CodeLlama-13B as the representative model for RQ1 and RQ2, as it is the most capable model in our evaluation. Comprehensive results covering all tested model architectures and sizes are included in Appendix A.

After identifying the most effective configurations of ANVIL, we compare it against various supervised-learning-based vulnerability detectors to address RQ3. To avoid potential overfitting from the previous configuration tuning process and to demonstrate ANVIL's broader generalizability, the comparison is done on a different dataset, namely the PrimeVul [14] dataset, comprising over 700 open-source C/C++ projects. Despite its scale, PrimeVul maintains a low labelling error rate, achieving over 90% accuracy via automated validation using descriptions from the NVD database [46].

Additionally, to address concerns of LLM data leakage, RQ4 evaluates ANVIL with vulnerabilities unseen by the LLMs during training. Because each LLM we tested is trained at a different time, it is difficult to find a large set of vulnerabilities that is guaranteed to be unseen by all models. Therefore, we focus on CodeLlama-13B for this leakage experiment because it generally provides the best performance as per Table 5. We leave the construction of leakage-free datasets and evaluation of other LLMs for future work. Using a script from the CVEFixes project [5], we collected 100 CVEs from 53 repositories, all published between Dec 31, 2023, and Apr 20, 2024 (after CodeLlama-13B's knowledge cutoff). Instead of only relying on lines changed by defect-fixing commits, which may contain irrelevant information, we manually extracted only vulnerable lines from each fix. The final dataset includes 147 vulnerable functions (235 vulnerable and 10,956 non-vulnerable LoC) and 3,589 benign functions (101,173 LoC). We made this dataset publicly available to support future research.

In all experiments, the calibration factor ($\overline{V}$) and the normalization factors used in our anomaly function (Equation 2) are derived from the Magma dataset. We also set the scaling constant C to 40 because it consistently yields strong separation between vulnerable and benign lines on Magma. To avoid dataset-specific tuning, we keep all normalization factors fixed when applying the scoring function to other datasets.

3.2 RQ1: Anomaly Score

To evaluate the effectiveness of the anomaly scoring function and its components (discussed in Section 2.3), we performed FIM (Section 2.2) using CodeLlama-13B

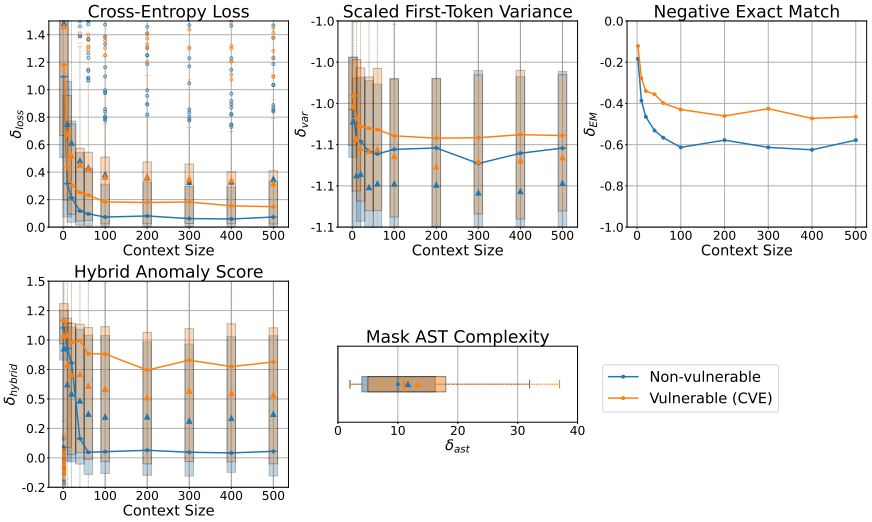

Fig. 4. Anomaly scores on various functions

on the Magma dataset. Specifically, we examine the score *values* and the *distributions* generated by each anomaly score for each line and evaluate the score's ability to discriminate between vulnerable and non-vulnerable lines. This evaluation was framed as a binary classification task, with the Area Under the Receiver Operating Characteristic Curve (ROC-AUC) used as a metric to measure the significance of the score in distinguishing between vulnerable and non-vulnerable lines. An ROC-AUC metric of 0.5 indicates random guessing, while higher values reflect a greater discrimination capability.

Table 1. ROC-AUC using different scoring functions

	Context Size									
Func	2	10	20	40	60	100	200	300	400	500
δ_{ast}	55.6%	55.6%	55.6%	55.6%	55.6%	55.6%	55.6%	55.6%	55.6%	55.6%
δ_{EM}	53.1%	55.5%	56.2%	58.8%	58.4%	59.2%	55.9%	59.4%	57.6%	55.7%
δ_{var}	54.5%	57.6%	54.5%	56.4%	56.9%	55.1%	52.9%	54.5%	55.3%	53.9%
δ_{loss}	51.8%	54.3%	53.8%	57.2%	57.8%	58.1%	**58.7%**	60.4%	59.5%	57.6%
δ_{hybrid}	**56.9%**	**57.8%**	**57.2%**	**60.7%**	**60.7%**	**60.8%**	57.7%	**61.5%**	**60.1%**	**57.7%**

To compute the values of each scoring function component in FIM, we used all 256 vulnerable LOC from the Magma dataset as vulnerable samples. To equally reward vulnerable and non-vulnerable lines in the comparative analysis, we balanced the dataset by randomly sampling 256 non-vulnerable lines (excluding comments and whitespace). For each reconstructed line of code, we calculate anomaly scores using five scoring functions (described in Section 2.3) across var-

ious fixed context sizes from 2 to 500 lines. Figure 4 presents the values of each component. For δ_{EM}, we report the averaged values due to their binary nature. For the δ_{loss}, δ_{var}, and δ_{hybrid} scores, we plot median values as connected lines across context sizes to reduce the influence of outliers. To provide a comprehensive view, we also included box plots [50], with mean values represented by small triangles. Additionally, since δ_{ast} is a static property of the masked line itself and independent of LLM output, we present its distribution only as a box plot.

Across all scoring function components, the median values reveal a noticeable gap between categories of vulnerable and non-vulnerable lines, and this gap typically stabilizes when the context size exceeds 100 lines. This observation reinforces the notion that the LLM processes these two categories differently. Conversely, as the context size decreases below 100 lines, the anomaly scores for both vulnerable and non-vulnerable lines gradually increase, and the gap between the two categories narrows, indicating that the LLM becomes less accurate in reconstructing the masked lines regardless of their categories. This performance decline occurs because smaller context sizes do not provide the model with sufficient information for accurate generation. Among the scoring functions, the δ_{hybrid} function exhibited a larger median gap between vulnerable and non-vulnerable lines compared to other scoring functions.

To quantify these findings, we computed ROC-AUC for the distributions of vulnerable and non-vulnerable lines for each scoring function component and context size pair, as shown in Table 1. For each context size, we highlighted in bold the component with the highest ROC-AUC. The results indicate that the combined scoring function (δ_{hybrid}, Equation 2) yields the highest ROC-AUC on nearly all context sizes (with statistical significance $p < 0.0002$, using the method of DeLong et al. [13]), with only a single exception where δ_{loss} slightly outperforms it on a context size of 200 lines. Further computing *Spearman's Rho Correlation* (SRC) and *Cohen's d* between each pair of components within δ_{hybrid} reveals strong separation across all pairs (SRC < 0.31 and d > 1.7, except δ_{loss} and δ_{EM}), highlighting that every component makes valuable contribution to δ_{hybrid} from a distinct perspective. We further validate these observations through an ablation study in Appendix C, which demonstrates that removing any component of δ_{hybrid} consistently degrades performance.

3.3 RQ2: Adaptive Context Size

As shown in Figure 4, varying fixed context sizes affect average discrimination ability. However, static windows may confuse the model by including irrelevant or omitting critical code. This leads to fluctuations in performance and makes it difficult to identify an optimal context size.

To address this limitation, we propose Adaptive Context (AC) as defined in Section 2.2 – a structure-aware approach that adaptively selects meaningful context for each line, aiming to provide a more stable and accurate anomaly score. We evaluate AC using the CodeLlama-13B model with the δ_{hybrid} score and compare it to fixed context sizes. Adopting AC significantly improves the

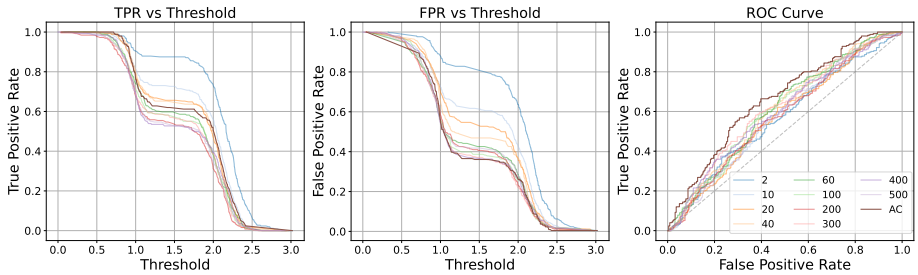

Fig. 5. TPR, FPR & ROC under different contexts

ROC-AUC to 66.3%[2], outperforming all fixed-size settings presented in Table 1 (with $p < 0.0001$ on all pairs). A deeper investigation (Figure 5) reveals that AC effectively reduces the false positive rate compared to fixed-size contexts, likely due to its ability to focus only on relevant context and avoid the lost-in-the-middle issue of longer inputs. Additionally, when using fixed contexts, we encountered three samples that triggered CUDA out-of-memory (OOM) errors at large sizes (400 and 500 lines) due to long line lengths. In contrast, AC avoided these OOMs and reduced peak memory usage to 67GB.

We further measured the time required to run experiments across different context sizes. As shown in Figure 6, inference time increases with larger fixed windows, with the 500-line context taking over 2 seconds to analyze a line – twice as long as the 100-line window. This slowdown is due to the greater computational demands of larger contexts, which increase hardware contention and reduce throughput. In contrast, using AC significantly reduced runtime, making it, on average, nearly as fast as the 100-line fixed window. To understand this, we analyzed the distribution of AC sizes across all samples and found that most were relatively short (median of 86.5 lines).

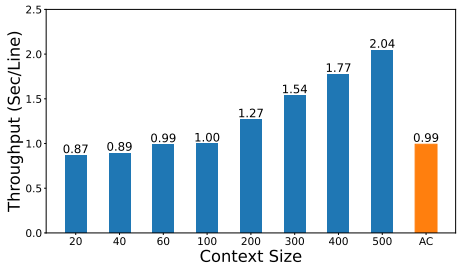

Fig. 6. Throughput (Sec/Line) on fixed context sizes vs. adaptive context

[2] Running on the full Magma dataset yielded a consistent ROC-AUC of 65.8%, confirming our sampling introduced no significant bias.

To summarize, AC dynamically selects semantically meaningful code blocks. By mitigating the limitations of fixed contexts, it achieves the best discrimination between vulnerable and non-vulnerable lines while improving computational efficiency by lowering GPU memory usage and speeding up inference.

3.4 RQ3: Line-level Vulnerability Detection

We compare ANVIL, which does not require any labelled data, against state-of-the-art supervised line-level vulnerability detectors: LineVul, LineVD, and LLMAO. These baselines were selected because they operate at the same line-level granularity as ANVIL, ensuring a fair and consistent comparison. LineVul [18] and LineVD [23] both fine-tune CodeBERT [16] on the BigVul [15] dataset; LineVul uses attention scores from transformer layers, while LineVD incorporates Joern's graph representations [24]. LLMAO [54] is an LLM-based line-level detector built on CodeGen [39], using models up to 16B parameters and a fixed 128-line context window. It was originally trained on Devign [57], which includes only two projects (FFmpeg and QEMU). When evaluated using their released checkpoint, LLMAO exhibits substantially lower performance on PrimeVul than reported in the original study, likely due to the limited coverage of its original training set (Devign). To ensure consistency with other baselines, we retrained LLMAO on BigVul, which covers 4,296 projects. While retraining improves performance across all metrics (Table 2, last row), the resulting ROC-AUC still remains below the 0.855 reported by Yang et al.

For experiments, we configured ANVIL to use the CodeLlama-13B model, the hybrid anomaly score (Equation 2), and Adaptive Context (AC), as this configuration generally yields the best performance (see previous RQs). We evaluate ANVIL using the same two types of metrics employed by our baselines.

1. We approach vulnerability detection as a binary classification task and measure the line-level classification performance using ROC-AUC. This metric treats each line of code as an independent data sample, assessing the overall classification accuracy without setting a specific threshold.
2. To gauge how effective ANVIL's anomaly score is as a vulnerability prioritization tool, we evaluate all detectors with a ranking task. We use Top-N accuracy [26] (a standard metric for measuring vulnerability prioritization ability) to assess whether any of the vulnerable lines are included in the detector's top-n most confident predictions within each function. Additionally, we report the *Mean First Ranking* (MFR) to provide an overall measure.

We performed evaluations on the PrimeVul dataset described in Section 3.1. We selected this dataset over Magma and BigVul to mitigate potential overfitting: The supervised-learning-based detectors were trained and tuned on the BigVul dataset, whereas ANVIL does not require labelled data; however, it uses calibration and scaling constants V and C, as well as normalization factors, that are derived from Magma (introduced in Section 2.3). These processes could introduce biases specific to their respective datasets. By using PrimeVul as a separate evaluation dataset, we ensure independence and promote a fairer comparison.

Table 2. Vulnerability Detection & Localization on PrimeVul Dataset

Method	Dataset	Top-1↑	Top-3↑	Top-5↑	MFR↓	N-MFR↓	ROC-AUC↑
ANVIL	N/A	**11.7%**	**31.3%**	**41.4%**	25.0	**0.18**	**61.8%**
LLMAO	BigVul	9.2%	17.3%	26.2%	49.3	0.39	51.8%
LineVD	BigVul	9.2%	15.4%	23.1%	38.0	0.30	49.6%
LineVul	BigVul	4.0%	17.0%	30.0%	**17.6**	0.71	49.6%
LLMAO	Devign	4.4%	14.0%	22.5%	50.7	0.40	50.7%

Line-level Vulnerability Classification. We first evaluate ANVIL against LineVul, LineVD, and LLMAO in their ability to classify lines with vulnerabilities using ROC-AUC scores. This experiment uses the PrimeVul dataset with several accommodations to ensure fairness and compatibility. First, following the *independent testing* principle outlined in [40,41,10], we ensured that the testing and training sets include data from different sources to avoid bias caused by duplicated data distributions. Specifically, we included only projects not part of the training datasets (BigVul and Devign) used for the baselines. For the remaining data, we included only functions with corresponding source files available to support varying context window sizes used in ANVIL and LLMAO. Furthermore, in line with LineVD and LLMAO, all whitespace lines were removed from the source files. We also considered only vulnerable lines with paired patched versions. Comments were then filtered out to create the final versions of vulnerable lines. This process resulted in a dataset containing 924 vulnerable LoC and 33,820 non-vulnerable LoC from 273 vulnerable functions, along with 325,353 LoC from 9,273 benign functions. These vulnerabilities span 40 different CWEs, with detailed breakdowns provided in [3].

Additionally, as ANVIL operates differently from the three other tools, we made the following adjustments. Due to the way LineVul uses CodeBERT (processing entire functions as single inputs), it inherits CodeBERT's 512-token limit. For functions exceeding this limit, LineVul truncates them to include only the first 512 tokens, discarding the rest. Consequently, we evaluate LineVul only on lines it retains, which is 455 vulnerable LoC and 192,382 non-vulnerable LoC in the dataset, whereas the other three tools are evaluated on the full dataset of approximately 360k LoC.

As shown in Table 2, ANVIL achieves ROC-AUC scores of 61.8% with CodeLlama-13B, outperforming all three baselines (with $p < 0.0001$). In contrast, LineVul, LineVD and LLMAO yield only ROC-AUC scores close to that of random guessing (approximately 50%). Although these results are lower than the original performance reported in the respective papers, they align with findings from the DiverseVul study [10], which demonstrated that supervised vulnerability detectors experience significant performance degradation when evaluated on projects not seen during training. This observation has been further validated by subsequent works [40,14]. In contrast, since ANVIL does not rely on labelled

training data, its anomaly-based detection strategy exhibits stronger generalization to different projects.

As ROC-AUC captures overall detection performance across all thresholds, we use it as our primary metric; Precision, Recall, and F1 scores are reported in Appendix B for completeness.

Vulnerability Prioritization. This experiment evaluates the ability of vulnerability detectors to prioritize vulnerable lines within a given function by comparing their Top-N accuracies and Mean First Ranking (MFR). Specifically, Top-N accuracy measures whether at least one vulnerable line is included among the top N lines ranked by each detector. For ANVIL, lines are ranked based on anomaly scores, while LineVul ranks lines using attention scores, and LineVD and LLMAO rely on logits from their respective neural network heads. We report Top-1, Top-3, and Top-5 accuracies, as these metrics are originally used by the baselines. Additionally, we calculate MFR (smaller is better) by averaging the rank of the highest-ranked vulnerable line in each function, providing a comprehensive measure of overall vulnerability prioritization performance. This experiment uses 273 vulnerable functions from the PrimeVul dataset, consisting of 924 vulnerable LoC and 33,820 non-vulnerable LoC.

As shown in Table 2, ANVIL outperforms all baselines in Top-N accuracy. While it achieves a modest 1.27 times improvement in Top-1 accuracy over the best-performing baseline, LLMAO, its advantage becomes more pronounced when the criterion is relaxed to Top-3 and Top-5, with ANVIL achieving 1.81 and 1.58 times higher accuracies, respectively. Notably, the 512-token limit of LineVul results in an average of only 25 available LoC per function, compared to an average function length of 141 LoC for the other detectors. This shortened context theoretically gives LineVul an advantage in achieving higher Top-N accuracy due to the smaller candidate pool. However, its performance falls short, achieving only 4.0% Top-1 accuracy (the lowest among all detectors), and its Top-3 and Top-5 accuracies remain below those of ANVIL.

The limited function length also explains why LineVul appears to achieve the best MFR. To enable a fairer comparison, we use Normalized Mean First Ranking (N-MFR), which adjusts the raw MFR based on average function length. For LineVul, we apply a normalization factor of 25 LoC, while for all other detectors, we use an average function length of 141 LoC. The normalized results show that ANVIL achieves the best N-MFR of 0.18, which is 54% lower than LLMAO and 40% lower than LineVD. Although LineVul originally records the lowest raw MFR, its normalized score rises to 0.71, yielding the worst N-MFR score.

3.5 RQ4: Data Leakage

In this section, we address concerns about potential data leakage that may arise if patches for the vulnerabilities in our evaluation dataset were present in the LLM training data, as this may enable the LLM to correctly generate the patched code instead of the vulnerable code, as discussed in Section 3.1. The evaluation

is conducted on CodeLlama-13B with the hybrid scoring function and Adaptive Context.

We assess the generalizability of ANVIL in detecting line-level vulnerabilities by comparing its detection performance on PrimeVul and on our newly collected 2024 CVEFixes dataset described in Section 3.1. Similar to RQ3, this evaluation examines both vulnerable line identification and prioritization performance.

Table 3. ANVIL's generalizability on 2024 CVEFixes Dataset

Tested On	Top-1	Top-3	Top-5	MFR	N-MFR	ROC
PrimeVul	11.7%	31.3%	41.4%	25.0	0.18	61.8%
CVEFixes	16.4%	33.6%	47.3%	16.7	0.22	68.6%

Similar to previous experiments, we evaluate vulnerability prioritization using Top-N accuracies, MFR, and N-MFR. For N-MFR, we apply a normalization factor of 76 LoC, representing the average vulnerable function length in the 2024 CVEFixes dataset. The evaluation covers 147 vulnerable functions from 100 CVEs. As shown in Table 3, ANVIL demonstrates similar Top-1 to Top-5 accuracies across both datasets, with slightly better performance on CVEFixes. This improvement is expected, as the shorter average function length in CVEFixes theoretically results in higher prioritization metrics. After normalization, the N-MFR scores converge to 0.18 and 0.22 for PrimeVul and CVEFixes, respectively. This consistency highlights the robustness of ANVIL's vulnerability prioritization capabilities, even when applied to previously unseen datasets.

For the evaluation of vulnerability classification, we assess all 3,736 functions in the dataset, which consists of 235 vulnerable LoC and over 110k non-vulnerable LoC. As shown in Table 3, ANVIL achieves a higher ROC-AUC score of 68.6% ($p < 0.0001$) on the 2024 CVEFixes dataset compared to 61.8% on the PrimeVul dataset. This result confirms that our anomaly-based approach generalizes and is robust to potential data leakage, accurately detecting vulnerabilities in unseen datasets.

4 Real-World Integration: Enhancing Fuzzing with ANVIL

In Section 3, ANVIL demonstrated superior vulnerability detection and prioritization capabilities compared to existing ML-based detectors. However, due to the inherent difficulty of vulnerability review, even a low false-positive rate can lead to hours of manual investigation, making ML-based detectors, including ANVIL, potentially impractical as standalone static analysis tools in production settings. Nonetheless, we hypothesize that ANVIL can still be effective, especially when used in combination with other vulnerability analysis tools, e.g., with fuzzers.

One promising application of ANVIL is to guide fuzzing seed selection. Traditional fuzzers use initial inputs or *seeds* to generate large numbers of mutated inputs in each iteration, aiming to maximize code coverage. However, many of these seeds, despite increasing coverage, may not exercise vulnerable code paths. This leads fuzzers to waste effort on "uninformative" inputs. In contrast, anomaly scores produced by ANVIL can help prioritize seeds that are more likely to interact with potentially vulnerable code regions, thereby accelerating bug discovery.

As a proof of concept, we integrated ANVIL with three fuzzers included in the Magma benchmark [21]: AFL++[17], Honggfuzz[20], and LibFuzzer [35]. The Magma benchmark includes 24 binaries across 9 fuzzing targets, and provides an initial corpus of 19,748 seeds. For each seed, we executed the binary to determine its code coverage. We then selected the top 10% most anomalous lines based on ANVIL's scoring results, and retained only those seeds that covered at least one of these lines. This filtering process removed 34% of seeds, leaving 13,059 prioritized ones.

We then conducted 24-hour fuzzing campaigns using both the original and the refined seed sets, repeating each run five times per fuzzer, on a 128-core Intel Xeon Gold 6530 server. Figure 7 shows the cumulative number of bugs triggered over time. After 24 hours, the seed set refined by ANVIL enabled fuzzers to discover 9.4% more bugs (39.6 vs. 36.2). We also report Normalized Area Under the Curve (N-AUC), which accounts for both the number of bugs found and the time taken to trigger them. N-AUC improved by 7.8%, showing that integrating ANVIL with fuzzers not only improves bug yield but also increases efficiency.

Zero-day Vulnerabilities. We applied ANVIL in combination with LibFuzzer and discovered two zero-day vulnerabilities in SFML[3], a widely used multimedia library with over 11,000 GitHub stars. SFML was selected from approximately ten popular C/C++ libraries after preliminary screening for reproducible fuzzing and in-depth analysis suitability. The two bugs are categorized as a heap-use-after-free and a heap-buffer-overflow, respectively.

The heap-use-after-free vulnerability was caused by missing synchronization between SFML's internal audio playback thread and the main thread when stopping a specially crafted audio track prioritized by ANVIL. We reported this bug as a GitHub issue[4], later patched by the developers.

The heap-buffer-overflow vulnerability was responsibly disclosed to MITRE and assigned the reserved identifier CVE-2025-50940. The bug resides in the `minimp3` [33] library used by SFML, and is caused by a mismatch between a pointer and its associated size, leading to an out-of-bounds memory access. The issue was then fixed by the SFML maintainers by replacing the affected library[5].

The two vulnerabilities span different temporal relationships with the LLM's knowledge cutoff: the heap-buffer-overflow originates from pre-cutoff code, while

[3] https://github.com/SFML/SFML
[4] https://github.com/SFML/SFML/issues/3503
[5] https://github.com/SFML/SFML/pull/2995

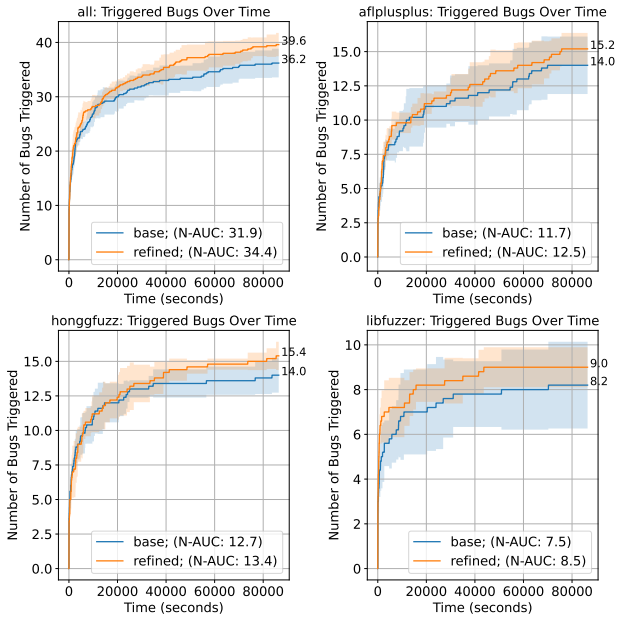

Fig. 7. Number of bugs triggered over time, filled with standard deviations

the heap-use-after-free involves post-cutoff modifications, demonstrating that ANVIL does not rely on memorization or training-data leakage. We also checked SFML using CodeQL [19], a SOTA static analyzer. When using its "extended" query suite (known for high false-positive rates), CodeQL flagged 48 potential issues. However, none were related to the two zero-days. In contrast, ANVIL was able to prioritize the root-cause lines of both vulnerabilities within the top 10% most anomalous lines, effectively guiding LibFuzzer to trigger the bugs.

5 Discussion

ANVIL demonstrates effective vulnerability detection by reframing the task as anomaly detection. From a systems perspective, ANVIL is intentionally designed to operate directly on raw source code and to be fully parallelizable, enabling independent LLM inference across lines at scale. This simplicity facilitates deployment while leaving room for refinement. For example, lightweight static analysis or preprocessing may further enhance anomaly discrimination by providing richer semantic signals, such as LLM-generated summaries or inter-function-level information injected as auxiliary context. At the same time, such techniques introduce additional overhead and design trade-offs that require careful evaluation. Exploring these extensions forms a natural part of our roadmap.

Beyond this, our current fuzzing integration shows that anomaly scores effectively prioritize vulnerability-prone regions, motivating further exploration of

anomaly-guided dynamic analysis. To our knowledge, while prior work has used ML models to select seeds for optimizing coverage [9,53], prioritizing seeds based on bug likelihood remains unexplored. While our current integration focuses on filtering the initial seed corpus, richer interactions with fuzzing workflows are possible. As fuzzing progresses, newly generated seeds may exercise highly anomalous code regions, suggesting that dynamically incorporating anomaly feedback could further improve performance. Moreover, anomaly scores naturally lend themselves to guiding optimization-based directed fuzzers such as AFLGo [7], where anomaly-ranked locations could serve as fuzzing targets. Realizing such integrations, however, requires careful consideration of how targets are selected and grouped, as anomaly-based scanning may identify many locations across diverse execution paths. Designing strategies that balance focus and coverage remains an important direction for future work.

6 Related Work

Anomaly Detection. Prior work has leveraged anomalous behaviour to identify bugs. Yun et al. [55] detect API misuse by learning correct usage patterns, while Ahmadi et al. [2] identify bugs by clustering inconsistent code within a project. To our knowledge, the only prior LLM-based anomaly-guided vulnerability detector compares LLM outputs to ground truth using string distance [1], which lacks contextual understanding and relies on pre-existing comments. In contrast, ANVIL incorporates context-aware, multi-faceted anomaly scoring without assuming comment availability.

ML-Based Vulnerability Detection. Prior work has explored CNNs, GNNs, and LSTM-based models for vulnerability detection [44,12,31], but such approaches often fail to generalize to unseen real-world bugs [8]. Other work considers alternative code representations, such as code slices and bug-triggering paths [37,11]. Machine learning has also been used to enhance fuzzing, including predicting inputs for smart contracts [22], enabling input-format-aware fuzzing via basic-block classification [45], and deriving protocol grammars using LLMs [36].

LLM-Based Vulnerability Detection. Recent work explores using LLMs for vulnerability detection. Zhang et al. [56] show that prompt-based ChatGPT usage can outperform traditional detectors, while Lin et al. [32] systematically evaluate open-source LLMs for file-level detection across model sizes, quantization settings, and context lengths. However, these approaches typically rely on fixed or maximum-length contexts and carefully crafted prompts. Wu et al. [52] leverage in-context learning and semantic similarity for function localization given known CVE descriptions, targeting localization rather than detection of unknown vulnerabilities. In contrast, ANVIL dynamically adjusts context size, requires no vulnerability descriptions, and exploits the FIM pretraining objective without task-specific prompt engineering.

Several studies combine LLMs with structured program representations. Li et al. [29] apply contrastive learning over code property graphs and vulnerability descriptions to learn vulnerability-aware representations, while Wen et al. [49] and Ni et al. [38] integrate LLM-derived features with program structures such as ASTs, CFGs, and PDGs for supervised vulnerability detection. Lekssays et al. [27] further extract vulnerability-related execution paths via an LLM-based graph query generator and augment them into code slices for classification.

These approaches incur nontrivial overhead due to program graph construction and typically rely on labelled data or auxiliary information such as vulnerability descriptions or execution paths. In contrast, ANVIL operates directly on raw source code, is fully parallelizable, and requires no labelled vulnerability data. Finally, it is worth noting that all LLM-based approaches discussed in this subsection evaluate vulnerability detection at coarse granularities, including function-, file-, or project-level detection, whereas ANVIL performs line-level detection, providing finer-grained and more actionable vulnerability localization.

7 Conclusion

To address the limitations of supervised-learning-based vulnerability detectors, we propose ANVIL, which leverages LLM mispredictions to guide vulnerability detection. Our method treats discrepancies between LLM-predicted and actual code as anomalies that can signal vulnerabilities. We demonstrate that Adaptive Context (AC) outperforms fixed-sized contexts for LLM predictions. Our approach generalizes across LLM sizes, architectures, and datasets, including vulnerabilities disclosed after model training. In comparative evaluations, ANVIL outperforms state-of-the-art supervised models, LineVul, LineVD, and LLMAO, despite requiring no labelled training data. Finally, by integrating ANVIL with fuzzers, we discover two previously unknown vulnerabilities in popular software.

Acknowledgments This research was supported in part by NSERC Discovery Grant RGPIN-2018-05931 and NSERC-CSE Research Communities Grant ALLRP 588144-23. David Lie is supported by Tier 1 Canada Research Chair CRC-2019-00242. Researchers funded through the NSERC-CSE Research Communities Grants do not represent the Communications Security Establishment Canada or the Government of Canada. Any research, opinions or positions they produce as part of this initiative do not represent the official views of the Government of Canada.

Appendix A Model Architecture and Model Size

This appendix evaluates whether anomaly-based vulnerability detection generalizes across different LLM architectures and model sizes. We perform the FIM workload on multiple open-source code LLMs that support FIM, selected from the top-ranked base models on Huggingface's Big Code Models Leaderboard as of

May 1, 2025 [6], and additionally study the effect of model scaling by comparing different sizes within the same architecture.

Table 4 summarizes the evaluated models, covering three architectures and sizes ranging from 0.5B to 33B parameters. For each model, we apply AC and compute ROC-AUC using the δ_{hybrid} anomaly score. As shown in Table 5, all models achieve ROC-AUC above 60% ($p < 0.0001$). Pairwise comparisons further show consistent anomaly rankings across models, with Spearman rank correlation exceeding 0.6 for all pairs and Cohen's d below 0.2 in 53 of 55 comparisons, indicating that the anomaly-based detection approach generalizes well across model architectures. Across model sizes within the same architecture, larger models generally achieve higher ROC-AUC, suggesting that discriminative capability improves with neural model scaling.

Table 4. Code LLMs used in the experiments

Model Name	Sizes	Training data [6]	Publication
CodeLlama	7B, 13B	29B LOC	Aug 2023
DeepSeek-Coder	1.3B, 6.7B, 33B	114B LOC	Jan 2024
Qwen2.5-Coder	0.5B, 1.5B, 3B,7B, 14B, 32B	314B LOC	Nov 2024

Table 5. ROC-AUC of different LLMs on the Magma Dataset

	Model Size					
LLM	0.5B	1.5B	3B	7B	14B	32B
Qwen	61.0%	61.4%	65.3%	64.0%	64.1%	65.6%
DeepSeek	N/A	60.5%	N/A	61.5%	N/A	62.3%
CodeLlama	N/A	N/A	N/A	63.9%	66.3%	N/A

Appendix B Precision, Recall and F1 Score

Table 6 reports Precision, Recall, and F1 scores for ANVIL and baselines on the full PrimeVul dataset (33,820 non-vulnerable vs. 924 vulnerable lines) and a balanced subset (924/924). For each method, the decision threshold is chosen to maximize F1, and the corresponding Precision and Recall are reported.

ANVIL achieves the highest F1 and Precision in both settings. While some baselines attain higher Recall at their F1-optimal thresholds, this comes at the cost of substantially lower Precision due to increased false positives, resulting in lower overall F1 scores than ANVIL. Because Precision and Recall are reported

[6] Approximated using 4 characters per token and 70 characters per LOC.

at method-specific operating points and reflect different trade-offs, we use ROC-AUC as the primary metric in the main text, and include Precision, Recall, and F1 results in the appendix for completeness and consistency with prior work.

Table 6. Precision, Recall and F1 Scores on PrimeVul Dataset

Method	Full Dataset			Balanced Dataset		
	Prec	Recall	F1↑	Prec	Recall	F1↑
ANVIL	**0.62%**	15.60%	**1.19%**	**52.75%**	93.95%	**67.56%**
LLMAO	0.43%	**77.95%**	0.85%	51.01%	97.60%	67.00%
LineVD	0.24%	21.95%	0.47%	50.00%	**100.00%**	66.67%
LineVul	0.33%	10.77%	0.64%	50.00%	**100.00%**	66.67%

Appendix C Ablation Study

While RQ1 (Section 3.2) evaluates anomaly score components in isolation, this ablation study assesses their contributions within the combined hybrid score δ_{hybrid} by selectively removing components. As shown in Table 7, the full hybrid score consistently achieves the best overall performance across metrics (with $p < 0.0001$ for ROC-AUC), and performance degrades as components are removed.

The only exception is Top-1 accuracy, which exhibits higher variance and where δ_{hybrid} narrowly misses the best result (11.7%). This behaviour arises because the model often assigns high anomaly scores to lines adjacent to the vulnerable line, occasionally ranking them above the exact root-cause line. Overall, the hybrid scoring function remains the most robust, with each component contributing to effective vulnerability detection.

Table 7. Ablation Study

loss	EM	var	ast	ROC↑	Top-1↑	Top-3↑	Top-5↑	N-MFR↓
✓	✓	✓	✓	**61.8%**	11.7%	**31.3%**	**41.4%**	**0.18**
✓	✓	✓		59.0%	10.2%	28.5%	41.0%	0.20
✓	✓			57.8%	**12.1%**	27.3%	39.8%	0.20
✓				57.1%	11.3%	26.6%	39.1%	0.22

References

1. Ahmad, B., Tan, B., Karri, R., Pearce, H.: FLAG: Finding Line Anomalies (in code) with Generative AI (Jun 2023), http://arxiv.org/abs/2306.12643, arXiv:2306.12643 [cs]

2. Ahmadi, M., Farkhani, R.M., Williams, R., Lu, L.: Finding Bugs Using Your Own Code: Detecting Functionally-similar yet Inconsistent Code. pp. 2025–2040 (2021), https://www.usenix.org/conference/usenixsecurity21/presentation/ahmadi
3. ANVIL: Distribution of CWEs in ANVIL's PrimeVul Validation and Test sets, https://anonymous.4open.science/r/anvil/appendices/CWE_Table.csv
4. Bavarian, M., Jun, H., Tezak, N., Schulman, J., McLeavey, C., Tworek, J., Chen, M.: Efficient Training of Language Models to Fill in the Middle (Jul 2022), http://arxiv.org/abs/2207.14255, arXiv:2207.14255 [cs]
5. Bhandari, G., Naseer, A., Moonen, L.: 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. pp. 30–39. PROMISE 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3475960.3475985
6. BigCode: Big Code Models Leaderboard - a Hugging Face Space by bigcode (Jul 2024), https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard
7. Böhme, M., Pham, V.T., Nguyen, M.D., Roychoudhury, A.: Directed Greybox Fuzzing. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 2329–2344. ACM, Dallas Texas USA (Oct 2017). https://doi.org/10.1145/3133956.3134020
8. Chakraborty, S., Krishna, R., Ding, Y., Ray, B.: Deep Learning Based Vulnerability Detection: Are We There Yet? IEEE Transactions on Software Engineering **48**(9), 3280–3296 (Sep 2022). https://doi.org/10.1109/TSE.2021.3087402, conference Name: IEEE Transactions on Software Engineering
9. Chen, Y., Ahmadi, M., farkhani, R.M., Wang, B., Lu, L.: MEUZZ: Smart seed scheduling for hybrid fuzzing. In: 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020). pp. 77–92. USENIX Association, San Sebastian (Oct 2020), https://www.usenix.org/conference/raid2020/presentation/chen
10. Chen, Y., Ding, Z., Alowain, L., Chen, X., Wagner, D.: 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. pp. 654–668. RAID '23, Association for Computing Machinery, New York, NY, USA (Oct 2023). https://doi.org/10.1145/3607199.3607242
11. Cheng, X., Nie, X., Li, N., Wang, H., Zheng, Z., Sui, Y.: How About Bug-Triggering Paths? - Understanding and Characterizing Learning-Based Vulnerability Detectors. IEEE Transactions on Dependable and Secure Computing pp. 1–18 (2022). https://doi.org/10.1109/TDSC.2022.3192419
12. Cheng, X., Wang, H., Hua, J., Xu, G., Sui, Y.: DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network. ACM Transactions on Software Engineering and Methodology **30**(3), 1–33 (May 2021). https://doi.org/10.1145/3436877
13. DeLong, E.R., DeLong, D.M., Clarke-Pearson, D.L.: Comparing the areas under two or more correlated receiver operating characteristic curves: A nonparametric approach. Biometrics **44**(3), 837–845 (1988), http://www.jstor.org/stable/2531595
14. Ding, Y., Fu, Y., Ibrahim, O., Sitawarin, C., Chen, X., Alomair, B., Wagner, D., Ray, B., Chen, Y.: Vulnerability Detection with Code Language Models: How Far Are We? (Jul 2024), http://arxiv.org/abs/2403.18624, arXiv:2403.18624 [cs]
15. Fan, J., Li, Y., Wang, S., Nguyen, T.N.: A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In: Proceedings of the 17th International Conference on Mining Software Repositories. pp. 508–512. MSR '20, Association

for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3379597.3387501
16. Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., Zhou, M.: CodeBERT: A Pre-Trained Model for Programming and Natural Languages (Sep 2020), http://arxiv.org/abs/2002.08155, arXiv:2002.08155 [cs]
17. Fioraldi, A., Maier, D., Eißfeldt, H., Heuse, M.: AFL++: Combining incremental steps of fuzzing research. In: 14th USENIX Workshop on Offensive Technologies (WOOT 20). USENIX Association (Aug 2020)
18. Fu, M., Tantithamthavorn, C.: LineVul: a transformer-based line-level vulnerability prediction. In: Proceedings of the 19th International Conference on Mining Software Repositories. pp. 608–620. ACM, Pittsburgh Pennsylvania (May 2022). https://doi.org/10.1145/3524842.3528452
19. GitHub: CodeQL, https://codeql.github.com/
20. Google: google/honggfuzz (May 2025), https://github.com/google/honggfuzz, original-date: 2015-05-07T15:43:01Z
21. Hazimeh, A., Herrera, A., Payer, M.: Magma: A Ground-Truth Fuzzing Benchmark. Proceedings of the ACM on Measurement and Analysis of Computing Systems **4**(3), 49:1–49:29 (Nov 2020). https://doi.org/10.1145/3428334
22. He, J., Balunović, M., Ambroladze, N., Tsankov, P., Vechev, M.: Learning to Fuzz from Symbolic Execution with Application to Smart Contracts. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 531–548. CCS '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3319535.3363230
23. Hin, D., Kan, A., Chen, H., Babar, M.A.: LineVD: statement-level vulnerability detection using graph neural networks. In: Proceedings of the 19th International Conference on Mining Software Repositories. pp. 596–607. ACM, Pittsburgh Pennsylvania (May 2022). https://doi.org/10.1145/3524842.3527949
24. Joern: Joern - The Bug Hunter's Workbench. https://joern.io/ (Jul 2024)
25. Kate, K., Pedapati, T., Basu, K., Rizk, Y., Chenthamarakshan, V., Chaudhury, S., Agarwal, M., Abdelaziz, I.: Longfunceval: Measuring the effectiveness of long context models for function calling (2025), https://arxiv.org/abs/2505.10570
26. scikit learn: top_k_accuracy_score, https://scikit-learn/stable/modules/generated/sklearn.metrics.top_k_accuracy_score.html
27. Lekssays, A., Mouhcine, H., Tran, K., Yu, T., Khalil, I.: Llmxcpg: context-aware vulnerability detection through code property graph-guided large language models. In: Proceedings of the 34th USENIX Conference on Security Symposium. SEC '25, USA (2025)
28. Li, J., Zhu, H., Liu, H., Shi, X., Zong, H., Dong, Y., Zhang, K., Jiang, S., Jin, Z., Li, G.: aixcoder-7b-v2: Training llms to fully utilize the long context in repository-level code completion (2025), https://arxiv.org/abs/2503.15301
29. Li, J., Cui, L., Zhao, S., Yang, Y., Li, L., Zhu, H.: CLeVeR: Multi-modal contrastive learning for vulnerability code representation. In: Findings of the Association for Computational Linguistics: ACL 2025. pp. 7940–7951. Vienna, Austria (Jul 2025). https://doi.org/10.18653/v1/2025.findings-acl.414
30. Li, Y., Wang, S., Nguyen, T.N.: Vulnerability detection with fine-grained interpretations. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 292–303. ESEC/FSE 2021, Association for Computing Machinery, New York, NY, USA (Aug 2021). https://doi.org/10.1145/3468264.3468597

31. Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., Zhong, Y.: VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In: Proceedings 2018 Network and Distributed System Security Symposium (2018). https://doi.org/10.14722/ndss.2018.23158, arXiv:1801.01681 [cs]
32. Lin, J., Mohaisen, D.: From Large to Mammoth: A Comparative Evaluation of Large Language Models in Zero-Shot Vulnerability Detection. In: Proceedings 2025 Network and Distributed System Security Symposium. San Diego, CA, USA (2025). https://doi.org/10.14722/ndss.2025.241491
33. Lion: lieff/minimp3 (Jan 2026), https://github.com/lieff/minimp3, original-date: 2018-01-09T14:58:36Z
34. Liu, N.F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., Liang, P.: Lost in the middle: How language models use long contexts. Transactions of the Association for Computational Linguistics **12**, 157–173 (2024). https://doi.org/10.1162/tacl_a_00638
35. LLVM: libFuzzer – a library for coverage-guided fuzz testing. — LLVM 21.0.0git documentation, https://llvm.org/docs/LibFuzzer.html
36. Meng, R., Mirchev, M., Böhme, M., Roychoudhury, A.: Large Language Model guided Protocol Fuzzing. In: Proceedings 2024 Network and Distributed System Security Symposium. Internet Society, San Diego, CA, USA (2024). https://doi.org/10.14722/ndss.2024.24556
37. Mirsky, Y., Macon, G., Brown, M., Yagemann, C., Pruett, M., Downing, E., Mertoguno, S., Lee, W.: {VulChecker}: Graph-based Vulnerability Localization in Source Code. pp. 6557–6574 (2023), https://www.usenix.org/conference/usenixsecurity23/presentation/mirsky
38. Ni, C., Yin, X., Li, X., Xu, X., Yu, Z.: Abundant modalities offer more nutrients: Multi-modal-based function-level vulnerability detection **35**(2) (Jan 2026). https://doi.org/10.1145/3731557
39. Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., Xiong, C.: CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis (Feb 2023). https://doi.org/10.48550/arXiv.2203.13474, arXiv:2203.13474 [cs]
40. Nong, Y., Fang, R., Yi, G., Zhao, K., Luo, X., Chen, F., Cai, H.: VGX: Large-Scale Sample Generation for Boosting Learning-Based Software Vulnerability Analyses. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. pp. 1–13. ACM, Lisbon Portugal (Apr 2024). https://doi.org/10.1145/3597503.3639116
41. Nong, Y., Ou, Y., Pradel, M., Chen, F., Cai, H.: VULGEN: Realistic Vulnerability Generation Via Pattern Mining and Deep Learning. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). pp. 2527–2539. IEEE, Melbourne, Australia (May 2023). https://doi.org/10.1109/ICSE48619.2023.00211
42. OpenAI: ChatGPT. https://openai.com/chatgpt/ (Jul 2024)
43. Ray, B., Hellendoorn, V., Godhane, S., Tu, Z., Bacchelli, A., Devanbu, P.: On the "naturalness" of buggy code. In: Proceedings of the 38th International Conference on Software Engineering. pp. 428–439. ICSE '16, Association for Computing Machinery, New York, NY, USA (May 2016). https://doi.org/10.1145/2884781.2884848
44. Russell, R., Kim, L., Hamilton, L., Lazovich, T., Harer, J., Ozdemir, O., Ellingwood, P., McConley, M.: Automated Vulnerability Detection in Source Code Using Deep Representation Learning. In: 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA). pp. 757–762 (Dec 2018). https://doi.org/10.1109/ICMLA.2018.00120

45. Shi, J., Wang, Z., Feng, Z., Lan, Y., Qin, S., You, W., Zou, W., Payer, M., Zhang, C.: AIFORE: Smart fuzzing based on automatic input format reverse engineering. In: 32nd USENIX Security Symposium (USENIX Security 23). pp. 4967–4984. USENIX Association, Anaheim, CA (Aug 2023), https://www.usenix.org/conference/usenixsecurity23/presentation/shi-ji
46. of Standards, N.I., Technology: National vulnerability database, https://nvd.nist.gov/
47. Steenhoek, B., Rahman, M.M., Roy, M.K., Alam, M.S., Barr, E.T., Le, W.: A Comprehensive Study of the Capabilities of Large Language Models for Vulnerability Detection (Mar 2024), http://arxiv.org/abs/2403.17218, arXiv:2403.17218 [cs]
48. Wang, Y., Wang, W., Joty, S., Hoi, S.C.H.: CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation (Sep 2021), http://arxiv.org/abs/2109.00859, arXiv:2109.00859 [cs]
49. Wen, X.C., Gao, C., Gao, S., Xiao, Y., Lyu, M.R.: SCALE: Constructing Structured Natural Language Comment Trees for Software Vulnerability Detection (Mar 2024). https://doi.org/10.48550/arXiv.2403.19096, arXiv:2403.19096 [cs]
50. Wikipedia: Box plot. https://en.wikipedia.org/w/index.php?title=Box_plot&oldid=1268089357 (Jan 2025)
51. Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Le Scao, T., Gugger, S., Drame, M., Lhoest, Q., Rush, A.: Transformers: State-of-the-Art Natural Language Processing. In: Liu, Q., Schlangen, D. (eds.) Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations. pp. 38–45. Association for Computational Linguistics, Online (Oct 2020). https://doi.org/10.18653/v1/2020.emnlp-demos.6
52. Wu, Y., Wen, M., Yu, Z., Guo, X., Jin, H.: Effective vulnerable function identification based on cve description empowered by large language models. In: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering. p. 393–405. New York, NY, USA (2024). https://doi.org/10.1145/3691620.3695013
53. Xu, H., Chen, L., Gan, S., Zhang, C., Li, Z., Ji, J., Chen, B., Hu, F.: Graphuzz: Data-driven seed scheduling for coverage-guided greybox fuzzing. ACM Trans. Softw. Eng. Methodol. **33**(7) (Aug 2024). https://doi.org/10.1145/3664603, https://doi.org/10.1145/3664603
54. Yang, A.Z.H., Le Goues, C., Martins, R., Hellendoorn, V.: Large Language Models for Test-Free Fault Localization. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. pp. 1–12. ICSE '24, Association for Computing Machinery, New York, NY, USA (Feb 2024). https://doi.org/10.1145/3597503.3623342
55. Yun, I., Min, C., Si, X., Jang, Y., Kim, T., Naik, M.: {APISan}: Sanitizing {API} Usages through Semantic {Cross-Checking}. pp. 363–378 (2016), https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/yun
56. Zhang, C., Liu, H., Zeng, J., Yang, K., Li, Y., Li, H.: Prompt-enhanced software vulnerability detection using chatgpt. p. 276–277. ICSE-Companion '24, New York, NY, USA (2024). https://doi.org/10.1145/3639478.3643065
57. Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y.: Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks (Sep 2019). https://doi.org/10.48550/arXiv.1909.03496, http://arxiv.org/abs/1909.03496, arXiv:1909.03496 [cs] version: 1