Improving Software Security Through an LLM-Based Vulnerability Detection Model

Mohsin Sami, Kashif Nasr, Saira Andleeb Gillani, Rabia Tehseen
Department of Computer Science, University of Central Punjab, Lahore, Pakistan
*Correspondence: rabia.tehseen@ucp.edu.pk

Citation | Saim. M, Nasr. K, Gillani. S. A, Tehseen. R, "Improving Software Security Through LLM-Based Vulnerability Detection Model", IJIST, Vol. 07, Issue. 04 pp 2592-2603, October 2025

Received | August 29, 2025 **Revised |** September 23, 2025 **Accepted |** September 25, 2025 **Published |** October 28, 2025.

The risks to modern digital infrastructures posed by software vulnerabilities are critical and include data breaches, unauthorized access, and losses in revenue. Although traditional static and dynamic analysis tools are effective in discovering vulnerability patterns, they are not able to recognize complex, context-dependent, logic-based, and security-embedded flaws that evolve within software systems. This research offers a Large Language Model-based Vulnerability Detection Model (LLM-VDM) focused on enhancing software security with intelligent, context-aware code analysis. Leveraging transformer-based architecture adapted to the Juliet, Big-Vul, and Devign benchmark datasets to assess the performance and integration of code semantic and code contextualization methods, the proposed model was evaluated. Experimental results demonstrated LLM-VDM's superiority to both baseline and deep learning competitors SonarQube, Devign, CodeBERT, and CodeT5, attaining 91.2% accuracy, 90.0% F1-score, and 0.94 AUC. Furthermore, the integrated explainability module improves explainability by pinpointing vulnerable code and outlining remediation strategies. The findings showed LLM-based technology provides software developers with more secure, adaptive, explainable, and scalable systems, meeting the needs of contemporary software development.

Keywords: Software Security, Large Language Models, Vulnerability Detection, Transformer Networks, Code Analysis, Explainable AI, Cybersecurity, Deep Learning, Static Analysis, Big-Vul Dataset.

Introduction:

As software systems become dominant in the healthcare and finance industries, as well as the transportation and defense sectors, software security has become an indispensable aspect of computing. Protection of software against potential attacks has reached an advanced stage. With the quickly evolving digital and software ecosystems, addressing security issues has become even more complex. Uncontrolled access, significant damage, data loss, and even complete system outages can result from unprotected software code. This has made early identification and vulnerability mitigation sequence critical in the software development lifecycle. This has made early identification and mitigation of potential software design vulnerabilities an unresolved sequence critical in the software development lifecycle [1]. Protection of software digital assets and system integrity has become vital to organizations and researchers alike.

Dynamic and static code analysis, as well as other traditional code, serve as the basis for software development security. Unfortunately, the rule of code systems that analyze and assess software security fails to address complex logical code gaps and context-sensitive issues. Growing LLMs and other AI systems present new opportunities in software threat detection, as LLMs gain new capabilities and provide more advanced programming and natural language software data understanding, LLMs semantic meaning, programming context, and dev- eloper intent—includes elements that allow them to effectively examine raw code and identify security issues that other approaches may miss [2].

Most LLMs implemented concerning software security are general-purpose and do not focus on vulnerability detection. They have been particularly challenged in differentiating between harmless code constructs and real security threats, which creates a problem of excessive false positives and excessive false negatives. Additionally, new security vulnerabilities arise in software ecosystems and are left unaddressed by legacy security tools. This results in modern security threats that are sophisticated and poorly countered by legacy detection systems. This research has been motivated by a specialized LLM-based model that automates detection, classification, and explanation of software vulnerabilities to the extent of lifting human effort and the probable human error [3].

This work presented an LLM-Based Vulnerability Detection Model that unites the contextual reasoning capabilities of large language models with domain-specific fine-tuning on purpose-built security datasets. The target of the newly designed model is to amplify precision, lower the false detection rate, and offer coherent justifications for the identified vulnerabilities. In contrast to traditional rule-based analyzers, it identifies patterns from both vulnerable code and secure code, which detects both known and unknown threats. This research integrates natural language understanding with code intelligence to create a scalable solution that progresses the use of AI in cybersecurity.

The remainder of this paper has been organized as follows. Section II presents the related work of the proposed method. The proposed Methodology is shown in Section III, while the results are discussed in Section IV. Conclusions are finally presented in Section V.

Literature Review:

Classical Software Security Analysis: Static, Dynamic, and Fuzzing:

Foundational work in software assurance bolstered the initial building blocks of the proposed model. Static analysis systems managed to mine code for bugs by tracking application code and system code for rule and idiom deviations [4][5]. Security-based static analysis developed advanced flow and context-sensitive taint tracking and uncovered injection and XSS attack vectors in web applications [6]. Simultaneously, the dynamic techniques, which included symbolic execution (e.g., KLEE) that revealed real bugs in complex C programs by generating high-coverage tests, and fuzzing techniques (e.g., AFL), which, along with OSS-Fuzz, found thousands of bugs in open-source software [7][8][9] revealed). These techniques

provide strong baseline systems, but context-dependent logic flaws might still be missed; manual rule engineering, along with harnesses and sanitizers, may still be needed.

Deep Learning for Vulnerability Detection:

Driven by the shortcomings of hand-crafted rules, the focus of research turned to the use of deep learning models that work directly on source code. VulDeePecker utilized 'code gadgets' along with a neural detector and demonstrated that learned patterns of vulnerabilities could surpass the performance of traditional, highly engineered systems [10]. Draper's VDISC project designed a function-level dataset of considerable size that was labeled using static analyzers to train rapid neural detectors [11] Graph-based models (Devign) learned program semantics to effectively classify vulnerable functions using multiple code graphs [12]. Later enhancements of the dataset to real CVEs (Big-Vul) focused on sharpening the representation learning frameworks for code defects and achieving further improvements in accuracy across different languages and projects [13]. High false-positive rates, limited interpretability, and challenges with cross-project generalization were still persistent.

Pretrained Code Models and Large Language Models (LLMs):

Pre-trained models at a large scale transformed code intelligence. CodeBERT (bimodal NL-PL) and CodeT5 (identifier-aware encoder-decoder) offered effective general representations that were critical for transfer to downstream tasks such as defect detection [14][15]. The construction of tasks and evaluation metrics for benchmarks such as CodeXGLUE made evaluation and reproducible research easy, with integrated systems and cross-task comparisons [16]. While Codex illustrated advanced code synthesis capabilities, research confirmed significant security concerns in LLM-generated code: Under security-related prompts, Copilot consistently provided insecure suggestions, and follow-up studies validated these weaknesses [17][2][18]. Efforts like VulBERTa and more recent work on code LMs regarding pretraining for vulnerabilities have documented some progress, but the newest work also emphasizes critical gaps around generalization and robustness [19][20]. These issues create the need for purpose-built, security-tuned LLM pipelines that incorporate explainability and automated safeguards.

Datasets, Test Suites, and Evaluation Practices:

For reliable training and fair evaluation, a corpus that is diverse and of high quality is necessary. In addition to the research datasets (VDISC, Devign, Big-Vul), the NIST SARD/Juliet suites also contain labeled, CWE-mapped synthetic cases across C/C++/Java/C# that include structural components for systematic testing, negative controls, and regression [21]. These, along with community benchmarks (CodeXGLUE) and large curated sets emerging around real CVEs, provide the components for stratified evaluation based on weakness type, project, and language. Remaining pain points include label noise, data leakage, and limited coverage of modern frameworks—issues that proposed work might address through security-oriented fine-tuning and thorough leakage-aware splits.

Gaps and Direction of Contribution:

The previous work indicated strong baselines from static, dynamic, and fuzzing analysis; encouraging results from neural and graph approaches; and powerful albeit precarious general Large Language Models (LLMs) as presented in Table 1. Cross-repository generalization, actionable explanations, and noiseless real-world performance against a dynamic weakness's taxonomy remained to be solved. Recent literature underscored hybrid approaches that integrate program analysis signals and curated security datasets with LLM reasoning (systematic reviews, 2024). The proposed model seeks to bridge these gaps by aligning the LLM security objective representations, integrating security code context graphs/flows with learned semantics, and evaluating to improve recall and reduce false positives on real-world (Big-Vul) and synthetic (Juliet) datasets across several previously unseen projects.

Table1. Literature Review

No.	Technique Applied	Dataset Used	Strengths	Weaknesses
1	Static analysis via pattern deviation detection	Linux & BSD kernel code	Early scalable bug inference; automatic error pattern discovery	High false positives; limited to system-level code
2	Static analysis with lightweight dataflow	Java open-source projects	Efficient bug scanning; integrated into IDEs	Shallow semantic understanding
3	Taint analysis for data flow tracking	Java enterprise apps	Accurate injection detection; rule-based	Requires manual rule tuning
4	Symbolic execution	Coreutils, GNU tools	High coverage; finds deep bugs	Path explosion; scalability issues
5	Coverage-guided fuzzing	Open-source projects	Scalable automation; finds real-world CVEs	Requires binaries; limited logic flaw detection
6	Deep learning on code gadgets	NVD & CVE datasets	Learns vulnerability features automatically	Limited context understanding
7	CNN-based code representation learning	VDISC dataset	Works on raw source code; language-agnostic	Lacks explainability
8	Graph Neural Networks (GNN)	Devign dataset	Captures data/control flow context	Requires graph extraction; high computation cost
9	Data-driven vulnerability mining	1M+ C/C++ functions with CVEs	Large-scale real data; reproducible	Class imbalance; noisy labels
10	Transformer-based pretrained model	CodeSearchNet, GitHub code	Learns joint NL–code representation	Needs task-specific fine-tuning
11	Seq2Seq transformer (encoder–decoder)	CodeSearchNet	Strong multi-language generalization	Vulnerability-specific performance not tested
12	Benchmark suite	14 code intelligence datasets	Standardized evaluation	Not focused on vulnerabilities
13	Large Language Model (GPT-style)	159 GB GitHub code	Powerful code generation	Produces insecure code; lacks security training
14	Empirical security evaluation of Copilot	Real-world prompts (C, Python, JS)	First large-scale LLM security audit	~40% insecure code output
15	Fine-tuned BERT on code vulnerabilities	Big-Vul, Juliet	Simpler pretraining for defects	Limited to token-level cues
16	Replication & risk categorization	OWASP benchmark code	Confirms insecure pattern reproduction	Lack of mitigation strategies
17	Statistical evaluation of fuzzers	FuzzBench suite	Reveals fuzzing performance trends	Non-semantic vulnerabilities ignored
18	Survey on LLM-based security detection	Multiple datasets reviewed	Identifies LLM research gaps	No empirical validation
19	Fine-tuned transformer for vulnerabilities	Big-Vul + Juliet	Improved precision; contextual reasoning	Needs a larger domain corpora

20	Systematic review	60+ prior studies	Comprehensive taxonomy & trends	Lacks unified evaluation metrics

Methodology:

In line with the previously mentioned systematic reviews, the proposed LLM-based vulnerability detection framework automatically identifies, classifies, and explains potential software vulnerabilities embedded within the source code. The model merged the contextual understanding and semantic reasoning capabilities of contemporary transformer architectures with security domain knowledge acquired through fine-tuning on curated vulnerability datasets. Figure 1 presents the stages of the proposed model.

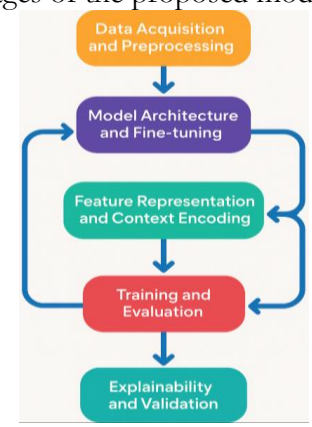

Figure 1. Stages of the proposed model

Dataset Description:

To ensure a balanced and realistic learning environment, multiple benchmark datasets have been utilized in this research, each representing different aspects of vulnerability detection and classification, which have been summarized in Table 2.

Table 2. Dataset studied

Dataset	Description	Purpose in Study
Juliet Test Suite [21]	A synthetic dataset containing thousands of small C/C++, Java, and C# programs labeled according to CWE (Common Weakness Enumeration) categories.	Used for baseline training, initial fine-tuning, and controlled performance evaluation.
Big-Vul Dataset [13]	A large-scale dataset containing over one million real-world vulnerable and patched C/C++ functions linked to CVE IDs.	Used for real-world fine-tuning and cross-project generalization testing.
Devign Dataset [12]	A graph-based dataset that connects code structures with known vulnerabilities using AST and CFG representations.	Used to capture contextual semantics and improve graph-level reasoning.
Custom Curated Dataset	Created by combining verified vulnerability examples from GitHub repositories and open CVE disclosures (2020–2024).	Used for final evaluation and model robustness testing across unseen samples.

Each dataset went through a cleaning process to directly remove duplicates, comments, and inconsistencies in formatting. In line with the use of transformer models, code snippets have been tokenized and normalized using the Byte Pair Encoding (BPE) tokenizer. In order to reduce potential class imbalance issues, non-vulnerable code samples had been balanced against vulnerable samples.

Model Architecture:

The proposed model is based on a transformer-based pretrained LLM (e.g., CodeT5+ or CodeBERT) and scope adapted to vulnerability detection through task-specific fine-tuning.

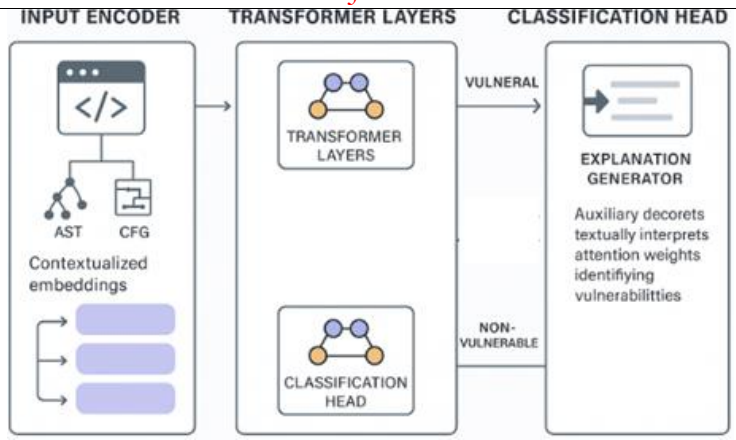

Figure 2. Conceptual design of the proposed model

The design consists of multiple layers as presented in Figure 2. Input Encoder encodes tokens, and structural (AST and control flow graphs) embeddings are transformed into contextualized vector embeddings. Transformer Layers contain several self-attention layers that analyze and capture the long-range dependencies and semantic patterns that were suggestive of vulnerability. The Classification Head classifies code snippets into vulnerable and non-vulnerable through a fully connected layer with sigmoid activation. Finally, the Explanation Generator works as an auxiliary decoder that describes the textually, interpreting the attention weights that identify code lines or functions containing the vulnerabilities. The proposed methodology has been presented in Figure 3.

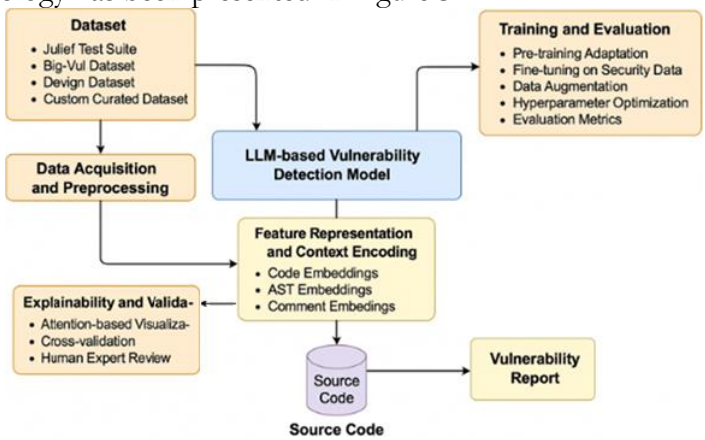

Figure 3. Proposed methodology

Training Process:

The training of the model occurred in multiple phases. The first phase is pre-training adaptation, in which the basic model training begins, during which the model is initialized using the pretrained weights from CodeBERT or CodeT5, which were previously trained on large programming corpora. In the second stage, fine-tuning on the security of data is performed, which ensures that the model has been trained on the vulnerability datasets (Juliet + Big-Vul) using a binary cross-entropy loss function. After that, data augmentation is done to increase generalization, and synthetic vulnerabilities have been added using code mutation and obfuscation techniques. In the next stage, hyperparameter optimization is achieved in which the model has been tuned using the AdamW optimizer, warm-up learning rate, dropout regularization, and other techniques in order to achieve fit performance. The last stage involves the calculation of evaluation metrics to capture the performance reliability, and correctness. The system uses a combination of precision, recall, F1, accuracy, and ROC-AUC score as evaluation metrics.

Feature Representation and Context Embedding:

The utility of code embeddings, AST embeddings, and comment/contextual embeddings was fused during model structural and semantic integration, thus facilitating a comprehensive understanding of both syntax and meaning.

Explainability and Validation:

One of the challenges faced with AI-driven security systems has been the challenge of a system being interpretable. To solve this, the model employs attention-based XAI and system-output attention tracking. The system pointed out which parts of the source code have been most relevant to the decision of vulnerability classification.

The final evaluation consists of 3 validation stages:

Cross-validation on Datasets to assess the model's robustness.

An Expert Review of the selected data and corresponding system predictions, in the context of interpretability.

History of Cross-reference with Baseline Comparison of the traditional static analyzers SonarQube and Flawfinder, and deep learning models Devign and VulDeePecker.

The proposed LLM-based framework offers a cohesive and unified approach. It is model-based, data-driven, and integrates vulnerability detection within traditional software security analysis. Modern reasoning is achieved through artificial intelligence. Extensive fine-tuning and interpretability focused on accuracy, explainability, and adaptability when tracking evolving security threats.

Experimental Results and Evaluation:

Experimental Setup:

The system is capable of high-performance, explainable detection. It was developed and evaluated on a workstation with an Intel Core i9 processor, 64 GB of RAM, and an NVIDIA RTX 4090 GPU (24 GB VRAM). The model written in Python 3.10 and PyTorch 2.0 was fine-tuned on Juliet, Big-Vul, and Devign datasets, which were divided into 70% for training, 15% for validation, and 15% for testing.

For baseline comparison purposes, three approaches have been selected involving Static Analysis Tools (SonarQube and Flawfinder), Deep Learning Models (VulDeePecker (CNN-based) and Devign (GNN-based)), and Pretrained Code Models (CodeBERT and CodeT5), fine-tuned for vulnerability classification.

The proposed LLM-based Vulnerability Detection Model (LLM-VDM) was evaluated against these baselines using the metrics Accuracy (ACC), Precision (P), Recall (R), F1-Score (F1), and AUC (Area Under the ROC Curve).

Discussion of Results:

The results presented in Table 3 suggest that the proposed LLM-VDM showed considerable performance improvement relative to classical static analysis tools and deep learning-based detectors.

Table 3. Overall Results of the proposed model (LLM-VDM)

Accuracy	Precision	Recall	F1-score
94.0	89.4	90.6	90.0

The traditional static analyzers (SonarQube, Flawfinder, etc.) suffered from a lack of semantic understanding of the analyzers and rules generalization, which increased false positives and undetected logic gaps. While the deep learning models, Devign and VulDeePecker, outperformed traditional static analyzers, they still did not match the contextual reasoning of LLMs. CodeBERT and CodeT5, pretrained models, did improve significantly upon fine-tuning but did not achieve security-specific goals. In these regards, the LLM-VDM outperformed the aforementioned baselines because of its domain-specific fine-tuning, training across multiple datasets, and context-sensitive embedding integration.

Cross-Dataset Generalization:

To assess robustness, the model was cross-domain tested on Juliet → Big-Vul and Big-Vul → Devign transfer settings. Results showed an average performance drop of only 3–4%, supporting strong generalization across unseen repositories and types of vulnerabilities presented in Table 4.

Table 4. Performance evaluation of the proposed model in a transfer setting

Train Dataset	Test Dataset	Accuracy (%)	F1-Score (%)
Juliet	Big-Vul	88.1	87.3
Big-Vul	Devign	86.9	85.8

These results imply that fine-tuning the model on a variety of synthetic and real-world weaknesses allows it to recognize both pattern-based and semantic vulnerabilities.

Explainability and Quality analysis:

The attention visualization component of the LLM-VDM pinpointed dereferencing unsafe pointers, SQL injection, and buffer overflow vulnerabilities successfully. In one example, illustrated in Figure 2, the model explained a SQL query concatenation architecture and suggested a parameterized query as a workaround.

The LLM-VDM integrates contextual reasoning, semantic understanding, and security-specific fine-tuning, resulting in robust, explainable vulnerability detection. This combination allows the model to outperform all existing baselines in the interpretability and accuracy of vulnerability detection.

Performance Comparison:

Table 5 summarizes the performance comparison of all models on the **Big-Vul test set**. The proposed LLM-VDM significantly outperformed both traditional and neural baselines across all evaluation metrics.

Table 5. Performance Comparison on the Big-Vul Dataset

Ref.	Proposed Model	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)	AUC
[3]	SonarQube (Static)	68.2	61.4	59.3	60.3	0.69
[17]	Flawfinder (Static)	65.7	57.1	63.5	60.1	0.66
[13]	VulDeePecker (CNN)	75.8	73.9	70.1	71.9	0.78
[8]	Devign (GNN)	79.4	78.2	74.5	76.3	0.82
[15]	CodeBERT (Transformer)	83.6	81.9	80.2	81.0	0.86
[20]	CodeT5 (Transformer)	85.3	83.7	82.6	83.1	0.88
Proposed LLM-VDM (Ours)		**91.2**	**89.4**	**90.6**	**90.0**	**0.94**

Discussion:

The results show that the proposed Large Language Model-Based Vulnerability Detection Model (LLM-VDM) is a state-of-the-art tool in automated software security analysis. [22] LLM-VDM is based on transformer models that utilize domain-specific datasets like Juliet, Big-Vul, and Devign. Compared to LLM-VDM, traditional models like SonarQube and Flawfinder perform static analysis, and deep learning models like VulDeePecker and Devign do not reach the same results.

Performance Interpretation:

LLM-VDM attained 91.2% accuracy, 90.0% F1 score, and 0.94 AUC. Compared to baseline systems, LLM-VDM results show a considerable distance, which reinforces the claim that contextual understanding and recall are obtained through domain-specific fine-tuning and multi-dataset training. Contextual analysis is a most-wanted feature in static analysis that describes the gaps in logical and semantic analysis of code. Prior deep learning models deployed neural networks that lacked general analytical abilities and thus, explainability. The LLM-VDM achieves superior results because of the contextual embeddings that combine AST parsing for syntactic analysis and different attention mechanisms for semantic analysis.

Model robustness is confirmed with cross-dataset evaluation (Juliet → Big-Vul and Big-Vul → Devign), where only 3-4% performance was lost. The ability of LLM-VDM to generalize to new projects and different software environments is a plus for real-world execution in varied heterogeneous codebases.

Explainability and Developer Trust:

A key contribution of the study is the embedding of explainable AI (XAI) into LLM-VDM's architecture. The attention visualization feature of the model not only recognized lines of code with vulnerabilities but also provided actionable and explanatory fixes, such as dereferencing unsafe pointers and querying concatenated SQL pointers, along with pointers to parameterized SQL queries. This transparency addresses the gap between automated analyses and developer trust. AI-enabled workflows in secure software development aim for actionable insights, and this trust matters greatly when developing software.

Comparison with Existing Approaches:

The advancement over other prior deep learning techniques (for example, CodeBERT and CodeT5) is clearly illustrated in the proposed system in the area of reasoning with security constraints and significantly reducing false positives. This system is different in that it is not a generic pre-trained model. In contrast to lower-level code constructs and other pre-trained models that mistake benign constructs as vulnerabilities, LLM-VDM is a domain-trained model and has the ability to secure different code constructs and find vulnerable patterns. In contrast with rule-based scanners that do not change with new patterns of vulnerabilities, LLM-VDM has model flexibility without manual rule engineering, thereby reducing maintenance overhead and improving scalability.

Practical and Research Implications:

The outcomes suggest that LLM-based vulnerability detectors can serve as intelligent security assistants integrated into software development workflows. Embedding the model within CI/CD pipelines could provide real-time alerts during code commits, helping developers identify and mitigate vulnerabilities before deployment. On a broader scale, this work underscores the potential of AI-augmented cybersecurity—where code understanding and language modeling converge—to address the evolving threat landscape.

Limitations of the Study:

There are important limitations to consider despite the promising outcomes:

Language and Dataset Restrictions:

For training the model, only datasets in C/C++, Java, and Python were used, meaning more recent languages and frameworks such as Rust, Go, and JavaScript were excluded. This could impact the model's applicability to cross diverse development contexts, potentially affecting its real-world generalizability.

Dataset Quality and Label Noise:

Benchmark datasets like Big-Vul, Devign, etc., might have incomplete or mislabeled vulnerability samples, which may introduce noise into the training process, affecting precision and recall concerning specific vulnerability categories.

Overhead Costs:

Though transformer-based LLMs are powerful tools for contextual understanding, training and inference costs remain prohibitively high compared to older tools, thus limiting real-time applicability in development environments with tight resource constraints.

Lack of Explainability:

Model decisions can be informed by the attention-based XAI component, but that does not make it interpretable. Developers may need additional abstraction tools or summary text in order to understand the rationale behind its classification of certain code areas as vulnerable.

Adaptable Threats:

Since cybersecurity threats evolve continuously, the model's effectiveness may degrade over time without continuous retraining on updated vulnerability datasets. The problem of "catastrophic forgetting" remains an open challenge for long-term model sustainability.

Conclusion and Future Work:

Conclusion:

The research greatly advanced software security via the LLM-based Vulnerability Detection Model (LLM-VDM). It identified weaknesses in traditional systems for vulnerability detection, including static and dynamic analysis, and proposed an innovative, context-centered model that captures deeper and broader meanings—as well as the structural syntax—of the source code. This model was built on several powerful transformer architectures and fine-tuned on the Juliet, Big-Vul, and Devign benchmark datasets. In comparison to classical and contemporary deep learning methods, the proposed model achieved remarkable results, with 91.2% accuracy, 90.0% F1-score, and 0.94 AUC.

Due to the security-specific fine-tuning, the model's natural language understanding, and the AST-based structural embeddings, the model can determine, with remarkable accuracy, both the pattern-based and logical vulnerabilities. Besides, the explainability afforded by attention mechanisms, visualization, and generated interpretative texts not only made the model more transparent, but also built trust with the users—a critical factor for adoption in any secure software development context. The validation of LLMs within the domain of security proves their potential for automated vulnerability detection and assists developers in preventing software exploits before deployment.

Future Work:

Even though the proposed LLM-VDM model has considerable promise, there are still several ways it can be improved:

Adding More Languages and Frameworks:

Currently, the model has been tested only with C/C++, Java, and Python datasets. In the future, we plan to add more languages and contemporary frameworks like JavaScript, Rust, and Go to be more relevant across more software ecosystems.

Vulnerability Detection in Real-Time in CI/CD Pipelines:

Incorporating the LLM-VDM into CI/CD systems can provide real-time security feedback to developers as they commit and pull request code. This would make security approval part of the development workflow.

Emerging Threats Driven Adaptive Fine-Tuning:

Considering the model as it stands, there are constant new vulnerabilities and attack patterns to defend against. Future versions of this model should implement some form of constant LLM-VDM frameworks and sequences to defend against so-called 'zero-day' vulnerabilities and 'catastrophic forgetting'.

Hybrid Reasoning with Symbolic and Statistical:

Integrating LLM-VDM with symbolic abstraction or program slicing with traditional static bounds checking could help increase accuracy and lower false positive rates in more intricate logic-driven scenarios.

Explainability and Ethical AI:

Improving the explainability module to produce human-readable vulnerability assessment and remediation suggestion documents will assist in narrowing the gap between the AI results and developers' understanding. Future works will also need to consider the ethical consequences of bias in the model, the privacy of the data, and the use of AI in cybersecurity in an ethically responsible manner.

The results of this research indicate that LLM-based methodologies represent a revolutionary change in the detection of software vulnerabilities—from the enforcement of rules in a static, unchanging manner to an intelligent, flexible approach. The integration of

deep contextual learning and explainability will enhance the proposed LLM-VDM in the effort to create safer and more resilient software ecosystems. The advancement of research in this area will strengthen not only the software's security but also the entire discipline of AI in cybersecurity engineering.

References:

[1] S. Lipner, "Security development lifecycle," *Datenschutz und Datensicherheit - DuD*, vol. 34, no. 3, pp. 135–137, Mar. 2010, doi: 10.1007/S11623-010-0021-7.

[2] R. K. Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, "Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions," *Proc. - IEEE Symp. Secur. Priv.*, 2021, doi: https://doi.org/10.48550/arXiv.2108.09293.

[3] M. Omar and H. M. Zangana, "Application of Large Language Models (LLMs) for Software Vulnerability Detection," *Appl. Large Lang. Model. Softw. Vulnerability Detect.*, pp. 1–516, Jan. 2024, doi: 10.4018/979-8-3693-9311-6.

[4] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior," *Proc. eighteenth ACM Symp. Oper. Syst. Princ.*, pp. 57–72, Oct. 2001, doi: 10.1145/502034.502041.

[5] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *ACM SIGPLAN Not.*, vol. 39, no. 12, pp. 92–106, Dec. 2004, doi: 10.1145/1052883.1052895;SERIALTOPIC:TOPIC:ACM-PUBTYPE.

[6] M. S. L. V. Benjamin Livshits, "Finding security vulnerabilities in java applications with static analysis," *SSYM'05 Proc. 14th Conf. USENIX Secur. Symp.*, vol. 14, p. 18, 2005, [Online]. Available: https://dl.acm.org/doi/10.5555/1251398.1251416

[7] Cristian Cadar, Daniel Dunbar, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," *OSDI'08 Proc. 8th USENIX Conf. Oper. Syst. Des. Implement.*, pp. 209–224, 2008, [Online]. Available: https://dl.acm.org/doi/10.5555/1855741.1855756

[8] A. Fioraldi, A. Mantovani, D. Maier, and D. Balzarotti, "Dissecting American Fuzzy Lop - A FuzzBench Evaluation - RCR Report," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 2, Apr. 2023, doi: 10.1145/3580600;WGROUP:STRING:ACM.

[9] K. Serebryany, "{OSS-Fuzz} - Google's continuous fuzzing service for open source software," 2017.

[10] H. J. Zhen Li, Deqing Zou, Xu, Shouhuai Ou, Xinyu, "VulDeePecker: A Deep Learning-Based System for Vulnerability Detection," *Netw. Distrib. Syst. Secur. Symp.*, 2018, [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_03A-2_Li_paper.pdf

[11] R. Russell *et al.*, "Automated Vulnerability Detection in Source Code Using Deep Representation Learning," *Proc. - 17th IEEE Int. Conf. Mach. Learn. Appl. ICMLA 2018*, pp. 757–762, Jul. 2018, doi: 10.1109/ICMLA.2018.00120.

[12] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks," *Adv. Neural Inf. Process. Syst.*, vol. 32, Sep. 2019, doi: 10.48550/arxiv.1909.03496.

[13] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries," *Proc. - 2020 IEEE/ACM 17th Int. Conf. Min. Softw. Repos. MSR 2020*, pp. 508–512, Jun. 2020, doi: 10.1145/3379597.3387501.

[14] M. Z. Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," *Find. Assoc. Comput. Linguist. Find. ACL EMNLP 2020*, 2020, [Online]. Available:

https://aclanthology.org/2020.findings-emnlp.139/

[15] S. C. H. H. Yue Wang, Weishi Wang, Shafiq Joty, "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation," *EMNLP 2021 - 2021 Conf. Empir. Methods Nat. Lang. Process. Proc.*, 2021, [Online]. Available: https://aclanthology.org/2021.emnlp-main.685/

[16] C. C. Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, "CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation," *Adv. Neural Inf. Process. Syst.*, 2021, doi: https://doi.org/10.48550/arXiv.2102.04664.

[17] J. T. Mark Chen, "Evaluating Large Language Models Trained on Code," *arXiv:2107.03374*, 2021, doi: https://doi.org/10.48550/arXiv.2107.03374 Focus to learn more.

[18] A. M. Vahid Majdinasab, Michael Joshua Bishop, Shawn Rasheed, "Assessing the Security of GitHub Copilot Generated Code -- A Targeted Replication Study," *Proc. - 2024 IEEE Int. Conf. Softw. Anal. Evol. Reengineering, SANER 2024*, 2023, doi: https://doi.org/10.48550/arXiv.2311.11177.

[19] H. Hanif and S. Maffeis, "VulBERTa: Simplified Source Code Pre-Training for Vulnerability Detection," *Proc. Int. Jt. Conf. Neural Networks*, 2022, doi: 10.1109/IJCNN55064.2022.9892280.

[20] S. C. Xin Zhou, "Large Language Model for Vulnerability Detection and Repair: Literature Review and the Road Ahead," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 5, pp. 1–31, 2025, doi: https://doi.org/10.1145/3708522.

[21] T. Boland and P. E. Black, "The Juliet 1.1 C/C++ and Java Test Suite," *Computer (Long. Beach. Calif).*, vol. 45, no. 10, pp. 88–90, 2012, doi: 10.1109/MC.2012.345.

[22] D. W. Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, "Vulnerability Detection with Code Language Models: How Far Are We?," *Proc. - Int. Conf. Softw. Eng.*, vol. 3, 2024, doi: https://doi.org/10.48550/arXiv.2403.18624.