

EnStack: An Ensemble Stacking Framework of Large Language Models for Enhanced Vulnerability Detection in Source Code

Shahriyar Zaman Ridoy¹, Md. Shazzad Hossain Shaon², Alfredo Cuzzocrea^{3,§}, and Mst Shapna Akter^{4,*}

¹Department of Electrical and Computer Engineering, North South University, Dhaka, Bangladesh

²Department of Computer Science and Engineering, Oakland University, Rochester, MI 48309, USA

³iDEA Lab, University of Calabria, Rende, Italy, and the Department of Computer Science, University of Paris City, Paris, France

⁴Department of Computer Science, Oakland University, MI, USA

shahriyar.ridoy@northsouth.edu, shaon@oakland.edu, alfredo.cuzzocrea@unical.it, akter@oakland.edu

Abstract—Automated detection of software vulnerabilities is critical for enhancing security, yet existing methods often struggle with the complexity and diversity of modern codebases. In this paper, we introduce EnStack, a novel ensemble stacking framework that enhances vulnerability detection using natural language processing (NLP) techniques. Our approach synergizes multiple pre-trained large language models (LLMs) specialized in code understanding—CodeBERT for semantic analysis, GraphCodeBERT for structural representation, and UniXcoder for cross-modal capabilities. By fine-tuning these models on the Draper VDISC dataset and integrating their outputs through meta-classifiers such as Logistic Regression, Support Vector Machines (SVM), Random Forest, and XGBoost, EnStack effectively captures intricate code patterns and vulnerabilities that individual models may overlook. The meta-classifiers consolidate the strengths of each LLM, resulting in a comprehensive model that excels in detecting subtle and complex vulnerabilities across diverse programming contexts. Experimental results demonstrate that EnStack significantly outperforms existing methods, achieving notable improvements in accuracy, precision, recall and F1-score. This work highlights the potential of ensemble LLM approaches in code analysis tasks and offers valuable insights into applying NLP techniques for advancing automated vulnerability detection.

Index Terms—CodeBERT, Ensemble Stacking, GraphCodeBERT, Large Language Models (LLMs), Source Code Analysis, UniXcoder, Vulnerability Detection.

I. INTRODUCTION

The increasing prevalence of software vulnerabilities in today’s rapidly advancing software development landscape poses significant security threats to individuals, organizations, and governments alike [1]. These vulnerabilities, often resulting from flaws in code implementation or design, can lead to serious consequences such as data breaches, system failures, and substantial financial losses. Traditional vulnerability detection methods, including manual code inspections and static analysis tools, are becoming less effective in managing the complexity and scale of modern software systems [2]. This situation underscores the urgent need for advanced techniques capable of efficiently identifying vulnerabilities within source code.

[§]This research has been made in the context of the Excellence Chair in Big Data Management and Analytics at University of Paris City, Paris, France.

Recent advancements in Artificial Intelligence (AI), particularly in Large Language Models (LLMs) [3]–[8], have opened new dimensions for code analysis [9]. Models like CodeBERT [10], GraphCodeBERT [11], and UniXcoder [12] have demonstrated remarkable abilities in understanding and processing programming languages by leveraging extensive code datasets. These models have been successful in tasks such as code completion, code retrieval, and bug detection by capturing both the syntactic and semantic aspects of code. Despite these achievements, challenges remain in effectively detecting complex software vulnerabilities.

Current LLM-based approaches for vulnerability detection often concentrate on specific facets of code representation. For example, CodeBERT focuses on capturing the semantic meaning [10] of code tokens, while GraphCodeBERT emphasizes structural relationships through data flow graphs [11]. UniXcoder seeks to unify cross-modal representations but may not fully integrate all dimensions of code features [12]. When utilized individually, these models might not fully encapsulate the multifaceted nature of software vulnerabilities, which often require a comprehensive understanding of both syntax and context [13]. Additionally, earlier models like VulDeePecker [14], SySeVR [15], and Devign [16] have made progress in applying deep learning to vulnerability detection but are constrained by their reliance on specific neural network architectures, such as Recurrent Neural Networks (RNNs) or Graph Neural Networks (GNNs), which may not generalize effectively across diverse vulnerability types.

To overcome these limitations, this paper introduces an ensemble-based stacking approach that combines multiple LLMs to enhance vulnerability detection in source code. By integrating CodeBERT, GraphCodeBERT, and UniXcoder through an ensemble stacking technique, we aim to leverage the unique strengths of each model to develop a more robust and comprehensive vulnerability detection system. This approach involves fine-tuning each model separately on the Draper VDISC dataset [17], which contains a wide range of labeled code snippets with various types of vulnerabilities. The outputs of these models are then combined using meta-classifiers, including Logistic Regression (LR) [18], Support

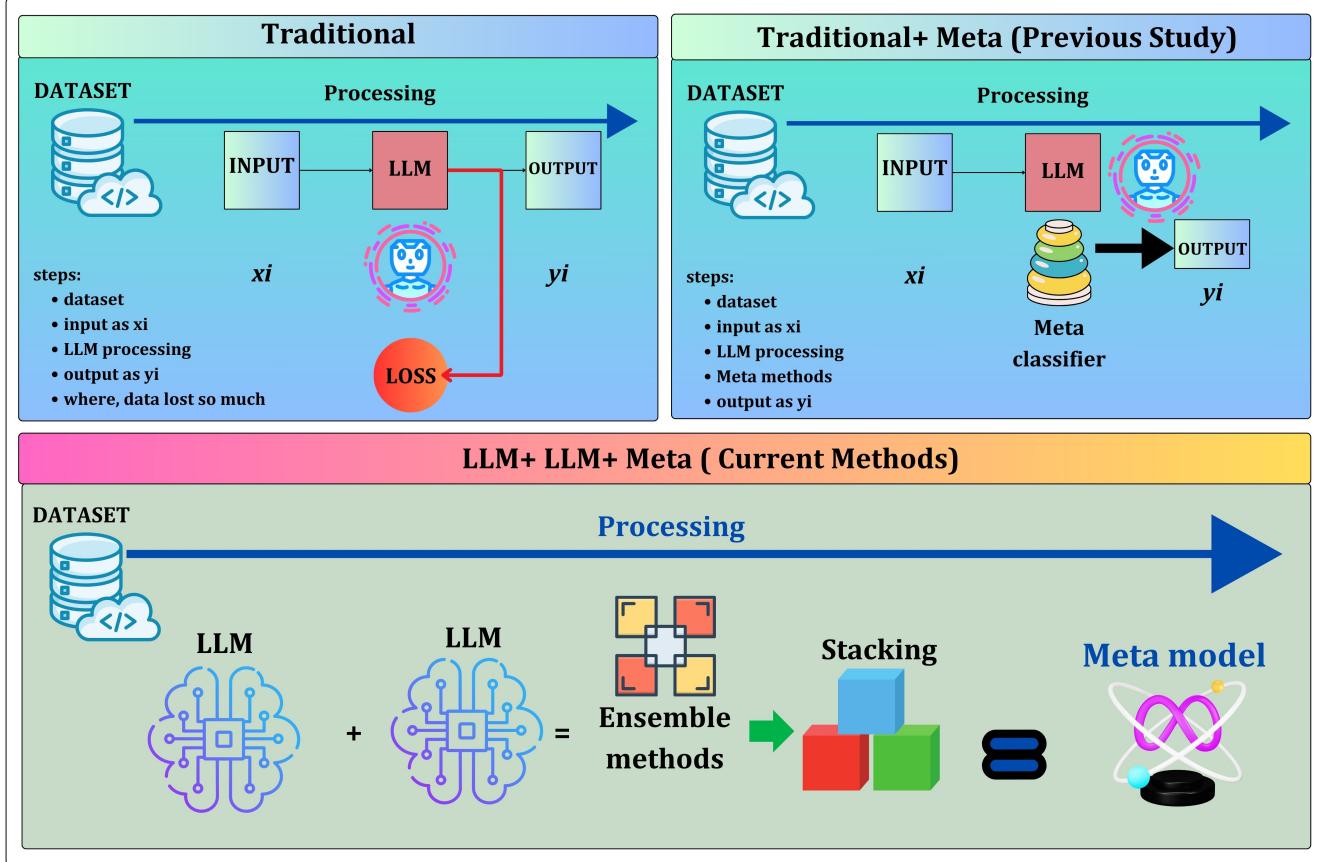


Fig. 1. A Comparative Overview of Vulnerability Detection Techniques. (1) Traditional LLM-based processing, which directly outputs predictions but experiences notable data loss, (2) Traditional+Meta models from previous studies that integrate a meta-classifier to enhance LLM outputs, and (3) the proposed EnStack framework, which leverages an ensemble of multiple LLMs combined through stacking methods. EnStack incorporates a meta-model to further refine predictions, aiming for improved accuracy in vulnerability detection by effectively combining strengths of various LLMs and meta-model architectures.

Vector Machine (SVM) [19], Random Forest (RF) [20], and XGBoost [21].

Our method is based on the premise that integrating models that capture different aspects of code—such as semantic meaning, structural relationships, and cross-modal representations—can lead to better detection performance than using any single model alone. The ensemble stacking technique enables the meta-classifier to learn the optimal way to combine the predictions from the base models, effectively mitigating the shortcomings of individual models and enhancing the overall detection capability.

The main contributions of this paper are as follows:

- **Proposing an ensemble-based stacking framework** that integrates multiple pre-trained large language models (LLMs) with meta-classifiers to enhance vulnerability detection in source code.
- **Conducting a comprehensive evaluation** of the proposed EnStack framework on the Draper VDISC dataset [17], demonstrating its superior performance compared to individual models and existing approaches across accuracy, precision, recall, F1-score, and AUC-score metrics.
- **Performing an ablation study** to analyze the impact

of different model combinations and meta-classifiers on detection performance, providing insights into the effectiveness of the stacking approach and guiding future research on model ensemble strategies.

II. LITERATURE REVIEW

Machine Learning Models in Vulnerability Detection. A vulnerability refers to a weakness in software, often stemming from errors in the code's implementation or design, that creates the potential for a security threat [22]. For instance, accessing an array with an invalid index is a weakness, and when this weakness occurs in a specific part of a program, it becomes a vulnerability. To assist security professionals, known weaknesses are compiled in the Common Weakness Enumeration (CWE) catalog, where each weakness is given a unique ID and detailed explanation. The CVE Program (Common Vulnerabilities and Exposures) keeps a publicly available record of known vulnerabilities, accessible through platforms like the National Vulnerability Database (NVD) [23].

Seas et al. [24] introduced a CNN-based model for identifying vulnerabilities in C source code. They utilized datasets from Draper Labs and the NIST SATE Juliet dataset. The

source code was divided into 91 token categories and vectorized using a binary approach. The model significantly improved precision by reducing false positives while maintaining consistent recall rates.

Wu et al. [25] proposed a graph-based method for classifying vulnerabilities in C/C++ functions. They used version 1.3 of the Juliet Test Suite and represented the code as simplified code property graphs to capture both syntactic and semantic features. Graph Neural Networks (GNNs) and multi-layer perceptrons were employed for learning graph representations and performing binary classification. The model achieved an F1-score of 0.836, outperforming previous CNN and RNN-based methods with F1-scores of 0.805 and 0.803.

Li et al. [15] introduced SySeVR, a deep learning framework designed to detect vulnerabilities in C/C++ code. This approach used both syntax-based (SyVC) [26] and semantics-based (SeVC) [27] vulnerability candidates, with word2vec employed for encoding SeVCs into vector representations. The dataset, containing 126 types of vulnerabilities, was split into 80% training and 20% testing. SySeVR achieved an F1-score of 0.926, significantly surpassing VulDeePecker [14], which scored 0.783. The use of stratified 5-fold cross-validation ensured the robustness of the results.

Large Language Models in Vulnerability Detection. Kim et al. [28] developed VulDeBERT, a BERT-based model fine-tuned on C/C++ source code for detecting specific vulnerabilities. They processed the source code using a method called code gadget. Tested on datasets representing CWE-119 (buffer errors) and CWE-399 (management errors), achieved results with F1-scores of 0.946 and 0.979, outperforming VulDeePecker by a large margin.

Hanif and Maffeis [29] presented VulBERTa, a vulnerability detection model that uses RoBERTa and a novel tokenization method. The model was evaluated using two architectures—VulBERTa-MLP and VulBERTa-CNN—achieving F1-scores of 93.03 and 90.86, respectively, on the VulDeePecker dataset and 43.34 and 57.92 on the Draper dataset.

Fu et al. [30] examined the performance of GPT-3.5 and GPT-4 for vulnerability classification tasks. The GPT models underperformed compared to fine-tuned models like CodeBERT, achieving lower accuracies in both function-level and line-level vulnerability classification tasks.

III. METHODOLOGY

Detecting vulnerabilities in source code is crucial for ensuring software security. Various models have been proposed to address this challenge, but improvements are needed to handle the complex structure of source code effectively. In this study, we employ an ensemble stacking approach that combines the strengths of multiple large language models (LLMs) to improve vulnerability detection. The models used include CodeBERT, GraphCodeBERT, and UniXcoder, and their outputs are combined using an ensemble stacking technique.

A. Problem Formulation

Given a dataset $D = \{(x_i, y_i)\}_{i=1}^n$, where each x_i represents a code snippet and $y_i \in \{0, 1, 2, 3, 4\}$ denotes the vulnerability class label (e.g., CWE categories), the objective is to develop a model that accurately predicts the vulnerability class \hat{y}_i for unseen code snippets.

To achieve this, we employ the EnStack ensemble stacking framework. The framework utilizes a set of base models $\mathcal{M} = \{M_k\}_{k=1}^K$ (e.g., CodeBERT, GraphCodeBERT, UniXcoder), each fine-tuned on the training data D_{train} to detect vulnerabilities independently. Each base model M_k produces an output vector for a given code snippet x_i , typically a probability distribution over the vulnerability classes.

We construct a meta-feature vector \mathbf{z}_i for each sample x_i by concatenating the outputs of all base models:

$$\mathbf{z}_i = [M_1(x_i), M_2(x_i), \dots, M_K(x_i)].$$

A meta-classifier $\mathcal{F}_{\text{meta}}$ is then trained on these meta-features to learn the optimal combination of base model predictions:

$$\hat{y}_i = \mathcal{F}_{\text{meta}}(\mathbf{z}_i).$$

The goal is to find the meta-classifier $\mathcal{F}_{\text{meta}}$ that minimizes the classification error on the validation set D_{val} and performs well on the test set D_{test} . This involves selecting $\mathcal{F}_{\text{meta}}$ from a set of candidate meta-classifiers $\mathcal{C} = \{C_m\}_{m=1}^M$ (e.g., logistic regression, random forest).

B. Dataset and Preprocessing

Dataset Details.

The Draper VDISC dataset is a large-scale collection of over 1.27 million code functions sourced from open-source software projects. These code functions are labeled based on potential vulnerabilities detected through static analysis. The dataset contains five primary vulnerability classes, each mapped to Common Weakness Enumeration (CWE) categories:

- CWE-119: Memory-related vulnerabilities
- CWE-120: Buffer overflow vulnerabilities
- CWE-469: Integer overflow vulnerabilities
- CWE-476: NULL pointer dereference vulnerabilities
- CWE-other: Miscellaneous vulnerabilities

For this study, the dataset is divided into training (80%), validation (10%), and test (10%) sets. The class distribution is imbalanced, with certain vulnerabilities, such as CWE-469 (integer overflow), being significantly less frequent than others like CWE-119 (memory vulnerabilities). Null entries were then removed from the training, validation, and test sets.

Data Balancing.

To address the class imbalance, we employed downsampling to reduce the size of the majority classes, such as CWE-119 and CWE-120, bringing them closer to the size of the minority classes, particularly CWE-469. Downsampling was selected as it effectively reduces the dominance of the majority classes without introducing synthetic data, which could

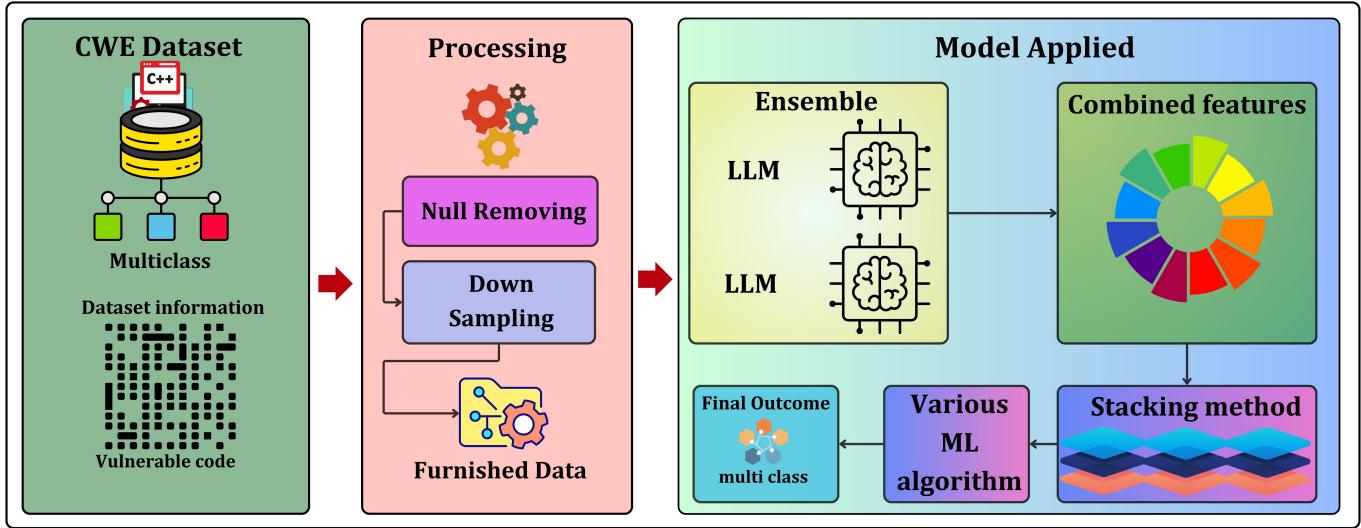


Fig. 2. Methodology Workflow for Enhanced Vulnerability Detection using EnStack Framework. The proposed methodology begins with the CWE dataset, which includes multiclass labels associated with various types of vulnerable code. During the processing phase, data cleansing steps, such as null value removal and down sampling, ensure balanced and representative input data. This furnished data is then fed into an ensemble of large language models (LLMs), where features are extracted and combined to capture diverse vulnerability patterns. These combined features are passed through a stacking framework that employs various machine learning algorithms to enhance model robustness. The stacked model produces a final multiclass prediction, facilitating accurate and refined detection of code vulnerabilities across different classes.

potentially introduce noise or lead to overfitting. Additionally, downsampling is computationally efficient, allowing faster model training while maintaining a representative sample of the different vulnerabilities.

Other techniques, such as oversampling or synthetic data generation were considered but ultimately not used due to the large dataset size. Downsampling was the preferred method as it prevents overfitting and preserves the dataset's natural characteristics. Table I presents the overall distribution of the dataset across the training, test, and validation sets.

TABLE I
CLASS DISTRIBUTION ACROSS TRAINING, VALIDATION, AND TEST SETS

Class Label	CWE Type	Training Samples	Validation Samples	Test Samples
0	CWE-119 (Memory)	5942	1142	1142
1	CWE-120 (Buffer Overflow)	5777	1099	1099
2	CWE-469 (Integer Overflow)	249	53	53
3	CWE-476 (Null Pointer)	2755	535	535
4	CWE-other	5582	1071	1071
Total		20305	3900	3900

C. Fine-Tuning and Training

To enhance the vulnerability detection capabilities of large language models (LLMs) for code-related tasks, we fine-tuned three pre-trained models: CodeBERT, GraphCodeBERT, and UniXcoder. These models were selected for their specialized architectures, which are well-suited to understanding code syntax, semantics, and structure. The fine-tuning process comprised two stages: (a) fine-tuning the models using vulnerability class labels and (b) combining the outputs of multiple LLMs through an ensemble stacking method to further improve classification performance. **Models Used:**

- 1) **CodeBERT [10]**, a model pre-trained on code and natural language, was fine-tuned using the Draper VDISC

dataset. The fine-tuning process optimized the model for multi-class classification using *cross-entropy loss*.

- 2) **GraphCodeBERT [11]** is designed to handle the graph-based structure of code. It was fine-tuned on the same dataset to enhance its ability to detect code vulnerabilities.
- 3) **UniXcoder [12]** was fine-tuned to improve its understanding of the syntactic and semantic structures of code snippets, making it suitable for vulnerability detection.

Hyperparameter Setup.

The training setup for each model is summarized in Table II.

Fine-Tuning using Labels.

For each model, the fine-tuning process involved training the models on the downsampled Draper VDISC dataset using the Cross Entropy Loss function. The input code snippets were tokenized and fed into the model to predict their vulnerability classes. The original labels from the dataset, denoting vulnerability types (CWE categories), were used to guide the training process, minimizing the cross-entropy loss between the predicted outputs \hat{y}_i and the ground truth labels y_i .

D. Ensemble Stacking Method

To further enhance the detection performance, we implemented an ensemble stacking method to combine the outputs of CodeBERT, GraphCodeBERT, and UniXcoder. This approach allowed us to leverage the strengths of each model to improve vulnerability classification.

Model Predictions.

Each model generated a probability distribution over the five vulnerability classes. These distributions, represented as $M_k(x_i)$, were concatenated to form feature vectors. For each

TABLE II
HYPERPARAMETER SETUP FOR FINE-TUNING LLMs AND MACHINE LEARNING MODELS

Hyperparameter	Value
Transformer Model	
Batch Size	16
Epochs	10
Learning Rate	2×10^{-5}
Optimizer	AdamW
Max Token Length	512
Logistic Regression	
Max Iterations	200
Solver	lbfgs
Random Forest	
Number of Estimators	200
Max Depth	10
SVM	
Kernel	RBF
Probability Estimation	True
Random State	42
XGBoost	
Number of Estimators	100
Eval Metric	mlogloss
Learning Rate	0.1
Max Depth	6

code snippet x_i , the model predictions from CodeBERT, GraphCodeBERT, and UniXcoder were combined into a single feature vector.

Meta-Classifier Training.

Four meta-classifiers were tested to combine the output vectors:

- Logistic Regression (LR) [18]
- Random Forest (RF) [20]
- Support Vector Machine (SVM) [19]
- XGBoost [21]

These classifiers were chosen for their ability to handle high-dimensional data and combine diverse outputs from base models. The meta-classifiers were trained on the concatenated output features and were evaluated on the test set.

E. Algorithm

The EnStack framework, outlined in Algorithm 1, is an ensemble stacking approach developed to enhance vulnerability detection accuracy. The process begins by preparing the dataset, where data is split into training, validation, and test subsets, and the training set is balanced through downsampling. Each base model within the ensemble is then fine-tuned on the training data. Meta-features are generated by aggregating predictions from all base models for each data sample, creating a comprehensive meta-feature vector. Multiple meta-classifiers are trained on these meta-features, with the classifier demonstrating the best validation performance selected as the optimal meta-classifier. This selected meta-classifier is finally evaluated on the test set to assess its performance, providing a robust solution for effective vulnerability detection.

Algorithm 1 EnStack: Ensemble Stacking Framework for Vulnerability Detection

Require: Dataset $D = \{(x_i, y_i)\}_{i=1}^n$; Base models $\mathcal{M} = \{M_k\}_{k=1}^K$; Meta-classifiers $\mathcal{C} = \{C_m\}_{m=1}^M$

Ensure: Optimal meta-classifier $\mathcal{F}_{\text{meta}}$ and performance metrics

- 1: **Data Preparation:**
- 2: Split D into D_{train} , D_{val} , and D_{test}
- 3: Balance D_{train} via downsampling
- 4: **Base Model Training:**
- 5: **for** each base model $M_k \in \mathcal{M}$ **do**
- 6: Fine-tune M_k on D_{train}
- 7: **end for**
- 8: **Meta-feature Generation:**
- 9: **for** each sample $(x_j, y_j) \in D_{\text{val}}$ **do**
- 10: Compute meta-feature vector:
- 11: $\mathbf{z}_j = [M_1(x_j), M_2(x_j), \dots, M_K(x_j)]$
- 12: **end for**
- 13: **Meta-classifier Training and Selection:**
- 14: **for** each meta-classifier $C_m \in \mathcal{C}$ **do**
- 15: Train C_m on $\{(\mathbf{z}_j, y_j)\}_j$
- 16: Evaluate C_m on D_{val}
- 17: **end for**
- 18: Select optimal $\mathcal{F}_{\text{meta}}$ with best validation performance
- 19: **Evaluation on Test Set:**
- 20: **for** each sample $(x_i, y_i) \in D_{\text{test}}$ **do**
- 21: Compute $\mathbf{z}_i = [M_1(x_i), M_2(x_i), \dots, M_K(x_i)]$
- 22: Predict $\hat{y}_i = \mathcal{F}_{\text{meta}}(\mathbf{z}_i)$
- 23: **end for**
- 24: Compute performance metrics on D_{test}
- 25: **return** Optimal meta-classifier $\mathcal{F}_{\text{meta}}$ and performance metrics

IV. EXPERIMENT

A. Hardware and Software Setup

All experiments were conducted using the NVIDIA Tesla P100 GPU. We used the PyTorch deep learning framework for model implementation and Hugging Face’s transformers library to fine-tune pre-trained models such as CodeBERT, GraphCodeBERT, and UniXcoder. For the ensemble stacking technique, scikit-learn was employed to implement meta-classifiers, including Logistic Regression, Random Forest, Support Vector Machine (SVM), and XGBoost.

B. Baselines

We compare our model with several baselines, including transformer-based models such as CodeBERT, GraphCodeBERT, and UniXcoder. Additionally, we included Attention LSTM [31] as a non-transformer baseline model to highlight the performance benefits of pre-trained language models.

Evaluation Metrics. The models’ performance was evaluated using several metrics, including:

- Accuracy: Proportion of correct predictions.
- Precision, Recall, and F1-Score: Precision represents the accuracy of positive predictions, recall measures the

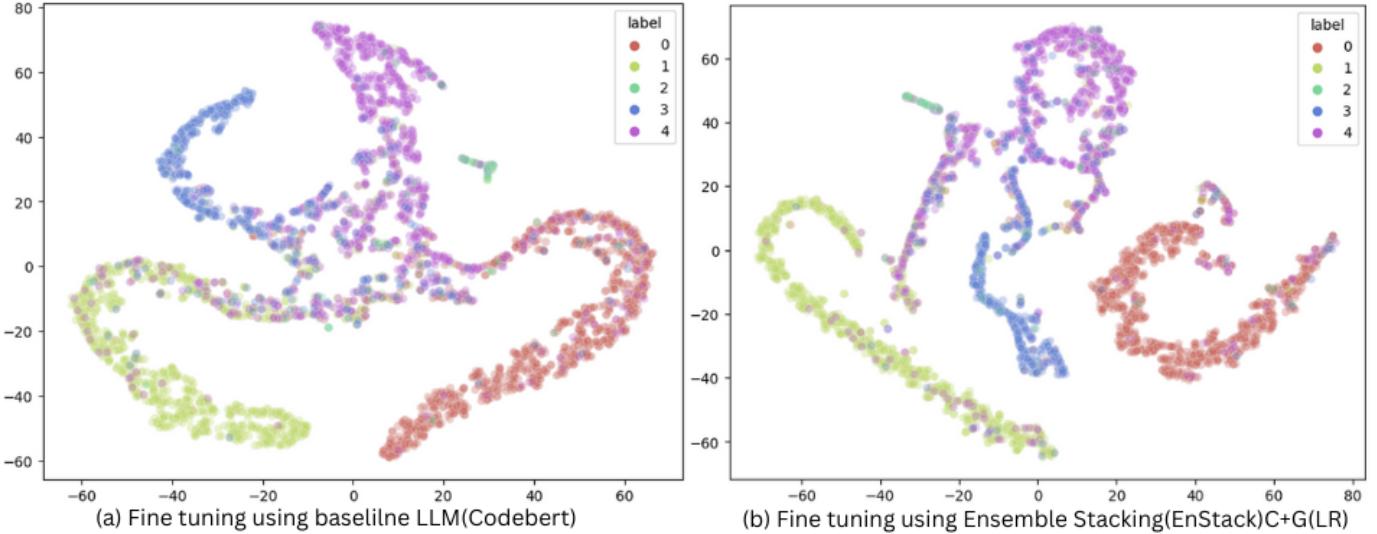


Fig. 3. t-SNE visualization of latent space representations for CWE categories in vulnerability detection. (a) baseline model fine-tuned using CodeBERT, exhibiting class overlap and reduced inter-cluster separation; (b) enhanced representations through ensemble stacking (EnStack) with CodeBERT and GraphCodeBERT (C+G) followed by logistic regression (LR), demonstrating improved cluster formation and class separability.

coverage of actual positives, and the F1-Score balances both precision and recall.

- AUC-Score: The Area Under the ROC Curve (AUC) was computed to assess the models' ability to discriminate between vulnerability classes.

C. Results and Analysis

Individual Model Performance.

We first evaluated the individual models, including Attention LSTM, CodeBERT (C), GraphCodeBERT (G), and UniXcoder (U), using accuracy, precision, recall, F1-score, and AUC-score. These results, summarized in Table III, highlight the distinct strengths and limitations of each model in detecting vulnerabilities within source code.

The non-transformer baseline, Attention LSTM, achieved an accuracy of 73.00%, reflecting its ability to capture sequential dependencies but also its limitations in handling the more intricate semantic and structural aspects of code. The model's low AUC-score of 77.54% further underscores its struggles with distinguishing between vulnerability classes effectively.

In contrast, CodeBERT, a transformer-based model, demonstrated significant improvement with an accuracy of 78.51% and an F1-score of 77.98%. Its ability to process both natural language and programming code semantics contributes to these gains, though its relatively lower recall (78.51%) indicates that it might not fully capture structural nuances essential for vulnerability detection.

GraphCodeBERT (G), which integrates both code semantics and structural information, outperformed CodeBERT with an accuracy of 80.05%, an F1-score of 79.86%, and a higher AUC-score of 93.36%. The model's enhanced understanding of code flow and data dependencies makes it more robust for identifying vulnerabilities.

UniXcoder (U) achieved the highest performance among the individual models, with an accuracy of 81.54% and an F1-score of 81.49%. UniXcoder's ability to represent both the syntactic and semantic properties of code through cross-modal learning allowed it to surpass the other models. Its AUC-score of 93.80%, the highest among all models, emphasizes its superior capacity to distinguish between different classes of vulnerabilities.

Stacking with a Single LLM and Meta-Classifier. We explored the impact of stacking a single LLM with traditional machine learning meta-classifiers, such as Logistic Regression (LR), Support Vector Machine (SVM), Random Forest (RF), and XGBoost. The results demonstrate that stacking improves performance over individual models. For example, stacking UniXcoder with SVM resulted in an accuracy of 81.36% and an F1-score of 81.89%, compared to the standalone UniXcoder model. Similarly, stacking GraphCodeBERT with Logistic Regression achieved an accuracy of 80.13%, further illustrating the benefits of combining an LLM with a meta-classifier.

These findings suggest that even without leveraging multiple LLMs, a single LLM can be effectively enhanced by traditional classifiers, refining its predictions and increasing its discriminatory ability. Among the meta-classifiers, SVM consistently delivered strong performance due to its capability of handling high-dimensional feature spaces.

Ensemble Stacking of Multiple LLMs with Meta Classifiers.

The most notable performance improvements, however, were observed when combining multiple LLMs in an ensemble stacking approach. Stacking GraphCodeBERT and UniXcoder (G+U) with SVM delivered the highest accuracy of 82.36% and the best F1-score of 82.28%. This result underscores the effectiveness of combining the structural insights from

TABLE III

COMPREHENSIVE EVALUATION OF INDIVIDUAL MODELS, SINGLE STACKED MODELS, AND ENSEMBLE STACKING METHODS ACROSS KEY PERFORMANCE METRICS: **ACCURACY**, **PRECISION**, **RECALL**, **F1-SCORE**, AND **AUC-SCORE**. ALL RESULTS ARE REPORTED AS PERCENTAGES, WITH ↑ INDICATING HIGHER VALUES ARE BETTER.

Model	Accuracy (%) ↑	Precision (%) ↑	Recall (%) ↑	F1-Score (%) ↑	AUC-Score (%) ↑
Individual Models					
Attention LSTM	73.00	72.97	73.00	72.95	77.54
CodeBERT (C)	78.51	77.85	78.51	77.98	92.16
GraphCodeBERT (G)	80.05	79.92	80.05	79.86	93.36
UniXcoder (U)	81.54	81.96	81.54	81.49	93.80
Stacked Models with Individual Base Models					
Stacking C (LR)	79.00	78.71	79.00	78.62	91.26
Stacking C (RF)	78.69	79.10	78.69	78.70	91.64
Stacking C (SVM)	78.74	79.85	78.74	78.64	88.55
Stacking C (XGBoost)	77.26	77.14	77.26	77.12	90.99
Stacking G (LR)	80.13	80.12	80.13	79.88	92.39
Stacking G (RF)	79.79	80.01	79.79	79.63	92.11
Stacking G (SVM)	80.00	80.82	80.00	80.04	89.60
Stacking G (XGBoost)	78.46	78.47	78.46	78.36	90.36
Stacking U (LR)	81.51	81.70	81.51	81.36	92.49
Stacking U (RF)	81.44	81.71	81.44	81.35	92.42
Stacking U (SVM)	81.31	81.89	81.31	81.33	90.76
Stacking U (XGBoost)	80.08	79.90	80.08	79.94	92.57
Ensemble Stacking Models					
Ensemble Stacking C+G (LR)	81.13	81.10	81.13	80.90	92.93
Ensemble Stacking C+G (RF)	81.56	81.87	81.56	81.44	92.32
Ensemble Stacking C+G (SVM)	81.46	81.77	81.46	81.40	89.96
Ensemble Stacking C+G (XGBoost)	80.28	80.22	80.28	80.06	91.31
Ensemble Stacking G+U (LR)	82.36	82.59	82.36	82.21	92.85
Ensemble Stacking G+U (RF)	82.28	82.45	82.28	82.13	92.45
Ensemble Stacking G+U (SVM)	82.36	82.85	82.36	82.28	90.53
Ensemble Stacking G+U (XGBoost)	80.67	80.47	80.67	80.46	92.28

GraphCodeBERT with the cross-modal learning of UniXcoder, allowing the meta-classifier to leverage both models' strengths. The highest AUC-score of 92.85% was achieved by stacking G+U with Logistic Regression, indicating the model's superior discriminatory power across classes.

D. Ablation Study

Impact of Model Combinations in Stacking. We conducted a detailed analysis of various model combinations in our stacking ensembles, focusing on CodeBERT (C), GraphCodeBERT (G), and UniXcoder (U). As illustrated in Figure 4 (a), stacking GraphCodeBERT and UniXcoder (Stacking G+U) consistently achieved superior performance compared to other combinations across different meta-classifiers. Specifically, stacking G+U with SVM attained the highest accuracy

of 82.36% and an F1-score of 82.28%, significantly outperforming stacking combinations like C+G, which reached a lower accuracy of 81.46% with SVM. This demonstrates that combining GraphCodeBERT's structural insights with UniXcoder's syntactic-semantic understanding results in a more comprehensive feature set, optimizing the performance of stacking ensembles for vulnerability detection. These results emphasize the importance of selecting complementary base models to maximize stacking effectiveness.

Impact of Meta-Classifier Choice in Stacking. In addition to model combinations, we investigated the impact of different meta-classifiers, including Logistic Regression (LR), Random Forest (RF), Support Vector Machine (SVM), and XGBoost, on the performance of stacking ensembles. As shown in Figure 4 (b), the choice of meta-classifier significantly affects perfor-

mance. Stacking G+U with SVM yielded the highest F1-score (82.28%) and precision (82.85%), while Logistic Regression achieved the highest AUC-Score (92.85%) in the same configuration. In contrast, XGBoost consistently underperformed, with an accuracy of only 80.67% when stacking G+U. This suggests that linear classifiers such as SVM and LR are better suited for leveraging the complementary features of base models in this task. More complex classifiers like XGBoost may introduce unnecessary complexity, which does not translate into better performance, further validating the suitability of simpler, interpretable classifiers for this application.

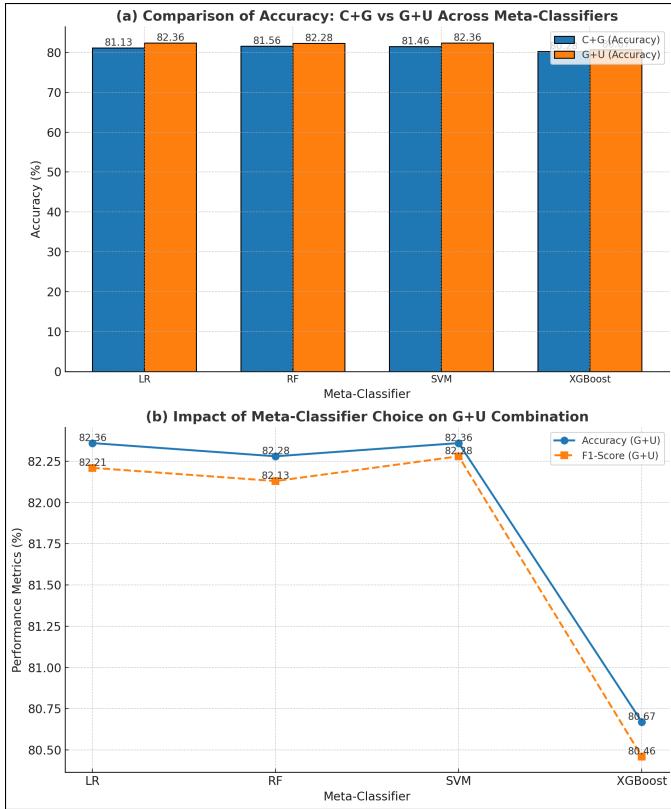


Fig. 4. Ablation study results showcasing the impact of model combinations and meta-classifier choice in stacking ensembles. (a) Performance comparison of model combinations (C+G vs. G+U) across meta-classifiers (LR, RF, SVM, and XGBoost). **G+U consistently outperforms C+G**, with SVM achieving the highest accuracy of **82.36%**. (b) Detailed analysis of meta-classifier performance on the G+U combination, highlighting the dominance of linear classifiers (SVM and LR) in capturing complementary features. **XGBoost underperforms**, indicating added complexity may not yield better results for this task.

E. Discussion

Ensemble Stacking for Vulnerability Detection. Our experimental results demonstrate that the ensemble stacking approach, which combines the strengths of multiple transformer-based models, outperforms individual models in detecting vulnerabilities in source code. The individual models, such as CodeBERT, GraphCodeBERT, and UniXcoder, achieved respectable accuracy and F1-scores, with UniXcoder performing the best at 81.54% accuracy. However, combining

these models through both weighted aggregation and stacking significantly enhanced the overall performance. The ensemble methods—particularly Stacking G+U with Logistic Regression (LR) and Support Vector Machine (SVM)—achieved the highest accuracy of 82.36%, showcasing the value of integrating diverse model strengths. SVM further excelled by attaining the highest F1-score of 82.28%, while Logistic Regression demonstrated superior discriminative power with an AUC score of 92.85%. These results highlight the effectiveness of ensemble stacking in improving vulnerability detection, especially by leveraging complementary features such as syntactic-semantic cross-modal understanding from UniXcoder and structural insights from GraphCodeBERT.

Limitations. The EnStack framework has several limitations. First, the dataset suffers from severe class imbalance, with vulnerabilities like CWE-469 (Integer Overflow) being significantly underrepresented, reducing the model’s ability to generalize effectively. Downsampling used to mitigate this imbalance further reduced the dataset size, compounding generalization challenges. Second, reliance on the Draper VDISC dataset restricts applicability, as it focuses on specific vulnerabilities and programming languages, raising concerns about generalization to other datasets and languages that may exhibit distinct vulnerability patterns. Third, the ensemble stacking approach imposes substantial computational overhead, with the integration of pre-trained models (CodeBERT, GraphCodeBERT, UniXcoder) and meta-classifier training requiring significant resources, limiting scalability for real-time or large-scale applications.

V. CONCLUSION

In this study, we proposed an ensemble stacking approach that combines the outputs of transformer-based models like CodeBERT, GraphCodeBERT, and UniXcoder with meta classifiers to improve vulnerability detection in source code. By fine-tuning each model on the Draper VDISC dataset and utilizing both ensemble and stacking methods, we demonstrated that the ensemble stacking approach significantly enhances the performance of individual models. Among the meta-classifiers, Logistic Regression (LR) and Support Vector Machine (SVM) produced the best results, achieving an accuracy of 82.36% and a high AUC score of 92.85%. These results validate the effectiveness of leveraging diverse models that capture different aspects of code—syntactic, semantic, and structural—to improve classification of vulnerabilities in a multi-class setting.

Future Work. While our EnStack framework demonstrated promising results, there is still room for improvement in addressing the challenges posed by underrepresented classes in vulnerability detection. In future work, we plan to experiment with multiple vulnerability datasets to evaluate the generalizability and robustness of our models across diverse codebases. By incorporating a wider range of datasets, we aim to capture a broader spectrum of vulnerability types and patterns, enhancing the models’ ability to detect rare and complex issues.

Furthermore, we intend to explore generative models such as LLaMA and Mistral to enhance code understanding and vulnerability detection. These models, with their powerful generative capabilities, can be leveraged for domain-specific pre-training, allowing for more comprehensive detection of nuanced vulnerabilities. Additionally, we aim to investigate how generative models can provide richer contextual information about code, potentially generating synthetic code snippets to augment training data and improve model robustness. Exploring transfer learning with these generative models will further allow us to tailor detection strategies, leading to more accurate and nuanced vulnerability identification.

ACKNOWLEDGMENT

This work was partially supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU

REFERENCES

- [1] Y. Perwej, S. Q. Abbas, J. P. Dixit, N. Akhtar, and A. K. Jaiswal, "A systematic literature review on the cyber security," *International Journal of scientific research and management*, vol. 9, no. 12, pp. 669–710, 2021.
- [2] S. M. Ghaffarian and H. R. Shahriari, "Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey," *ACM computing surveys (CSUR)*, vol. 50, no. 4, pp. 1–36, 2017.
- [3] P. P. F. Balbin, J. C. Barker, C. K. Leung, M. Tran, R. P. Wall, and A. Cuzzocrea, "Predictive analytics on open big data for supporting smart transportation services," *Procedia computer science*, vol. 176, pp. 3009–3018, 2020.
- [4] M. J. H. Faruk, H. Shahriar, M. Valero, F. L. Barsha, S. Sobhan, M. A. Khan, M. Whitman, A. Cuzzocrea, D. Lo, A. Rahman, et al., "Malware detection and prevention using artificial intelligence techniques," in *2021 IEEE international conference on big data (big data)*, pp. 5369–5377, IEEE, 2021.
- [5] M. Masum, H. Shahriar, H. Haddad, M. J. H. Faruk, M. Valero, M. A. Khan, M. A. Rahman, M. I. Adnan, A. Cuzzocrea, and F. Wu, "Bayesian hyperparameter optimization for deep neural network-based network intrusion detection," in *2021 IEEE International Conference on Big Data (Big Data)*, pp. 5413–5419, IEEE, 2021.
- [6] C. K. Leung, P. Braun, and A. Cuzzocrea, "Ai-based sensor information fusion for supporting deep supervised learning," *Sensors*, vol. 19, no. 6, p. 1345, 2019.
- [7] R. C. Camara, A. Cuzzocrea, G. M. Grasso, C. K. Leung, S. B. Powell, J. Souza, and B. Tang, "Fuzzy logic-based data analytics on predicting the effect of hurricanes on the stock market," in *2018 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, pp. 1–8, IEEE, 2018.
- [8] P. Howlader, K. K. Pal, A. Cuzzocrea, and S. M. Kumar, "Predicting facebook-users' personality based on status and linguistic features via flexible regression analysis techniques," in *Proceedings of the 33rd annual ACM symposium on applied computing*, pp. 339–345, 2018.
- [9] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.
- [10] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al., "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.
- [11] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, et al., "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.
- [12] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixer: Unified cross-modal pre-training for code representation," *arXiv preprint arXiv:2203.03850*, 2022.
- [13] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE symposium on security and privacy*, pp. 590–604, IEEE, 2014.
- [14] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *arXiv preprint arXiv:1801.01681*, 2018.
- [15] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "Sysevr: A framework for using deep learning to detect software vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 4, pp. 2244–2258, 2021.
- [16] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *Advances in neural information processing systems*, vol. 32, 2019.
- [17] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "Automated vulnerability detection in source code using deep representation learning," in *2018 17th IEEE international conference on machine learning and applications (ICMLA)*, pp. 757–762, IEEE, 2018.
- [18] D. W. Hosmer Jr, S. Lemeshow, and R. X. Sturdivant, *Applied logistic regression*. John Wiley & Sons, 2013.
- [19] D. A. Pisner and D. M. Schnyer, "Support vector machine," in *Machine learning*, pp. 101–121, Elsevier, 2020.
- [20] R. Genuer, J.-M. Poggi, R. Genuer, and J.-M. Poggi, *Random forests*. Springer, 2020.
- [21] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pp. 785–794, 2016.
- [22] B. Yuce, P. Schaumont, and M. Wittman, "Fault attacks on secure embedded software: Threats, design, and evaluation," *Journal of Hardware and Systems Security*, vol. 2, pp. 111–130, 2018.
- [23] H. KEKÜL, B. ERGEN, and H. ARSLAN, "Comparison and analysis of software vulnerability databases," *International Journal of Engineering and Manufacturing*, vol. 12, no. 4, p. 1, 2022.
- [24] C. Seas, G. Fitzpatrick, J. A. Hamilton, and M. C. Carlisle, "Automated vulnerability detection in source code using deep representation learning," in *2024 IEEE 14th Annual Computing and Communication Workshop and Conference (CCWC)*, pp. 0484–0490, IEEE, 2024.
- [25] Y. Wu, J. Lu, Y. Zhang, and S. Jin, "Vulnerability detection in c/c++ source code with graph representation learning," in *2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC)*, pp. 1519–1524, IEEE, 2021.
- [26] G. Lin, S. Wen, Q.-L. Han, J. Zhang, and Y. Xiang, "Software vulnerability detection using deep neural networks: a survey," *Proceedings of the IEEE*, vol. 108, no. 10, pp. 1825–1848, 2020.
- [27] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, Y. Zhang, Z. Chen, and D. Li, "Vuldeelocator: A deep learning-based system for detecting and locating software vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, 2021.
- [28] S. Kim, J. Choi, M. E. Ahmed, S. Nepal, and H. Kim, "Vuldebert: A vulnerability detection system using bert," in *2022 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 69–74, IEEE, 2022.
- [29] H. Hanif and S. Mafeis, "Vulberta: Simplified source code pre-training for vulnerability detection," in *2022 International joint conference on neural networks (IJCNN)*, pp. 1–8, IEEE, 2022.
- [30] M. Fu, C. K. Tantithamthavorn, V. Nguyen, and T. Le, "Chatgpt for vulnerability detection, classification, and repair: How far are we?," in *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 632–636, IEEE, 2023.
- [31] Y. Wang, M. Huang, X. Zhu, and L. Zhao, "Attention-based lstm for aspect-level sentiment classification," in *Proceedings of the 2016 conference on empirical methods in natural language processing*, pp. 606–615, 2016.