International Journal of Future Engineering Innovations

Impact of Full Fine-Tuning on Performance of Bert-Family Models in Source Code Vulnerability Detection Task

Nin Ho Le Viet [1], Tuan Nguyen Kim [2*], Cuong Dang Van [3], Chieu Ta Quang [4]
[1] School of Computer Science, Duy Tan University, Danang, Vietnam
[2] Phenikaa School of Computing, Phenikaa University, Hanoi, Vietnam
[3, 4] Faculty of Computer Science and Engineering, Thuyloi University, Hanoi, Vietnam

* Corresponding Author: **Tuan Nguyen Kim**

Article Info

ISSN (online): 3049-1215
Impact Factor (RSIF): 8.25
Volume: 03
Issue: 03
May-June 2026
Received: 03-03-2026
Accepted: 01-04-2026
Published: 02-05-2026
Page No: 26-35

Abstract
Deep learning-based approaches for source code vulnerability detection have gained considerable attention in software security. Among existing pretrained language models, BERT, CodeBERT, and GraphCodeBERT are widely adopted; however, their performance under a unified fine-tuning strategy has not been systematically examined. This study conducts a comparative assessment of the three models in the context of vulnerability detection, employing a full fine-tuning setup in which all model parameters are updated on task-specific training data. Experiments are carried out on three representative datasets, Devign, Big-Vul, MegaVul, and Juliet, covering diverse vulnerability patterns and levels of semantic complexity. The results indicate that GraphCodeBERT consistently delivers the strongest performance, particularly for vulnerabilities governed by data-flow semantics; CodeBERT shows stable behavior on pattern-oriented cases, whereas BERT lags behind due to its lack of pretraining on programming languages. The empirical findings highlight how architectural differences and pretraining objectives influence downstream vulnerability detection. This work establishes a reliable and reproducible baseline for subsequent studies in a broader investigation of fine-tuning strategies for language models in code analysis.

DOI: https://doi.org/10.54660/IJFEI.2026.3.3.26-35

Keywords: Source code vulnerability detection, Pretraining, Full fine-tuning, BERT, CodeBERT, GraphCodeBERT

1. Introduction

Detecting vulnerabilities in source code plays a central role in software security, especially as modern systems continue to grow in scale and complexity. Pretrained language models [1], such as BERT [2], CodeBERT [3], and GraphCodeBERT [4], have introduced new possibilities, enabling learning mechanisms that capture the semantic and structural properties of programs more effectively than earlier approaches. Despite their widespread adoption, a systematic and fair comparison of these three models under a unified training strategy, namely full fine-tuning, has not yet been established.

In addition, most existing work evaluates models on a single dataset, which limits the ability to draw conclusions about their generalization capabilities. The three datasets commonly used in vulnerability research, Devign [5], Big-Vul [6], Juliet [7], and MegaVul [8], reflect markedly different categories of weaknesses, ranging from pattern-based flaws to logic-level vulnerabilities [9] and real-world CVE instances [10]. Their diversity makes them well suited for a comprehensive assessment of model performance.

Motivated by these gaps, this study compares the behavior of BERT, CodeBERT, and GraphCodeBERT under full fine-tuning, evaluating them across all three representative datasets. These datasets collectively cover real-world project code, large-scale CVE-linked vulnerabilities, and synthetic CWE-style patterns. Our goal is to establish a reliable baseline, highlight performance differences among the models, and provide a clearer foundation for subsequent research on advanced fine-tuning strategies for code analysis.

The contributions of this work include: (i) Developing a unified and comprehensive evaluation framework for comparing BERT, CodeBERT, and GraphCodeBERT under identical full fine-tuning settings; (ii) Conducting experiments on Devign, Big-Vul, and Juliet to examine model performance and generalization behavior; and (iii) Offering quantitative and qualitative analyses
that clarify how architectural choices and pretraining data shape the distinct behaviors of each model.

Taken together, these observations highlight the need for a deeper examination of how different pretrained architectures behave when fully fine-tuned for vulnerability detection. To lay the groundwork for this analysis, the next section reviews the core characteristics of BERT, CodeBERT, and GraphCodeBERT, along with the pretraining objectives and representational assumptions that shape their behavior during downstream adaptation.

Unlike studies that aim to introduce novel architectures or achieve state-of-the-art perfor-mance, this work focuses on conducting a controlled comparative analysis within the BERT-family under a full fine-tuning strategy. By restricting the investigation to BERT, CodeBERT, and GraphCodeBERT, we aim to isolate the impact of pretraining objectives and structural information (e.g., data-flow awareness) under a consistent experimental setting. The primary objective of this study is therefore to establish a clear, reproducible, and practically meaningful performance baseline on widely used vulnerability detection datasets.

2. Background

Pretrained language models based on the Transformer architecture [11] have significantly advanced source code analysis by enabling richer semantic and syntactic representations. In vulnerability detection, these pretrained representations directly influence optimization behavior during fine-tuning [12], affecting convergence stability and overall performance. Therefore, understanding their architectural design and pretraining characteristics is essential for interpreting experimental outcomes.

This study examines three BERT-family models, BERT, CodeBERT, and GraphCodeBERT, which represent increasing levels of code awareness. By comparing them under a unified full fine-tuning setting, we aim to clarify how pretraining shapes adaptation behavior and contributes to vulnerability detection performance.

2.1. Characteristics of BERT-family Models

This study considers three representative models within the BERT family, BERT, CodeBERT, and GraphCodeBERT, which differ primarily in their pretraining data and the level of structural information they encode.

- BERT is pretrained on natural language using objectives such as masked language modeling and next sentence prediction. While effective for general semantic understanding, it does not explicitly capture programming-specific structures such as syntax rules or variable dependencies. As a result, when applied to source-code tasks, BERT must adapt from a representation space that is not aligned with program semantics, which limits its effectiveness in vulnerability detection. Nevertheless, BERT remains an important reference baseline, allowing us to quantify the impact of domain-specific pretraining.
- CodeBERT extends BERT by incorporating both natural language and programming language data during pretraining. Its training objectives include masked language modeling and replaced token detection, enabling the model to better capture programming syntax and common coding patterns. This reduces the domain gap observed in BERT and leads to more stable adaptation when fine-tuned for code-related tasks. However, CodeBERT primarily models code as token sequences and does not explicitly encode deeper structural relationships such as data-flow dependencies.
- GraphCodeBERT further enhances code understanding by integrating data-flow information into the pretraining process. In addition to token-level representations, it learns relationships between variables through data flow graphs, allowing it to capture how values propagate within a program. This richer structural awareness makes GraphCodeBERT more effective in detecting vulnerabilities that depend on semantic interactions or variable dependencies, particularly in real-world code.

Taken together, these models represent increasing levels of code awareness: BERT provides a general-language baseline, CodeBERT introduces programming-oriented representations, and GraphCodeBERT incorporates explicit structural information. This progression forms the basis for analyzing how pretraining influences performance under full fine-tuning.

2.2. Optimization Gradients and Full Fine-Tuning in Transformer Models

In deep learning [13], the optimization gradient represents how sensitively the loss function responds to each model parameter. It indicates both the direction and the magnitude of adjustment required to reduce the error during training [14].

A common update rule used in Transformer-based models takes the form: $\theta' = \theta - \eta \nabla_\theta L$, where θ denotes the current parameter set, θ' the updated parameters, η the learning rate, and $\nabla_\theta L$ the gradient of the loss with respect to each parameter. Because the gradient governs how quickly and how stably a model moves toward an optimal solution, its behavior plays a central role in determining whether a pretrained model can effectively adapt to a new task.

This concept becomes particularly critical in the context of full fine-tuning, where every parameter of the model is updated throughout training. Unlike parameter-efficient approaches such as Adapter tuning [15] or LoRA [16], full fine-tuning enables a complete reshaping of the model's representation space, from the lower layers that capture embeddings and early attention patterns to the higher layers responsible for semantic and logical reasoning. When a model has been pretrained on a domain that differs substantially from the target task, its gradients tend to be larger and more volatile, which can slow convergence or even destabilize training. In contrast, a model whose pretraining is closer to the target domain typically exhibits more controlled gradients and adapts more quickly and reliably.

In this study, full fine-tuning is chosen because it reveals the intrinsic adaptability of each BERT-family model. Instead of masking architectural limitations through restricted update strategies, this approach allows gradients to freely modify all layers, making the differences between models visible not only in their final performance but also in their optimization behavior. This provides a clearer view of the influence of pretraining: BERT exhibits large, fluctuating gradients due to

its natural-language-only pretraining; CodeBERT shows steadier gradients thanks to its NL-PL corpus; and GraphCodeBERT achieves the most directed gradient flow due to its integration of both token sequences and data-flow graphs.

For these reasons, full fine-tuning [15] forms the methodological backbone of this work, enabling an objective comparison among the three models and highlighting the role of specialized pretraining in source-code vulnerability detection.

2.3. The Relationship Between Pretraining, Optimization Gradients, and Fine-Tuning Performance

In Transformer-based models, pretraining serves as the stage where the initial representation space is shaped. This space encodes what the model has learned about its training domain, from surface-level syntactic patterns to deeper semantic regularities. When the model is later adapted to a downstream task, the suitability of this learned representation directly influences the gradients produced during optimization. If the pretrained space is already aligned with the nature of the target task, the resulting gradients tend to be moderate, stable, and directionally coherent. By contrast, when the pretrained domain is far removed from the new objective, gradients often become larger and more erratic, making convergence more difficult. This divergence between human-aligned reasoning and model-driven optimization behavior reflects broader decision-making conflicts observed in human–AI systems [15].

These gradients, in turn, govern how quickly and how reliably a model converges during fine-tuning. Under full fine-tuning, every parameter is exposed to gradient updates, which means differences introduced during pretraining manifest clearly in the model's optimization dynamics. A model with well-behaved gradients typically converges faster and with fewer oscillations, whereas one with noisy or oversized gradients may progress slowly, lose stability, or drift away from previously learned useful representations. Ultimately, the convergence behavior during optimization determines the model's fine-tuning performance: pretrained models whose representation space closely matches the target domain generally achieve higher accuracy and maintain stable learning curves, while those with a large domain mismatch may struggle to adapt, resulting in lower performance or heightened sensitivity to hyperparameters.

In our study, this relationship forms the conceptual backbone for comparing BERT, CodeBERT, and GraphCodeBERT. Although the three architectures share the same Transformer encoder structure, their pretraining corpora and objectives differ substantially, leading to distinct gradient behaviors during full fine-tuning and, consequently, notable differences in detection performance on source-code vulnerabilities.

Taken together, these theoretical considerations clarify why BERT, CodeBERT, and GraphCodeBERT, despite belonging to the same model family, exhibit divergent representation spaces and optimization dynamics shaped by their pretraining strategies. Such differences translate into varying levels of stability, convergence speed, and adaptability when full fine-tuning [15] is applied to the vulnerability-detection task. A controlled experimental setup with unified training procedures is therefore essential to isolate the true impact of pretraining on gradient behavior and final model performance.

Building on this foundation, the next section outlines the experimental design, including preprocessing, training configuration, and evaluation metrics, ensuring a fair comparison among the three models and enabling empirical validation of the hypotheses established in the Background section.

3. Experimental Setup

The experimental component of this study evaluates three pretrained models that represent different levels of source-code understanding: BERT, CodeBERT, and GraphCodeBERT. All models are trained using full fine-tuning, meaning that every parameter in the network is updated directly with respect to the loss function of the vulnerability-detection task. This configuration aligns with our objective of examining each model's intrinsic ability to adapt, without relying on parameter-efficient techniques such as LoRA or Adapter modules. By avoiding these restricted fine-tuning strategies, we ensure a fair comparison that highlights the true representational capacity and learning behavior of each model.

To guide the experimental design and evaluation, this study is structured around the following research questions: (i) RQ1: What is the impact of full fine-tuning on the performance of different BERT-family models in source code vulnerability detection? (ii) RQ2: Under what conditions does structural information, particularly data-flow awareness, contribute to improved detection performance? (iii) RQ3: How consistent are these performance differences across commonly used vulnerability detection datasets?

3.1. Experimental Setup

- **Experimental datasets**

To provide a comprehensive evaluation and assess generalization behavior, this study employs four datasets that represent distinct types of vulnerabilities and varying levels of complexity: (i) Devign [5] consists of function-level code extracted from real-world projects and includes a wide range of logic-related vulnerabilities, requiring models to capture deeper program semantics; (ii) Big-Vul [6] is a large-scale dataset annotated with CVE information, containing noisy and heterogeneous real-world code, making it well suited for testing model robustness under practical conditions; (iii) Juliet Test Suite [7] is a synthetic benchmark organized around standardized CWE patterns; its clean structure and low noise make it appropriate for evaluating pattern-oriented vulnerability detection; (iv) MegaVul [8] is a more recent large-scale dataset reflecting modern and diverse vulnerability patterns, providing a more realistic setting for evaluating model generalization under distribution shift.

For each dataset, we adopt a stratified random split into training, validation, and test sets to ensure balanced class distribution across splits. The test set is strictly separated from the training and validation data, and no sample overlap is allowed between splits. This protocol is applied consistently across Devign, Big-Vul, Juliet, and MegaVul datasets to ensure fair comparison among models.

The selection of these datasets aligns closely with the objectives of this work: To examine model performance across diverse scenarios and to clearly identify the strengths and limitations of each approach under a full fine-tuning setting.

- **Full Fine-Tuning Configuration**

All models are trained under a unified experimental configuration to ensure fairness in comparison. The optimization setup employs AdamW, with a batch size of 16, and training runs for 5 to 10 epochs depending on dataset size. The maximum sequence length is fixed at 256 tokens, with a warmup ratio of 10%, and Binary Cross Entropy is used as the loss function [17].

Learning rates are selected individually for each model to reflect differences in their pretraining characteristics and to maintain stable fine-tuning behavior [15]. Specifically, BERT is trained with a learning rate of 1×10^{-5}, as it has not been pretrained on source code and therefore requires moderate update steps to relearn program-related features. CodeBERT, which has prior exposure to NL-PL data, exhibits greater stability and can accommodate a slightly higher learning rate of 2×10^{-5} to accelerate convergence. In contrast, GraphCodeBERT uses the smallest learning rate, 5×10^{-6}, to avoid disrupting layers that encode data-flow structure, which are particularly sensitive to aggressive parameter updates.

This choice of learning rates is grounded in prior findings and aligns with the observed convergence behavior of each model. BERT benefits from a lower learning rate to reduce the risk of catastrophic forgetting [12], CodeBERT tolerates faster updates due to its code-aware pretraining, while GraphCodeBERT requires more conservative adjustments because of its integration of Data Flow Graph information. Together, these settings ensure that fine-tuning proceeds in a controlled and comparable manner across models.

It should be noted that the models were trained using a comprehensive fine-tuning strategy with identical configurations. All models were initialized from publicly available pre-trained checkpoints, and their weights were fully updated during the fine-tuning process without altering the model architecture.

- **Model Evaluation Metrics**

In this study, four evaluation metrics are employed to provide a comprehensive view of model performance in the binary classification setting. These include: (i) Accuracy, which reflects overall prediction correctness; (ii) Precision, which assesses the model's ability to limit false alarms; (iii) Recall, which measures how effectively vulnerable cases are identified; and (iv) F1-score, a composite metric that balances Precision and Recall [18].

Among these measures, F1-score is considered the most informative for comparison. It is particularly sensitive to class imbalance, a common characteristic of vulnerability datasets, especially Big-Vul, and therefore offers a more faithful indication of a model's true predictive quality than accuracy alone.

3.2. Experimental Procedure

The experimental workflow is organized into three main stages, designed to ensure consistency and enable a fair comparison across models.

- **Preprocessing**: A unified preprocessing pipeline is applied to all datasets to eliminate biases arising from input-handling differences. Source code is first syntactically normalized, with comments and redundant whitespace removed. For Devign, Big-Vul, and MegaVul, code samples are further segmented at the function level to maintain consistency across instances, whereas Juliet already follows a fixed function structure and therefore requires no additional splitting. Each code fragment is then converted into a token sequence compatible with the tokenizer of the corresponding model. Importantly, the normalization and function-level segmentation steps remain identical across all models and datasets, ensuring that differences in performance are attributable to the models themselves rather than to variations in input preparation.

  Deduplication and Data Integrity: Prior to model training, we perform a deduplication step at the function level to remove exact duplicate code instances. This helps reduce the risk of unintended information leakage between training and test sets. Additionally, overlap checks are conducted to ensure that identical code samples do not appear across different splits, including in the MegaVul dataset.
- **Model training:** All three models, BERT, CodeBERT, and GraphCodeBERT, are trained on each dataset using full fine-tuning, with the same core set of hyperparameters to preserve comparability. Training follows an identical optimization schedule, and model checkpoints are saved at every epoch. This setup allows us to monitor convergence behavior as well as changes in optimization dynamics throughout the training process.

  Random Initialization and Stability: To improve experimental reliability, model training is conducted with controlled random seeds. All experiments use fixed and documented seeds to ensure reproducibility and consistent initializa-tion across runs. To account for training variability, each experiment is repeated multiple times (5 times) with different random seeds. Reported performance metrics represent the average results across runs.
- **Evaluation and analysis:** In the final stage, models are evaluated on a held-out test set that is completely separated from the training data. Performance is compared across models for each dataset, and their behaviors are analyzed in light of their representational characteristics: BERT's syntax-oriented representations, CodeBERT's NL-PL representations, and GraphCodeBERT's combined syntax and data-flow representations.

  Together, this procedure supports the central objectives of the study: establishing a reliable baseline, examining model adaptability under full fine-tuning [15], and clarifying performance differences when handling diverse categories of source-code vulnerabilities.

3.3. Experimental Results

Table 1: Experimental results on the Devign dataset

Model	Accuracy	Precision	Recall	F1-score
BERT	68.2%	66.5%	63.1%	64.7%
CodeBERT	76.9%	75.4%	73.8%	74.6%
GraphCodeBERT	80.3%	81.1%	78.4%	79.7%

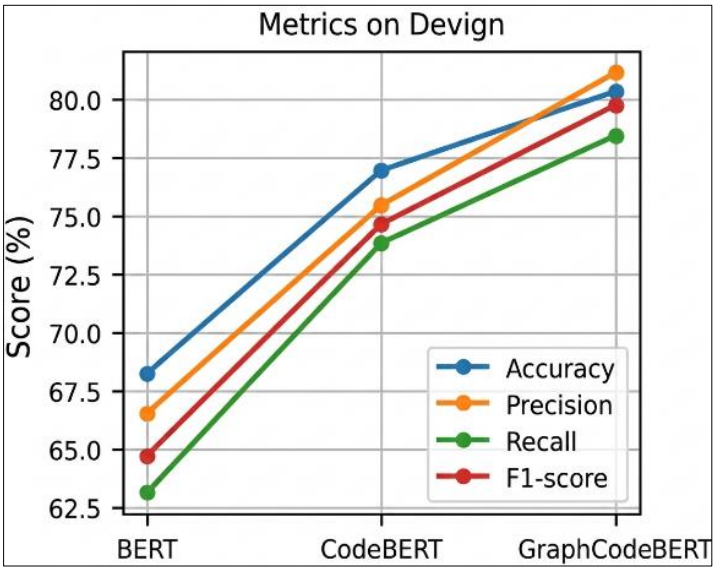

Fig 1: Experimental results on the Devign.

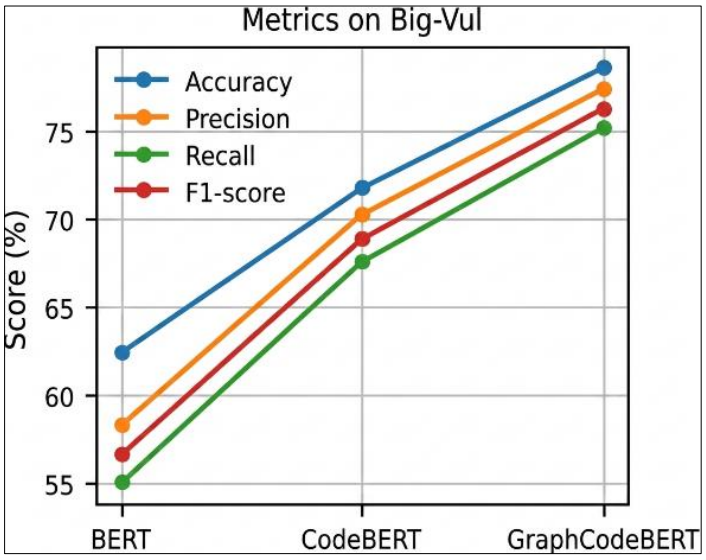

Fig 2: Experimental results on the Big-Vul.

The experimental results are summarized using four primary evaluation metrics: Accuracy, Precision, Recall, and F1-score. The reported results reveal clear performance differences among the three models when full fine-tuning is applied to the Devign, Big-Vul, Juliet and MegaVul datasets. These differences closely reflect the extent to which each model has acquired source-code understanding during the pretraining stage. On the Devign dataset, GraphCodeBERT maintains the highest performance, whereas BERT is limited by its weaker ability to capture program semantics (see Table 1 and Fig 1).

On the Big-Vul dataset, GraphCodeBERT maintains the highest performance, whereas BERT is noticeably affected by data noise and class imbalance (see Table 2 and Fig 2).

Table 2: Experimental results on the Big-Vul dataset

Model	Accuracy	Precision	Recall	F1-score
BERT	62.4%	58.3%	55.0%	56.6%
CodeBERT	71.7%	70.2%	67.5%	68.8%
GraphCodeBERT	78.5%	77.3%	75.1%	76.2%

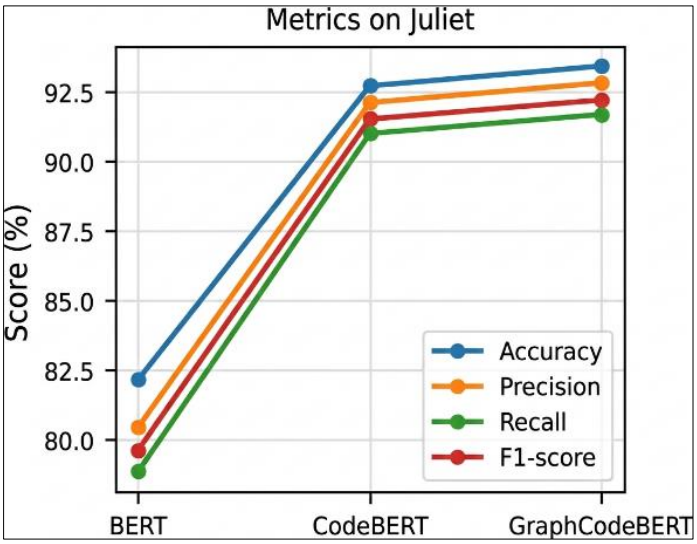

Fig 3: Experimental results on the Juliet.

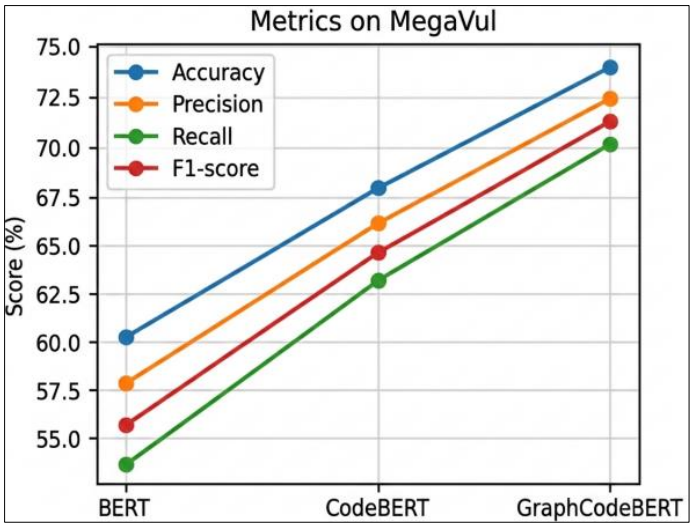

Fig 4: Experimental results on MegaVul.

On the Juliet dataset, CodeBERT and GraphCodeBERT achieve comparable performance, while BERT remains at a lower level (see Table 2 and Fig 3).

Table 3: Experimental results on the Juliet Test Suite dataset

Model	Accuracy	Precision	Recall	F1-score
BERT	82.1%	80.4%	78.8%	79.6%
CodeBERT	92.7%	92.1%	91.0%	91.5%
GraphCodeBERT	93.4%	92.8%	91.7%	92.2%

On the Juliet dataset, CodeBERT and GraphCodeBERT achieve comparable performance, while BERT remains at a lower level (see Table 4 and Fig 4).To address dataset recency, we include a pilot evaluation on MegaVul using the same full fine-tuning protocol. As shown in Table 4, all models exhibit a performance decrease compared to Big-Vul, reflecting the increased complexity and diversity of modern vulnerabilities (see Fig 4). Nevertheless, the relative ranking remains consistent, with GraphCodeBERT achieving the best performance, followed by CodeBERT and BERT.

Table 4: Experimental Results on The Megavul Dataset

Model	Accuracy	Precision	Recall	F1-score
BERT	60.3%	57.8%	53.6%	55.6%
CodeBERT	67.9%	66.1%	63.4%	64.7%
GraphCodeBERT	73.8%	72.4%	70.2%	71.3%

Table 5 summarizes the F1-scores across four datasets, including MegaVul. GraphCodeBERT achieves the highest average (0.799), followed by CodeBERT (0.749) and BERT (0.641). Although performance decreases on MegaVul, the ranking remains consistent, confirming that the advantage of code-aware pretraining persists under more realistic conditions.

Table 5: Summary of F1-scores across all four datasets

Model	Devign	Big-Vul	Juliet	MegaVul	Average (F1)
BERT	0.647	0.566	0.796	0.556	0.641
CodeBERT	0.746	0.688	0.915	0.647	0.749
GraphCodeBERT	0.797	0.762	0.922	0.713	0.799

- On Devign dataset, GraphCodeBERT achieves the best performance due to its ability to capture data-flow dependencies in semanti-cally complex code. CodeBERT shows stable mid-level results, while BERT performs worst due to the lack of code-specific pretraining.
- On Big-Vul dataset, the performance gap becomes more pronounced. GraphCodeBERT remains the strongest under noisy and hetero-geneous data, CodeBERT is moderately stable, and BERT shows the lowest and least robust performance.
- On Juliet dataset, all models perform better due to the structured nature of CWE patterns. CodeBERT performs competitively, sometimes matching GraphCodeBERT, while BERT improves but still lags behind.
- On MegaVul dataset, all models experience a performance drop due to increased complexity and diversity. However, the ranking remains consistent, with GraphCodeBERT leading, followed by CodeBERT and BERT, confirming the benefit of structural pretraining under realistic conditions.

Across all datasets, F1-score, as the most informative metric under class imbalance, reveals a stable comparative pattern. On Devign, Big-Vul, and MegaVul, GraphCodeBERT consistently performs best, followed by CodeBERT, while BERT remains the weakest. In contrast, on Juliet, where vulnerabilities follow more regular patterns, CodeBERT performs comparably to GraphCodeBERT, with both models clearly outperforming BERT.

These findings reinforce the central hypothesis of the study: The extent of code understanding gained during pretraining significantly shapes a model's adaptability under full fine-tuning and strongly influences its performance across diverse vulnerability-detection scenarios. GraphCodeBERT exhibits the best generalization on real-world data, CodeBERT maintains reliable performance on structured or pattern-based datasets, and BERT serves as a lower-bound baseline, highlighting the limitations of models not pretrained on source code.

The experimental results directly address the research questions defined in Section III: (i) RQ1: The comparative results across Devign, Big-Vul, and Juliet clearly demonstrate that full fine-tuning yields different performance levels among BERT-family models, with GraphCodeBERT consistently achieving the highest scores; (ii) RQ2: The superior performance of GraphCodeBERT on Devign and Big-Vul confirms that incorporating structural information, particularly data-flow awareness, significantly improves detection capability in semantically complex scenarios; and (iii) RQ3: The observed performance ranking (GraphCodeBERT, followed by CodeBERT, and then BER) remains consistent across all datasets, indicating stable generalization behavior despite differences in dataset characteristics.

3.4. Case Study: Data-Flow-Dependent Vulner-ability Detection

To complement the quantitative results in Section 3.3, we present a representative example of a data-flow dependent vulnerability.

```
#include <stdio.h>
#include <string.h>
void vulnerable(char *input) {
char buffer[16];
int len = strlen(input);
if (len < 32) { strcpy(buffer, input);}
```

In this code, the externally controlled variable input propagates into buffer via strcpy. Although the condition checks len < 32, the buffer size is only 16 bytes. If len is between 16 and 31, a buffer overflow occurs. Detecting this vulnerability requires understanding the data-flow relationship between input, len, and buffer.

Model Behavior and Analysis: In representative cases, BERT fails to classify this function as vulnerable, while GraphCodeBERT correctly identifies the issue. The difference stems from structural awareness: BERT relies primarily on token-level context, whereas GraphCodeBERT incorporates data-flow information during pretraining, enabling better reasoning over inter-statement variable dependencies.

This qualitative example aligns with the quantitative findings in Section III.C and supports the conclusion that data-flow awareness contributes to improved vulnerability detection performance.

4. Discussion

The results obtained from the three datasets reveal clear performance gaps among BERT [2], CodeBERT [3], and GraphCodeBERT [4] under a full fine-tuning setting for vulnerability detection. These discrepancies arise not only from how much programming-related knowledge each model acquires during pretraining, but also from differences in architecture, optimization behavior, and the characteristics of each dataset. Building on these observations, this section examines the comparative performance of the three models, explains the underlying causes of the identified trends, and draws practical implications together with directions for further investigation.

- **Overall performance comparison**

Across all datasets, a consistent pattern emerges: GraphCodeBERT achieves the strongest results, CodeBERT follows closely behind, and BERT performs the weakest. This outcome supports the initial hypothesis that the degree

to which a model is exposed to code during pretraining strongly influences its ability to adapt during fine-tuning.

Because BERT is trained solely on natural language, it struggles to capture the syntactic structure and semantic dependencies inherent to source code. CodeBERT narrows this gap through joint training on natural-language and programming-language pairs, enabling it to form more code-aware representations. GraphCodeBERT, however, benefits from an additional learning signal based on data-flow relations, allowing it to reconstruct program behavior more faithfully and converge more stably during optimization.

Taken together, these findings align with the goal of the study: assessing the intrinsic adaptability of each model when all parameters are updated through full fine-tuning.

- **Model assessment and recommenda-tions for vulnerability detection**

1. BERT model: The empirical results indicate that BERT performs inconsistently on realistic datasets such as Devign and Big-Vul, where effective detection requires capturing deeper semantic cues and variable dependencies. In contrast, the model behaves notably better on well-structured and low-noise datasets like the Juliet Test Suite, which primarily encode rule-based or pattern-oriented vulnerabilities. This suggests that BERT is more suitable for tasks where the indicators of a flaw appear at the syntactic level and do not require an understanding of program logic.
2. Since BERT is not pretrained on code, it struggles with vulnerabilities that hinge on data-flow relations or subtle semantic constraints, such as memory-management issues, logical inconsistencies, or defects influenced by cross-block data propagation. Nevertheless, it can still offer acceptable performance on CWE patterns with fixed structural signatures, for example, missing condition checks, unsafe hardcoded values, or basic API misuse.
3. Overall, BERT is best positioned as a baseline or as a lightweight option for environments where the data is clean and relatively homogeneous, rather than for scenarios requiring nuanced semantic reasoning about source code.
4. CodeBERT model: CodeBERT shows stable performance across all datasets and excels particularly on structured or semi-synthetic corpora such as Juliet. Its joint training on natural-language and programming-language pairs enables the model to form meaningful representations of both syntax and common programming idioms. As a result, CodeBERT handles pattern-based or syntactically salient vulnerabilities effectively, examples include conditional errors, simple API misuse, and many CWE variants with predictable forms.
5. For vulnerabilities requiring semantic reasoning but not deeply dependent on data flow, CodeBERT generally outperforms BERT, although it still falls short of models designed to capture execution-level behavior. From a practical standpoint, CodeBERT offers a balanced trade-off: it is easier to fine-tune, has moderate computational demands, and yields reliable performance, though it may underperform when data-dependency modeling becomes critical.
6. In summary, CodeBERT is well-suited for mid-scale detection systems or organizations seeking solid accuracy without the cost of more specialized graph-based architectures.
7. GraphCodeBERT model: GraphCode-BERT consistently achieves the strongest results on realistic datasets such as Devign and Big-Vul, where many vulnerabilities arise from variable interactions or data-flow-dependent behavior. Its pretraining strategy, which integrates Data Flow Graph structure, equips the model to capture vulnerabilities involving pointer misuse, tainted data propagation, resource-lifetime violations, and logic flaws that span multiple program regions.

This model is particularly advantageous when dealing with heterogeneous, noisy, or large-scale real-world codebases, as it demonstrates strong generalization and stable convergence under fine-tuning. The primary drawback lies in its higher computational requirements and the need for a carefully selected learning rate to avoid disrupting the pretrained data-flow representations. Moreover, its performance gain is less pronounced on datasets dominated by simple syntactic patterns, such as Juliet, where flow information plays a smaller role.

GraphCodeBERT is therefore the preferred option for production-oriented vulnerability detection pipelines, especially those involving semantically rich or data-flow-intensive vulner-abilities.

The analysis shows that no single model is optimal for every dataset or vulnerability category. Effectiveness varies with the structure of the data and the complexity of the flaw being detected. BERT is most effective on clean, structured datasets with rule-driven patterns; CodeBERT offers a balanced and resource-efficient middle ground; and GraphCodeBERT stands out in real-world, semantically complex scenarios where understanding program behavior is essential.

In practice, model selection should follow the needs of the target system: BERT for standardized datasets and simple patterns, CodeBERT for mid-scale systems requiring stable performance, and GraphCodeBERT for advanced software-security tasks involving complex semantic and data-flow relationships.

- **Impact of pretraining on optimization behavior**

One of the main sources of divergence among the three models during full fine-tuning lies in the representations shaped during pretraining.

Since BERT is trained solely on natural language, it enters the optimization phase without prior knowledge of code-level syntax or semantics. As a result, the model must relearn many fundamental patterns, producing large and highly variable gradients, an effect that slows convergence and often introduces instability. CodeBERT, having been exposed to paired natural-language and programming-language data, begins from a representation space already aligned with the task, leading to more moderate updates and a smoother optimization path. GraphCodeBERT benefits even further due to its integration of data-flow information: parameter updates are guided in directions that match the structure of typical code vulnerabilities, resulting in faster, cleaner convergence and superior performance on real-world datasets.

- **Architectural influence on generaliza-tion**

The experiments indicate that each model's ability to generalize is closely tied to the depth of code understanding embedded during pretraining. GraphCodeBERT exhibits the strongest generalization on practical datasets such as Big-Vul, where source code is diverse, noisy, and often requires

reasoning about data-flow relations. Its data-flow-aware pretraining helps maintain reliable performance even when input distributions vary.

CodeBERT performs well on synthetic or clean datasets with strong structural regularities, particularly the Juliet Test Suite, which organizes vulnerabilities around recurring patterns. While its NL-PL design captures syntactic cues effectively, it is less capable of handling the broader variability seen in real-world projects.

BERT shows the weakest generalization and exhibits strong dependence on the characteristics of each dataset. Without code-specific semantic priors, the model tends to perform adequately only when the data is uniform, low-noise, and structurally simple; performance drops markedly when shifting to realistic or semantically complex distributions.

- **Limitations of the study**

This work focuses exclusively on full fine-tuning and does not compare against parameter-efficient tuning methods such as LoRA, Adapter-tuning, or Prefix-tuning, limiting the scope of conclusions regarding optimization strategies.

In addition, all experiments rely on the base versions of the models; larger variants (e.g., GraphCodeBERT-large) may exhibit different optimization dynamics or generalization beha-viors.

Finally, the study does not include a detailed error analysis by CWE category, leaving open questions about which model is best suited for specific classes of vulnerabilities.

- **Practical implications**

The findings of this study offer meaningful guidance for selecting appropriate models in real-world vulnerability detection systems. For large software projects, where source code tends to be complex, unevenly distributed, and often contains flaws tied to data-flow or deeper semantic interactions, GraphCodeBERT stands out as the most capable option. Its integration of data-flow information allows the model to reconstruct program behavior more accurately and maintain stable performance across diverse inputs.

In scenarios with limited computational resources or tasks that do not require detailed reasoning about data propagation, CodeBERT provides a well-balanced alternative, combining solid accuracy with relatively low training cost and strong performance on vulnerabilities characterized primarily by syntactic cues.

Although BERT is not ideal for deployment in practical vulnerability-detection pipelines, it remains valuable as a reference baseline. Using BERT as a comparative foundation clarifies the extent to which code-oriented pretraining enhances performance and helps standardize evaluation procedures for future research.

- **Directions for future work**

Building on the insights and limitations identified, several promising avenues remain open for further investigation. One direction is to examine parameter-efficient tuning techniques, such as LoRA, Adapter-tuning, Prefix-tuning, and other PEFT methods, to assess whether they can preserve pretrained knowledge while achieving performance on par with or better than full fine-tuning. Another extension involves experimenting with larger variants of the models, such as CodeBERT-large or GraphCodeBERT-large, to better understand how model scale influences semantic learning and generalization.

Additionally, conducting a detailed error analysis across different CWE categories would help determine which model aligns best with specific classes of vulnerabilities, thereby supporting more targeted deployment in practical security tools.

Although PEFT approaches offer substantial benefits in terms of training cost and stability, they tend to narrow the natural performance differences between models since most parameters remain unchanged. For this reason, the present study adopts full fine-tuning as the primary setting, allowing the optimization dynamics of each model to manifest fully and enabling a clear assessment of how their respective pretraining strategies influence downstream vulnerability detection performance.

5. Conclusion

This work presents a thorough comparison of BERT, CodeBERT, and GraphCodeBERT under a full fine-tuning regime for the task of source-code vulnerability detection. Experiments conducted on three representative datasets, Devign, Big-Vul, MegaVul, and the Juliet Test Suite, demonstrate that the knowledge acquired during pretraining plays a decisive role in shaping each model's performance and its ability to adapt to downstream security tasks. BERT serves primarily as a reference baseline, CodeBERT delivers consistent results on well-structured datasets, and GraphCodeBERT achieves the strongest performance on real-world code thanks to its capacity to capture data-flow and deeper semantic relations.

The findings highlight that selecting an appropriate model should be guided by the characteristics of the dataset and the nature of the vulnerabilities being targeted, rather than assuming a single architecture can generalize across all scenarios. Moreover, the observed differences in optimization behavior across the three models suggest promising directions for refining fine-tuning strategies.

Future work may explore parameter-efficient tuning methods, larger model variants, and fine-grained error analysis across CWE categories. Such extensions would further enhance detection accuracy and support the deployment of vulnerability-analysis systems in practical software-engineering environments.

References

1. Qiu X, Sun T, Xu Y, Shao Y, Dai N, Huang X. Pre-trained models for natural language processing: A survey. Sci China Technol Sci. 2020;63(10):1872-97. doi: 10.1007/s11431-020-1647-3
2. Devlin J, Chang MW, Lee K, Toutanova K. BERT: Pre-training of deep bidirectional transformers for language understanding. In: Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. Minneapolis, MN; 2019. p. 4171-86.
3. Feng Z, Guo D, Tang L, *et al* . CodeBERT: A pre-trained model for programming and natural languages. In: Findings of the Association for Computational Linguistics: EMNLP 2020. 2020. p. 1536-47. doi: 10.18653/v1/2020.findings-emnlp.139
4. Guo D, Ren S, Lu S, Feng Z, Tang L, Li D, *et al* . GraphCodeBERT: Pre-training code representations with data flow. In: Proceedings of the International Conference on Learning Representations (ICLR). 2021.
5. Zhou Z, Liu Y, Siow J, Du X, Liu Y. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. Adv

Neural Inf Process Syst. 2019.
6. Fan J, Li Y, Wang S, Nguyen TN. A C/C++ code vulnerability dataset with code changes and CVE summaries. In: Proceedings of the 17th International Conference on Mining Software Repositories (MSR '20). 2020. doi:10.1145/3379597.3387501
7. Boland F Jr, Black PE. The Juliet 1.1 C/C++ and Java Test Suite. IEEE Comput. 2012;45(10):88-90. doi:10.1109/MC.2012.345
8. Ni C, Shen L, Yang X, Zhu Y, Wang S. MegaVul: A C/C++ Vulnerability Dataset with Comprehensive Code Representations. In: Proceedings of the 21st International Conference on Mining Software Repositories (MSR). 2024.
9. Soud M, Nuutinen W, Liebel G. Soley: Identification and automated detection of logic vulnerabilities in Ethereum smart contracts using large language models. arXiv:2406.16244 [Preprint]. 2024.
10. Seo M, Choi W, Shin S, You M. AutoPatch: Multi-agent framework for patching real-world CVEs generated by outdated LLMs. arXiv:2505.04195 [Preprint]. 2025.
11. Bui VC, Do XC. Detecting software vulnerabilities based on source code analysis using GCN transformer. In: 2023 RIVF International Conference on Computing and Communication Technologies (RIVF). Hanoi, Vietnam; 2023. doi: 10.1109/RIVF60135.2023.10471834
12. Yang ZAZH, Tian H, Ye H, Martins R, Le Goues C. Security Vulnerability Detection with Multitask Self-Instructed Fine-Tuning of Large Language Models. arXiv:2406.05892 [Preprint]. 2024.
13. Kedia A, Zaidi MA, Khyalia S, Jung J, Goka H, Lee H. Transformers Get Stable: An End-to-End Signal Propagation Theory for Language Models. arXiv:2403.09635 [Preprint]. 2024.
14. Wen Q, Zeng X, Zhou Z, Liu S, Hosseinzadeh M, Rawassizadeh R. GradES: Significantly Faster Training in Transformers with Gradient-Based Early Stopping. arXiv:2509.01842 [Preprint]. 2025.
15. Tasleem N, Raghav RS, Ansari MN, Sharma AJ. A decision intelligence framework: Integrating human intuition with AI models. J Artif Intell Gen Sci. 2024;7(1):304-19.
16. Houlsby N, Giurgiu A, Jastrzebski S, *et al* . Parameter-Efficient Transfer Learning for NLP. In: Proceedings of the 36th International Conference on Machine Learning (ICML). 2019.
17. Hu E, Shen Y, Wallis P, Chen Z, Sachan M, Liu V. LoRA: Low-Rank Adaptation of Large Language Models. In: International Conference on Learning Representations (ICLR). 2022.
18. Goodfellow I, Bengio Y, Courville A. Deep Learning. Cambridge, MA: MIT Press; 2016.
19. Powers DM. Evaluation: From Precision, Recall and F-measure to ROC, Informedness and Markedness. J Mach Learn Technol. 2011;2(1):37-63.

How to Cite This Article
Le Viet NH, Nguyen Kim T, Dang Van C, Ta Quang C. Impact of full fine-tuning on performance of Bert-family models in source code vulnerability detection task. International Journal of Future Engineering Innovations. 2026 May-Jun;3(3):26-35. doi:10.54660/IJFEI.2026.3.3.26-35.