Dependency-Informed GraphCodeBERT Framework for Java Method-Level Vulnerability Detection

Urusha Shakhakarmi
Chonnam National University
South Korea
urusha7403@jnu.ac.kr

Geonmin Kim
Chonnam National University
South Korea
geonminkim@jnu.ac.kr

Hyeonji Jang
Chonnam National University
South Korea
gka1225@jnu.ac.kr

Kyungbaek Kim
Chonnam National University
South Korea
kyungbaekkim@jnu.ac.kr

ABSTRACT

Detecting security vulnerabilities in Java source code is challenging because vulnerable behavior often depends on complex interactions among variables, API calls, and statements within a method. Rule-based static analysis tools are widely used but rely on manually defined specifications and generalize poorly across projects. To address this, we propose a dependency-informed GraphCodeBERT framework for Java method-level vulnerability detection. The framework applies leakage-aware preprocessing, extracts lightweight dependency and data-flow information and uses a multi-task design to predict vulnerability labels, CWE categories, and suspicious-line scores. We evaluate the framework on Juliet Java, SVD-Benchmark, CWE-Bench-Java, and Vul4J under synthetic-to-real, project-disjoint, cross-corpus, and few-shot settings. Across five project-disjoint folds of SVD-Benchmark, the model achieves a mean AUC of 0.907 ± 0.040 and F1-score of 0.833 ± 0.059 (AUC 0.914 on the primary split). Few-shot fine-tuning with only 5% real SVD-Benchmark data improves AUC from 0.450 to 0.832. Overall, the results show that GraphCodeBERT-based representations are effective for project-disjoint Java vulnerability detection within the same corpus.

KEYWORDS

Java vulnerability detection, software security, GraphCodeBERT

1 INTRODUCTION

Java is widely used in modern software systems, including enterprise, financial and government applications. Because Java programs often process user input, database queries, file operations, and authentication logic, they can be difficult to secure. Vulnerabilities such as SQL injection, cross-site scripting, path traversal, and sensitive information exposure may arise from interactions among variables, framework APIs, and method calls. Manual vulnerability detection is difficult because modern Java projects are large and usually spread across many classes, methods, and frameworks. Recent Java-focused studies have shown that vulnerabilities in Java web applications are often related to complex program behaviors, including data-flow, control-flow, and interprocedural relationships, rather than isolated code statements [1] – [5]. Therefore, automated vulnerability detection for Java source code remains an important software security problem.

Explanation:
The highlighted line is vulnerable because it builds the SQL query by concatenating the untrusted username directly into the string. This allows an attacker to inject SQL code through the username parameter, leading to SQL injection (CWE-089). The correct fix is to use a prepared statement with parameter binding instead of string concatenation.

```
001  import java.sql.*;
002
003  public class SQLInjectionExample {
004      public void getUser(Connection conn, String
     username) throws SQLException {
005          String query = "SELECT * FROM users WHERE name =
     '" + username + "'";
006          Statement stmt = conn.createStatement();
007          ResultSet rs = stmt.executeQuery(query);
008      }
009  }
```

Figure 1: Example of a SQL injection vulnerability (CWE-089) in Java

Traditional static analysis techniques have been widely used for vulnerability detection because they can analyze program structure, control flow, data flow, and relationships without executing the program. These techniques are useful for identifying potential security weaknesses and enforcing predefined security rules. However, rule-based static analysis tools often depend on manually defined vulnerability specifications across different Java frameworks and application domains. This makes it difficult to scale across different projects and frameworks. IRIS addresses this problem by combining large language models with static analysis

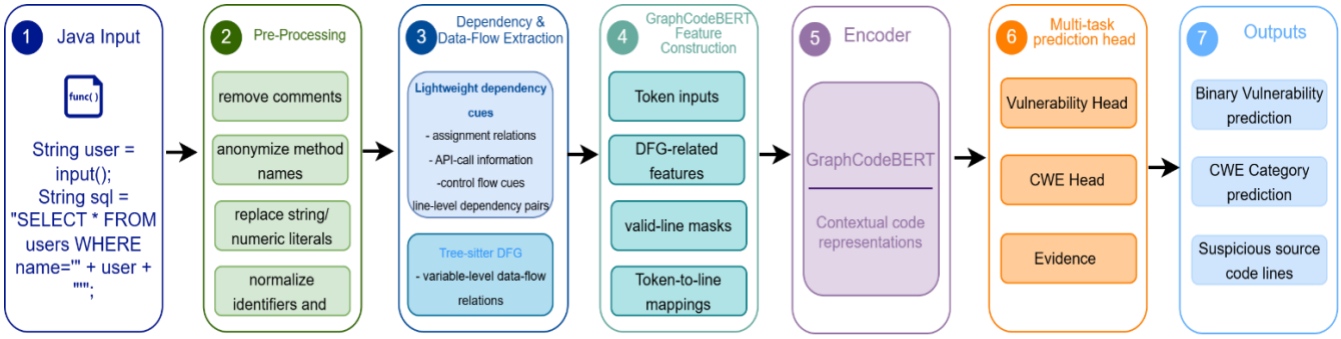

Figure 2: Architecture of the Dependency-Informed GraphCodeBERT Framework for Java Method-Level Vulnerability Detection

to infer taint specifications and perform contextual reasoning for Java projects [1]. Similarly, recent Java web vulnerability detection studies use code property graphs, interprocedural analysis and concolic execution to capture deeper program behavior [2-5]. These works show structural and semantic program information is important for detecting Java vulnerabilities.

Deep-learning based approaches have recently been used to improve vulnerability detection. Deep-learning methods can learn vulnerability patterns from source code without relying on manually designed rules. For Java source code, Hussain et al. proposed a hybrid deep learning model that combines graph-based and sequence-based feature extraction with CodeBERT to improve syntactic and semantic code representation [6]. Other recent methods use GraphCodeBERT, data-flow graphs, contrastive learning and graph attention networks to improve vulnerability representation learning [7], [8]. StagedvulBERT further shows that vulnerability detection can benefit from multi-granular modeling where both coarse-grained detection and fine-grained vulnerable statement localization are considered [9]. These studies suggest that pretrained code models are useful, but also need structural cues and fine-grained supervision to support practical vulnerability detection.

Although these studies have advanced Java vulnerability detection, several limitations remain. First, many repository-level systems that use static analysis, code property graphs or LLM assisted reasoning can capture deeper program behavior, but they are often complex and computationally expensive to apply [1], [3], [5], [9]. In contrast, many neural vulnerability detection methods are easier to train and deploy, but they often focus mainly on binary classification and provide limited explanations for developers [6], [7], [8]. Recent LLM-based studies also show that vulnerability detection can change significantly depending on the selected model, prompt design, context size, and evaluation setting [1]. These observations suggest that Java vulnerability detection should not only focus on prediction accuracy but also consider generalization, evidence of localization, and practical usefulness for code review.

To address these limitations, this paper introduces a dependency-informed GraphCodeBERT framework for Java method-level vulnerability detection. The framework applies leakage-aware preprocessing, extracts lightweight dependency and data-flow information, and converts the extracted information into GraphCodeBERT-compatible features. It uses a multi-task learning design to predict the vulnerability label, classify the CWE category, and produce suspicious-line scores. This design aims to support both automatic detection and developer-oriented inspection. The dependency information is used as lightweight structural context, while GraphCodeBERT provides the main pretrained code representation.

The main contributions of this paper are as follows:

1. This paper introduces a dependency-informed GraphCodeBERT framework for Java method-level vulnerability detection.
2. We formulate Java vulnerability detection as a multi-task learning problem in which binary vulnerability classification is the primary task and CWE-category and suspicious-line supervision act as auxiliary signals; we quantitatively evaluate binary detection and line localization.
3. The proposed framework is evaluated under synthetic-to-real transfer, project-disjoint evaluation, cross-corpus evaluation, and few-shot adaptation using Java vulnerability datasets, including Juliet Java, SVD-Benchmark, CWE-Bench-Java and Vul4J.

2 METHODOLOGY

This section describes the proposed framework for Java-method-level vulnerability detection. The framework analyzes Java methods or code snippets and produces three outputs: a binary vulnerability prediction, a CWE-category prediction, and suspicious-line scores. The overall goal is to support vulnerability detection while also providing line-level information that can help developers inspect the predicted result.

A Code Representation

The input to the framework is a Java method or code snippet collected from Java vulnerability datasets including Juliet Java, SVD-Benchmark, CWE-Bench-Java and Vul4J. Each sample contains a binary vulnerability label, and CWE labels and

vulnerable-line annotations are used when available. The CWE labels are mapped into consistent label space, and rare CWE classes may be grouped into broader security categories to reduce label imbalance during training.

For each raw Java sample, the framework performs dependency extraction and leakage-aware preprocessing as code preparation steps. Dependency and data-flow information are extracted from the raw source code to preserve API names, variable relations and original line positions. Before tokenization, leakage-aware preprocessing removes comments, anonymizes declared method names, replaces string and numeric literals with placeholder tokens, and normalizes identifiers. Security-relevant API names are preserved during normalization because they may be important for vulnerability detection. The preprocessing step also preserves line alignment so that token representations can later be mapped back to corresponding source-code lines.

The framework supports lightweight dependency extraction and Tree-sitter-based DFG extraction. The lightweight extractor identifies assignment relations, API-call information, control-flow-inspired line relations, source/sink/sanitizer API cues, and line-level dependency pairs. The Tree-sitter-based extraction is used to obtain variable-level data-flow nodes and edges for GraphCodeBERT. The extracted dependency information is converted into GraphCodeBERT-compatible features, including token inputs, DFG related nodes and edges, a graph-guided attention mask, valid-line masks, and token-to-line mappings. The features allow the model to use method-level code representation and line-level information for suspicious-line scoring.

B Multi-Task Detection Model

The proposed model uses GraphCodeBERT as the shared encoder. In the dependency-informed setting, a graph-guided attention mask is used so that data-flow information can be incorporated into the encoded representation. GraphCodeBERT produces contextual representations for the input Java code. The method-level representation is used for vulnerability classification and CWE classification, while token-level representations are used for suspicious-line scoring.

The model contains three prediction heads. The first head performs binary vulnerability classification and predicts whether the input method is vulnerable or safe. The second head performs CWE category classification and predicts the likely vulnerability type or grouped CWE category. The third head performs suspicious-line scoring by aggregating token embeddings for each valid source-code line and assigning a score to each line. The multi-task design allows the model to learn related vulnerability information from the same shared code representation. Binary classification provides the main detection result, CWE classification provides additional information about the vulnerability type, and suspicious-line scores provide developer-oriented inspection support.

C Training and Inference

The model is trained using a multi-task objective that combines vulnerability classification, CWE category prediction and suspicious-line scoring. The overall loss is a weighted sum of task-specific losses, allowing the shared GraphCodeBERT encoder to learn both method-level and line-level vulnerability information. Binary vulnerability prediction is optimized as the primary task, while CWE classification and suspicious-line scoring provide auxiliary supervision. The auxiliary tasks do not require complete annotations for every training sample. CWE supervision is applied only when a valid CWE label is available, while samples without a mapped CWE category are excluded from the corresponding loss. For line-level evidence, explicit vulnerable-line annotations are used when available. When annotations are unavailable, the framework can use weak supervision derived automatically from dependency information or exclude the sample from the evidence loss when no usable supervision is available.

3 EXPERIMENTS

Experiments are conducted on four Java vulnerability datasets: Juliet Java, SVD-Benchmark, CWE-Bench-Java, and Vul4J. Juliet Java is used as a synthetic training source, while SVD-Benchmark, CWE-Bench-Java and Vul4J are used to evaluate performance on real Java vulnerability data. For leakage-safe evaluation, the dataset splits are project-disjoint, meaning that the same project does not appear in more than one training, validation, or test split.

The evaluation includes four settings. First, synthetic-to-real transfer is evaluated by training on Juliet Java and testing on SVD-Benchmark. Second, project-disjoint evaluation is conducted within SVD-Benchmark to measure generalization across unseen Java projects from the same corpus. Third, cross-corpus evaluation is performed by training on SVD-Benchmark and testing on CWE-Bench-Java and Vul4J. Fourth, few-shot adaptation is evaluated by fine-tuning the Juliet-pretrained model using different proportions of real SVD-Benchmark training data. The model is implemented using GraphCodeBERT with lightweight dependency information extracted through Tree-sitter DFG extraction. AUC and F1-score are used as the main metrics for binary vulnerability detection.

All models use the microsoft/graphcodebert-base checkpoint with a maximum input length of 512 tokens, up to 160 source lines, and at most 96 data-flow nodes extracted with the official GraphCodeBERT Tree-sitter DFG procedure for Java. Training uses the AdamW optimizer with a learning rate of 2e-5, batch size 16, weight decay 0.01, a 6% linear warmup, up to 10 epochs with early stopping (patience 4) on validation AUC, and random seed 42. The multi-task loss weights are 1.0 (vulnerability), 0.7 (CWE), and 0.8 (evidence); the CWE loss is applied only to samples with a mapped label. After training, temperature scaling is fitted on the validation split to calibrate probabilities, and F1-score is computed at the 0.5 threshold. Splits are project-disjoint using each dataset's project identifier, so no project contributes to more than one of train, validation, and test.

The corpora contain 65,536/8,339/7,930 (Juliet Java), 3,241/767/1,045 (SVD-Benchmark; 17/1/2 disjoint projects), 318/230/225 (CWE-Bench-Java; 71/7/10 projects), and 72/48/48 (Vul4J; 34/6/8 projects) train/validation/test methods, with approximately balanced vulnerable/safe labels in each split.

For few-shot adaptation, we sample balanced subsets of the SVD-Benchmark training split that preserve the vulnerable/safe ratio, using a fixed random seed; the subsets are nested (the 5% subset is contained in the 10% subset, and so on). The validation split is kept unchanged for model selection. "100% fine-tune" continues training the Juliet-pretrained model on the full real training split, whereas "from scratch" trains the same architecture on the full real training split without Juliet pretraining. Results are reported for a single sampling seed.

Because consistent CWE labels are available for only a subset of real-world samples and the label space is highly imbalanced, the CWE head is treated as auxiliary supervision during training, and we leave its quantitative evaluation to future work.

4 RESULTS AND DISCUSSION

Figure 3 shows the generalization performance across different evaluation settings. When the model is trained on the synthetic Juliet dataset and evaluated on SVD-Benchmark, it achieves AUC of 0.450 and an F1-score of 0.465 indicating that synthetic-only training does not transfer to real-world code. In contrast, training and evaluating on the primary project-disjoint split of SVD-Benchmark yields substantially higher performance (an AUC of 0.914 and an F1-score of 0.819; see Figure 4 for the five-fold evaluation). The low performance on CWE-Bench-Java and Vul4J suggests that external real-world evaluation remains difficult.

Table 1 compares cross-corpus performance with direct in-corpus training on CWE-Bench-Java and Vul4J. The SVD-trained model achieves AUC values of 0.494 and 0.491 on the two external datasets. However, even when the same architecture is trained directly on these target corpora, the AUC remains low at 0.539 for CWE-Bench-Java and 0.542 for Vul4J. The corresponding training partitions contain only 318 and 72 methods respectively. These weak in-corpus results suggest that the limited amount of target-corpus training data (318 and 72 methods) is also a contributing

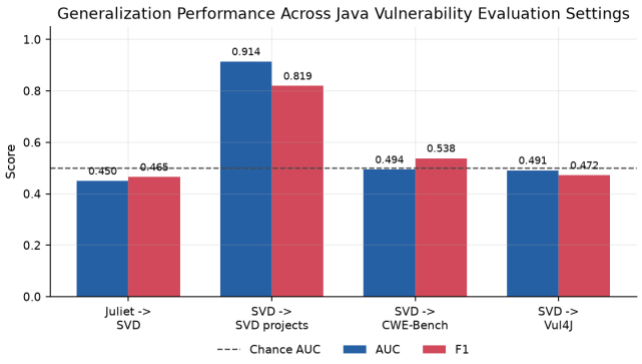

Figure 3: Generalization Performance Across Synthetic-To-Real, In-Corpus Cross-Project and Cross-Corpus Java Vulnerability Evaluation

Table 1: Cross-Corpus and In-Corpus Performance on External Java Vulnerability Corpora

Target Corpus	Target Train Method	In-Corpus AUC	SVD-Trained AUC
CWE-Bench-Java	318	0.539	0.494
Vul4J	72	0.542	0.491

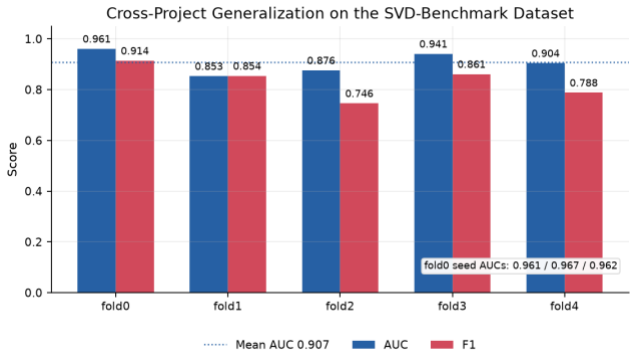

Figure 4: Cross-Project Generalization on the SVD-Benchmark Dataset

factor, in addition to potential cross-corpus distribution shift. Zero-shot cross-corpus transfer therefore remains challenging, and target-corpus data scarcity is an additional important limitation for external evaluation.

Figure 4 shows the 5-fold cross-project evaluation on SVD-Benchmark. AUC remains relatively stable across folds (0.853-0.961) while F1-score shows more variation (0.746-0.914). This suggests that the model learns vulnerability-relevant patterns that transfer reasonably well across unseen Java projects within the same corpus. The mean cross-project performance over the five folds is an AUC of 0.907 ± 0.040 and an F1-score of 0.833 ± 0.059.

Table 2 compares the token-only GraphCodeBERT baseline with the dependency-informed model. The dependency-informed model improves AUC only slightly from 0.911 to 0.914 while the token-only model obtains a higher F1-score. To further examine whether the trained model functionally depends on the extracted DFG, we perform inference-time perturbation analysis in Table 3. Shuffling the DFG edges changes only 0.7% of binary predictions, and AUC remains essentially unchanged (0.913 vs. 0.912). Randomizing the node-to-line positions changes 2.2% of

Table 2 : Ablation Comparison Between Token-Only and Dependency-Informed GraphCodeBERT on Project-Disjoint SVD-Benchmark

Model Setting	AUC	F1
Token-Only GraphCodeBERT	0.911	0.837
Dependency-Informed GraphCodeBERT+ tree-sitter DFG	0.914	0.819

Table 3: Effects of DFG Perturbations on Vulnerability Prediction Performance

DFG Condition	AUC	Prediction Flips
True DFG	0.913	-
Shuffled Edges	0.912	0.7%
Randomized Node Lines	0.915	2.2%

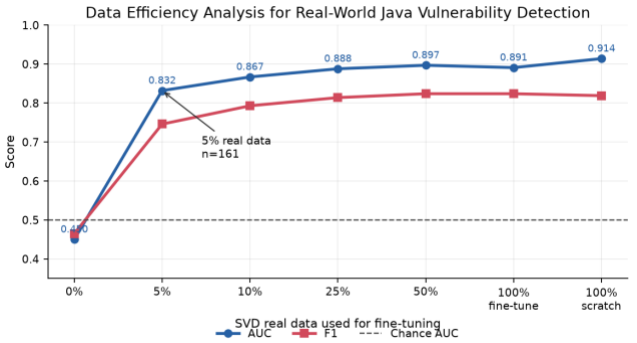

Figure 5: Effect of Real SVD-Benchmark Fine-Tuning Data Size on Model Adaptation After Synthetic-Data Training

predictions and likewise does not reduce AUC. These results indicate that binary vulnerability prediction depends only weakly on the precise DFG topology and is driven mainly by the semantic representations learned by GraphCodeBERT.

To evaluate the suspicious-line head directly, we measured standard localization metrics on the Juliet test set, the only corpus in our study with ground-truth vulnerable-line annotations. Ranking lines by the evidence head yields hit@1 of 0.973, hit@5 of 0.987, and a mean reciprocal rank (MRR) of 0.981 over 374 annotated vulnerable methods, versus hit@1 of 0.061 and MRR of 0.204 for a length-matched random-ranking baseline. The real-world corpora used in this study do not provide line-level ground truth; on those corpora the evidence head is trained with weak supervision derived automatically from dependency analysis, so we do not claim localization accuracy on real data. Juliet is split at the level of test-case families rather than individual methods (605/79/70 disjoint families for train/validation/test, with no family shared across splits), so these localization scores are not inflated by near-duplicate templates. A token-only model achieves comparable localization (hit@1 of 0.983 and MRR of 0.991), indicating that, as with binary detection, localization ability stems mainly from the pretrained semantic representation rather than the precise DFG.

Figure 5 shows that fine-tuning with a small amount of real SVD-Benchmark data substantially improves performance after synthetic data training. The AUC increases from 0.450 without real-data fine-tuning to 0.832 with only 5% of the SVD training data. Additional real data further improves performance, but the gain becomes more gradual. This result indicates that limited real Java vulnerability samples can significantly improve model adaptation to real-world code.

5 CONCLUSIONS

In this paper, we presented a dependency-informed GraphCodeBERT framework for Java method-level vulnerability detection. The framework combines pretrained code representation with lightweight dependency features and a multi-task design that predicts the vulnerability label, CWE category and suspicious source code lines. The framework also supports partially labeled training by ignoring unavailable auxiliary labels and using dependency-derived weak supervision when explicit line-level annotations are not available. The experimental results show that the model performs well in project-disjoint evaluation within the same corpus while performance decreases when transferring to different vulnerability corpora. The in-corpus control experiments further indicate that the limited amount of training data in external datasets also affects the interpretation of cross-corpus performance. In addition, the DFG perturbation analysis shows that binary vulnerability prediction is mainly driven by the pretrained semantic representation and has limited sensitivity to the precise DFG structure. In future work, we plan to improve evidence localization on real-world Java code, develop stronger methods for utilizing structural information and evaluate the framework on larger and more diverse vulnerability datasets.

ACKNOWLEDGMENTS

This work was supported by the Institute of Information & Communications Technology Planning & Evaluation(IITP)-Innovative Human Resource Development for Local Intellectualization program grant funded by the Korea government(MSIT)(IITP-2026-RS-2022-00156287)(34%). This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) under the Artificial Intelligence Convergence Innovation Human Resources Development (IITP-2026-RS-2023-00256629) grant funded by the Korea government(MSIT)(33%). This research was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korean government (MSIT) (RS-2024-00398993, Development of Security Threat Response Technology for Electric Vehicle Chargers)(33%).

REFERENCES

[1] Li, Z., Dutta, S. and Naik, M., 2025, May. IRIS: LLM-assisted static analysis for detecting security vulnerabilities. In *International conference on learning representations* (Vol. 2025, pp. 35735-35758).

[2] Chen, X., Wang, B., Zhang, M., Cao, Y. and Liu, Q., 2025, April. VulKiller: Java Web Vulnerability Detection with Code Property Graph and Large Language Models. In *ICASSP 2025-2025 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (pp. 1-5). IEEE.

[3] Zhang, B., Zhi, X., Wang, M., Ren, R. and Dong, J., 2025. Enhancing java web application security: Injection vulnerability detection via interprocedural analysis and deep learning. *IEEE Transactions on Reliability*, *74*(3), pp.3642-3656.

[4] Huang, X., Zhang, L., Liu, Y., Deng, P., Cao, Y., Zhang, Y. and Yang, M., 2025. Towards automatic detection and exploitation of java web application vulnerabilities via concolic execution guided by cross-thread object manipulation. In *34th USENIX Security Symposium (USENIX Security 25)* (pp. 8367-8384).

[5] Chen, X., Li, Y., Jin, Z., Tan, Y., Li, X., Wang, B., Zhang, M. and Liu, Q., 2026. iDetector: Unraveling and Automating the Detection of Modern Java Web Injection Vulnerabilities. *ACM Transactions on Software Engineering and Methodology*.

[6] Hussain, S., Nadeem, M., Baber, J., Hamdi, M., Rajab, A., Al Reshan, M.S. and Shaikh, A., 2024. Vulnerability detection in Java source code using a quantum convolutional neural network with self-attentive pooling, deep sequence, and graph-based hybrid feature extraction. *Scientific Reports*, *14*(1), p.7406.

[7] Wang, R., Xu, S., Tian, Y., Ji, X., Sun, X. and Jiang, S., 2024. SCL-CVD: Supervised contrastive learning for code vulnerability detection via GraphCodeBERT. *Computers & Security*, *145*, p.103994.

[8] Rangapuram, D.S. and Ratha, N., 2025, May. GraphCodeBERT-augmented graph attention networks for code vulnerability detection. In *2025 IEEE Conference on Artificial Intelligence (CAI)* (pp. 912-917). IEEE.

[9] Jiang, Y., Zhang, Y., Su, X., Treude, C. and Wang, T., 2024. StagedVulBERT: Multigranular vulnerability detection with a novel pretrained code model. *IEEE Transactions on Software Engineering*, *50*(12), pp.3454-3471.

[10] Lin, J. and Mohaisen, D., 2025, February. From Large to Mammoth: A Comparative Evaluation of Large Language Models in Vulnerability Detection. In *NDSS*.