K-ASTRO: Structure-Aware Adaptation of LLMs for Code Vulnerability Detection

Anonymous Author(s)

ABSTRACT

Large Language Models (LLMs) are transforming software engineering tasks, including code vulnerability detection—a critical area of software security. However, existing methods often rely on resource-intensive models or graph-based techniques, limiting their accessibility and practicality. This paper introduces K-ASTRO, a lightweight Transformer model that combines semantic embeddings from LLMs with structural features of Abstract Syntax Trees (ASTs) to improve both efficiency and accuracy in code vulnerability detection. Our approach introduces an AST-based augmentation technique inspired by mutation testing, a structure-aware attention mechanism that incorporates augmented AST features, and a joint adaptation pipeline to unify code semantics and syntax. Experimental results on three large-scale datasets—BigVul, DiverseVul, and PrimeVul—demonstrate state-of-the-art performance while enabling rapid inference on CPUs with minimal training time. By offering a scalable, interpretable, and efficient solution, K-ASTRO bridges the gap between LLM advancements and practical software vulnerability detection, providing open-sourced tools to foster further research.

ACM Reference Format:
Anonymous Author(s). 2025. K-ASTRO: Structure-Aware Adaptation of LLMs for Code Vulnerability Detection . In *Companion Proceedings of the 33rd ACM Symposium on the Foundations of Software Engineering (FSE '25), June 23–27, 2025, Trondheim, Norway.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Large Language Models (LLMs) have demonstrated remarkable capabilities in tasks such as question answering, code generation, and text summarization [18, 43, 44]. Built upon the Transformer architecture [32], LLMs leverage large datasets to solve domain-specific problems with unprecedented efficiency, making them increasingly integral in software engineering. Among the various applications of LLMs, code vulnerability detection holds particular significance, where the goal is to determine whether a given piece of code is vulnerable to security threats. Early and reliable detection of vulnerabilities minimizes the risk of exploitation and reduces the cost of addressing these issues later in the software lifecycle.

Conference'17, July 2017, Washington, DC, USA
© 2025 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Effectively leveraging LLMs for vulnerability detection presents two key challenges. First, pre-training or fine-tuning Transformer-based LLMs is computationally demanding, often requiring extensive GPU resources unavailable to many practitioners [45]. Second, capturing the intricate nuances of software vulnerabilities in standalone code functions remains a significant hurdle. For example, it is challenging to infer whether inputs to a function have been sanitized prior to their use. While pretrained models [9, 13, 31, 36] with hundreds of millions to billions of parameters offer state-of-the-art baselines, they remain resource-intensive and underperform on nuanced tasks like vulnerability detection when used off-the-shelf [5, 7, 11, 29]. Notably, GitHub employs LLMs with CodeQL [12], combining generative models with static code analysis to identify vulnerabilities at scale, but this setup requires significant infrastructure investments.

In this paper, we introduce K-ASTRO, a novel and lightweight Transformer-based model that combines semantic embeddings from LLMs with structural features derived from Abstract Syntax Trees (ASTs) to enhance code vulnerability detection. Our approach addresses the challenges of efficiency and nuance by introducing three key innovations: (i) **Diversity-Introducing AST Augmentation**, which enhances feature diversity through an AST-based mutation method inspired by mutation testing; (ii) **Structure-Aware Attention Bias**, a novel mechanism that incorporates augmented AST features into the Transformer block, guiding the model's attention to structural relationships; and (iii) **Joint LLM Adaptation**, a training pipeline that unifies structural and semantic information for improved prediction accuracy. These innovations bridge the gap between off-the-shelf LLM capabilities and the domain-specific requirements of code vulnerability analysis. Our contributions are as follows:

- We propose K-ASTRO, a lightweight, single-layer, encoder-only Transformer that unifies code syntax (via AST features) and semantics (via LLM embeddings), significantly improving both binary vulnerability prediction and CWE classification.
- We evaluate K-ASTRO on three large-scale, real-world datasets: BigVul, DiverseVul, and PrimeVul. These datasets cover hundreds of open-source projects. K-ASTRO achieves state-of-the-art performance with minimal computational requirements.
- We conduct an ablation study to validate K-ASTRO's design, comparing it with simpler models that classify code embeddings without leveraging AST structure.
- To foster further research, we open-source all code, datasets, and tools, including scripts for data preprocessing, LLM API interactions, model training, evaluation, and embedding generation.

The remainder of this paper is organized as follows: Section 2 outlines the problem formulation and background. Section 3 describes the datasets and architectural details of K-ASTRO. Section 4 presents experimental results, guided by four research questions. Section 5 discusses related work in LLM-based vulnerability detection. Finally, Section 6 summarizes our findings and highlights the limitations of our approach.

2 PROBLEM STATEMENT

In this section, we introduce the problem of vulnerability detection, motivate our approach with an example of vulnerable code, and describe the Abstract Syntax Tree (AST) representation that underpins our design.

2.1 Vulnerability Detection

Detecting vulnerabilities in real-world software is a challenging task, particularly given the sheer scale of modern codebases—often comprising millions of lines of code contributed by multiple developers. Despite rigorous testing [2], vulnerabilities persist due to their dispersed nature and the potential for fixes to inadvertently introduce new issues. Common automated approaches, such as static analysis [47], fuzzing [48], and software testing [39], often suffer from high rates of false positives and negatives.

Vulnerability detection can be categorized into two complementary tasks: (i) *vulnerability prediction*, which determines *whether* a piece of code is vulnerable, and (ii) *CWE classification*, which identifies *how* the code is vulnerable by mapping it to a Common Weakness Enumeration (CWE). CWE identifiers provide an abstract description of vulnerability types, while specific instances of vulnerabilities are documented as Common Vulnerabilities and Exposures (CVEs). Our approach addresses both tasks by leveraging the structural and semantic information encapsulated in Abstract Syntax Trees (ASTs).

2.2 Motivating Example

Code vulnerabilities often arise from issues such as buffer overflows or improper memory management. Consider the example in Listing 1, which lacks null termination in its arrays and uses an insecure call to `strcpy()`. This omission risks injecting unexpected data, making the code vulnerable. For binary vulnerability prediction, this function would be labeled as 1 (vulnerable), and for CWE classification, it would be assigned CWE-170, "Improper Null Termination" [1]. Our methodology uses the AST of the source code to effectively represent and analyze such vulnerabilities, as described further in Section 2.3.

2.3 Abstract Syntax Tree Representation

The Abstract Syntax Tree (AST) is a hierarchical representation of a program's structure, where each node corresponds to a construct in the source code. ASTs are generated during the parsing phase of compilation and encode both syntactic and semantic relationships, such as variable declarations, control flow, and data dependencies. Internal nodes represent operators, while leaf nodes represent operands, and edges capture relationships such as loop conditions and variable assignments.

This structured representation is widely used in machine learning models for tasks like code classification [34, 35, 46]. In our approach, we enhance the standard AST by introducing augmented variants. Specifically, we replace selected nodes with subtrees from other functions sharing the same vulnerability label, rooted at matching node types (Section 3.1). Figure 1 shows part of the AST derived from Listing 1. This augmentation improves the model's ability to generalize across diverse code samples while maintaining the structural context essential for accurate vulnerability detection.

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <unistd.h>
4  #define MAXLEN 1024
5
6  int main() {
7      char *inputbuf;
8      char pathbuf[MAXLEN];
9      read(0, inputbuf, MAXLEN);
10     strcpy(pathbuf, inputbuf);
11     return 0;
12 }
```

Listing 1: Example Vulnerable Function. Vulnerable C code that lacks null termination, resulting in a risky `strcpy()` call.

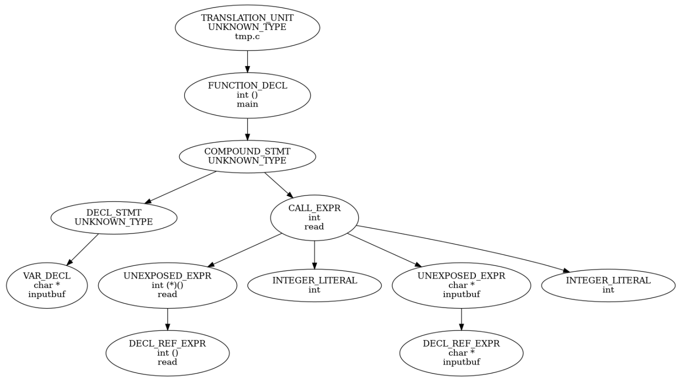


Figure 1: Example AST. AST representation of lines 6, 7, and 9 of Listing 1, parsed with Clang 14.0 and visualized with Graphviz.

3 APPROACH: K-ASTRO

We address the related tasks of vulnerability prediction (binary classification) and CWE classification from source code. Given a standalone source code function $f \in \mathcal{F}$ in C/C++, the goal is to reliably predict either the presence of a vulnerability or the corresponding Common Weakness Enumeration (CWE) exhibited in the function, without leveraging surrounding code context such as function callers or repository-level information.

To achieve this, we propose K-ASTRO, a lightweight yet powerful framework that addresses the limitations of existing approaches by combining structural and semantic information effectively. The

[1]https://cwe.mitre.org/data/definitions/170.html

framework integrates three core components that work in tandem to improve vulnerability detection accuracy:

First, Diversity-Introducing AST Augmentation enriches the structural representation of code by introducing controlled variations in the Abstract Syntax Tree (AST). This step ensures that the model is exposed to a diverse range of structural patterns during training, making it more robust to real-world scenarios where vulnerable patterns may vary significantly. By enhancing structural diversity, this component helps the model generalize better to unseen data.

Second, Structure-Aware Attention Bias leverages these augmented ASTs to encode structural patterns directly into the Transformer's attention mechanism. This step injects a deeper understanding of code structure into the model by highlighting the relationships and interactions between nodes in the AST. The integration of co-occurrence patterns allows the attention mechanism to focus on the most relevant structural features, reducing noise and improving the model's ability to identify vulnerabilities.

Finally, Joint LLM Adaptation brings everything together by combining the semantic information captured by pre-trained LLM embeddings with the structural insights derived from the augmented ASTs. This joint representation bridges the gap between the high-level semantic understanding of code and the low-level structural details, ensuring that the model benefits from both perspectives. Together, these components enable K-ASTRO to provide accurate and efficient vulnerability predictions.

3.1 Diversity-Introducing AST Augmentation

The first step in K-ASTRO is to address the inherent sparsity and diversity of vulnerability patterns in real-world codebases. Vulnerable code fragments often constitute a small fraction of large repositories, making it difficult for models to generalize effectively. To mitigate this issue, we introduce a novel AST augmentation technique inspired by mutation testing.

AST Generation and Subtree Extraction: For a given source code function f, the AST $T(f)$ is parsed using Clang. Subtrees are extracted recursively via depth-first search and stored in a catalog C_{AST}, organized by CWE classes c. This catalog serves as a repository of structural patterns for augmentation.

Augmentation via Subtree Replacement: To augment the AST, we randomly select a node v in $T(f)$ and replace the subtree rooted at v with another subtree s' from $C_{\text{AST}}(c)$, ensuring that the root types match. This process generates a structurally diverse augmented AST $T_{\text{aug}}(f)$, which is formally expressed as:

$$T_{\text{aug}}(f) = \mathcal{A}_{\text{aug}}(T(f), C_{\text{AST}}).$$

This augmentation introduces controlled variations that enhance the model's ability to learn robust representations of vulnerability patterns. While the augmented ASTs may not preserve the exact semantics of the original code, they provide a rich structural feature set that complements the model's learning process. Figure 3 illustrates the augmentation pipeline.

3.2 Structure-Aware Attention Bias

Building on the augmented ASTs, the second component of K-ASTRO focuses on integrating structural context into the model's attention mechanism. While traditional attention mechanisms treat all input tokens equally, our structure-aware attention bias prioritizes important structural relationships derived from the ASTs.

Co-Occurrence Matrix Construction: For each AST $T(f)$, we compute a co-occurrence matrix M that captures adjacency relationships between node types:

$$M_{ij} = \text{frequency}(t_i \to t_j), \quad \forall t_i, t_j \in T(f).$$

Logarithmic Binning and Bias Computation: To prevent the model from overfitting to specific patterns, we apply logarithmic binning to the co-occurrence values:

$$B_{ij} = \lceil \log_{10}(M_{ij} + 1) \rceil.$$

Bias Integration into Attention: For K AST variants (one original and $K - 1$ augmented), we compute a separate bias matrix B^k for each variant and combine them to form the final bias matrix:

$$B_{\max,ij} = \max(B^1_{ij}, B^2_{ij}, \dots, B^K_{ij}).$$

The final attention map is computed by adding this bias to the original attention map A:

$$A' = A + B_{\max}.$$

This structure-aware bias ensures that the attention mechanism focuses on the most relevant structural features, improving the model's ability to identify vulnerabilities. Figure 4 provides an overview of this mechanism.

3.3 Joint LLM Adaptation

The final component of K-ASTRO unifies the structural and semantic embeddings into a joint representation, ensuring that the model benefits from both detailed structural insights and high-level semantic understanding. By combining these complementary perspectives, K-ASTRO delivers robust and efficient predictions for both binary vulnerability detection and CWE classification.

The semantic embedding T is derived from a pre-trained LLM and encodes the high-level contextual meaning of the source code. It represents the function as a dense vector in $\mathbb{R}^d$, capturing linguistic and semantic nuances. In contrast, the structural embedding A', generated through the structure-aware attention mechanism, encodes hierarchical relationships and structural interactions within the AST. These two embeddings reflect different but equally important aspects of the source code, making their combination a powerful tool for vulnerability detection.

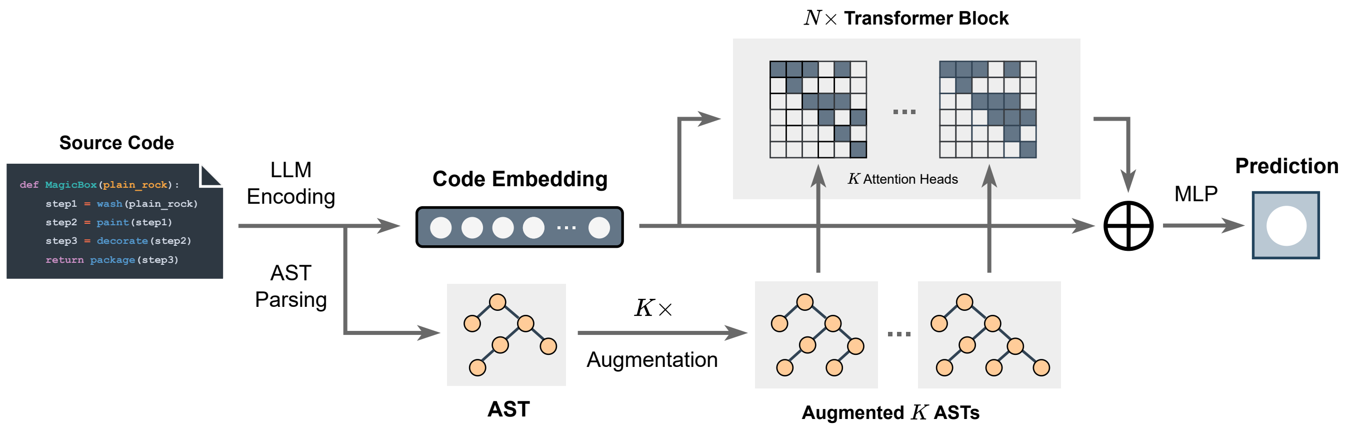
To integrate these embeddings, we concatenate T and A' to form a unified representation:

$$\text{Input to Classifier: } [A'; T].$$

This joint representation ensures that the model can simultaneously leverage high-level semantic context and low-level structural details. The combined embedding is passed through a ResNet-styled MLP classifier, where the concatenated vector undergoes transformation through learnable parameters:

$$L = \sigma(W \cdot [A'; T] + b),$$

where W is the weight matrix, b is the bias term, and σ is an activation function such as ReLU. This architecture ensures stable gradient flow during training and provides sufficient capacity for learning complex patterns inherent in vulnerabilities.



```
def MagicBox(plain_rock):
    step1 = wash(plain_rock)
    step2 = paint(step1)
    step3 = decorate(step2)
    return package(step3)
```

Figure 2: Overview of K-ASTRO. The framework processes source code into semantic embeddings via LLMs and structural embeddings via AST augmentation. Augmented ASTs provide structural insights through a structure-aware attention mechanism, which is combined with LLM embeddings for final vulnerability prediction using a single lightweight Transformer block.

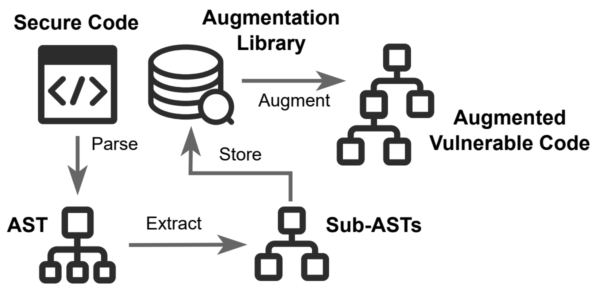


Figure 3: AST Augmentation Pipeline. The process involves AST generation, subtree extraction, and augmentation through node replacement to create structurally diverse representations for vulnerability detection.

By unifying the semantic and structural embeddings, K-ASTRO addresses both the broad contextual variability in codebases and the localized, nuanced patterns of vulnerabilities. This approach not only improves prediction accuracy but also maintains computational efficiency. The joint adaptation mechanism allows K-ASTRO to scale effectively to large datasets while remaining lightweight enough for practical deployment, demonstrating its utility in real-world software engineering scenarios.

4 EXPERIMENTS

In this section, we describe the datasets considered in our experiments and the data preparation process. The following four research questions (RQs) guide our investigation:

(1) **RQ1: Other LLMs vs. K-ASTRO** How well does K-ASTRO perform in comparison to prompting common off-the-shelf LLMs for vulnerability prediction and CWE classification, and in comparison to recent larger models fine-tuned on code?
(2) **RQ2: CWE-Specific Performance** What trends exist in the performance of K-ASTRO for specific CWE classes?
(3) **RQ3: Model Efficiency** Here we assess the training overhead of K-ASTRO by considering the number of parameters in the model, training time, and inference throughput on different datasets.
(4) **RQ4: Ablation Study** We compare K-ASTRO to 3 simpler models trained on the same CWE classification for C/C++ source code to justify the proposed K-ASTRO model.

4.1 Datasets and Data Summary

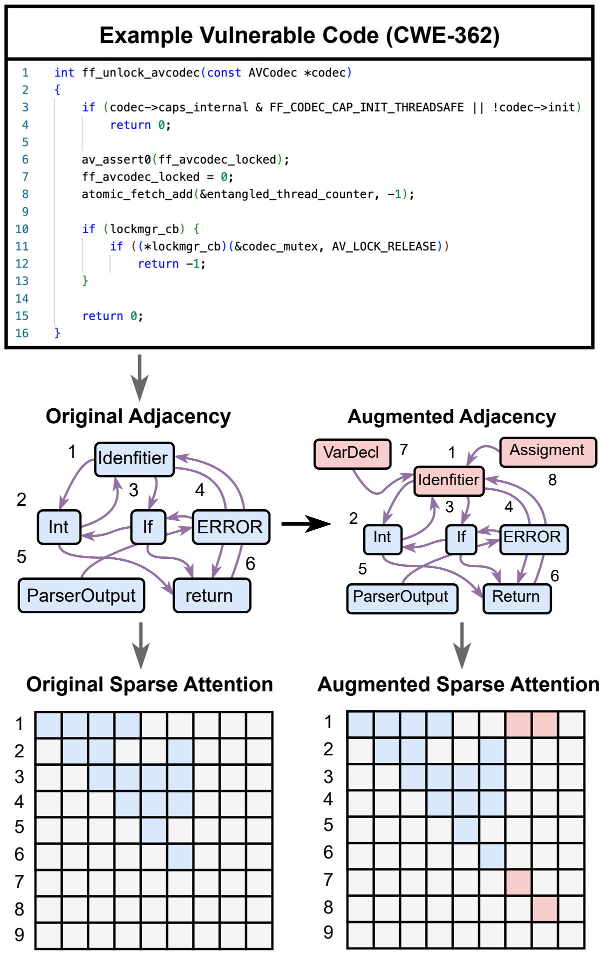
We focus on binary vulnerability classification and multi-class CWE classification of C/C++ source code functions, utilizing three large-scale datasets: BigVul [8], DiverseVul [4], and PrimeVul [6]. Table 1 provides a summary of the datasets, including train, validation, and test split sizes, as well as the distribution of vulnerable and non-vulnerable samples and the number of unique CWE classes.

Table 1: Summary of datasets used in this study.

Dataset	Train	Val	Test	Vuln	Not Vuln	CWEs
BigVul [8]	148,067	32,045	31,978	170,613	41,477	36
DiverseVul [4]	206,962	24,615	24,813	207,632	48,758	49
PrimeVul [6]	183,673	25,211	25,706	83,191	151,399	4

The datasets are selected to ensure diverse sources and high-quality labeling:

- **BigVul** is derived from CVE database crawls, featuring functions from 91 CWEs with an emphasis on code prior to bug-fixing commits.
- **DiverseVul** introduces a larger variety of CWEs with a more recent collection strategy, focusing on eliminating heuristic biases from commit messages.
- **PrimeVul** enhances dataset quality through rigorous deduplication and chronological data splitting, addressing data leakage issues inherent in prior datasets.



```
int ff_unlock_avcodec(const AVCodec *codec)
{
    if (codec->caps_internal & FF_CODEC_CAP_INIT_THREADSAFE || !codec->init)
        return 0;

    av_assert0(ff_avcodec_locked);
    ff_avcodec_locked = 0;
    atomic_fetch_add(&entangled_thread_counter, -1);

    if (lockmgr_cb) {
        if ((*lockmgr_cb)(&codec_mutex, AV_LOCK_RELEASE))
            return -1;
    }

    return 0;
}
```

Figure 4: Structure-Aware Attention Mechanism. The mechanism incorporates co-occurrence matrices from ASTs into the Transformer's attention map, enhancing its structural understanding.

4.2 Data Preparation

To ensure consistency and compatibility across the datasets used in this study, we followed a structured data preparation pipeline, which is outlined below.

Pre-Processing. We converted each dataset into Parquet files while preserving train, validation, and test splits. To enable consistent tracking, a unique identifier (UUID) was added to each function. Invalid CWE entries, such as empty strings or lists, were mapped to a "No CWE" class, and rows containing multiple CWEs were discarded. Source code comments were removed using the CodeTF ApexCodeUtility [3]. Token counts for each function were calculated using the `tiktoken`[2] library, ensuring compatibility with embedding models.

[2]https://pypi.org/project/tiktoken/

Embedding Collection. We utilized OpenAI's *text-embedding-ada-002*[3] and *text-embedding-3-small*[4] models to generate 1536-dimensional embeddings for each function. Functions exceeding the token limit of 8191 were excluded. This process produced approximately 600,000 embeddings across all splits, which were used as input representations for classification.

AST Collection and Augmentation. Abstract Syntax Trees (ASTs) for each function were extracted using Clang version 14.0 and stored in JSON format. Subtrees were recursively collected and categorized by CWE label and root node type. For augmentation, we randomly replaced AST nodes with subtrees that matched the function's original CWE label. This procedure generated $K = 4$ augmented samples per function, enriching the training data.

Adjacency Matrix Generation. After generating the augmented ASTs, we vectorize each AST by producing an adjacency matrix based on the node types present in the AST. We cap the maximum number of node types at 64 and observed 50 node types in practice from Clang. The dimensionality of this matrix directly affects the architectural parameters of the K-ASTRO model (Section 3). Each of these augmented matrices is incorporated into K-ASTRO using sparse attention mechanisms, as detailed in Section 3.2.

4.3 Implementation and Training Details

To train and evaluate K-ASTRO, we designed an efficient implementation strategy, ensuring reproducibility and scalability across datasets and experiments.

Model Architecture. K-ASTRO integrates an MLP and a Transformer layer in its joint adaptation module. The MLP has a hidden dimension of 512 and 3 layers, while the Transformer is a single-layer encoder with a hidden size of 64. These dimensions balance expressiveness and computational efficiency, enabling effective modeling of source code features and structural representations.

Training Configuration. The model was trained using the Adam optimizer [20], with a batch size of 32 and a learning rate of 1e−3. A total of 25 epochs were conducted, comprising 5 runs of 5 epochs each with distinct random seeds to ensure robust evaluation. To incorporate augmented AST adjacency matrices, we set $K = 4$, which balances the diversity of syntactic patterns introduced during training with computational efficiency.

Hardware and Efficiency. Training was performed on a single NVIDIA RTX A6000 GPU with 48GB of VRAM, while inference was tested on both GPU and Intel(R) Xeon(R) Gold 6330N CPUs. Training time ranged from 1 to 3 hours per experiment depending on dataset size, while inference over test sets, containing 24,000 to 32,000 samples, took approximately 10–30 seconds per dataset.

Scalability. Despite its small size of 1 million trainable parameters, K-ASTRO demonstrated competitive performance with minimal computational resources. The model's ability to train and infer efficiently makes it suitable for practical applications in resource-constrained environments.

4.4 Prompted LLM Performance

To evaluate the feasibility of using general-purpose LLMs for vulnerability classification tasks, we conducted experiments with three

[3]https://platform.openai.com/docs/guides/embeddings
[4]https://platform.openai.com/docs/api-reference/embeddings

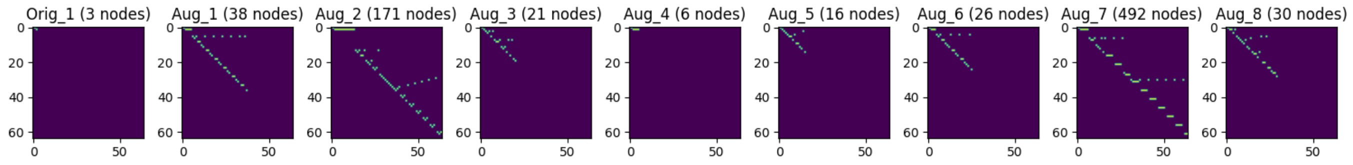


Figure 5: Visualization of Augmented ASTs. Adjacency matrices generated during the AST augmentation process, where subtrees from functions with corresponding vulnerability labels replace selected nodes in the original ASTs. The original matrix (left column) and $K = 8$ augmented matrices (right columns) depict node connectivity in the resulting ASTs. We consider a total of $n = 50$ node kinds, as observed in the datasets using Clang. These structures are incorporated into K-ASTRO via sparse attention to enhance vulnerability prediction performance (Section 3.2). We set $K = 4$ in our experiments to balance feature diversity and experimentation speed.

Table 2: Performance Comparison: Prompted LLMs vs. K-ASTRO Metrics for CWE classification and binary vulnerability prediction.

Dataset	Metric	GPT-3.5	GPT-4o	K-ASTRO
BigVul	F1	7.19	9.83	**76.31**
	Precision	8.76	10.51	**76.92**
	Recall	19.56	18.70	**76.47**
BigVul-bin	F1	7.38	9.39	**47.19**
	Precision	6.05	5.25	**59.17**
	Recall	9.44	**44.26**	39.24

popular models: GPT-3.5, GPT-4o, and Claude 3 Haiku. These evaluations focus on assessing their performance on CWE classification and binary vulnerability prediction, comparing them against the specialized model, K-ASTRO.

Evaluation Setup. We used programmatic APIs to query these models for predictions, using the BigVul dataset's test set as input. For each sample, we requested outputs in JSON format: `is_vuln` (binary label indicating vulnerability), `cwe_label` (CWE class prediction), and `reasoning` (a brief explanation in up to 100 words). Chain-of-thought prompting techniques [38] were adopted to refine the prompt, incorporating explicit format instructions, a synopsis of vulnerabilities, and examples of CWE categories. Due to malformed JSON outputs from Claude (~72% of responses), we report complete results only for GPT-3.5 and GPT-4o.

Key Findings. K-ASTRO significantly outperforms GPT-3.5 and GPT-4o across all metrics. On BigVul, K-ASTRO achieves a weighted F1 score of 76.31 compared to 7.19 (GPT-3.5) and 9.83 (GPT-4o). Similarly, for binary vulnerability classification (BigVul-bin), K-ASTRO achieves an F1 of 47.19, surpassing the performance of GPT-3.5 (7.38) and GPT-4o (9.39). These results underscore the limitations of generic LLMs in vulnerability classification tasks.

Limitations of LLMs. While the LLMs provide reasoning for their predictions, their inability to handle complex CWE-specific classifications highlights the necessity for specialized models like K-ASTRO. Detailed reasoning analysis for LLM predictions remains an avenue for future exploration.

Table 3: K-ASTRO Model Performance. Weighted metrics across datasets. BigVul: 36 classes, BigVul-bin: 2 classes, DiverseVul: 49 classes, PrimeVul: 4 classes. PrimeVul's test set contains only ≈3% vulnerable functions, hence binary results are omitted.

Dataset	F1	Precision	Recall	Accuracy
BigVul [8]	76.31	76.92	76.47	76.47
BigVul-bin	47.19	59.17	39.24	97.28
DiverseVul [4]	69.12	69.64	69.14	69.14
PrimeVul [6]	92.33	90.99	93.78	93.78

4.5 RQ1: Other LLMs vs. K-ASTRO

Objective. This research question evaluates how K-ASTRO performs in comparison to state-of-the-art LLMs (e.g., GPT-3.5 and GPT-4o) on CWE classification and binary vulnerability prediction tasks across multiple datasets.

Model Variants. K-ASTRO was tested with two configuration options:

- **with-mask**: Enforces masking in the learned attention matrix, ensuring the model focuses on connected nodes in augmented ASTs. After experimentation, we fixed **with-mask** to True.
- **embedding-type**: Two embedding models were evaluated, OpenAI's [5] ("small") and [6] ("ada"). The "small" embedding model demonstrated superior performance and was selected for all experiments.

Findings. K-ASTRO consistently outperforms GPT-3.5 and GPT-4o on all datasets (Table 3). For instance:

- On BigVul, GPT-3.5 achieves a weighted F1 of 7.19 for CWE classification, and GPT-4o scores 9.83, while K-ASTRO achieves 76.31.
- On BigVul-bin, K-ASTRO achieves a weighted F1 of 47.19 compared to 7.38 (GPT-3.5) and 9.39 (GPT-4o).

Comparison with Larger Models. Despite having only 1M parameters, K-ASTRO achieves competitive results compared to significantly larger models like GraphCodeBERT [14] (125M), PolyCoder [40] (2.7B), and T5 [37] (220M), none of which exceed 50%

[5] `text-embedding-3-small`
[6] `text-embedding-ada-002`

F1 on the DiverseVul dataset. This suggests that specialized small models can outperform general-purpose large models for domain-specific tasks.

4.6 RQ2: CWE-Specific Performance

The results from K-ASTRO inference on the test sets of BigVul and DiverseVul provide a comprehensive evaluation of its ability to handle diverse CWE classes. To assess its performance, we conducted experiments focusing on class-specific metrics, including Precision, Recall, and F1 scores, averaged across individual classes for each dataset.

Experimental Setup. For this evaluation, K-ASTRO was tasked with classifying functions based on their associated CWE labels in the BigVul and DiverseVul test sets. These datasets include a mix of specific CWEs and broader categories, offering a diverse challenge. By analyzing the class-wise metrics, we aimed to identify the strengths and weaknesses of the model across different types of CWEs.

Performance Consistency on BigVul. On BigVul, K-ASTRO exhibits consistent performance across the 36 CWE classes, achieving an average F1 score of approximately 0.8. The model performs particularly well on CWE-269: "Improper Privilege Management" and CWE-704: "Incorrect Type Conversion or Cast," achieving F1 scores above 0.9. These results highlight the model's capability to identify specific and well-defined vulnerabilities. However, the model struggles with broader categories like CWE-388: "7PK - Errors," which aggregate multiple specific CWEs, making classification inherently more challenging. This discrepancy is likely due to the lack of granularity in such categories, which can obscure the patterns needed for accurate classification.

Variability in DiverseVul Results. In contrast, results on DiverseVul exhibit greater variability across its 49 CWE classes. CWE-212: "Improper Removal of Sensitive Information Before Storage or Transfer" achieves an almost perfect F1 score, demonstrating the model's ability to handle specific and clearly defined vulnerabilities. Other strong performers include CWE-191: "Integer Underflow" and CWE-613: "Insufficient Session Expiration." However, some classes, such as CWE-122: "Heap-based Buffer Overflow" and CWE-19: "Data Processing Errors," exhibit lower F1 scores, highlighting areas where the model's performance is limited. Notably, many of these underperforming classes are categorized as general groupings, which further complicates classification.

4.7 RQ3: Model Efficiency

We assess K-ASTRO's efficiency by evaluating its inference performance across different datasets using GPU and CPU setups. Despite its lightweight architecture, K-ASTRO demonstrates excellent scalability and speed, making it well-suited for processing large datasets.

K-ASTRO is a compact Transformer model tailored for binary vulnerability prediction and CWE classification. It includes a single encoder layer with multi-head attention that integrates K augmented AST interaction matrices. With approximately 1 million trainable parameters, K-ASTRO occupies only ≈4MB of disk space when trained, showcasing remarkable storage efficiency.

In our experiments, K-ASTRO was trained for 5 rounds of 5 epochs, each with distinct random seeds for reproducibility. Training completes within 1-3 hours, depending on dataset size, and inference takes just seconds per full pass. This lightweight design ensures minimal overhead while maintaining high performance.

Table 4 summarizes the evaluation metrics across datasets. On the GPU, K-ASTRO achieves an impressive throughput of 5,434 samples per second for BigVul-bin, while on the CPU, it processes up to 12,989 samples per second for the same dataset. Across all datasets, throughput consistently exceeds 1,500 samples per second on both GPU and CPU setups.

Compact and Efficient Design. These results highlight K-ASTRO's suitability for practical deployment in resource-constrained environments. Its compact size and rapid inference capabilities make it an ideal choice for real-world applications requiring efficient and accurate vulnerability detection.

5 RELATED WORK

This section reviews previous studies in three primary areas: ML-driven vulnerability detection, the application of LLMs in security, and strategies for LLM adaptation to specific tasks.

5.1 ML-Driven Vulnerability Detection

Machine learning approaches have become increasingly desirable for vulnerability detection as they require less manual effort compared to pattern-based techniques like FlawFinder [10] and ITS4 [33]. Early ML-based methods, such as VulDeePecker [23], utilized Bi-LSTMs with word2vec encodings of code gadgets, demonstrating significant improvements over traditional rule-based systems.

Abstract syntax tree (AST)-based methods have also gained traction. [24] explored AST serialization combined with Bi-LSTMs for function-level vulnerability detection. Building on these, [26] employed neural sub-tree encodings to capture fine-grained syntactic features, while [41] extended these approaches for generalized vulnerability extrapolation using AST-based representations.

Recently, transformer-based models have dominated vulnerability detection tasks. [30] demonstrated that fine-tuned models like CodeBERT [9] and GraphCodeBERT [14] significantly outperform Bi-LSTMs. Further, [19] emphasized learning from syntax-based execution paths to enhance detection performance. Despite these advances, existing methods often fall short in leveraging augmented AST structures to address generalization and interpretability, a gap addressed by K-ASTRO.

5.2 LLMs in Computer Security

Large Language Models (LLMs) have seen growing use in security applications, including vulnerability repair [1, 28], CWE mapping [25], and policy analysis [27]. Despite their promise, off-the-shelf LLMs like GPT-3.5 and GPT-4o have shown limited success in vulnerability detection [5, 11].

Recent frameworks have sought to improve LLM-driven detection through tailored strategies. For instance, [42] introduced deep-learning-augmented prompting frameworks, while [26] utilized vulnerability-preserving data augmentation to enrich training data. However, challenges remain in adapting these general-purpose

Table 4: K-ASTRO Inference Efficiency. Inference throughput and evaluation time for different datasets on GPU and CPU. The trained model is compact (≈4MB) and achieves competitive inference speeds, processing thousands of samples per second.

Dataset	# Samples	Eval Time GPU (s)	Eval Time CPU (s)	GPU Throughput (samples/s)	CPU Throughput (samples/s)
BigVul	31,894	20.52	20.64	1,554	1,545
BigVul-bin	31,953	5.88	2.46	5,434	12,989
DiverseVul	24,601	14.65	17.07	1,679	1,441
PrimeVul	24,990	15.03	15.91	1,662	1,570

models for domain-specific tasks. Our work addresses this by integrating AST-based structural insights with lightweight fine-tuning techniques, enabling robust performance even with limited computational resources.

5.3 LLM Adaptation

LLM adaptation focuses on resource-efficient strategies for task-specific fine-tuning. Techniques like prompt tuning [21] and prefix tuning [22] enable parameter-efficient updates by optimizing input embeddings. Similarly, LoRA [16] employs low-rank approximations to reduce the number of trainable parameters.

Adapter-based methods, such as [15] and [17], introduce small auxiliary modules between transformer layers to achieve efficient fine-tuning. These methods have been widely adopted for various NLP and security tasks. In contrast, our approach avoids auxiliary modules, opting instead for a single lightweight transformer block with multi-head attention to integrate AST-derived biases into the final prediction.

By combining text embeddings with augmented AST structures, K-ASTRO achieves strong performance on vulnerability detection tasks while maintaining computational efficiency. This unique adaptation mechanism sets our approach apart from existing parameter-efficient methods and bridges the gap between LLM generalization and domain-specific performance.

6 CONCLUSION

We present K-ASTRO, a lightweight Transformer model tailored for few-shot vulnerability detection in C/C++ source code. By integrating AST-based data augmentation and sparse attention with text embeddings, K-ASTRO harmonizes structural and semantic features to enhance vulnerability detection and CWE classification. Unlike off-the-shelf LLMs, which struggle with this task due to biases towards non-vulnerable code, K-ASTRO leverages its compact architecture to achieve competitive performance with state-of-the-art models while maintaining efficiency and simplicity.

With ≈1M parameters occupying only 4MB on disk, K-ASTRO trains within hours and performs rapid inference on CPUs, enabling secure, local processing of sensitive code without reliance on resource-intensive GPU servers. Our results demonstrate that K-ASTRO matches or exceeds the performance of larger industrial LLMs, underscoring the value of combining structural and semantic insights for vulnerability detection. To encourage further research, we release all software artifacts to the community.

6.1 Limitations

K-ASTRO focuses exclusively on C/C++ source code, chosen for its prevalence, available data, and the complexity of vulnerabilities in these languages. While the approach is conceptually applicable to other programming languages, dataset limitations and embedding constraints shaped our experiments. Specifically, our input length is limited by the token capacity of embedding models, resulting in the omission of a small subset of functions. Moreover, we do not explore variations in embedding dimensionality or broader dataset refinements, which could provide opportunities for future work. Despite these constraints, K-ASTRO demonstrates robust performance and offers a promising foundation for lightweight vulnerability prediction.

REFERENCES

[1] Baleegh Ahmad, Shailja Thakur, Benjamin Tan, Ramesh Karri, and Hammond Pearce. 2023. Fixing Hardware Security Bugs with Large Language Models. *arXiv preprint arXiv:2302.01215* (2023). https://arxiv.org/pdf/2302.01215.pdf

[2] Paul Ammann and Jeff Offutt. 2016. *Introduction to software testing*. Cambridge University Press.

[3] Nghi D. Q. Bui, Henry Le, Yue Wang, Akhilesh Deepak Gotmare, Junnan Li, and Steven Hoi. 2023. CodeTF: A Transformer-based Library for CodeLLM and Code Intelligence. arXiv:2209.09019 [cs.CV]

[4] 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. *arXiv preprint arXiv:2304.00409* (2023). https://arxiv.org/pdf/2304.00409.pdf

[5] Anton Cheshkov, Pavel Zadorozhny, and Rodion Levichev. 2023. Evaluation of ChatGPT Model for Vulnerability Detection. *arXiv preprint arXiv:2304.07232* (2023). https://arxiv.org/pdf/2304.07232.pdf

[6] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. 2024. Vulnerability Detection with Code Language Models: How Far Are We? arXiv:2403.18624 [cs.SE] https://arxiv.org/abs/2403.18624

[7] Yangruibo Ding, Ben Steenhoek, Kexin Pei, Gail Kaiser, Wei Le, and Baishakhi Ray. 2023. TRACED: Execution-aware Pre-training for Source Code. *arXiv preprint arXiv:2306.07487* (2023). https://arxiv.org/pdf/2306.07487.pdf

[8] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories* (Seoul, Republic of Korea) *(MSR '20)*. Association for Computing Machinery, New York, NY, USA, 508–512. https://doi.org/10.1145/3379597.3387501

[9] 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 *Findings of the Association for Computational Linguistics: EMNLP 2020*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, Online, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[10] Oliver Ferschke, Iryna Gurevych, and Marc Rittberger. 2012. FlawFinder: A Modular System for Predicting Quality Flaws in Wikipedia.. In *CLEF (Online Working Notes/Labs/Workshop)*. 1–10.

[11] Chakkrit (Kla) Fu, Michael Tantithamthavorn, Van Nguyen, and Trung Le. 2023. ChatGPT for Vulnerability Detection, Classification, and Repair: How Far Are We? *arXiv preprint arXiv:2310.09810* (2023). https://arxiv.org/pdf/2310.09810.pdf

[12] Tiferet Gazit. 2024. Fixing security vulnerabilities with AI. https://github.blog/2024-02-14-fixing-security-vulnerabilities-with-ai/

[13] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics*

(Volume 1: Long Papers), Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (Eds.). Association for Computational Linguistics, Dublin, Ireland, 7212–7225. https://doi.org/10.18653/v1/2022.acl-long.499

[14] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. arXiv:2009.08366 [cs.SE] https://arxiv.org/abs/2009.08366

[15] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin de Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-Efficient Transfer Learning for NLP. In *Proceedings of the 36th International Conference on Machine Learning (ICML)*. http://proceedings.mlr.press/v97/houlsby19a/houlsby19a.pdf

[16] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-Rank Adaptation of Large Language Models. In *Proceedings of the International Conference on Learning Representations*. https://openreview.net/forum?id=nZeVKeeFYf9

[17] Zhiqiang Hu, Lei Wang, Yihuai Lan, Wanyu Xu, Ee-Peng Lim, Lidong Bing, Xing Xu, Soujanya Poria, and Roy Ka-Wei Lee. 2023. LLM-Adapters: An Adapter Family for Parameter-Efficient Fine-Tuning of Large Language Models. *arXiv preprint arXiv:2304.01933* (2023). https://arxiv.org/pdf/2304.01933.pdf

[18] Ehsan Kamalloo, Nouha Dziri, Charles Clarke, and Davood Rafiei. 2023. Evaluating Open-Domain Question Answering in the Era of Large Language Models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (Eds.). Association for Computational Linguistics, Toronto, Canada, 5591–5606. https://doi.org/10.18653/v1/2023.acl-long.307

[19] Dongwoo Kim and Jaewoo Choi. 2022. Enhancing Vulnerability Detection by Learning from Syntax-Based Execution Paths of Code. In *Proc. of ICSE*.

[20] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014). https://arxiv.org/pdf/1412.6980.pdf

[21] Brian Lester, Rami Al-Rfou, and Noah Constant. 2021. The Power of Scale for Parameter-Efficient Prompt Tuning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 3045–3059. https://doi.org/10.18653/v1/2021.emnlp-main.243

[22] Xiang Lisa Li and Percy Liang. 2021. Prefix-Tuning: Optimizing Continuous Prompts for Generation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli (Eds.). Association for Computational Linguistics, Online, 4582–4597. https://doi.org/10.18653/v1/2021.acl-long.353

[23] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In *Proceedings 2018 Network and Distributed System Security Symposium (NDSS 2018)*. Internet Society. https://doi.org/10.14722/ndss.2018.23158

[24] Guanjun Lin, Jun Zhang, Wei Luo, Lei Pan, and Yang Xiang. 2017. Vulnerability Discovery with Function Representation Learning from Unlabeled Projects. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2539–2541. https://doi.org/10.1145/3133956.3138840

[25] Xin Liu, Yuan Tan, Zhenghang Xiao, Jianwei Zhuge, and Rui Zhou. 2023. Not The End of Story: An Evaluation of ChatGPT-Driven Vulnerability Description Mappings. In *Findings of the Association for Computational Linguistics: ACL 2023*, Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (Eds.). Association for Computational Linguistics, Toronto, Canada, 3724–3731. https://doi.org/10.18653/v1/2023.findings-acl.229

[26] Anil Mishra and Supriya Gupta. 2023. Enhancing Code Vulnerability Detection via Vulnerability-Preserving Data Augmentation. In *Proc. of ICML*.

[27] Sudipta Paria, Aritra Dasgupta, and Swarup Bhunia. 2023. DIVAS: An LLM-based End-to-End Framework for SoC Security Analysis and Policy-based Protection. *arXiv preprint arXiv:2308.06932* (2023). https://arxiv.org/pdf/2308.06932.pdf

[28] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining Zero-Shot Vulnerability Repair with Large Language Models. In *Proceedings of the 2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 2339–2356. https://doi.org/10.1109/SP46215.2023.10179420

[29] Moumita Das Purba, Arpita Ghosh, Benjamin J. Radford, and Bill Chu. 2023. Software Vulnerability Detection using Large Language Models. In *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 112–119. https://doi.org/10.1109/ISSREW60843.2023.00058

[30] Chandra Thapa, Seung Ick Jang, Muhammad Ejaz Ahmed, Seyit Camtepe, Josef Pieprzyk, and Surya Nepal. 2022. Transformer-Based Language Models for Software Vulnerability Detection. In *Proceedings of the 38th Annual Computer Security Applications Conference* (<conf-loc>, <city>Austin</city>, <state>TX</state>, <country>USA</country>, </conf-loc>) *(ACSAC '22)*. Association for Computing Machinery, New York, NY, USA, 481–496. https://doi.org/10.1145/3564625.3567985

[31] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. LlaMA: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023). https://arxiv.org/pdf/2302.13971.pdf

[32] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[33] John Viega, J.T. Bloch, Yoshi Kohno, and Gary McGraw. 2000. ITS4: a static vulnerability scanner for C and C++ code. In *Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00)*. 257–267. https://doi.org/10.1109/ACSAC.2000.898880

[34] Kesu Wang, Meng Yan, He Zhang, and Haibo Hu. 2022. Unified abstract syntax tree representation learning for cross-language program classification. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension* (Virtual Event) *(ICPC '22)*. Association for Computing Machinery, New York, NY, USA, 390–400. https://doi.org/10.1145/3524610.3527915

[35] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 261–271. https://doi.org/10.1109/SANER48275.2020.9054857

[36] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. https://aclanthology.org/2021.emnlp-main.685.pdf

[37] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. arXiv:2109.00859 [cs.CL] https://arxiv.org/abs/2109.00859

[38] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv:2201.11903 [cs.CL] https://arxiv.org/abs/2201.11903

[39] Yan Wu, Jingyi Su, David D. Moran, and Chris D. Near. 2023. Automated Software Testing Starting from Static Analysis: Current State of the Art. *arXiv preprint arXiv:2301.06215* (2023). https://arxiv.org/ftp/arxiv/papers/2301/2301.06215.pdf

[40] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent J. Hellendoorn. 2022. A Systematic Evaluation of Large Language Models of Code. arXiv:2202.13169 [cs.PL] https://arxiv.org/abs/2202.13169

[41] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. 2012. Generalized vulnerability extrapolation using abstract syntax trees. In *Proceedings of the 28th Annual Computer Security Applications Conference* (Orlando, Florida, USA) *(ACSAC '12)*. Association for Computing Machinery, New York, NY, USA, 359–368. https://doi.org/10.1145/2420950.2421003

[42] Kaizhi Yu and Yixuan Chen. 2023. DLAP: A Deep Learning Augmented Prompting Framework for Vulnerability Detection. In *Proc. of NeurIPS*.

[43] Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Wang Yongji, and Jian-Guang Lou. 2023. Large Language Models Meet NL2Code: A Survey. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (ACL)*, Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (Eds.). https://doi.org/10.18653/v1/2023.acl-long.411

[44] Haopeng Zhang, Xiao Liu, and Jiawei Zhang. 2023. Extractive Summarization via ChatGPT for Faithful Summary Generation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 3270–3278. https://aclanthology.org/2023.findings-emnlp.214

[45] Jieyu Zhang, Ranjay Krishna, Ahmed H Awadallah, and Chi Wang. 2023. EcoAssistant: Using LLM Assistant More Affordably and Accurately. *arXiv preprint arXiv:2310.03046* (2023). https://arxiv.org/pdf/2310.03046.pdf

[46] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A Novel Neural Source Code Representation Based on Abstract Syntax Tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 783–794. https://doi.org/10.1109/ICSE.2019.00086

[47] Jiang Zheng, Laurie Williams, Nachiappan Nagappan, Will Snipes, John P Hudepohl, and Mladen A Vouk. 2006. On the value of static analysis for fault detection in software. *IEEE transactions on software engineering* 32, 4 (2006), 240–253. https://collaboration.csc.ncsu.edu/laurie/Papers/TSE-0197-0705-2.pdf

[48] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: a survey for roadmap. *ACM Computing Surveys (CSUR)* 54, 11s (2022), 1–36. https://dl.acm.org/doi/abs/10.1145/3512345