The Richer Representation Fallacy: Are We Just Adding Noise to LLM-based Software Vulnerability Detectors?

Hazim Hanif
Centre of Research for Cyber Security and Network (CSNET)
Faculty of Computer Science and Information Technology
Universiti Malaya
Kuala Lumpur, Malaysia
hazimhanif@um.edu.my

Sergio Maffeis
Department of Computing
Imperial College London
London, United Kingdom
sergio.maffeis@imperial.ac.uk

Nor Badrul Anuar
Centre of Research for Cyber Security and Network (CSNET)
Faculty of Computer Science and Information Technology
Universiti Malaya
Kuala Lumpur, Malaysia
badrul@um.edu.my

Abstract—**Large Language Models (LLMs) have established strong baselines for software vulnerability detection, leading to a common assumption that their performance can be enhanced by augmenting them with supplementary information such as Abstract Syntax Trees (ASTs), software metrics, or expanded pre-training data. However, the actual efficacy of these computationally expensive techniques over a robust LLM baseline remains unevaluated, potentially misdirecting research efforts. This paper aims to empirically test this "more is better" assumption by conducting a large-scale study that evaluates four supplementary techniques: multi-task learning, software metrics injection, data expansion, and hybrid graph representations against a high-performing LLM baseline, VulBERTa, on the CodeXGLUE benchmark for C/C++ code. Our findings demonstrate that none of these complex techniques provides a statistically significant performance improvement, as the baseline model's tokenization and attention mechanisms already capture the necessary information, rendering the additions redundant. However, we identify software metrics injection as an effective method for tuning the precision-recall trade-off, a critical capability for practitioners needing to minimize false negatives. This paper concludes that for LLM-based vulnerability detection, adding external complexity offers diminishing returns, and future efforts should focus on core model improvements, supporting a "less is more" approach.**

Keywords — vulnerability detection, software security, representation learning, large language models, deep learning

I. Introduction

The introduction of Large Language Models (LLMs) has led to significant developments in code intelligence, showing strong performance in a range of software engineering tasks, from automated program synthesis to bug detection [1]. In the critical area of software security, specialized LLMs have shown success in identifying vulnerabilities from source code. Models such as VulBERTa [2], pre-trained on large datasets of C/C++ functions, have established strong baselines and achieved success by learning complex patterns from sequences of code tokens, outperforming DL approaches on benchmarks like CodeXGLUE [3] and D2A [4].

This success leads to a common hypothesis in the research community: if these models perform well on token sequences, their capabilities have the potential to be further enhanced by adding more explicit structural and semantic information to their input. This "more is better" assumption [20] is motivated by known limitations in LLMs, such as challenges in deep contextual understanding and a tendency to produce incorrect information, which indicates that providing more structured knowledge results in more reliable models [1]. As a result, several research directions have been explored to improve these baseline models.

- Multi-Task Learning (MTL): Trains a model on the main vulnerability detection task and a related secondary task (e.g., predicting code complexity) simultaneously. This encourages the model to learn more generalized and robust features [5].
- Domain-Specific Feature Injection: Directly feeds expert-crafted software metrics (e.g., lines of code or complexity scores) into the model to guide its decisions using established indicators of potential defects [6].
- Data and Model Scaling: Increases the amount of pre-training data and the model's size, based on the principle that a larger, more broadly trained model yields better performance on downstream tasks [2].
- Hybrid Code Representations: Combines code tokens with structural information, such as Abstract Syntax Trees (ASTs), to provide a deeper semantic context than sequential data alone can offer [7].

While these techniques are well-reasoned on their own and represent active areas of research, a crucial question remains unanswered: Do they deliver a genuine, significant performance improvement when applied to a pre-trained LLM baseline for vulnerability detection? The assumption of their utility is widespread, yet it remains to be systematically compared and validated. It is unclear whether these methods offer a true enhancement or merely introduce computational complexity for little or few benefits.

This paper presents a large-scale empirical study designed to evaluate the "more is better" assumption. We systematically assess the efficacy of the four supplementary techniques against a strong baseline LLM, VulBERTa, on the task of function-level vulnerability detection in C/C++ code. The techniques are multi-task learning, software metrics injection, pre-training data expansion and hybrid code representations. Contrary to common assumptions, evaluation results show that these complex supplementary techniques provide few significant performance improvements. Our results indicate that a well-designed baseline LLM, with an advanced tokenization pipeline and a powerful attention mechanism, already captures the necessary signals from raw code sequences, rendering these elaborate additions mostly redundant or, in some cases, even harmful to performance.

In summary, our main contributions are:

- A large-scale comparative study to systematically evaluate four distinct and popular classes of supplementary techniques (multi-task learning, software metrics injection, pre-training data

This work was supported by Universiti Malaya Research Programme under grant UMG010D-2025.

expansion, and hybrid graph representations) against a unified, high-performing LLM baseline.

- Our primary findings indicate that there is strong empirical evidence that these complex techniques failed to outperform a simpler baseline in a meaningful way. The findings encourage future research efforts to focus more on improving core LLM components, such as pre-training objectives and tokenization strategies, rather than adding external complexity for minor improvements.

- Supplementary techniques, while less effective in improving overall accuracy, are still useful for adjusting a model's predictions. For instance, the evaluation showed that software metrics injection is a useful technique for increasing recall at the cost of precision, a practical finding for practitioners who need to minimize false negatives.

II. Related Works

A. *The Rise of LLMs for Code Intelligence*

The application of deep learning to source code analysis has undergone significant evolution. Early approaches leveraged architectures like LSTMs and CNNs on token sequences, while others explored graph neural networks on program structures like ASTs [2]. The introduction of the Transformer architecture, however, marked a new phase [8]. Transformer-based LLMs such as CodeBERT [9], CoTexT [10], and CodeT5 [11] have established themselves as the state of the art, achieving strong performance on comprehensive benchmarks like CodeXGLUE [3] for a wide range of tasks, including code completion, translation, and defect detection.

B. *Recent Advances and Challenges in LLM-based Vulnerability Detection*

In recent years, we have seen a surge in research applying LLMs to C/C++ vulnerability detection, moving beyond initial proofs-of-concept to more rigorous real-world evaluations [11]. However, this body of work has also highlighted significant challenges, particularly the discrepancy between model performance on curated benchmarks versus more realistic scenarios. A key theme is the critical impact of dataset quality. For instance, Primevul dataset [12] was introduced to address data quality issues in earlier benchmarks. On this more challenging dataset, a state-of-the-art 7B parameter model achieved an F1-score of only 3.09%, a stark contrast to the 68.26% F1-score it achieved on the widely used BigVul dataset [13].

Other recent evaluations corroborate this trend. The SecVulEval [14] benchmark, introduced in 2025 for fine-grained, statement-level detection, found that the best-performing model, Claude-3.7-Sonnet, only reached a 23.83% F1-score. A comprehensive study of 14 SOTA LLMs on the SVEN C/C++ dataset reported a balanced accuracy of just 54.5%, with researchers concluding that current LLMs perform poorly at this task [15]. While many recent LLM-based approaches show modest results, other deep learning techniques continue to be explored. In response to these challenges, recent research has focused on providing LLMs with richer context. A significant trend is the shift towards interprocedural and repository-level analysis, with new benchmarks such as ReposVul [19] and VulEval [16] being developed in 2024 to incorporate caller-callee relationships. Another promising direction is the integration of LLMs with traditional static analysis tools.

C. *State-of-the-art in Supplementary Techniques*

The four techniques evaluated in this paper are specifically chosen as they represent primary, active research directions for enhancing code intelligence models.

- Multi-Task Learning (MTL): Improves model generalization by learning shared representations across tasks. In code, it predicts properties of code snippets jointly through different classification tasks. For example, some studies used MTL to predict token for code completion, showing better performance than single-task models [5].

- Software Metrics in ML-based Security: Traditional metrics like cyclomatic complexity, lines of code, and coupling measures were used before deep learning as features for vulnerability prediction models. These metrics quantify code complexity, often correlating with defects, and have been successfully used with various machine learning algorithms to identify vulnerable code units.

- Scaling Laws and Data Expansion: The principle of "scaling laws" suggests model performance improves predictably with increased size, dataset, and compute [1]. Empirical evidence from successive models supports this. Chinchilla [17] demonstrated that for optimal training, model size and training tokens should be scaled equally, revealing that many large models were undertrained.

- Hybrid Code Representations: Hybrid models combine token sequences, which show linear code flow, with graph representations like ASTs and CFGs that explicitly show hierarchical and control-flow structures. Recent research highlights the advantages of hybrid graphs in tasks like code clone detection [7].

While the literature contains numerous proposals for models that utilize one of these supplementary techniques, a significant gap remains in the research. There is a lack of rigorous, comparative analysis that evaluates the additional benefit of these techniques when added to a single, unified, and already high-performing LLM baseline. Prior work has focused on demonstrating the viability of a new complex model, asking, "Can we build a model with technique X?" This paper addresses a more critical and practical question for the field: "Given a state-of-the-art LLM, should we add the complexity of technique X?" By providing a direct, empirical answer, this work aims to help direct future research.

III. Methodology

In this section, we introduce the framework for our large-scale empirical evaluation of the supplementary techniques. The framework consists of four main supplementary techniques: a) Multi-task learning, b) Software metrics injection, c) Pre-training data expansion, and d) Hybrid code representation, which we implement as modifications to the baseline model, allowing for a direct comparison of their impact on the baseline model. We describe the baseline model and these supplementary techniques in detail in subsequent subsections.

A. *The Baseline LLM: VulBERTa-CNN*

The baseline LLM selected for this paper is VulBERTa-CNN, due to its effectiveness despite its small size. It efficiently freezes the pre-trained embedding layer and uses it to initialize a Text-CNN architecture. This reduces the number of trainable parameters to approximately 2 million, significantly accelerating training while keeping the knowledge from pre-training in the embeddings.

A critical component of its success is its advanced tokenization pipeline. This pipeline is a language-specific pre-processing stage designed to retain as much information as possible from the raw source code. This tokenization strategy ensures that key syntactic and semantic elements are never broken down into subwords, providing the Transformer's attention mechanism with a stable and informative sequence. The strong performance of these baseline models on the CodeXGLUE benchmark sets a high standard for any supplementary technique to be considered beneficial.

B. *Technique 1: Multi-Task Learning (MTL)*

This technique aims to improve model generalization by training it on multiple related tasks simultaneously. We modify the VulBERTa-CNN architecture to incorporate a second classification head, allowing it to learn two tasks simultaneously. The primary task remains vulnerability detection. For the secondary task, we implement five task variations to provide different types of contextual information:

- **Frequency of standard API calls**: The syntactic occurrence of standard C/C++ API or library calls in a function.
- **Frequency of security-related API calls**: The syntactic occurrence of security-related C/C++ API or library calls in a function.
- **Frequency of memory-related API calls**: The syntactic occurrence of memory-related C/C++ API or library calls in a function.
- **Frequency of all API calls**: The syntactic occurrence of all C/C++ API or library calls in a function.
- **Cyclomatic complexity level**: The level of complexity of a function by measuring the number of linearly independent paths in a function.

The idea is that by learning a related secondary task, the model develops a richer, more generalized internal representation. During training, the losses from both tasks were averaged and backpropagated through the shared network layers, forcing the model to find a balance in learning features useful for both objectives. Fig. 2 illustrates the MTL architecture by altering the VulBERTa-CNN model.

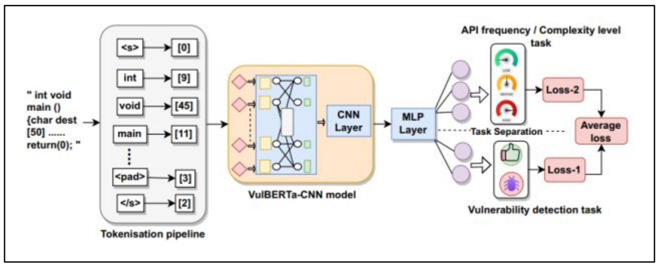

Fig. 1. Software metrics injection architecture with VulBERTa-CNN model.

C. *Technique 2: Software Metrics Injection*

This technique involves supplementing the model with external, expert-defined features that have traditionally been used to indicate code quality and complexity. We modify the baseline VulBERTa-CNN architecture to include a separate input path for these numerical metrics. Three distinct sets of metrics were extracted for each function:

- **Standard software metrics**: Statistical measures extracted from the raw C/C++ code using the static analysis tool, Scitools Understand (e.g., *CountStmt, AvgCyclomatic, CountLine*).
- **AST-based metrics**: Statistical measures extracted based on the elements of Abstract Syntax Trees of a function using Clang (e.g., *NumASTNode, DepthAST, NumTokens*).
- **Flawfinder metrics**: Statistical measures extracted from a full report produced by a Flawfinder run for each function (e.g., *NumFlagged, L5count, buffer)*

Fig. 1 illustrates the software metrics injection architecture in conjunction with the baseline model. The supplementary mechanism operates by injecting these numerical features into the network immediately before the final classification layers, where they are concatenated with the learned representation from the CNN. This approach evaluates whether providing explicit, quantitative measures of code complexity is able to aid the LLM's decision-making process effectively.

D. *Technique 3: Pre-training Data Expansion*

This technique evaluates the "scaling laws" hypothesis, which hypothesizes that model performance scales with the amount of data. To achieve this, we developed **VulDeBERTa**, a new 125-million-parameter model, by implementing two significant modifications to the baseline. First, we replace the underlying RoBERTa architecture with the more advanced DeBERTa v2 [21], which enhances performance through novel techniques such as a disentangled attention mechanism and an improved mask encoder. Second, we expand the pre-training dataset by over 4.5 times, moving from 2.2 million functions to a 10-million-function subset of the GitHub-L dataset.

We also increase the vocabulary with over 800 new memory and security-related API call tokens to capture more domain-specific knowledge. Finally, we fine-tune VulDeBERTa using the same lightweight CNN head as the baseline, which allows for a direct comparison to measure the impact of the modern architecture and increased data. Table I summarizes the main component differences between VulBERTa and the newly developed VulDeBERTa model.

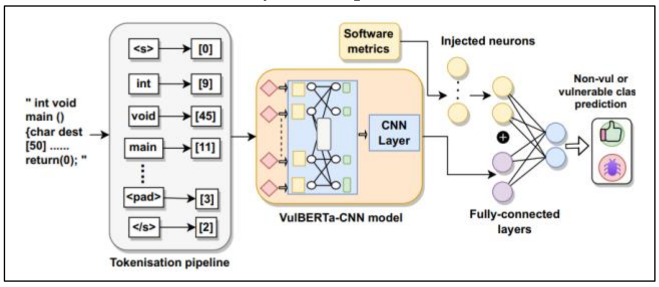

Fig. 2. Multi-task learning architecture with VulBERTa-CNN model

TABLE I.
Main component difference between VulBERTa and VulDeBERTa model

Components	VulBERTa	VulDeBERTa (this paper)
Architecture	RoBERTa-base	DeBERTa v2-base
Pre-training data	2.2 million C/C++ functions	10 million C/C++ functions
Special tokens	Standard API calls	Standard, memory-related, security-related API calls
Number of special tokens	444	1250
Pre-training time	5 days	25 days

E. *Technique 4: Hybrid Code Representation*

This technique provides the model with a richer understanding of code by combining multiple representations. We design a new architecture, **VulCAS**, to process three parallel input streams for each function. Fig. 3 visualizes the VulCAS architecture.

The first stream feeds the standard sequence of code tokens into the baseline VulBERTa-CNN to capture sequential patterns. The other two streams process the function's Abstract Syntax Tree (AST) and Control Flow Graph (CFG) using separate Graph Attention Networks (GATv2), a state-of-the-art architecture for learning from graph data. The AST provides hierarchical syntactic structure, while the CFG outlines the execution flow.

Towards the end, the architecture concatenates the outputs from all three streams: the token-based CNN and the two graph-based GATs, and passes the combined result to a set of fully-connected layers for classification. This hybrid approach evaluates whether explicitly providing structural and control-flow information improves upon what the model learns implicitly from tokens alone.

F. *Datasets*

This subsection describes the datasets used in this paper. They consist of function-level C/C++ source code from various codebases, mainly from open-source repositories. All datasets mentioned below are in the public domain and available for download without restriction. They consist of function-level C/C++ source code from various codebases, mainly from open-source repositories. All datasets mentioned below are in the public domain and available for download without restriction.

1) Devign (CodeXGLUE benchmark): The Devign dataset is a real-world software vulnerability detection dataset with function-level C/C++ source code from QEMU and FFmpeg. It is a binary detection dataset labeled as non-vulnerable or vulnerable, manually verified by security researchers in two rounds. Chosen for its widespread use and as a benchmark in the Microsft CodeXGLUE leaderboard.

2) GitHub-L (BigQuery): The GitHub-L dataset is an extensive collection of function-level C/C++ source code from various open-source software projects on GitHub. It consists of 18 million real-world functions extracted using Google BigQuery. Since this dataset aims to collect a large number of functions, we avoid using the GitHub API due to the rate-limit restrictions. From there, we sample 10 million function-level C/C++ source code for the pre-training of the VulCAS model.

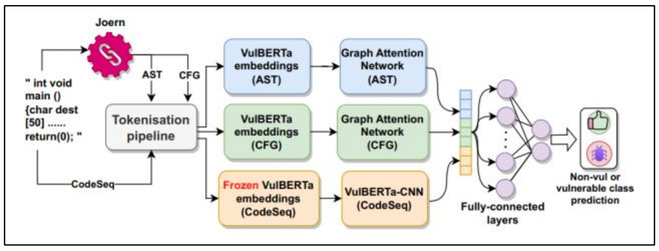
Joern
AST
CFG
" int void main () {char dest [50] ...... return(0); "
Tokenisation pipeline
CodeSeq
VulBERTa embeddings (AST)
Graph Attention Network (AST)
VulBERTa embeddings (CFG)
Graph Attention Network (CFG)
Frozen VulBERTa embeddings (CodeSeq)
VulBERTa-CNN (CodeSeq)
Fully-connected layers
Non-vul or vulnerable class prediction

Fig. 3. The VulCAS architecture

IV. Experimental Evaluation

In this section, we describe how we evaluate the proposed supplementary techniques against the baseline LLM.

A. *Experimental Setup*

1) Hardware and software: We use PyTorch 1.10 with CUDA 11.3 on top of Python 3.9 for all experiments. For pre-training VulDeBERTa, we use the High-Performance Computing (HPC) cluster from the university with 32 cores Intel Xeon CPU, 192GB RAM, and 8 NVIDIA RTX 6000 GPUs, each with 24GB of video memory. On top of that, we also use Weights & Biases as our training management platform to track training sessions throughout the work.

2) Performance criteria: The primary evaluation metric is Accuracy, as it is the standard for the CodeXGLUE benchmark. However, to provide a more complete picture of performance, we also report Precision, Recall, and F1-score. The F1-score is particularly important as it provides a balanced measure of a model's performance on imbalanced classification tasks like this one.

3) Dataset: All comparative fine-tuning experiments were conducted solely on the Devign dataset. As a key part of the Microsoft CodeXGLUE benchmark, Devign is a well-known and challenging dataset for real-world vulnerability detection, ensuring that our results are both reproducible and directly comparable to other published work.

B. *Results*

Table II summarises the experimental results on the Devign dataset for the vulnerability detection task. We highlight the highest score for each evaluation metric. Additionally, we also analyze the results using various supplementary techniques and discuss them accordingly.

1) Multi-task learning: None of the multi-task learning approaches improved the baseline model's accuracy. The best was using cyclomatic complexity as a secondary task, with 64.16% accuracy, the highest true positives (488), and lowest false negatives (767). It also had a recall of 61.11%, about 10% above the baseline, and the highest F1 score of 61.04%. Including complexity as a second task improved positive sample detection. The memory API calls task had the most false positives and true negatives, with the highest precision of 66.95%, but overall accuracy was only 62.62%.

2) Software metrics injection: The results showed that by injecting standard software metrics, we achieved 64.24% accuracy and 65.11% precision, the highest among different approaches for software metrics injection. Conversely, the injection of AST metrics achieved higher recall at 50.52%, compared to standard software metrics. Flawfinder metrics

TABLE II
Evaluation results for all supplementary techniques on the CodeXGLUE benchmark.

Supplementary technique	**Approach**	**FN**	**FP**	**TN**	**TP**	**Accuracy (%)**	**Precision (%)**	**Recall (%)**	**F1 (%)**
Baseline	VulBERTa-CNN	615	357	1120	640	64.42	64.19	51.00	56.84
Multi-task learning	Frequency of standard API calls	580	409	1068	675	63.80	62.26	53.78	57.72
	Frequency of security API calls	533	503	974	722	62.07	58.93	57.52	58.22
	Frequency of memory API calls	793	**228**	**1249**	462	62.62	**66.95**	36.81	47.50
	Frequency of all API calls	581	452	1025	674	62.19	59.86	53.70	56.61
	Level of cyclomatic complexity	**488**	491	986	**767**	64.16	60.97	**61.11**	**61.04**
Software metrics injection	Standard software metrics	656	321	1156	599	64.24	65.11	47.73	55.08
	AST metrics	621	381	1096	634	63.32	62.46	50.52	55.88
	Flawfinder metrics	519	505	972	736	62.52	59.31	58.65	58.97
Pre-training data expansion	VulDeBERTa	649	319	1158	606	**64.57**	65.51	48.29	55.60
Hybrid code representation	VulCAS	570	398	1079	685	**64.57**	63.25	54.58	58.60

also drag VulBERTa-CNN down and reduce its detection performance by 1.9%. The results indicated that all software metrics injection approaches are still unable to improve the detection performance beyond the baseline.

3) Pre-training data expansion: VulDeBERTa achieved 64.57% accuracy, slightly better than VulBERTa-CNN by 0.15%. It achieves better precision due to higher true negatives, but at the expense of lower recall and F1 scores compared to baselines, indicating a more effective classification of non-vulnerable samples than vulnerable ones.

4) Hybrid code representation: The results showed that VulCAS achieved the same accuracy as VulDeBERTa, which is also higher than the VulBERTa-CNN baseline. However, it has a lower precision of 63.35% than the baselines. We achieved higher true positives using the hybrid representation architecture, resulting in higher recall and F1 scores compared to both baseline models. This shows that it predicts vulnerable samples more accurately than baseline models.

C. Discussion

We divide our discussion into four research questions based on the supplementary techniques proposed in this paper.

1) RQ1: Do multiple tasks share different information learned between them, and do they compete with each other?

Multi-task learning (MTL) aims for tasks to share information to improve performance, but it only works if tasks don't compete during training. Our Devign dataset evaluation shows that MTL was harmful, reducing primary vulnerability detection. The secondary task maintained high accuracy over 90%, but at the expense of the primary task, indicating one-sided learning. Adding a third task worsened performance, suggesting the model was reaching capacity. Simple tasks dominate and harm others. Even with weighting to prioritize vulnerability detection, no improvement was seen. Overall, different learning tasks shared information among each other but ended up competing, causing negative results

2) RQ2: Does injection of software metrics provides relevant information to the detection model?

We hypothesize that adding external software metrics would improve VulBERTa-CNN's detection accuracy. However, the results show that it is less likely and sometimes worsens performance compared to the baseline. This indicates that the model doesn't benefit from such data, which is common for neural networks with tabular inputs, unlike traditional models like SVM or Random Forest. These metrics also performed poorly alone, with less than 55% accuracy. However, by combining standard and Flawfinder metrics, we were able to increase true positives and achieve better recall and F1 scores than the baseline. While metrics injection does not improve overall accuracy, it serves as a valuable control knob. Practitioners can use it to deliberately shift a model's behavior towards higher recall, a critical function in security settings where the cost of a missed vulnerability (a false negative) far outweighs that of a false alarm.

3) RQ3: Does increasing pre-training data help in a vulnerability detection task?

Based on prior work showing larger datasets benefit Large Language Models (LLMs), we increased pre-training data by over 450% (from 2.2 to 10 million C/C++ functions) and added new API call tokens. We hypothesize this would help the model learn a more general code representation and improve vulnerability detection. Despite the effort, the results were underwhelming. This data increase only improved accuracy by 0.15% over the VulBERTa-CNN baseline. The abundance of general C/C++ functions is unable to introduce better generalization, likely because vulnerability detection needs project-specific code, as vulnerabilities have complex, hidden patterns best captured within a single project's style. Different vulnerability types also hinder detection, making a general approach less effective. We conclude that increasing general pre-training data does little to improve the baseline models for this task.

4) RQ4: Does hybrid code representation introduce additional semantic and contextual knowledge to the model?

We combine code tokens, ASTs, and CFGs into a hybrid architecture to learn richer syntactic and semantic information from C/C++ source code. The ASTs and CFGs should provide additional contextual knowledge to improve vulnerability detection. However, the hybrid model, VulCAS, performed similarly to VulDeBERTa, with 64.57% accuracy. This is slightly higher than VulBERTa-CNN, suggesting that ASTs and CFGs provide only a marginal enhancement to code representation. This likely stems from a foundational representational mismatch. The VulBERTa embeddings are highly optimized to capture semantic patterns from sequential code tokens. In contrast, the GATs learn structural patterns from graph nodes, and in our setup, they were trained from scratch only on the downstream task data. Without a joint pre-training phase to harmonize these disparate modalities, the model struggles to effectively fuse them. The powerful, pre-trained signals from the token-based CNN likely overshadow the weaker signals from the graph networks, rendering their structural information redundant.

D. Limitations

Our work faced two main limitations: resource constraints restricted us to models with fewer than 500 million parameters on a 24GB GPU, which impacted downstream performance improvements demonstrated by research such as AlphaCode [18].

Additionally, our techniques were applied to pre-trained VulBERTa embeddings, which were optimized solely for source code and created without considering software metrics or other objectives. Proper evaluation of these methods would require pre-training the model from scratch with integrated information.

In low-signal environments, the supplementary techniques we evaluated could provide a more significant benefit by injecting necessary structural or expert-defined context that the model fails to learn from tokens alone. Therefore, while our work questions the utility of added complexity on established benchmarks, the value of these techniques in more difficult, realistic scenarios remains an open question.

V. Conclusion And Future Work

This paper demonstrates that computationally expensive supplementary techniques offer diminishing returns for LLM-based software vulnerability detection. Our large-scale study found that four popular techniques which are multi-task learning, software metrics injection, data expansion, and hybrid representations. These techniques failed to significantly outperform a simpler, well-tuned baseline. While metrics injection is a useful tool for tuning the precision-recall trade-off, we conclude that the path to better performance lies in improving core model architectures and pre-training objectives, not in adding external complexity. In this domain, our evidence suggests that less, is in fact, more.

Future work will proceed in two primary directions. First, we will focus on scaling and validation by testing the "less is more" hypothesis on larger models and across challenging, repository-level benchmarks such as VulEval and ReposVul. Second, to properly evaluate hybrid methods, we will pursue joint pre-training, where a model is trained from scratch with integrated graph and token representations to overcome the limitations identified in this study.

References

[1] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. Salakhutdinov, and Q. V. Le, ‘XLNet: generalized autoregressive pretraining for language understanding’, in Proceedings of the 33rd International Conference on Neural Information Processing Systems, Red Hook, NY, USA. 2019.

[2] Hanif, H., & Maffeis, S. (2022, July). Vulberta: Simplified source code pre-training for vulnerability detection. In 2022 International joint conference on neural networks (IJCNN) (pp. 1-8). IEEE.

[3] S. Lu et al., “Codexglue: A machine learning benchmark dataset for code understanding and generation,” CoRR, vol. abs/2102.04664, 2021

[4] Y. Zheng et al., “D2A: A dataset built for AI-based vulnerability detection methods using differential analysis,” in 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). Los Alamitos, CA, USA: IEEE Computer Society, may 2021, pp. 111–120.

[5] O. Sener and V. Koltun, ‘Multi-task learning as multi-objective optimization’, in Proceedings of the 32nd International Conference on Neural Information Processing Systems, Montréal, Canada, 2018, pp. 525–536.

[6] M. U. Zahid, S. Kiranyaz, and M. Gabbouj, ‘Global ECG Classification by Self-Operational Neural Networks With Feature Injection’, IEEE Transactions on Biomedical Engineering, vol. 70, no. 1, pp. 205–215, 2023.

[7] Z. Zhang and T. Saber, ‘AST-Enhanced or AST-Overloaded? The Surprising Impact of Hybrid Graph Representations on Code Clone Detection’, arXiv [cs.AI]. 2025.

[8] A. Vaswani et al., ‘Attention is all you need’, in Proceedings of the 31st International Conference on Neural Information Processing Systems, Long Beach, California, USA, 2017, pp. 6000–6010.

[9] Z. Feng et al., ‘CodeBERT: A Pre-Trained Model for Programming and Natural Languages’, in Findings of the Association for Computational Linguistics: EMNLP 2020, 2020, pp. 1536–1547.

[10] L. Phan et al., ‘CoTexT: Multi-task Learning with Code-Text Transformer’, in Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021), 2021, pp. 40–47.

[11] Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi, ‘CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation’, in Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, 2021, pp. 8696–8708.

[12] Y. Ding et al., ‘Vulnerability Detection with Code Language Models: How Far are We?’, in 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE), 2025, pp. 1729–1741.

[13] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, ‘A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries’, in Proceedings of the 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 2020, pp. 508–512.

[14] M. B. U. Ahmed, N. S. Harzevili, J. Shin, H. V. Pham, and S. Wang, ‘SecVulEval: Benchmarking LLMs for Real-World C/C++ Vulnerability Detection’, arXiv [cs.SE]. 2025.

[15] J. He and M. Vechev, ‘Large Language Models for Code: Security Hardening and Adversarial Testing’, in Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, Copenhagen, Denmark, 2023, pp. 1865–1879.

[16] X.-C. Wen, X. Wang, Y. Chen, R. Hu, D. Lo, and C. Gao, ‘VulEval: Towards Repository-Level Evaluation of Software Vulnerability Detection’, arXiv [cs.SE]. 2024.

[17] J. Hoffmann et al., ‘Training compute-optimal large language models’, in Proceedings of the 36th International Conference on Neural Information Processing Systems, New Orleans, LA, USA, 2022.

[18] Y. Li et al., ‘Competition-level code generation with AlphaCode’, Science, vol. 378, no. 6624, pp. 1092–1097, 2022.

[19] X. Wang, R. Hu, C. Gao, X.-C. Wen, Y. Chen, and Q. Liao, ‘ReposVul: A Repository-Level High-Quality Vulnerability Dataset’, in Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings, Lisbon, Portugal, 2024, pp. 472–483.

[20] J. B. Simon, D. Karkada, N. Ghosh, and M. Belkin, ‘More is Better: when Infinite Overparameterization is Optimal and Overfitting is Obligatory’, in The Twelfth International Conference on Learning Representations, 2024.

[21] P. He, X. Liu, J. Gao, and W. Chen, ‘DeBERTa: Decoding-Enhanced BERT with Disentangled Attention’, in 2021 International Conference on Learning Representations, 2021.