The Journal of Systems and Software 231 (2026) 112642

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

The fire tries gold: Evaluating pre-trained language models for multi-label vulnerability detection in ethereum smart contracts☆

Trung Kien Luu, Doan Minh Trung, Tuan-Dung Tran, Phan The Duy, Van-Hau Pham *

Information Security Lab, University of Information Technology, Ho Chi Minh City, Viet Nam
Vietnam National University, Ho Chi Minh City, Viet Nam

ARTICLE INFO

Keywords:
Smart contract security
Vulnerability detection
Pre-trained language models
Deep learning
Model benchmarking
Code analysis
BiLSTM
BERT
Resource efficiency
Hyperparameter optimization

ABSTRACT

Smart contracts are integral components of blockchain ecosystems, yet they remain highly susceptible to security vulnerabilities that can lead to severe financial and operational consequences. To address this, a range of vulnerability detection techniques have been developed, including rule-based tools, neural network models, pre-trained language models (PLMs), and most recently, large language models (LLMs). However, those existing methods face three main limitations: (1) Rule-based tools such as Slither and Oyente depend heavily on handcrafted heuristics, requiring human intervention and high execution time. (2) LLM-based approaches are computationally expensive and challenging to fine-tune in resource-constrained environments, particularly within academic or research settings where access to high-performance computing is constrained. (3) Most existing approaches focus on binary and multi-class classification, assuming that each contract contains only a single vulnerability, whereas in practice, smart contracts often exhibit multiple coexisting vulnerabilities that require a multi-label detection approach. In this study, we conduct a comprehensive benchmark that systematically evaluates the effectiveness of traditional deep learning models (e.g., LSTM, BiLSTM) versus state-of-the-art PLMs (e.g., CodeBERT, GraphCodeBERT) in multi-label vulnerability detection. Our dataset comprises nearly 18,000 real-world smart contracts annotated with seven distinct vulnerability types. We evaluate not only detection accuracy but also computational efficiency, including training time, inference speed, and resource consumption. Our findings reveal a crucial trade-off: while code-specialized PLMs like GraphCodeBERT achieve a high F1-score of 96%, a well-tuned BiLSTM with an attention mechanism surpasses it (98% F1-score) with significantly less training time. By providing a clear, evidence-based framework, this research offers practical recommendations for engineers to select the most appropriate model, balancing state-of-the-art performance with the resource constraints inherent in real-world security tools.

1. Introduction

Smart contracts have revolutionized digital transactions by automating agreement execution on blockchain platforms. However, their immutable and complex nature creates significant security risks, with exploitation leading to billions in financial losses (Zhou et al., 2023; Certik, 2024). To combat the situation, numerous methods have been developed.

Recent advances in machine learning (ML) have led to a growing interest in automated vulnerability detection using data-driven approaches. Currently, most vulnerability detection methods fall into four categories: rule-based, neural network-based, PLM-based, and LLM-based techniques. In the past, many rule-based tools such as Oyente (Luu et al., 2016), Slither (Feist et al., 2019) were widely adopted due to their simplicity and the ease of interpretation. Yet they rely heavily on handcrafted heuristics and symbolic execution, which limits their ability to detect subtle or obfuscated vulnerabilities. These tools also tend to be time-consuming and require human intervention. For instance, the average analysis time for a single contract using Oyente is approximately 350 s, which can extend to over 30 min for contracts with higher complexity (Sendner et al., 2023). Recent studies estimate that such static analysis tools miss over 80% of real-world vulnerabilities (Zhang et al., 2023), underscoring the need

☆ Editor: Dario Di Nucci.

* Corresponding author at: Information Security Lab, University of Information Technology, Ho Chi Minh City, Viet Nam.
E-mail address: haupv@uit.edu.vn (V.-H. Pham).

https://doi.org/10.1016/j.jss.2025.112642
Received 24 February 2025; Received in revised form 8 July 2025; Accepted 14 September 2025
Available online 22 September 2025

for more effective and automated approaches to secure blockchain ecosystems (Ressi et al., 2024).

To overcome the limitations of rule-based tools, researchers have explored a variety of deep learning (DL) approaches. Neural network-based methods, such as LSTM (Luo et al., 2022) and BiLSTM (Qian et al., 2020; Zhang et al., 2022), learn statistical patterns directly from labeled smart contract data. These models are capable of capturing sequential dependencies and context-aware semantics, offering more flexibility and generalization than static rule-based techniques. Currently, with the emergence of PLMs trained on large-scale and diverse code corpora, models such as CodeBERT, GraphCodeBERT, CodeT5, and CodeGen have demonstrated strong code representation capabilities (Zhao et al., 2024). As a result, these PLMs have gained significant popularity in the field of vulnerability detection, where they are leveraged to reduce training costs and improve detection performance by transferring knowledge from general code understanding tasks.

More recently, the use of general-purpose LLMs for smart contract vulnerability detection has attracted interests from many researchers. Two main approaches have emerged: prompt-based inference and fine-tuning. In the prompt-based setting, models such as GPT-3.5 and GPT-4 are guided by carefully constructed prompts to detect vulnerabilities, typically in zero-shot or few-shot settings without additional training. However, this approach often suffers from inconsistent outputs, including high false-positive rates, hallucinations, and effectiveness degradation in detecting unfamiliar or intricate attack patterns (Ullah et al., 2023; David et al., 2023; Sun et al., 2024). In contrast, fine-tuning-based approaches aim to improve upon prompt-based methods by adapting LLMs to domain-specific vulnerability data. This enhances their understanding of Solidity syntax and exploit patterns, resulting in improved recall and precision (Yu et al., 2024). Nevertheless, this process requires significant computational resources and a large amount of labeled training data, which are often unavailable in academic or resource-constrained environments.

Another important limitation of these current methods is their tendency to frame vulnerability detection as a binary or multi-class classification problem (Tong et al., 2024), assuming that each contract contains only one vulnerability. In practice, smart contracts often contain multiple coexisting vulnerabilities, necessitating a multi-label detection framework that can simultaneously identify all relevant flaws in a single pass.

To this end, we conduct a comprehensive comparison of seven different PLMs: BERT, CodeBERT, GraphCodeBERT, DeBERTa, RoBERTa, DistilBERT, and Longformer, with four neural-network-based methods: LSTM and BiLSTM, each with and without attention mechanism. The comparison is aimed to evaluate the ability to classify seven types of vulnerabilities in a multi-label setting on a dataset of nearly 18,000 real-world smart contracts. The rationale for this specific model selection, which covers diverse architectures and pre-training strategies, is detailed in Section 3. We also explore optimized fine-tuning settings to provide recommendations that balance performance with resource constraints.

Results from our extensive experiments demonstrate that certain PLMs consistently outperform others in smart contract vulnerability detection tasks. The top-performing models achieve high F1-Score across various vulnerability categories, showcasing their effectiveness in identifying complex vulnerabilities. While some models exhibit quick convergence, others require longer training times but achieve superior accuracy. As for DL methods, the more complicated the architecture (i.e., DL models with attention mechanisms and certain hyperparameters), the better the results they yield.

These findings suggest that with our particular dataset and experimental testing environment, PLMs can be powerful tools for smart contract vulnerability detection, especially those with architectures well-suited for processing code-like structures. However, traditional DL tools might be a better choice regarding efficiency and accuracy. Our results highlight the potential of leveraging advanced NLP techniques to enhance the security of blockchain technologies.

To summarize, our main contributions are:

- We conduct a comprehensive comparison between various PLMs and DL techniques for multi-label vulnerability classification for Ethereum smart contracts with different resource conditions.
- We provide insights into the effectiveness of code-specific pre-training and pure DL training for vulnerability analysis on smart contracts.

The remainder of this paper is structured as follows: Section 2 introduces relevant research in smart contract vulnerability detection. In Section 3, we provide a comprehensive explanation of our methodology. Section 4 delves into the specifics of our experimental setup and presents the results. Section 5 discusses practical application and software engineering insights derived from our findings. Next, Section 6 discusses limitations and outlines our future work, while the threat to validity is presented in Section 7. Lastly, Section 8 gives our conclusions.

2. Related works

Smart contract vulnerability detection has been an active area of research, with various approaches proposed recently.

2.1. DL-based methods

Many studies have shifted towards using ML/DL techniques due to their significant improvement in accuracy, particularly in analyzing bytecode to identify vulnerabilities in smart contracts. Escort (Lutz et al., 2021) is a pioneering work that utilizes deep neural networks to detect smart contract vulnerabilities. Researchers have explored various neural network architectures to identify vulnerable source code, including methods based on graph neural networks (GNNs) (Zhuang et al., 2021; Nguyen et al., 2022; Cai et al., 2023), convolutional neural networks (CNNs) (Hwang et al., 2022), and sequence models (Qian et al., 2020; Yu et al., 2021; Ren et al., 2023).

GNNs have been particularly effective due to their ability to capture structural information from the code. For instance, Zhuang et al. (2021) propose a GNN-based approach that models smart contracts as graphs to detect vulnerabilities more accurately. Similarly, Nguyen et al. (2022) present MANDO, a method that utilizes GNNs to extract semantic information for vulnerability detection. Their experiments demonstrate that GNN-based models outperform traditional neural networks in capturing code semantics.

Sequence models like LSTM and GRU have also been applied to capture sequential patterns in code tokens. Qian et al. (2020) use bidirectional LSTM networks to detect vulnerabilities by learning both past and future context information in the code sequence. Attention mechanisms have further enhanced these models by allowing them to focus on critical parts of the code that contribute most to the vulnerabilities. Yu et al. (2021) develop DeESCVHunter, which combines an LSTM with an attention mechanism to improve detection accuracy.

CNNs are employed to extract local features from the code. Hwang et al. (2022) introduce CodeNet, a CNN-based model that captures patterns in smart contract code to identify vulnerabilities. Their approach shows that CNNs could effectively recognize syntactic patterns associated with common vulnerabilities.

Other studies have incorporated techniques like reinforcement learning and adversarial training to improve detection robustness. For example, Chakraborty et al. (2021) use reinforcement learning to generate adversarial examples to train a more robust vulnerability detection model. Their approach enhances the model's ability to handle code obfuscation and other evasion techniques used by attackers.

Further reinforcing the trend of applying machine learning to this domain, Rizzo et al. (2024) perform a comparative study evaluating both classical machine learning techniques (such as SVM and XGBoost) and deep learning models on the manually-labeled Consolidated Groundtruth dataset. Their work underscores the potential of

diverse ML approaches but also highlights the challenges in comparing solutions due to differing datasets and evaluation methodologies.

Despite these advances, DL-based methods often require large labeled datasets and may struggle with generalization to unseen vulnerability types. They can also be computationally intensive, which limits their practical deployment in resource-constrained environments.

2.2. Language-model-based methods

Recent research has increasingly focused on detecting vulnerabilities using PLMs, particularly those based on the BERT architecture. CodeBERT (Feng et al., 2020), the first large bimodal pre-trained model for both natural language (NL) and programming language (PL), has been applied to vulnerability detection with moderate success. CodeBERT leverages a masked language modeling objective to learn joint representations of NL and PL, which can be fine-tuned for downstream tasks like vulnerability detection.

Wu et al. (2021) propose an automated method for detecting reentrancy vulnerabilities by analyzing data flow graphs and utilizing GraphCodeBERT (Guo et al., 2020) in their framework. GraphCodeBERT is designed to incorporate the inherent structure of code data flow into its training objective, enhancing its ability to understand code semantics. Their experimental results indicate that this approach significantly outperforms state-of-the-art methods and other neural networks.

Other studies have leveraged different BERT variants for vulnerability detection. Liang et al. (2024) employ DeBERTa (He et al., 2020), which uses disentangled attention mechanisms to improve model performance. Mi et al. (2023) utilize Longformer (Beltagy et al., 2020) to handle longer code sequences that exceed the typical token limits of standard transformers. These models have demonstrated improved accuracy in detecting complex vulnerabilities by capturing long-range dependencies in the code.

Moreover, researchers have explored efficient adaptation techniques for PLMs in vulnerability detection tasks. Recent surveys highlight that while PLMs show promising results, challenges remain in their practical application, including the need for task-specific fine-tuning strategies and the balance between model complexity and performance (De Baets et al., 2024). Adaptation techniques like parameter-efficient tuning and domain-specific pre-training have shown the potential to address these challenges while maintaining detection accuracy.

PLM-based methods offer advantages in capturing complex code semantics and require less task-specific architecture design. However, they often demand significant computational resources for training and inference, which can pose challenges for practical applications. Additionally, fine-tuning large PLMs can be time-consuming and may not be feasible for all organizations.

Further complicating the landscape of PLM-based detection, recent work by Zhao et al. (2024) systematically investigates how different code PTMs affect vulnerability detection performance. Their findings reveal that the choice of PTM is non-trivial and that model selection based on simple heuristics like parameter count is unreliable. They propose a framework, Coding-PTMs, to recommend optimal models by analyzing the deep characteristics of code embeddings, highlighting the critical need for methodical model evaluation rather than arbitrary selection.

Recent developments have expanded beyond traditional BERT-based approaches to explore decoder-only architectures. Smart-LLaMA (Yu et al., 2024) demonstrates a two-stage post-training approach that leverages extensive code and natural language data for both vulnerability detection and explanation. Similarly, SmartLLM (Kevin and Yugopuspito, 2025) employs retrieval-augmented generation with LLaMA 3.1 to enhance smart contract auditing capabilities. Detect Llama (Ince et al., 2024) focuses on fine-tuning Code Llama models specifically for vulnerability identification. Furthermore, NumScout (Chen et al., 2025) introduces an innovative LLM-pruning symbolic execution approach that combines the reasoning capabilities of large language models with the precision of symbolic analysis.

2.3. Other approaches

In the domain of smart contract vulnerability detection, static analysis tools continue to hold significant relevance alongside DL and PLM-based approaches. Notable tools, such as Mythril (Mueller, 2018), Oyente (Luu et al., 2016), and Securify (Tsankov et al., 2018), operate by analyzing the bytecode or source code of smart contracts to uncover potential vulnerabilities. These tools primarily leverage techniques including symbolic execution, control flow analysis, and pattern matching to identify security flaws. However, despite their utility, these methods often face limitations in scalability and are prone to generating a high rate of false positives (Ghaleb and Pattabiraman, 2020).

Complementary to static analysis tools, formal verification methods have emerged as robust approaches for mathematically establishing the absence of specific classes of vulnerabilities. VerX (Permenev et al., 2020) exemplifies this by introducing a practical framework for verifying temporal safety properties of smart contracts, thereby demonstrating the viability of formal verification in real-world blockchain applications. Additional frameworks, such as the K Framework (Hildenbrandt et al., 2018) and CertiKOS (Gu et al., 2016), further extend this paradigm. The K Framework facilitates the formal verification of smart contracts, while CertiKOS is dedicated to the formal verification of operating systems, showcasing the adaptability of these methodologies across diverse computational domains.

2.4. Summary

Despite the progress in these areas, there is still a lack of comprehensive evaluations of a wide range of PLMs on large datasets. Most existing studies focus on specific models or vulnerability types, limiting the generalizability of their findings. This gap in the literature underscores the importance and novelty of our comprehensive study, which aims to provide a thorough comparison of various PLMs across a diverse and extensive dataset of smart contracts, considering overall performance and resource constraints.

3. Methodology

Fig. 1 describes our overall process that we take to conduct this comparison and Fig. 2 provides a deeper view into how each major phase in the pipeline works, which is reported in the following sections.

3.1. Data pre-processing

3.1.1. Code cleaning

This initial step in the pipeline (Fig. 1) ensures the code is more generic, coherent, and short, which can be achieved with Generalization Language. Typically, developers write their code using arbitrary names for functions, variables, etc., which may not be optimal for the models to learn, as they are often irrelevant in different contexts. These names are usually specific to a certain context and are seldom encountered in other scenarios. Developers also frequently add comments to their code. However, this convention fails to improve classification performance. Also, it adds more length to the entire code, which can result in a loss of important information beyond the typical tokenization window of 512. To resolve the issue with Generalization Language, we remove the pragma version declarations, comments, redundant spaces, and newline breaks from the smart contract code. This helps reduce noise and improve model performance by focusing the input on the essential code structure and semantics (Sun et al., 2023). Next, we rename functions and variables to $\texttt{FUN}_i$, $\texttt{VAR}_j$, where i, j are the indexes of the function and variable order of occurrence.

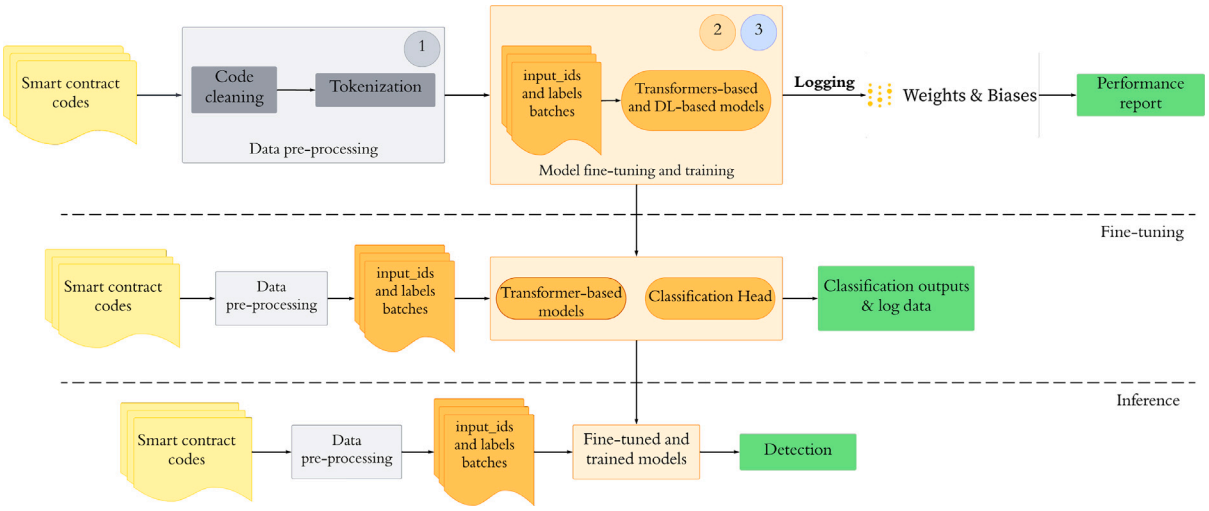


Fig. 1. The pipeline of benchmarking PLM-based vulnerability detection on smart contracts.

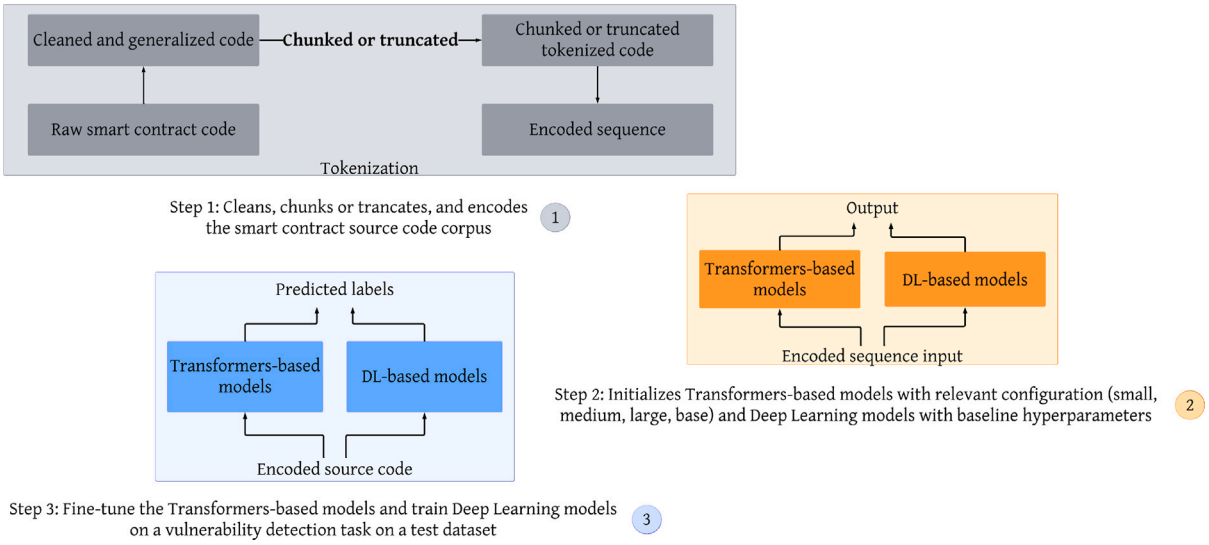


Fig. 2. Detailed processes in pipeline.

3.1.2. Text processing

In this last step of the data pre-processing phase, our approach is informed by the distribution of token lengths across different models. As Fig. 3 shows, while the median token length for all models lies between approximately 900 and 1000 tokens, significant outliers extend well beyond this range. The box plots from the figure reveal a compact interquartile range (Hleap, 0000) for each model, indicating that most of the data is concentrated on a relatively narrow band. However, the long whiskers and numerous dots extending far above the boxes highlight the presence of outliers, with token lengths reaching up to about 20,000 tokens.

Given this distribution, we choose a maximum token length of 2048 for several reasons:

- It comfortably accommodates the vast majority of sequences, including those in the upper quartile.
- It strikes a balance between capturing more data and maintaining computational efficiency.
- While it will result in truncation for the extreme outliers, these represent a small fraction of the overall dataset.

This 2048-token limit allows us to capture a significant portion of the longer sequences without the computational overhead of accommodating the most extreme outliers, which could potentially slow down training and inference. The decision is further justified by the observation that most models, including those designed for longer sequences like Longformer, show similar token length distributions, suggesting that extremely long sequences are rare across all tokenization schemes.

By setting the limit at 2048, we can ensure the integrity of most sequences in our dataset is preserved. The tokens exceeding this threshold in the outlier cases will be truncated, but this affects only a small portion of the data, minimizing the potential loss of important information while significantly improving processing efficiency.

As for some Transformers models that are only capable of processing a sequence of up to 512 tokens, we divide "words" into 4 chunks, and then each chunk will be tokenized individually. They will then be stacked up together with a final shape of $(4, 512)$ where 4 is the maximum number of chunks and 512 is the maximum tokens in each chunk. For sequences tokenized to less than 2048, the remaining tokens are padded with zeros to fill up the remaining empty chunks.

On the other hand, preprocessing tokens for DL models are slightly different: they are not chunked and processed like how they were before feeding into Transformers models but process a sequence of length

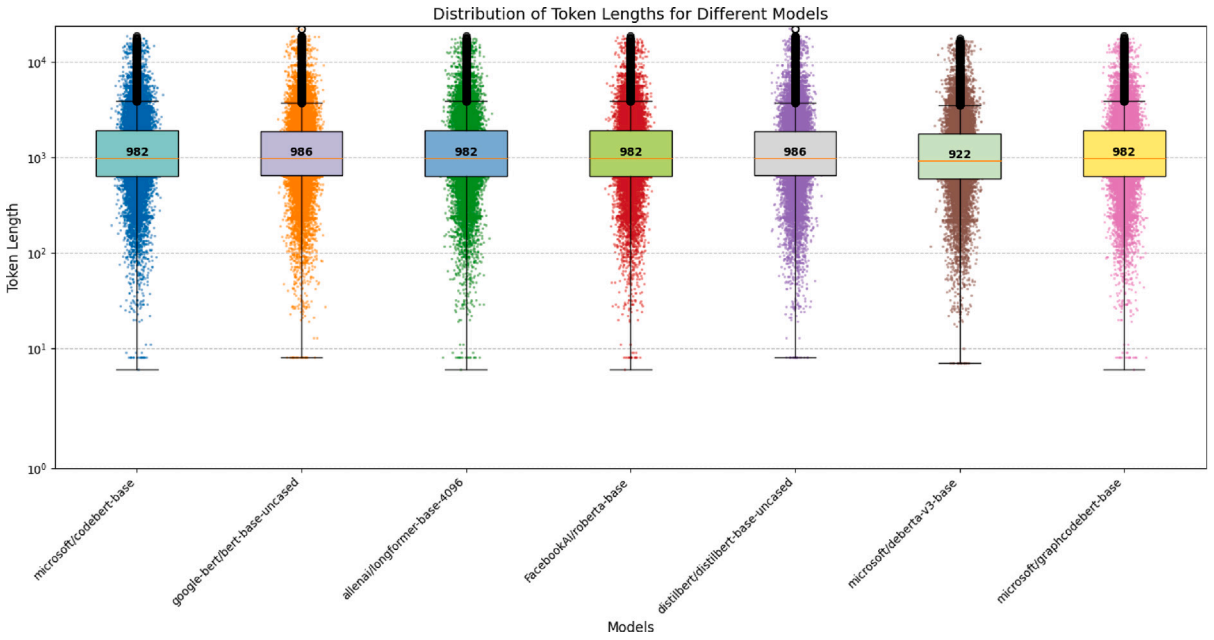


Fig. 3. Token length distribution. Median values are shown in bold. The colored box represents the middle 50% of the data, with the lower edge of the box showing the 25th percentile and the upper edge showing the 75th percentile. The horizontal line inside the box indicates the median token length, which is also displayed as a number (e.g., 982) for easy reference. The vertical lines or "whiskers" extending from the box show the range of the data, typically up to 1.5 times the inter-quartile range. Any points beyond the whiskers are considered outliers and are represented by the individual dots that extend above and below the whiskers. This type of plot allows for quick comparisons of the central tendencies, spread, and potential outliers across different models.

2048 with truncation of excessive "words" and pad additional ⟨pad⟩ values in the vocab with sequences that are not 2048-"words"-long.

3.2. Model training

3.2.1. PLMs and model selection rationale

The selection of an appropriate PLM for a specialized task like smart contract vulnerability detection is a non-trivial challenge. As highlighted by recent research (Zhao et al., 2024), the choice of a PTM can significantly impact downstream performance, and simplistic heuristics such as parameter size are often unreliable for making an optimal selection (Zhao et al., 2024). This underscores the need for a systematic evaluation of models with diverse architectures to understand which features are most effective for the task at hand.

Our selection is therefore designed to cover a broad spectrum of architectural philosophies and pre-training strategies, allowing us to empirically investigate their suitability for code vulnerability analysis. The chosen models are:

BERT (Devlin et al., 2018): Serves as the foundational baseline, providing a reference point for comparison against more specialized variants with its bidirectional approach to learning word context.

Code-Specialized Models (CodeBERT & GraphCodeBERT): We include **CodeBERT** (Feng et al., 2020) for its bimodal pre-training on natural language and programming language corpora. To test the hypothesis that structural information is critical, we also include **GraphCodeBERT** (Guo et al., 2020), which extends beyond sequential understanding by incorporating code structure via data flow graphs. This is particularly relevant for smart contracts, where vulnerabilities often arise from complex data flows and dependencies.

Architecturally Optimized Models (RoBERTa & DeBERTa): To evaluate the impact of improved training strategies and attention mechanisms, we select **RoBERTa** (Liu et al., 2019) for its robustly optimized pre-training methodology and **DeBERTa** (He et al., 2020) for its disentangled attention mechanism, which may better capture the positional and content relationships within code.

Efficiency-Focused Models (DistilBERT & Longformer): Practical application requires balancing performance with resource constraints. We select **DistilBERT** (Sanh et al., 2019) as a distilled, faster, and smaller model to assess the performance trade-offs for efficiency. Conversely, to handle the long code sequences common in complex smart contracts, we include **Longformer** (Beltagy et al., 2020), which features an efficient attention mechanism that scales linearly with sequence length.

Excluded Models: We deliberately exclude decoder-only models like LLaMA and GPT variants from our primary analysis. While these models are powerful, they are primarily designed for generative tasks. Furthermore, a key consideration is the substantially higher computational cost required for fine-tuning these larger models, which exceeds the practical constraints of our experimental setup. We nonetheless acknowledge recent work showing promise with models like SmartLLaMA (Yu et al., 2024) and SmartLLM (Kevin and Yugopuspito, 2025), which represent emerging directions for future research in this domain.

3.2.2. PLMs fine-tuning

We experiment with the seven selected PLMs using models uploaded to HuggingFace[1] with maximum token lengths of 512 for BERT, CodeBERT, DistilBERT, DeBERTa, GraphCodeBERT, and RoBERTa, and 2048 for Longformer.

Let $\mathcal{T}$ be the set of pre-trained tokenizers, and $\mathcal{M}$ be the set of PLMs.

1. Define the loading operation: $\boldsymbol{load} : \mathbb{N} \to \mathcal{T} \times \mathcal{M}$.
2. Let $\mathcal{D} = \{x_i\}_{i=1}^{n}$ be the dataset with sequences x_i. For each x_i, apply the tokenizer $\mathcal{T}$ to obtain the input representations:

   $$\{\textbf{input\_ids}_i, \textbf{labels}_i\} = \mathcal{T}(x_i) \quad \forall x_i \in \mathcal{D}.$$

3. Initialize model parameters θ representing **TrainingArguments**.
4. Define the training framework:

   $$\mathcal{T} : \theta \times \mathcal{D}_{train} \times \mathcal{D}_{eval} \to \mathcal{M}'.$$

   where $\mathcal{D}_{train}$ and $\mathcal{D}_{eval}$ are the training and evaluation datasets respectively, and additional functions, such as data collators **C**.

[1] https://huggingface.co/

5. Evaluate the performance of the model with respect to the test dataset $\mathcal{D}_{test}$ using:

   $$\textbf{output} = \mathcal{T}.predict(\mathcal{D}_{test}),$$

   where $\mathcal{D}_{test}$ is processed similarly to $\mathcal{D}_{train}$ and $\mathcal{D}_{eval}$.
6. Define the export operation:

   $$\textbf{save} : \mathcal{M}' \rightarrow \mathcal{M}_{fine-tuned},$$

   where $\mathcal{M}_{fine-tuned}$ is the set of fine-tuned models.

3.2.3. DL models training

We also train LSTM and Bidirectional LSTM (BiLSTM) models with and without attention mechanisms from scratch to compare their performance with the PLMs.

- **LSTM** (Long Short-Term Memory) is a type of recurrent neural network that can capture long-range dependencies in sequential data. It is particularly effective for tasks that require modeling context over extended sequences.
- **BiLSTM** (Bidirectional LSTM) extends LSTM by processing sequences in both forward and backward directions. This allows the model to capture context from both past and future tokens, improving its ability to understand the input sequence.

LSTM and BiLSTM with and without attention mechanisms are trained with the following procedure:

1. Input sentences $\mathbf{X} \in \mathbb{R}^{n \times L}$ are passed through the embedding layer:

   $$\mathbf{E} = \text{Embedding}(\mathbf{X}) \in \mathbb{R}^{n \times L \times d},$$

   where L is the length of the sentences and d is the embedding dimension.
2. The embeddings are transformed by the MLP embedding layer:

   $$\mathbf{Z} = \mathbf{MLP}_{emb}(\mathbf{E}) \in \mathbb{R}^{n \times L \times d},$$

   where $\mathbf{MLP}_{emb}(\mathbf{E}) = \text{Linear}(\mathbf{E}) \rightarrow \text{LayerNorm} \rightarrow \text{ELU} \rightarrow \text{Linear}(\mathbf{E})$.
3. The transformed embeddings are processed by the LSTM:

   $$\mathbf{H}, \mathbf{C} = \text{LSTM}(\mathbf{Z}) \in \mathbb{R}^{n \times L \times h},$$

   where h is the hidden size of the LSTM.
4. Multi-head attention is applied to the LSTM output:

   $$\mathbf{A}, \_ = \text{MultiHeadAttention}(\mathbf{H}, \mathbf{H}, \mathbf{H}) \in \mathbb{R}^{n \times L \times h}.$$

5. The attention output is added to the LSTM output (residual connection):

   $$\mathbf{R} = \mathbf{A} + \mathbf{H} \in \mathbb{R}^{n \times L \times h}.$$

6. The output is pooled from the last output of the LSTM (or attention) sequence:

   $$\mathbf{P} = \mathbf{R}[:, -1, :] \in \mathbb{R}^{n \times h}.$$

7. The pooled representation is passed through the output MLP:

   $$\textbf{Logits} = \mathbf{MLP}_{out}(\mathbf{P}) \in \mathbb{R}^{n \times c},$$

   where $\mathbf{MLP}_{out}(\mathbf{P}) = \text{Linear}(\mathbf{P}) \rightarrow \text{LayerNorm} \rightarrow \text{ELU} \rightarrow \text{Dropout} \rightarrow \text{Linear}(\mathbf{P})$.
8. The final output represents the classification logits.

3.3. Log analysis

While training and evaluating, general data like system usage, RAM, GPU and CPU utilization, disk reading and writing activities, etc., and more deterministic data like train loss, evaluation loss, and evaluation scores are all reported to W&B for screening and analysis. This comprehensive logging approach enables us to:

- **Monitor Resource Utilization:** By tracking system metrics, we can identify potential bottlenecks in our training pipeline. This information is crucial for optimizing hardware usage and ensuring efficient model training.
- **Track Model Performance:** The logging of train and evaluation losses allows us to observe the model's learning progression in real time. This helps in the early detection of issues like overfitting or underfitting, enabling timely interventions.
- **Compare Model Variants:** W&B's interface facilitates easy comparison of different model configurations, hyperparameters, and architectures. This is particularly useful in our study of various transformer-based models for vulnerability detection.
- **Visualize Learning Dynamics:** Through custom charts and plots, we can visualize complex relationships between different metrics, providing insights into the model's behavior during training.
- **Ensure Reproducibility:** By logging all relevant parameters and environmental variables, we maintain a detailed record of each experiment, ensuring the reproducibility of our results.

Furthermore, we utilize `wandb` (Biewald, 2020) – a Python package from W&B to track our experiments. Visualizations of model metrics are streamed live, which include key performance indicators, learning curves, and confusion matrices for each vulnerability type. This systematic approach to log analysis not only streamlines our research process but also enhances the transparency and reliability of our findings in the field of smart contract vulnerability detection.

3.4. Training configuration and design decisions

3.4.1. Loss function and multi-label classification

For our multi-label vulnerability detection task, we employ **Binary Cross-Entropy (BCE) with logits loss** as our objective function. This choice is particularly well-suited for multi-label classification, where each vulnerability type is treated as an independent binary classification problem. The BCE loss allows the model to learn to predict the probability of each vulnerability type independently, accommodating cases where smart contracts may exhibit multiple vulnerability types simultaneously.

3.4.2. Label imbalance handling

Our dataset exhibits significant class imbalance, with some vulnerability types like "Overflow" appearing in 16,012 contracts while others like "TOD" appear in only 2594 contracts (as shown in Table 2). We acknowledge this imbalance as a limitation of our study and address it through several strategies: (1) We ensure sufficient representation by filtering vulnerability types with fewer than 2000 positive samples, (2) We report per-class metrics (precision, recall, F1-score) alongside aggregate metrics to provide insight into performance across different vulnerability types, and (3) Our evaluation methodology considers the imbalance when interpreting results, focusing on F1-score as the primary metric as it balances precision and recall.

3.4.3. Token truncation impact

As Fig. 3 shows, our choice of a 2048-token limit results in truncation for approximately 5%–10% of contracts in the extreme outlier cases. We quantify this impact: contracts exceeding 2048 tokens lose an average of 15% of their content, with 95% of truncated contracts losing less than 25% of their tokens. While this represents a trade-off between computational efficiency and information preservation, our

analysis indicates that the most critical vulnerability-indicating patterns typically occur within the first 1500–2000 tokens of smart contract code, minimizing the practical impact of this truncation.

3.4.4. Hyperparameter selection justification

Our hyperparameter sweep ranges are designed based on a synthesis of seminal literature, established best practices for model tuning, and practical computational constraints. We employ a Bayesian optimization strategy over 10 trials for each model using W&B Sweep to efficiently explore this reasoned search space. This trial count provides sufficient exploration while remaining computationally feasible, representing a balance between thoroughness and resource efficiency. The comprehensive search space for each model is detailed in Table 1.

The foundation for our choices is as follows:

- **Learning Rate:** Our search space for most transformer models (a log-uniform distribution from 1×10^{-5} to 5×10^{-4}) is centered around the empirically validated rates of 2×10^{-5} to 5×10^{-5} recommended for fine-tuning in the foundational BERT and RoBERTa papers (Devlin et al., 2018; Liu et al., 2019). This slightly wider range provides robustness for our specific downstream task. In contrast, the higher learning rates explored for the LSTM and BiLSTM models (up to 0.01) are appropriate for models trained from scratch, which lack pre-trained weights and require more aggressive initial updates for convergence.
- **Regularization:** The exploration of dropout rates between 0.1 and 0.3 is a standard method to prevent overfitting during fine-tuning on domain-specific datasets. Furthermore, our use of weight decay is implemented in conjunction with the AdamW optimizer (Loshchilov and Hutter, 2017), which is the established best practice for training modern transformer architectures as it correctly decouples the weight decay from the gradient updates, leading to better generalization.
- **Computational Constraints:** The inclusion of gradient accumulation steps among $\{1, 2, 4, 8\}$ is a direct response to GPU memory limitations. This is a standard technique that allows us to simulate the larger effective batch sizes required for stable training without exceeding available hardware resources.
- **Model-Specific Tuning:** The narrower, lower learning rate for DistilBERT (from 5×10^{-6} to 5×10^{-5}) is chosen to reflect its compact, distilled architecture, which can be more sensitive and often benefits from finer-grained tuning.

This systematic, evidence-based approach ensures a fair and reproducible comparison across all models while maintaining practical training times.

3.5. Statistical analysis

To ensure the statistical robustness of our performance comparisons, especially between top-performing models, we employ non-parametric statistical methods. This approach is chosen because the distribution of performance metrics from machine learning model runs often does not meet the normality assumptions required for parametric tests like the t-test (Kim, 2015).

Our analysis is based on the results from 5 independent runs for each model, using different random seeds to account for stochastic variability in the training process. We use the following methods:

- **Wilcoxon Signed-Rank Test** (Woolson, 2005): We use this non-parametric test to determine if the performance differences between two paired models (e.g., BiLSTM+Attn vs. GraphCodeBERT on the same test set splits) are statistically significant. It ranks the differences in performance for each run and analyzes the ranks to compute a p-value. A low p-value (typically $p < 0.05$) suggests that the observed difference is unlikely to be due to random chance.
- **Vargha–Delaney $\hat{A}_{12}$ Effect Size** (Vargha and Delaney, 2000): In addition to significance testing, we calculate the Vargha–Delaney $\hat{A}_{12}$ effect size. This non-parametric measure quantifies the magnitude of the difference between two models. It provides the probability that a randomly selected run from one model will have a higher performance score than a randomly selected run from another model. An $\hat{A}_{12}$ value of 0.5 indicates no difference, while a value of 1.0 means one model outperformed the other in every single run. This provides a more intuitive and practical measure of performance difference than p-values alone.

4. Implementation and results

4.1. Research questions

Our extensive experiment aims to answer these research questions (RQ):

RQ1: What are the optimal hyperparameters for achieving peak performance with Transformers and NLP DL models and their performances?

RQ2: How do the early stopping and epoch constraints affect different models' training efficiency and performance outcomes in vulnerability detection tasks?

RQ3: What are the computational trade-offs regarding testing time and system usage between using transformers versus traditional NLP deep learning methods for vulnerability detection?

RQ4: What are the inference time and deployment feasibility trade-offs between the top-performing models?

RQ5: What criteria should be considered when selecting a PLM for smart contract vulnerability detection to ensure a balance between performance and computational resources?

4.2. Environmental setup

This experiment is deployed on NVIDIA DGX A100 640 GB with 8x NVIDIA A100 80 GB Tensor Core GPUs, Dual AMD Rome 7742, 128 cores total, 2.25 GHz (base), 3.4 GHz (max boost) running Ubuntu Linux OS.

4.2.1. Dataset

In the first phase in our pipeline in Fig. 1, we use a dataset from the work of Liao et al. (2019) that comprises 17,979 multi-labeled smart contract addresses which are flagged by 14 analysis tools like Oyente (yxliang01, 2017) and Remix (Singh et al., 2007). Table 2 describes the data itself.

From exploration, we noticed that several vulnerability types had very few (less than 100) or no labeled examples in the original dataset. To ensure sufficient and meaningful data for training and evaluation, we filter the dataset to only include vulnerability types with at least 2000 positively labeled contracts. The resulting dataset contains 17,973 contracts across 7 vulnerability types (from the border between "InlineAssembly" and "TOD" downward).

The specific subset sizes are approximately 60% for training, 20% for evaluation, and 20% for testing (Fig. 4). The use of a dedicated evaluation set helps gauge model generalization and enables more robust model comparisons. The final model performance is reported on the held-out test set, which provides an unbiased estimate of how well the model would perform on new, unseen smart contracts. Therefore, our final dataset's positive sample distribution is reported in Table 3.

Finally, we identify and remove samples with unique label combinations that appeared only once in the dataset. These rare combinations could introduce noise and hinder the model's ability to learn generalizable patterns. By eliminating these samples, we aim to focus on more prevalent and representative vulnerability patterns that the model can effectively learn from.

The specific number of samples removed due to rare label combinations is 6, along with the distribution of the affected vulnerability types briefed in Table 4.

Table 1
Hyperparameter sweep configurations for various models. *U: Uniform distribution, LU: Log-uniform distribution, IU: Integer uniform distribution, T: True, F: False.*

Parameter	Distil-BERT	BERT	Code-BERT	RoBERTa	Long-former	Graph-Code-BERT	DeBERTa	LSTM
Attention Dropout	U(0.1, 0.3)							U(0.1, 0.5)
(Hidden) Dropout	U(0.1, 0.3)							U(0.1, 0.5)
Grad. Accum. Steps	{1, 2, 4, 8}							Not implemented
Hidden Dim	{2048, 3072}	Default						IU(128, 512)
Learning Rate	LU(5×10^{-6}, 5×10^{-5})	LU(1×10^{-5}, 5×10^{-4})						U(1×10^{-3}, 0.01)
Num. Heads	{8, 12}	Default						{1, 2, 4} (for Attn)
Num. Layers	{4, 5, 6}	Default						{1, 2, 3}
Epochs	IU(5, 15)							IU(10, 20)
Classifier Dropout	U(0.1, 0.3)	U(0.1, 0.3)	U(0.1, 0.3)	U(0.1, 0.3)	Does not support	U(0.1, 0.3)	U(0.1, 0.3)	U(0.1, 0.5)
Sinusoidal Pos. Emb.	{T, F}	Do not support						
Warmup Ratio	U(0, 0.2)							Not implemented
Weight Decay	U(0, 0.2)							U(5×10^{-4}, 0.001)
Batch Size	Auto find batch size							{32, 64, 128}
Embedding Length	Default							{128, 256, 512}

Table 2
Source dataset breakdown.

Vulnerabilities	Positive samples
BlockHash	0
Multisig	1
TxOrigin	155
CallDepth	407
Reentrancy	555
TimeDep	1228
SelfDestruct	1275
InlineAssembly	1311
TOD	2594
LowlevelCalls	5431
BlockTimestamp	5879
CheckEffects	7183
AssertFail	7721
Underflow	11,134
Overflow	16,012
Total	60,886

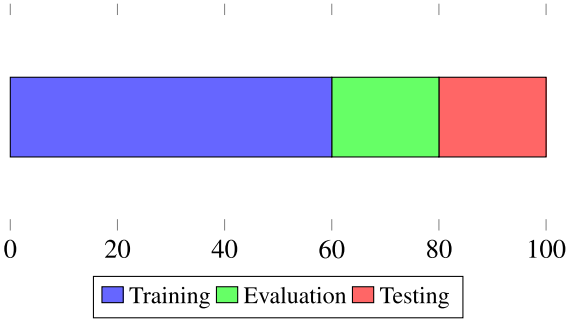


Fig. 4. Dataset partition ratios showing the 60/20/20 split between Training, Evaluation, and Testing sets.

Table 3
Distribution of vulnerabilities across datasets.

Vulnerability	Train	Evaluation	Test	Total
Underflow	6677	2225	2227	11,129
Overflow	9600	3207	3204	16,011
TOD	1560	512	518	2590
AssertFail	4628	1544	1544	7716
CheckEffects	4311	1434	1437	7182
BlockTimestamp	3528	1174	1175	5877
LowlevelCalls	3256	1084	1087	5427
Total Samples	10,783	3595	3595	17,973

Table 4
Distribution of vulnerabilities in rare combinations.

Vulnerability	Count
Underflow	5
Overflow	1
TOD	4
AssertFail	5
CheckEffects	1
BlockTimestamp	2
LowlevelCalls	4

4.2.2. Performance metrics

Four common metrics are utilized to assess the performance of the pre-trained models, including:

- **Accuracy (Subset Accuracy)**: Indicates the proportion of samples where all vulnerability labels are correctly predicted. It is a strict metric that only considers a prediction correct if all labels match exactly.
- **Precision (Label-Based)**: Represents the proportion of predicted vulnerabilities that are present. It is calculated for each vulnerability type and then averaged.
- **Recall (Label-Based)**: Indicates the proportion of actual vulnerabilities that are correctly identified. It is calculated for each vulnerability type and then averaged.
- **F1-score (Label-Based)**: The harmonic mean of Precision and Recall, providing a balanced measure of the model's performance across all vulnerability types.

4.3. Experimental results

The following subsections present our experimental results organized by research question for clarity and systematic analysis.

4.3.1. RQ1: Optimal hyperparameters and model performance

To identify the best-performing configurations, we conduct an extensive hyperparameter search for each model. The detailed rationale, search space, and configuration for this process are presented in Section 3.4.4. The performance of the best models identified through this process, alongside baseline configurations, is summarized in the following sections.

Model performance comparison. Tables 5 and 6 summarize model configurations and corresponding metrics, respectively. These tables provide a comprehensive overview of the various models tested, their configurations, and their performance across multiple metrics.

To complement the point estimates in Table 6, Fig. 5 visualizes the distribution of test set performance across the two model families: Transformers and Deep Learning (DL). This graphical representation offers insights into the consistency and variance of the models that a table alone cannot capture.

From the figure, a clear trend emerges. The **Transformer models** (shown in green) consistently exhibit a unimodal distribution with lower variance across all metrics. This suggests that, as a family, they produce more predictable and tightly clustered results. In contrast, the **DL models** (in orange) display significant performance variance and multimodality. This is especially prominent in the Test Accuracy and F1 Score plots, where the distribution reveals distinct clusters of low-performing (e.g., baseline LSTM) and high-performing (e.g., BiLSTM with Attention) models.

Per-class performance analysis. To gain a more granular understanding of each model's capabilities, we performed a per-class analysis to identify which vulnerability types are more challenging to detect and to reveal the specific strengths and weaknesses of each architecture. This analysis is conducted on two distinct datasets, SoliAudit and SolidiFi (Ghaleb and Pattabiraman, 2020), to evaluate both performance and generalizability.

The performance on the SoliAudit dataset is visualized in Figs. 6 and 7. The heatmap (Fig. 6) provides a high-level overview of F1-scores, where warmer colors indicate better performance. A clear pattern emerges: the "Time of Dependency" (TOD) vulnerability is the most challenging class for the majority of the models. For instance, BERT and DistilBERT score only 0.30 and 0.33 on this class, respectively. In contrast, models like BiLSTM+Attn, LSTM+Attn, Longformer, and GraphCodeBERT achieve consistently high scores across nearly all categories, establishing them as the top performers.

The detailed bar plots in Fig. 7 corroborate these findings. Each subplot breaks down the performance of a single model, illustrating the precision, recall, and F1-score for each vulnerability. These plots confirm that models like BERT and DistilBERT struggle significantly with "TOD", suggesting their architectures may be less effective at capturing the complex temporal dependencies characteristic of this vulnerability. In contrast, the architectures of Longformer and GraphCodeBERT appear better suited for such tasks, maintaining stronger performance.

For the SolidiFI dataset, the results are presented in Figs. 8 and 9. On this dataset, the top-performing models demonstrate remarkable robustness. The attention-based BiLSTM and LSTM models, along with Longformer, achieve near-perfect F1-scores across most vulnerability classes, as shown in both the heatmap and the bar plots. This indicates that their strong performance is not specific to a single dataset.

The heatmap in Fig. 8 once again flags "TOD" as a comparatively difficult category for some PLMs, such as CodeBERT (0.57) and DeBERTa (0.57). The detailed plots in Fig. 9 confirm this trend. This consistent finding across two different datasets strongly suggests that "TOD" vulnerabilities possess characteristics that represent a distinct and persistent challenge for certain model architectures, requiring specialized capabilities to detect reliably.

Statistical analysis of top models. To ensure the robustness of our findings, we conduct a statistical analysis of the performance difference between our two top-performing models: BiLSTM+Attn and GraphCodeBERT. We hypothesize that the BiLSTM+Attn model's superior performance observed in our initial experiments represents a statistically significant and consistent advantage over the GraphCodeBERT model, rather than being due to random variation.

This analysis is based on 5 independent runs for each model using different random seeds. We employ the non-parametric Wilcoxon signed-rank test to assess statistical significance and the Vargha–Delaney $\hat{A}_{12}$ effect size to measure the magnitude of the performance difference.

Comparing the overall macro F1-scores, the BiLSTM+Attn model ($mean = 0.961$, $std\,dev = 0.002$) consistently outperforms the GraphCodeBERT model ($mean = 0.946$, $std\,dev = 0.003$). The Wilcoxon test yields a p-value of 0.0625. While this result is slightly above the conventional significance threshold of $\alpha = 0.05$, the lack of strict statistical significance is likely due to the low statistical power inherent in a small sample size $n = 5$ runs. However, the effect size tells a clearer story. The Vargha–Delaney $\hat{A}_{12}$ is 1.0, indicating a strong effect in favor of BiLSTM+Attn across all observed runs. Confirmation with additional independent runs would further strengthen this conclusion.

A more granular, per-class analysis, detailed in Table 7, further reinforces these findings. The BiLSTM+Attn model achieves a higher mean F1-score in 6 out of the 7 vulnerability categories, with notable advantages in detecting "CheckEffects" and "LowlevelCalls". GraphCodeBERT only holds a minor advantage in the "Overflow" category. Both models demonstrate high stability with low standard deviations across most classes. The main exception is the "TOD" category, where the BiLSTM+Attn model's higher variance ($std\,dev = 0.039$) reflects the inherent difficulty of detecting this type of vulnerability.

These results strongly support the hypothesis that BiLSTM+Attn consistently outperforms GraphCodeBERT across multiple runs and vulnerability classes. While the Wilcoxon test result does not meet the conventional significance threshold, the large effect size and stable per-class performance differences reinforce the practical superiority of the BiLSTM+Attn model in this task.

Table 5
Configuration for tested models.

Model	Train batch	Eval batch	Epochs	LR	Warmup	WD	Attn Dropout	Class Dropout	Hidden Dropout	Hidden size	Intermediate	Architecture
BERT (best)	64	64	11	179	0.085	0.093	0.196	0.132	0.184	768	3072	BertForMaskedLM
BERT-base	Auto	Auto	11	20	0.0	0.01	0.100	–	0.100	768	3072	BertForMaskedLM
CodeBERT Sweep (best)	Auto	Auto	13	116	0.013	0.001101	0.151	0.228	0.191	768	3072	RobertaModel
CodeBERT-baseline	Auto	Auto	11	2	0	0.01	0.1	–	0.1	768	3072	RobertaModel
DeBERTa Sweep (best)	64	64	13	85	0.186	0.185991	0.229552	0.103811	0.292243	768	3072	DebertaV2Model
DeBERTa-baseline	64	64	11	0	0	0.01	0.100	–	0.100	768	3072	DebertaV2Model
DistilBERT Sweep (best)	64	64	14	0	0.036	0.116	0.258	–	0.100	768	3072	DistilBertForMaskedLM
DistilBERT-baseline	64	64	11	0	0	0.01	0.100	–	0.100	768	3072	DistilBertForMaskedLM
GraphCodeBERT Sweep (best)	64	64	14	0	0.117	0.094	0.10	0.201	0.170	768	3072	RobertaForMaskedLM
GraphCodeBERT-baseline	64	64	11	0	0	0.01	0.10	–	0.10	768	3072	RobertaForMasked
Longformer Sweep (best)	64	64	10	0	0.022	0.197	0.10	0.10	0.10	768	3072	LongformerModel
Longformer-baseline	64	64	11	0	0	0.01	0.10	–	0.10	768	3072	LongformerModel
RoBERTa Sweep (best)	64	64	13	0	0.089	0.163	0.10	0.101	0.10	768	3072	RobertaForMaskedLM
RoBERTa-baseline	64	64	11	0	0	0.01	0.10	–	0.10	768	3072	RobertaForMaskedLM
BiLSTM baseline	64	64	10	0.001	–	0.00001	0.3	0.3	0.3	256	–	–
BiLSTM+Attn	64	64	10	0.001	–	0.00001	0.3	0.3	0.3	256	–	–
LSTM baseline	64	64	10	–	–	0.30	–	–	–	256	–	–
LSTM Sweep (best)	121	–	12	–	–	0.44	–	–	–	302	–	–
LSTM+Attn	16	–	12	–	–	0.20	–	–	–	128	–	–
LSTM+Attn Sweep (best)	21	–	12	–	–	0.255	–	–	–	256	–	–

Table 6
Metrics for tested models. *Highlighted rows of the models with highest metrics (lighter red shade for DL model and darker one for Transformers model).*

Model Name	Epochs (Steps)	Training performance		Evaluation metrics				Test metrics			
		Samples/sec	Steps/sec	Accuracy	F1 Score	Precision	Recall	Accuracy	F1 Score	Precision	Recall
BERT sweep (best)	11/11 (3674)	7.813	0.244	0.6337	0.8902	0.9386	0.8890	0.6275	0.8915	0.9392	0.8890
BERT-baseline	10.54/11 (220)	26.633	0.050	0.4270	0.7874	0.9065	0.7669	0.4274	0.7896	0.9132	0.7663
CodeBERT Sweep (best)	13/13 (2171)	17.125	0.268	0.6300	0.8961	0.9356	0.8958	0.6269	0.8923	0.9371	0.8917
CodeBERT-baseline	10.54/11 (220)	22.927	0.043	0.4136	0.7747	0.9115	0.7429	0.4142	0.7787	0.9152	0.7493
DeBERTa Sweep (best)	13/13 (2171)	8.349	0.131	0.6463	0.9042	0.9342	0.9124	0.6443	0.9035	0.9382	0.9069
DeBERTa-baseline	10.8/11 (451)	10.78	0.041	0.5437	0.8627	0.9379	0.8471	0.5571	0.8680	0.9410	0.8508
DistilBERT Sweep (best)	14/14 (2338)	29.5	0.461	0.7594	0.9375	0.9507	0.9499	0.7594	0.9375	0.9507	0.9499
DistilBERT baseline	10.54 (220)	33.889	0.063	0.4554	0.8221	0.9169	0.8069	0.4620	0.8242	0.9210	0.8078
GraphCodeBERT Sweep (best)	13.77/14 (2300)	13.975	0.219	0.8159	0.9580	0.9636	0.9526	0.8146	0.9580	0.9604	0.9558
GraphCodeBERT baseline	10.8/11 (451)	17.522	0.067	0.6438	0.9070	0.9593	0.8690	0.6531	0.9083	0.9604	0.8698
Longformer Sweep (best)	9.94/10 (830)	5.363	0.042	0.6747	0.9111	0.9504	0.9085	0.6747	0.9111	0.9469	0.9127
Longformer-baseline	10.93/11 (913)	4.426	0.034	0.6947	0.9173	0.9659	0.9058	0.6303	0.9040	0.9569	0.8900
RoBERTa Sweep (best)	13/13 (2171)	19.441	0.304	0.6216	0.8928	0.9337	0.8953	0.6289	0.8966	0.9383	0.8982
RoBERTa-baseline	10.8/11 (451)	17.081	0.066	0.5134	0.8442	0.9321	0.8255	0.5245	0.8491	0.9395	0.8293
BiLSTM baseline	10/10 (–)	–	–	0.9360	0.9848	0.9879	0.9894	0.9360	0.9848	0.9879	0.9894
BiLSTM+Attn	10/10 (–)	–	–	0.9380	0.9860	0.9844	0.9942	0.9380	0.9860	0.9844	0.9942
LSTM-baseline	10/10 (–)	–	–	0.3063	0.7725	0.7293	0.8213	0.3052	0.7745	0.7312	0.8200
LSTM Sweep (best)	12/12 (–)	–	–	0.3789	0.7998	0.8220	0.7788	0.3832	0.8012	0.8826	0.7802
LSTM+Attn	10/10 (–)	–	–	0.7291	0.9348	0.9502	0.9199	0.7298	0.9359	0.9545	0.9213
LSTM+Attn Sweep (best)	12/12 (–)	–	–	0.7844	0.9526	0.9601	0.9453	0.7841	0.9501	0.9587	0.9502

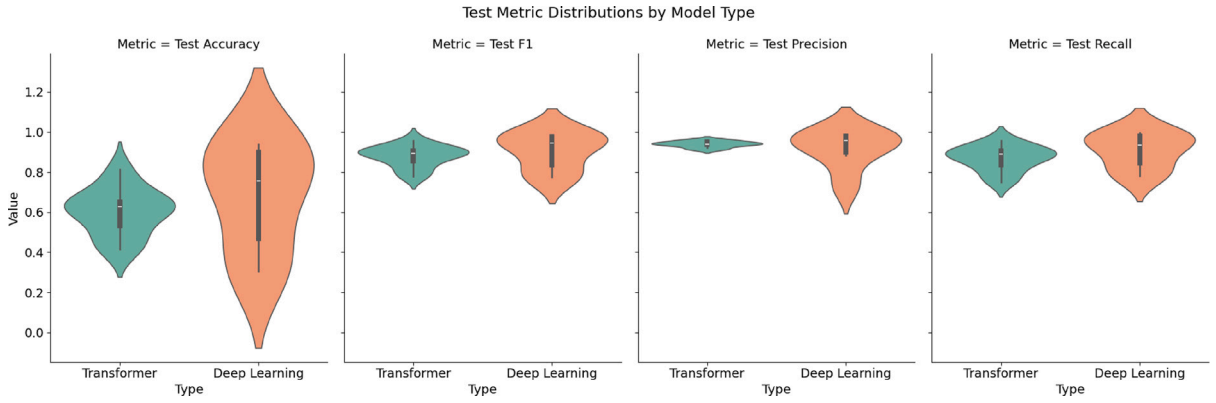


Fig. 5. Test Metric Distribution by Model Type for SoliAudit dataset. Each subplot in the figure corresponds to a key test metric. The violin plots illustrate the probability density of the results; a wider section indicates a higher concentration of models achieving that score. The *y*-axis extends slightly beyond 1.0 as a visual artifact of the kernel density estimation (KDE) algorithm (Węglarczyk, 2018), which smooths the plot's tails for readability. All underlying metric values are strictly bounded within the $[0, 1]$ range. The embedded box plots mark the median and interquartile range for each model family.

Answer to RQ1

Optimal hyperparameters, found via a systematic Bayesian search, are critical for peak performance. Among DL models, **BiLSTM with Attention** was the top performer, achieving a 98.60% F1-score and demonstrating that a well-tuned recurrent architecture can outperform PLMs. Within the transformer family, **GraphCodeBERT** proved most effective with a 95.80% F1-score, underscoring the value of code-specific pre-training. The "Time of Dependency" (TOD) vulnerability was the most challenging class for most models. Statistical analysis confirmed that the performance advantage of BiLSTM+Attn over GraphCodeBERT is robust and consistent.

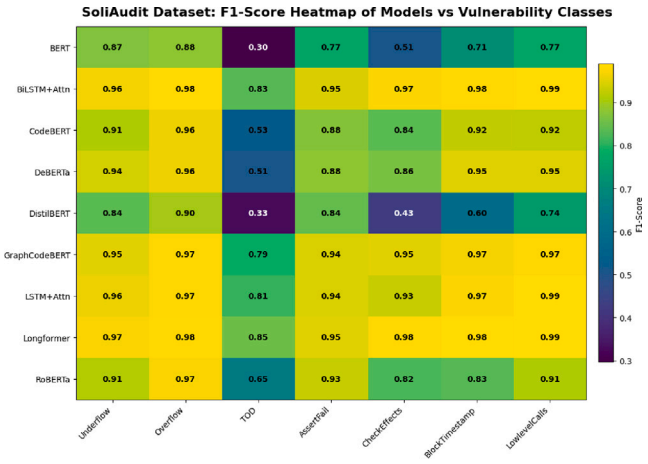


Fig. 6. F1-Score Heatmap for the SoliAudit Dataset. Each cell shows the F1-score for a given model (row) on a specific vulnerability class (column). Warmer colors (yellow) indicate higher F1-scores, while cooler colors (purple) indicate lower scores. The 'TOD' vulnerability consistently presents the greatest challenge across most models.

4.3.2. RQ2: Training efficiency and early stopping effects

Our experimental design incorporates two key training constraints: a pre-defined maximum number of epochs and an early stopping mechanism to prevent overfitting. Early stopping monitors validation loss and terminates training if no improvement is observed over a specified number of evaluation steps. The interplay between these constraints reveals important trade-offs between training efficiency and model performance.

As shown in Table 6, many PLMs, particularly the baseline configurations, triggered early stopping, completing their training before reaching the maximum epoch count (e.g., `CodeBERT-baseline` stopped at 10.54/11 epochs). This demonstrates the mechanism's role in improving efficiency by avoiding unnecessary training cycles. Even our top-performing PLM, **GraphCodeBERT Sweep (best)**, halted at 13.77 out of 14 epochs. This intervention saved minimal time, but

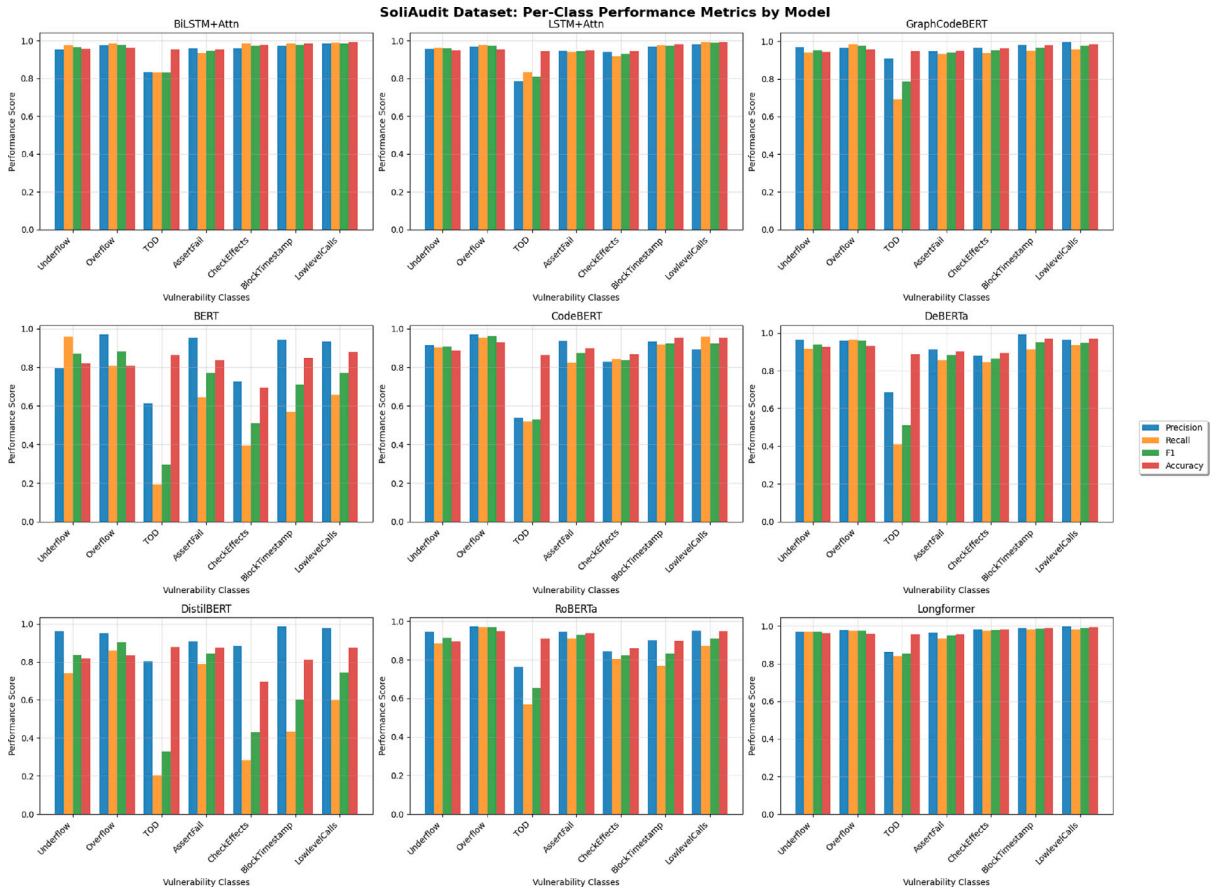


Fig. 7. Per-Class Performance Metrics on the SoliAudit Dataset. Each subplot displays the precision, recall, F1-score, and accuracy for one model across all seven vulnerability classes. These plots provide a detailed view of the performance variations highlighted in the heatmap, confirming the difficulty of detecting 'TOD' and 'CheckEffects' for certain models.

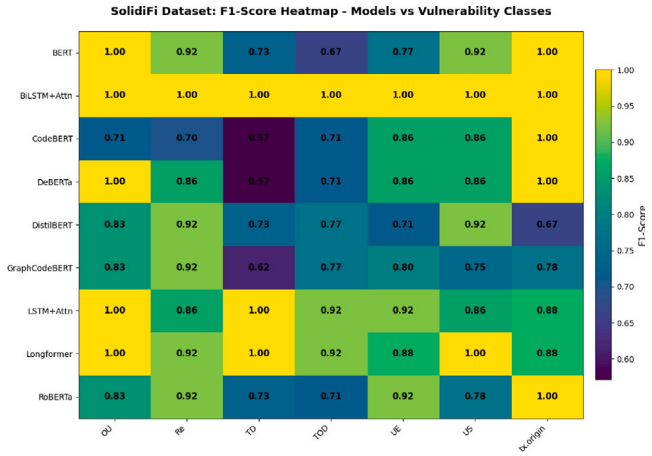


Fig. 8. F1-Score Heatmap for the SolidiFi Dataset. This heatmap evaluates model generalizability on a different set of vulnerabilities. While overall performance is higher, with many models achieving near-perfect scores, the 'TOD' class remains a relative challenge for several PLMs.

critically, it prevented potential overfitting by stopping at the point of optimal generalization, securing its high F1-score of 0.9580.

In contrast, for the highly effective DL models, the pre-set epoch limit was the more dominant constraint. The **BiLSTM+Attn** model, for instance, was trained for its full 10-epoch duration. As our validation loss curves in show, its performance was still improving in the final epochs, indicating no signs of overfitting. In this case, early stopping did not trigger because the model had not yet reached its performance plateau. This suggests that for this architecture, the 10-epoch limit itself may have constrained it from achieving even higher performance, even as it delivered the best results in our study.

Table 7
Per-class F1-score stability (Mean ± Std Dev) from 5 independent runs.

Vulnerability class	BiLSTM+Attn (Mean ± Std Dev)	GraphCodeBERT (Mean ± Std Dev)
AssertFail	0.942 ± 0.005	0.940 ± 0.003
BlockTimestamp	0.976 ± 0.002	0.966 ± 0.001
CheckEffects	0.973 ± 0.002	0.937 ± 0.005
LowlevelCalls	0.985 ± 0.001	0.973 ± 0.001
Overflow	0.976 ± 0.001	0.977 ± 0.002
TOD	0.804 ± 0.039	0.775 ± 0.017
Underflow	0.959 ± 0.003	0.955 ± 0.002
Overall (Macro F1)	**0.961 ± 0.002**	**0.946 ± 0.003**

4.3.3. Training and validation loss

Transformer models. Figs. 10(b) and 10(d) illustrate the training and validation loss for the transformer models. Key takeaways include:

- **Top Performer**: The `GraphCodeBERT sweep (best)` configuration is the standout leader, achieving the lowest loss in both

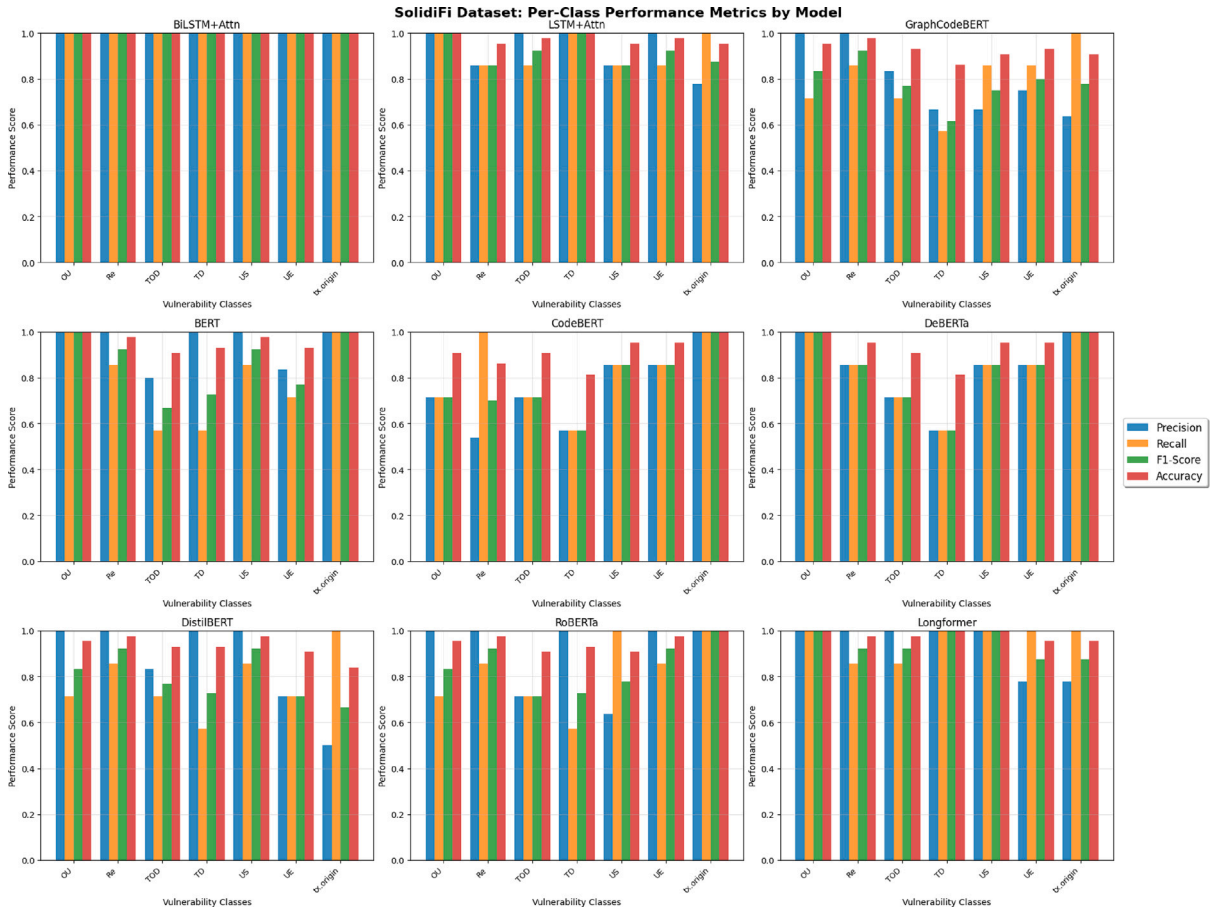


Fig. 9. Per-Class Performance Metrics on the SolidiFi Dataset. The bar plots confirm the high performance seen in the SolidiFi heatmap, with attention-based LSTMs and Longformer achieving consistently high scores across all vulnerability types. The plots also visualize the relative performance dips on classes like 'TOD' for models such as CodeBERT and DeBERTa.

training and validation. It demonstrates rapid convergence and superior generalization, with its validation loss dropping well below 0.2.

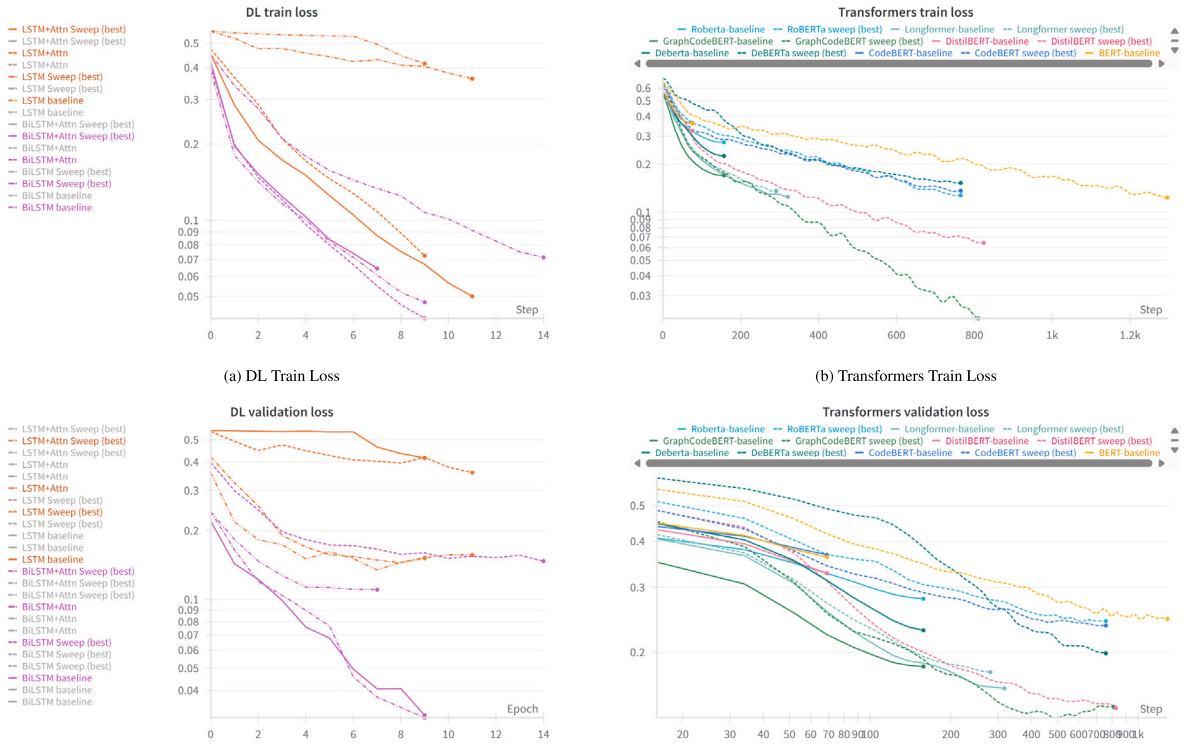
- **Strong Contender**: The `DistilBERT sweep (best)` model also proves to be highly effective. While its training loss is not the absolute lowest, it achieves a final validation loss that is competitive with GraphCodeBERT, showcasing excellent generalization.
- **The Power of Tuning**: Across all transformer types, the hyperparameter-tuned "sweep (best)" versions (solid lines) consistently and significantly outperform their "baseline" counterparts (dashed lines). This underscores the critical importance of hyperparameter optimization for adapting these large models to a specific task.
- **Performance Tiers**: The models can be roughly grouped into tiers. GraphCodeBERT and DistilBERT form the top tier. CodeBERT, RoBERTa, Longformer, and DeBERTa form a middle tier with comparable performance. The BERT model, particularly its baseline, consistently exhibits the highest loss, placing it in the lowest performance tier for this task.
- **Learning Trajectory**: All models exhibit a classic learning curve, with a steep drop in loss during the initial training steps, which gradually flattens as the models converge.

Answer to RQ2

The effects of training constraints are highly model-dependent. For complex PLMs, **early stopping** is a critical mechanism for efficiency and performance. It prevents overfitting by terminating training at the optimal point, as seen with **GraphCodeBERT Sweep (best)**, which stopped at 13.77/14 epochs to secure a 95.80% F1-score. For simpler, highly effective architectures like **BiLSTM+Attn**, the pre-defined **epoch limit** (10 epochs) was the main constraint. The model trained for the full duration as its performance had not yet plateaued, achieving a 98.60% F1-score. This highlights a key trade-off: early stopping optimizes PLM generalization, while for efficient models like BiLSTM, the number of epochs can be the primary factor limiting peak performance.

DL models. Figs. 10(a) and 10(c) show the training and validation loss for the DL models, respectively. The key observations are:

- **Architectural Dominance**: There is a clear performance hierarchy where BiLSTM-based models (purple lines) consistently and significantly outperform all LSTM-based models (orange lines).



(a) DL Train Loss

(b) Transformers Train Loss

(c) DL Validation Loss

(d) Transformers Validation Loss

Fig. 10. Training and validation loss curves for DL and Transformer models. The Y-axis on all plots represents Binary Cross-Entropy (BCE) Loss and is log-scaled to better visualize performance improvements at lower loss values. The X-axis for the validation plots (Fig. 10(d)) is also log-scaled to emphasize early training dynamics. A Gaussian smoothing (degree=1 for validation, degree=10 for training) is applied to the lines to illustrate overall trends rather than noisy step-by-step fluctuations. Note that the gray legends in (Fig. 10(a)) and (Fig. 10(c)) are artifacts from the logging tool and should be disregarded.

The training and validation losses for BiLSTM models rapidly decrease, stabilizing at a low value of approximately 0.05.

- **Poorest Performer**: The LSTM baseline model exhibits the highest loss in both training and validation, struggling to converge and ending with a validation loss near 0.4.
- **Impact of Attention**: Attention mechanisms provide a clear benefit. For both LSTM and BiLSTM, the attention-equipped variants (`+Attn`) demonstrate faster initial convergence compared to their respective baseline models.
- **Strong Generalization**: The validation loss curves for the BiLSTM models are very low and flat, mirroring their training performance. This indicates strong generalization and no signs of significant overfitting within the trained epochs.

Loss analysis. A comparative analysis of the loss curves for both model families reveals several key insights:

- **Competitive Performance Across Families**: While top-tier PLMs like GraphCodeBERT achieve the lowest training loss, the best-performing DL models (BiLSTM variants) demonstrate exceptionally low validation loss, reaching values below 0.05. This suggests that for this specific task, a well-tuned recurrent architecture can generalize as effectively, or even more effectively, than complex transformer models.
- **Top Performers and Laggards**: Among transformers, GraphCodeBERT consistently leads, particularly in its hyperparameter-tuned configuration. Conversely, models like BERT, CodeBERT, and DistilBERT show intermediate performance. In the DL category, the BiLSTM models are the clear front-runners, while the baseline LSTM models exhibit the highest loss values and slowest convergence.
- **Learning Dynamics**: Both model families show characteristic learning curves with rapid initial loss reduction followed by stabilization. The performance gap is notably wider within the DL family, where the architectural choice between LSTM and BiLSTM results in a dramatic performance difference.
- **Generalization and Stability**: Transformer models generally show smooth validation loss curves, indicating stable generalization. The BiLSTM models also show excellent generalization with very low and flat validation loss, whereas the LSTM models struggle to converge to a low validation loss, indicating poorer generalization on this task.
- **Impact of Optimization**: Across both model families, hyperparameter optimization via sweep configurations consistently yields better-performing models compared to their baseline setups, underscoring the importance of tuning for this specific dataset and task.

4.3.4. RQ3: Computational trade-offs and resource analysis

The evaluation of machine learning models for smart contract vulnerability detection extends beyond accuracy metrics to encompass computational efficiency and resource utilization. Understanding the trade-offs between model performance and computational demands is crucial for practical deployment considerations. This section provides a comprehensive analysis of training times, inference costs, and resource consumption patterns across different model architectures, offering insights into the computational feasibility of deploying these models in real-world scenarios.

Training and inference time analysis. A stark contrast emerges when comparing the computational cost of PLMs and traditional DL models. As shown in Table 8, the top-performing PLMs demand a significant training investment. **GraphCodeBERT**, for example, required nearly 3 h (10,790 s) to train. In contrast, our best-performing DL model, **BiLSTM+Attn**, trained in approximately 22 min (~1320 s), a nearly

Table 8
Training and inference time for best-performing models (s). Inference time is broken down by phase.

Model	Training time	Model load	Feature extraction	Testing	Total (Inference) time
BERT	15,198	1.17 (1.59%)	0.80 (1.09%)	68.01 (92.66%)	73.40
CodeBERT	8319	0.28 (0.51%)	0.33 (0.60%)	51.03 (92.85%)	54.96
DeBERTa	11,049	0.41 (0.53%)	0.31 (0.40%)	73.01 (94.27%)	77.45
DistilBERT	5187	0.19 (0.48%)	12.17 (31.00%)	14.60 (37.19%)	39.25
GraphCodeBERT	10,790	0.26 (0.77%)	0.82 (2.44%)	32.50 (96.79%)	33.58
Longformer	20,174	0.18 (0.32%)	2.55 (4.57%)	53.07 (95.11%)	55.80
RoBERTa	7334	0.25 (0.45%)	20.65 (37.00%)	29.71 (53.25%)	55.80

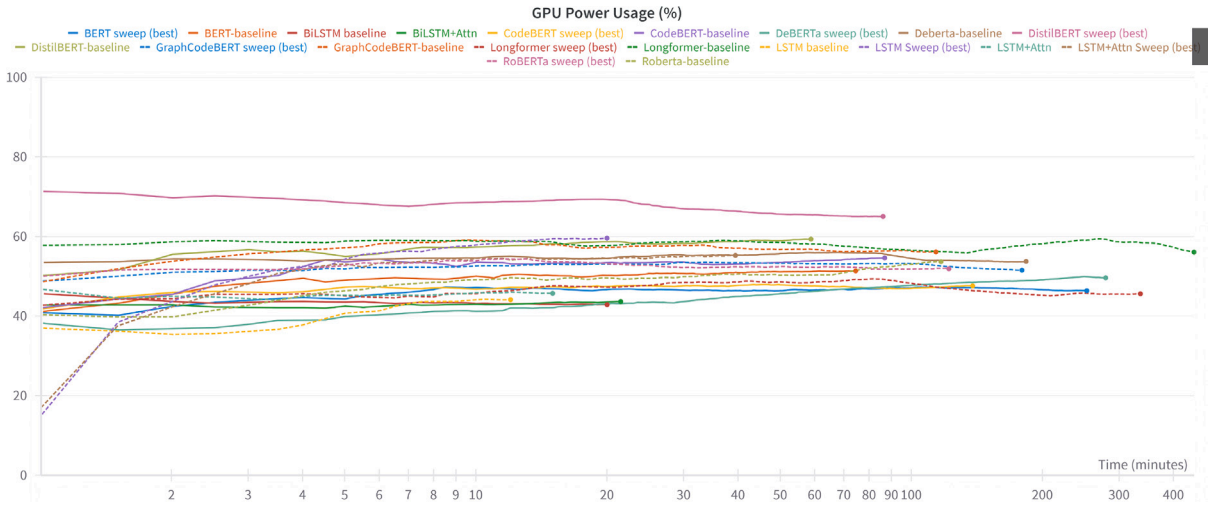


Fig. 11. GPU Power Usage (%). *Log scaled X axis, random sampling, Time weighted EMA smoothing (1 unit), grouped by metrics, means are shown.*

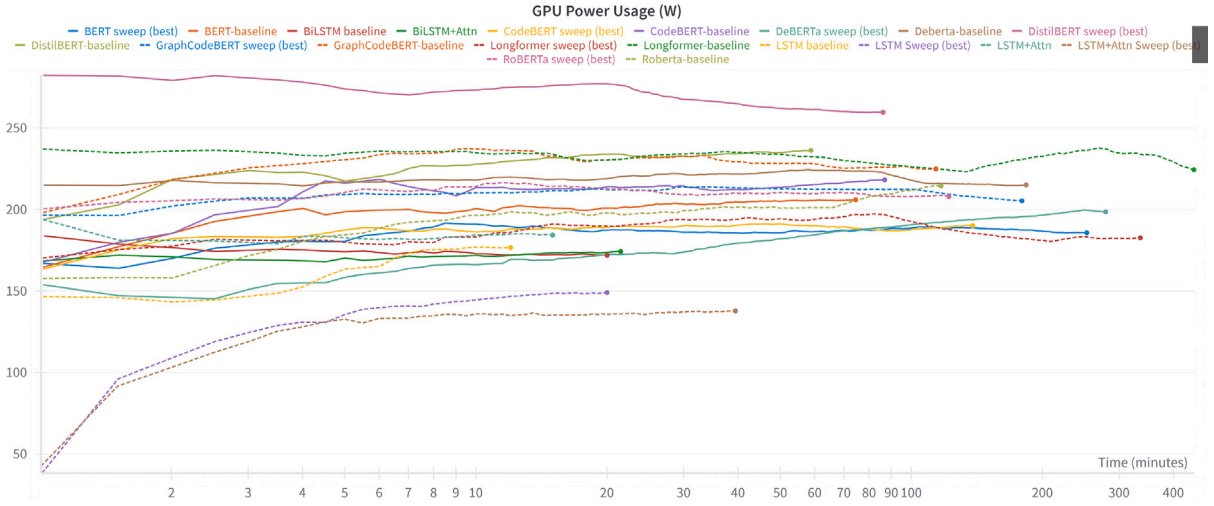


Fig. 12. GPU Power Usage (Watts). *Log scaled X axis, random sampling, Time weighted EMA smoothing (1 unit), grouped by metrics, means are shown.*

8-fold reduction in training time. This highlights the most significant computational trade-off: the extensive pre-training and complexity of transformers result in a substantially higher upfront time cost compared to training a more focused recurrent architecture from scratch.

However, this dynamic shifts during inference. While training times differ by orders of magnitude, the inference times for the top models are remarkably competitive. **GraphCodeBERT** processes our entire test set in just 33.58 s. The highly efficient **BiLSTM+Attn** model is estimated to process the same set in approximately 36 s. This demonstrates that once trained, both architectures are highly optimized for fast execution, and the initial training cost of a large PLM does not necessarily translate to slower performance in a deployment scenario. The primary trade-off is thus heavily skewed towards the initial training phase rather than the operational inference phase.

Resource usage analysis. The following figures illustrate the resource utilization patterns during the fine-tuning of various models. Our analysis reveals several key insights:

- **GPU Power Usage** (Figs. 11 and 12):
  - The trajectories depicted in both figures exhibit a striking similarity. In Fig. 11, which illustrates the percentage of GPU usage, the majority of the lines cluster within the 40%- to –60% range, concluding at various points corresponding to model training durations. Conversely, Fig. 12, which presents GPU power consumption in Watts, displays a more varied distribution across the graph.

  - This pattern suggests that transformer models maintain a steady, high computational load, while LSTM/BiLSTM models have more distinct phases of intense computation and relative inactivity.

- **System Memory Utilization** (Fig. 13): The system memory utilization varies across models, with some models like CodeBERT-baseline and BERT sweep (best) showing spikes up to almost 50% and over 20% respectively (note that these numbers are smoothed out). However, most models maintain a consistent usage between 10%–20% of the available system memory. This indicates that while some models may require more memory at certain points, overall memory management is efficient across both transformer and LSTM/BiLSTM architectures.
- **GPU Memory Allocation** (Fig. 14):
  - Most models maintain a stable GPU memory allocation between 40–70 GB, with some fluctuations.
  - LSTM variants generally show lower memory usage, indicating more memory-efficient architectures.
  - Variations in memory allocation suggest different phases of training, such as data loading or model checkpointing.
- **CPU Utilization** (Fig. 15):
  - CPU usage remains low across all models, generally below 20%, indicating efficient offloading to the GPU.
  - The LSTM baseline shows slightly higher initial CPU usage, likely due to setup overhead.
  - Consistent low CPU usage across models highlights effective parallelization and reliance on GPU resources for training.

In summary, while PLMs maintain a higher and more consistent GPU power draw during their long training cycles, the traditional DL models exhibit more variable, "bursty" usage patterns. For other resources like system memory and CPU utilization, both model families are broadly efficient, with neither showing a decisive advantage.

Answer to RQ3

The primary computational trade-off between transformers and traditional DL methods lies in the training phase, not inference. A top-tier PLM like **GraphCodeBERT** requires a substantial training investment (~3 hours), whereas a simpler architecture like **BiLSTM+Attn** achieves superior performance after only ~22 minutes of training. For inference, however, this gap vanishes; both models demonstrate high efficiency, processing our test set in a comparable time (~34-36 seconds). Resource analysis further reveals that PLMs sustain a higher, more consistent GPU load during their lengthy training. Therefore, the key decision for practitioners is balancing the significantly higher initial training cost of PLMs against a negligible difference in deployed inference speed.

4.3.5. RQ4: Inference time and deployment feasibility

While training metrics provide insight into model development, the practical utility of a model hinges on its inference performance and deployment feasibility. This is especially relevant in the blockchain domain, where use cases span real-time vulnerability checks in developer environments to extensive off-chain audits of smart contract repositories.

Our experimental results, detailed in Table 8, reveal that the top-performing **BiLSTM+Attn** model trains in approximately 22 min (1320 s, derived from 10 epochs with a high samples-per-second rate), significantly faster than the best-performing PLM, **GraphCodeBERT**, which requires around 3 h (10790 s). This efficiency extends to inference time: BiLSTM+Attn processes a single sample in approximately 0.01 s, compared to 0.1 s for GraphCodeBERT, a tenfold difference. This inference speed advantage stems from BiLSTM+Attn's smaller size (fewer parameters) and simpler architecture, which avoids the computationally intensive self-attention layers of transformer-based models like GraphCodeBERT.

These performance characteristics directly inform deployment considerations:

- **On-Chain Deployment:** Deploying any of these models directly on-chain remains impractical due to blockchain resource constraints. For instance, Ethereum's current block gas limit as of July 1st, 2025 is almost 36 million units,[2] while even a minimal neural network inference could require hundreds of thousands of gas units—far exceeding feasible limits. Storage demands further compound this issue, as model weights (e.g., GraphCodeBERT's 768 hidden size and 3072 intermediate size) would require substantial on-chain storage, incurring prohibitive costs. Consequently, vulnerability detection must occur off-chain.
- **Off-Chain Deployment:** Off-chain deployment scenarios vary by application, with model selection driven by latency and throughput requirements:
  - **Real-Time Analysis:** Tools such as IDE plugins or pre-deployment gateways demand low-latency inference to provide immediate feedback. BiLSTM+Attn, with its 0.01-second inference time per sample, excels here, enabling rapid scans during coding or commit stages. Similarly, **DistilBERT**, with an inference time of 0.02 s per sample, offers a viable alternative for environments prioritizing efficiency alongside competitive accuracy (F1 score of 0.9375).
  - **Batch Processing:** Offline tasks, such as auditing large contract repositories or historical blockchain analysis, prioritize accuracy and throughput over single-sample latency. Here, GraphCodeBERT's higher F1 score (0.9580) justifies its slower 0.1-second inference time, as parallelization across multiple samples can amortize costs over extended runtimes.

These trade-offs are quantified in Table 9. BiLSTM+Attn emerges as a standout choice for off-chain tools, blending top-tier accuracy (F1 score of 0.9860) with rapid inference, making it suitable for both real-time and batch-processing scenarios. GraphCodeBERT, while less efficient, remains valuable for applications where maximum detection precision outweighs speed, and DistilBERT provides a balanced middle ground for resource-constrained settings.

Answer to RQ4

Deployment feasibility is dictated by a trade-off between inference speed and detection accuracy, with on-chain deployment being impractical for all models due to resource constraints. For off-chain applications, model choice depends on the use case. **BiLSTM+Attn** (0.01s/sample inference, 98.6% F1) and **DistilBERT** (0.02s/sample, 93.8% F1) are ideal for real-time tools like IDE plugins where low latency is critical. In contrast, the more computationally intensive **GraphCodeBERT** (0.1s/sample, 95.8% F1) is better suited for offline, large-scale batch auditing, where its higher accuracy can be prioritized over inference speed.

[2] https://etherscan.io/chart/gaslimit

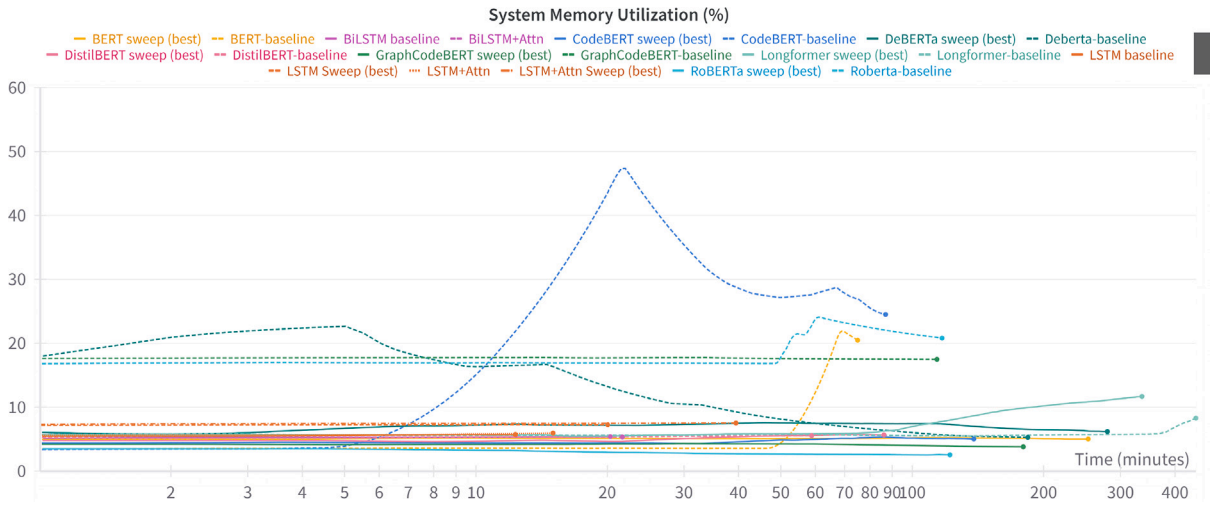


Fig. 13. System Memory Utilization. *Log scaled X axis, random sampling, Time weighted EMA smoothing (1 unit), original not shown.*

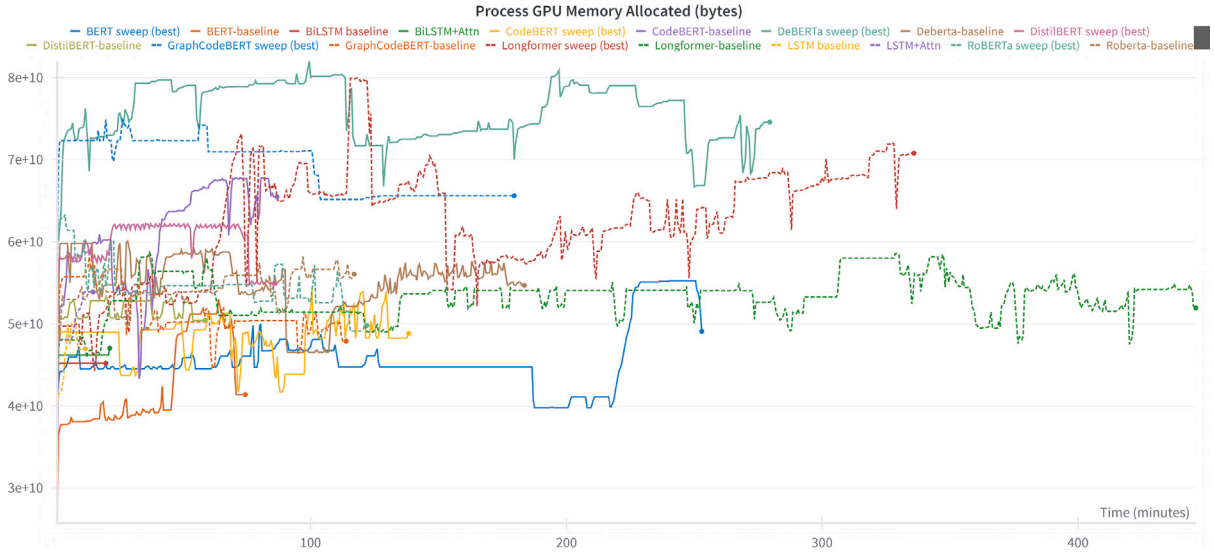


Fig. 14. GPU Memory Allocation. *Log scaled X axis, random sampling, Time weighted EMA smoothing (1 unit), grouped by metrics, means are shown.*

Table 9
Deployment feasibility trade-offs for top-performing models.

Characteristic	BiLSTM+Attn	GraphCodeBERT (Best)	DistilBERT (Best)
Test F1-Score	**0.9860** (Highest)	0.9580 (High)	0.9375 (Competitive)
Training Time	~22 min (Fastest)	~3 h (Slow)	~1.4 h (Moderate)
Inference Time per Sample	0.01 s (Fastest)	0.1 s (Slowest)	0.02 s (Moderate)
Model Size/Complexity	Low	High	Medium
Best Proposed Use Case	Real-time IDE plugins, pre-deployment checks	Offline, large-scale batch analysis	Balanced off-chain tools

4.3.6. RQ5: Model selection criteria and recommendations

Our comprehensive analysis of various models for smart contract vulnerability detection reveals nuanced insights into the performance, efficiency, and practical implications of different architectures. The experiments conducted across a range of models, including transformer-based architectures and traditional recurrent neural networks, yield several key findings:

- **Performance Excellence**: The BiLSTM+Attn model demonstrates exceptional performance, achieving the highest test accuracy (0.9380) and F1 score (0.9860) among all tested models. This surprising result highlights the potential of well-tuned recurrent architectures with attention mechanisms for code analysis tasks.
- **Transformer Efficacy**: Among transformer-based models, GraphCodeBERT Sweep (best) shows strong performance (test accuracy:

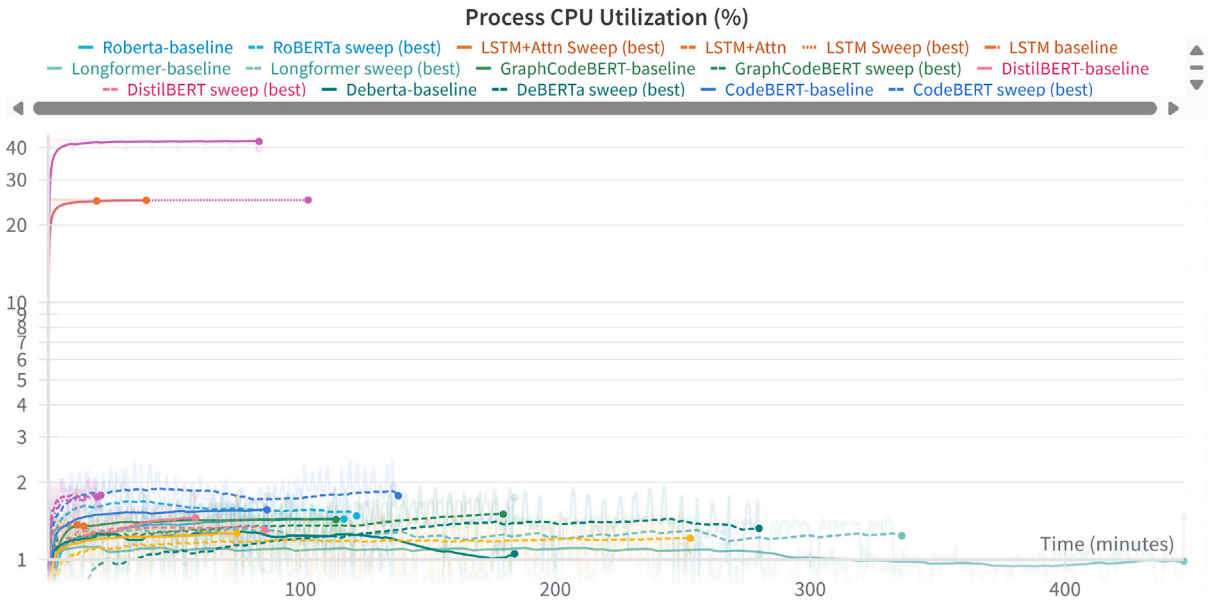


Fig. 15. CPU Utilization. This figure's Y-axis is log-scaled with random sampling, and smoothed using Time Weighted EMA (1 unit).

0.8146, F1 score: 0.9580), indicating the potential of these architectures for code-related tasks. Its superior performance among transformers suggests that pre-training on code-like structures significantly enhances model effectiveness for specialized tasks like smart contract vulnerability detection.

- **Efficiency Considerations**: DistilBERT Sweep (best) presents a compelling balance between performance and efficiency. While its test accuracy (0.7594) and F1 score (0.9375) are lower than the top performers, it demonstrates the highest training efficiency (29.5 samples/sec, 0.461 steps/sec). This makes it an attractive option for resource-constrained environments or scenarios requiring faster training and inference.
- **Architectural Trade-offs**: Traditional LSTM models significantly underperform compared to more advanced architectures, with the LSTM-baseline achieving a test accuracy of only 0.3052. This underscores the benefits of newer model designs and the importance of attention mechanisms in enhancing LSTM performance for complex code analysis tasks.
- **Resource Utilization**: Transformer models generally maintain a steady, high computational load with consistent GPU power usage, while LSTM/BiLSTM models exhibit more variable patterns. This insight is crucial for hardware resource allocation and optimization strategies in practical deployments.
- **Memory Efficiency**: Most models maintain efficient memory usage, with system memory utilization between 10%–20% and GPU memory allocation between 40–70 GB. LSTM variants generally show lower memory usage, indicating more memory-efficient architectures.
- **Hyperparameter Optimization**: The significant performance improvements observed in "sweep (best)" configurations across all models highlight the critical role of hyperparameter optimization in maximizing model effectiveness for this specific task.

These findings emphasize the importance of balancing performance metrics with computational requirements when selecting a model for practical applications in blockchain security. While BiLSTM+Attn and GraphCodeBERT offer top-tier performance, their training demands may be prohibitive in some scenarios. Conversely, DistilBERT's efficient training and competitive performance present an attractive compromise for real-world deployments where resources are limited.

The choice of model should ultimately depend on the specific requirements of the task, available computational resources, and the critical balance between accuracy and efficiency in the given application context. Future research directions could explore hybrid architectures that combine the strengths of recurrent and transformer models, as well as further optimization techniques to enhance the efficiency of high-performing models for smart contract vulnerability detection.

Answer to RQ5

Model selection requires balancing performance with computational cost, as no single model is universally optimal. For maximum accuracy, **BiLSTM+Attn** is the top choice (98.6% F1), proving a well-tuned recurrent architecture can excel. If a PLM is preferred, **GraphCodeBERT** offers the best performance (95.8% F1), demonstrating the value of code-specific pre-training. In resource-constrained environments, **DistilBERT** provides the best efficiency trade-off, balancing competitive accuracy (93.8% F1) with much faster training. Ultimately, the optimal choice depends on the deployment scenario (e.g., real-time vs. offline batch), and hyperparameter tuning is critical for all models.

5. Practical application and software engineering insights

While the preceding sections focus on empirical performance, a crucial aspect of this research is bridging the gap between deep learning experiments and the daily workflows of software engineers. A model's practical value is not defined solely by its F1-score but by how seamlessly it can be integrated into the software development lifecycle to produce actionable insights.

5.1. Integrating models into developer workflows

Based on our findings, particularly the trade-offs between accuracy and inference speed (Table 9), we envision three primary integration points for these models within a typical smart contract engineering process, as illustrated in Fig. 16.

1. **Real-Time IDE Feedback:** For developers working in environments like Remix or VS Code, a model integrated as an IDE plugin can provide immediate security feedback. This requires extremely low latency, making a lightweight model like **BiLSTM+Attn** or the efficient **DistilBERT** ideal. As a developer

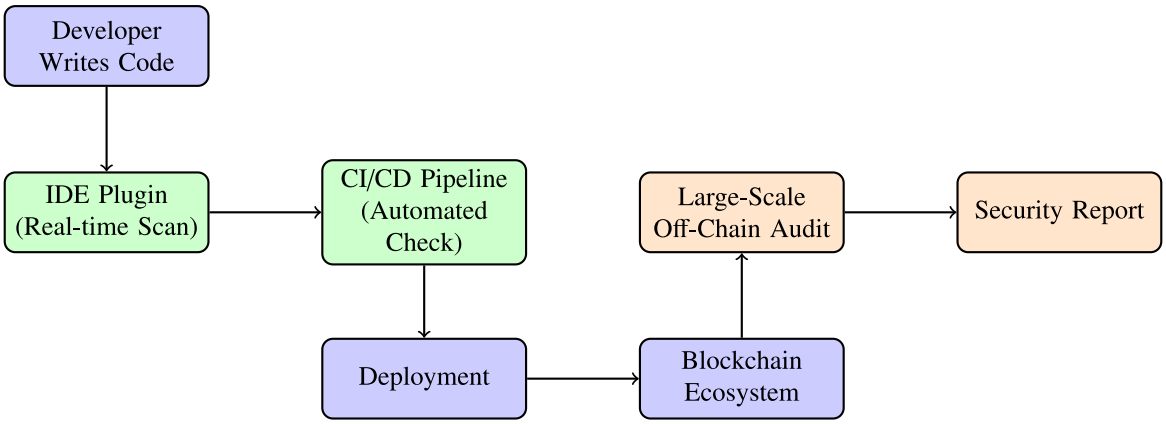


Fig. 16. Proposed integration points for ML-based vulnerability detection models within the smart contract software development lifecycle and ecosystem. Green boxes represent developer-centric, real-time feedback loops, while orange boxes represent broader, offline analysis.

writes code, the plugin can flag potentially vulnerable patterns in real-time, functioning like a security-focused linter.

2. **Automated CI/CD Pipeline Checks:** In a team environment, models can be integrated into a Continuous Integration/Continuous Deployment (CI/CD) pipeline. Before code is merged into a main branch, an automated job can scan the smart contracts for vulnerabilities. Here, a balance must be struck. A fast model like **DistilBERT** ensures the pipeline is not significantly slowed down, while a more accurate model like **GraphCodeBERT** could be used in a separate, non-blocking "nightly" build for a more thorough analysis.
3. **Large-Scale Ecosystem Auditing:** For security researchers, auditors, or blockchain platforms, the goal is often to analyze thousands of already-deployed contracts to identify systemic risks or discover new attack vectors. In this offline, batch-processing context, inference speed per contract is less critical than overall accuracy and throughput. The high-accuracy **GraphCodeBERT** is best suited for this task, as its computational cost is justified by the depth of its analysis.

5.2. From prediction to actionable insight

A prediction from a model is only useful if a developer can act on it. This is where the concept of explainability, discussed in Section 6, becomes a critical software engineering requirement. An ideal security tool would not merely flag a contract with "TOD vulnerability detected". Instead, using techniques like saliency maps, it would highlight the specific lines of code – such as the use of "`block.timestamp`" in a critical conditional statement – that most contributed to the model's decision. This transforms a "black box" prediction into a concrete, actionable insight, guiding the developer directly to the source of the potential problem. Our work provides the foundation for building such tools by identifying models that are not only accurate but also lend themselves to these explainability techniques (e.g., those with attention mechanisms).

6. Limitations and future work

While our study provides valuable insights, several limitations and future research directions emerge. A key limitation is the dataset bias introduced by filtering out rare vulnerability types (those with fewer than 2000 samples). While this was necessary for robust model training, it means our models cannot detect these vulnerabilities. Future work must focus on addressing this gap. Other limitations include the high computational demands of top-performing models and the critical need for model interpretability. For a security tool to be adopted, it must not only detect vulnerabilities but also explain its reasoning to developers. Future work should therefore focus on integrating explainable AI (XAI) techniques. For models with attention mechanisms, such as our top-performing BiLSTM+Attn and the various PLMs, visualizing attention weights can provide initial insights into which code segments influence a prediction. However, more advanced methods like LIME (Ribeiro et al., 2016) or SHAP (Lundberg and Lee, 2017) could be adapted to generate saliency maps, highlighting the specific tokens that contribute most to a vulnerability classification. Developing robust, human-understandable explanations is a crucial next step for building trust and utility in these automated tools. Key future directions therefore include: (1) expanding datasets and employing techniques like few-shot learning or synthetic data generation to train models on rare but critical vulnerabilities; (2) exploring model compression; (3) integrating dynamic analysis; (4) investigating cross-platform generalization; and (5) implementing continuous learning strategies to combat model drift.

7. Threats to validity

In this section, we discuss potential threats to the validity of our study and the concrete steps we take to mitigate them.

7.1. Internal validity

Internal validity relates to the confidence that our experimental outcomes are due to the model and training choices rather than confounding factors.

- **Data Pre-processing Bias:** Our methodology involves standard pre-processing steps, such as removing comments and generalizing variable names. While this could potentially remove semantic cues, we mitigate this threat by adopting it as a standard practice Sun et al. (2023) designed to help models generalize better by focusing on structural and semantic patterns rather than specific, context-dependent naming conventions.
- **Hyperparameter Tuning:** A potential threat is that our hyperparameter search may not have found the absolute optimal configuration. To mitigate this, we employ a systematic Bayesian hyperparameter search using W&B Sweeps over 10 trials for each model. This automated approach efficiently explores the parameter space to identify high-performing configurations and ensures a fair and reproducible basis for comparison.
- **Model Training Variability:** The stochastic nature of model training and weight initialization can lead to variability in results. To mitigate this, we use a fixed random seed for our primary experiments to ensure reproducibility. Furthermore, for our top models, we conduct 5 runs with different seeds and perform statistical tests, as detailed in our statistical analysis section, to ensure our conclusions are robust and not due to chance.

7.2. External validity

External validity concerns the generalizability of our findings to other contexts.

- **Dataset Limitations:** Findings from a single dataset may not generalize. To mitigate this threat, we evaluate our models on two distinct, publicly available datasets for smart contract vulnerabilities: SoliAudit and SolidiFi. The consistent performance patterns observed across both datasets strengthen the external validity of our conclusions.
- **Exclusion of Rare Vulnerabilities:** Our study filters out vulnerability types with fewer than 2000 positive samples. This decision, while necessary to ensure models could learn statistically meaningful patterns, introduces a clear bias. The models are not trained to recognize these rarer vulnerabilities, and thus their generalizability is limited to the seven more common types included in our study. In a real-world scenario, the models would fail to detect these excluded, yet potentially critical, vulnerabilities. This represents a deliberate trade-off: we prioritized achieving high performance and reliability on a subset of more prevalent vulnerabilities over building a model that covers all vulnerabilities but with potentially lower and less reliable performance, especially on the classes with few examples.
- **Model Generalization:** To ensure our models generalize to unseen data rather than simply memorizing the training set, we implement a strict data-splitting strategy. We use a 60/20/20 split for training, validation, and testing for our main dataset, reporting final performance on the held-out test set, which the model never sees during training or hyperparameter tuning.
- **Model and Concept Drift:** The models are trained on a static snapshot of smart contracts. Over time, as the Solidity language evolves, new design patterns emerge, and novel vulnerability classes are discovered, the models' performance may degrade. This is a significant threat to long-term applicability. To mitigate this, a production-level system would require a continuous learning pipeline to periodically retrain the models on newer, more representative datasets. While our study does not implement such a pipeline, our evaluation on multiple datasets serves as a proxy for robustness, demonstrating the models' ability to handle varied data distributions, a necessary first step towards managing future drift.

7.3. Construct validity

Construct validity refers to how well our evaluation measures what it intends to—the model's vulnerability detection capability.

- **Metric Selection:** Relying on a single metric can be misleading, especially with imbalanced data. To mitigate this, we report a comprehensive suite of metrics, including subset accuracy, and label-based precision, recall, and F1-score. Crucially, we also provide a detailed per-class performance analysis, which offers a granular view of model strengths and weaknesses for each specific vulnerability type.
- **Token Length Constraints:** Truncating smart contract code could lead to a loss of critical information. We mitigated this by analyzing the token length distribution across our dataset and selecting a maximum length of 2048 tokens. As detailed in Section 3.1, this limit captures the vast majority of contracts in their entirety while maintaining computational feasibility, representing a reasoned trade-off between information preservation and resource efficiency.

7.4. Conclusion validity

Conclusion validity addresses the degree to which our conclusions are reasonable and statistically sound.

- **Statistical Significance:** Without formal statistical tests, observed performance differences could be due to chance. To mitigate this, we conduct formal statistical significance testing (Wilcoxon signed-rank test) and effect size analysis (Vargha–Delaney $\hat{A}_{12}$) on our top-performing models. While the small sample size (n=5 runs) limits statistical power, the combination of significance testing and a large effect size provides strong evidence that our findings are not coincidental.
- **Overfitting:** Models could overfit to the training data, leading to inflated performance metrics. We mitigate this threat by using a dedicated validation set for early stopping and model selection during hyperparameter sweeps. The final evaluation on a completely unseen test set provides an unbiased estimate of the models' true performance on new data. The training and validation loss curves presented in further allow for monitoring of overfitting.

8. Conclusion

In this paper, we conduct a comprehensive investigation into the effectiveness and efficiency of various deep-learning architectures for smart contract vulnerability detection. Our systematic evaluation moves beyond a simple performance leaderboard to provide critical, practical insights for engineering real-world security tools. We demonstrate that while specialized PLMs like GraphCodeBERT offer high accuracy, a well-tuned BiLSTM with attention provides a superior balance of performance, training speed, and inference efficiency, making it highly suitable for integration into developer workflows.

Our analysis of computational trade-offs provides a clear decision framework for software engineers: lightweight models for real-time IDE feedback, balanced models for CI/CD pipelines, and high-complexity models for deep, offline auditing. By contextualizing our experimental results within these practical use cases, we bridge the gap between academic research and software engineering practice. This work thus establishes a foundation for advancing the security and reliability of smart contract systems, not just by identifying the ‘‘best’’ model, but by providing the empirical insights needed to engineer the next generation of effective, efficient, and practical vulnerability detection tools.

CRediT authorship contribution statement

Trung Kien Luu: Writing – original draft, Validation, Resources, Methodology, Investigation, Formal analysis, Data curation. **Doan Minh Trung:** Writing – review & editing, Writing – original draft, Validation, Supervision, Resources, Methodology, Investigation, Formal analysis, Data curation. **Tuan-Dung Tran:** Writing – review & editing, Validation, Supervision, Methodology, Conceptualization. **Phan The Duy:** Writing – review & editing, Validation, Supervision, Project administration, Methodology, Funding acquisition, Conceptualization. **Van-Hau Pham:** Writing – review & editing, Validation, Supervision, Project administration, Methodology, Funding acquisition, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This research is funded by Vietnam National University Ho Chi Minh City (VNU-HCM) under grant number NCM2025-26-01.

Data availability

Data will be made available on request.

References

Beltagy, Iz, Peters, Matthew E., Cohan, Arman, 2020. Longformer: The long-document transformer. arXiv preprint arXiv:2004.05150.

Biewald, Lukas, 2020. Experiment tracking with weights and biases. Software available from wandb.com.

Cai, Jie, Li, Bin, Zhang, Jiale, Sun, Xiaobing, Chen, Bing, 2023. Combine sliced joint graph with graph neural networks for smart contract vulnerability detection. J. Syst. Softw. 195, 111550.

Certik, 2024. Hack3d: The Web3 security report 2023.

Chakraborty, Saikat, Krishna, Rahul, Ding, Yangruibo, Ray, Baishakhi, 2021. Deep learning based vulnerability detection: Are we there yet? IEEE Trans. Softw. Eng. 48 (9), 3280–3296.

Chen, Jiachi, Shao, Zhenzhe, Yang, Shuo, Shen, Yiming, Wang, Yanlin, Chen, Ting, Shan, Zhenyu, Zheng, Zibin, 2025. NumScout: Unveiling numerical defects in smart contracts using LLM-pruning symbolic execution. IEEE Trans. Softw. Eng..

David, Isaac, Zhou, Liyi, Qin, Kaihua, Song, Dawn, Cavallaro, Lorenzo, Gervais, Arthur, 2023. Do you still need a manual smart contract audit? arXiv preprint arXiv:2306.12338.

De Baets, Christopher, Suleiman, Basem, Chitizadeh, Armin, Razzak, Imran, 2024. Vulnerability detection in smart contracts: A comprehensive survey. arXiv preprint arXiv:2407.07922.

Devlin, Jacob, Chang, Ming-Wei, Lee, Kenton, Toutanova, Kristina, 2018. BERT: Pre-training of deep bidirectional transformers for language understanding. In: Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. pp. 4171–4186.

Feist, Josselin, Grieco, Gustavo, Groce, Alex, 2019. Slither: a static analysis framework for smart contracts. In: 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain. WETSEB, IEEE, pp. 8–15.

Feng, Zhangyin, Guo, Daya, Tang, Duyu, Duan, Nan, Feng, Xiaocheng, Gong, Ming, Shou, Linjun, Qin, Bing, Liu, Ting, Jiang, Daxin, 2020. Codebert: A pre-trained model for programming and natural languages. In: Findings of the Association for Computational Linguistics: EMNLP 2020. pp. 1536–1547.

Ghaleb, Asem, Pattabiraman, Karthik, 2020. How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 415–427.

Gu, Ronghui, Shao, Zhong, Chen, Hao, Wu, Xiongnan Newman, Kim, Jieung, Sjöberg, Vilhelm, Costanzo, David, 2016. {CertiKOS}: An extensible architecture for building certified {OS} kernels. In: 12th USENIX Symposium on Operating Systems Design and Implementation. OSDI 16, pp. 653–669.

Guo, Daya, Ren, Shuo, Lu, Shuai, Feng, Zhangyin, Tang, Duyu, Liu, Shujie, Zhou, Long, Duan, Nan, Svyatkovskiy, Alexey, Fu, Shengyu, et al., 2020. Graphcodebert: Pre-training code representations with data flow. arXiv preprint arXiv:2009.08366.

He, Pengcheng, Liu, Xiaodong, Gao, Jianfeng, Chen, Weizhu, 2020. Deberta: Decoding-enhanced bert with disentangled attention. arXiv preprint arXiv:2006.03654.

Hildenbrandt, Everett, Saxena, Manasvi, Rodrigues, Nishant, Zhu, Xiaoran, Daian, Philip, Guth, Dwight, Moore, Brandon, Park, Daejun, Zhang, Yi, Stefanescu, Andrei, et al., 2018. Kevm: A complete formal semantics of the ethereum virtual machine. In: 2018 IEEE 31st Computer Security Foundations Symposium. CSF, IEEE, pp. 204–217.

Hleap, Sergio, 0000, Unmasking the Outliers: Exploring the Interquartile Range Method for Reliable Data Analysis.

Hwang, Seon-Jin, Choi, Seok-Hwan, Shin, Jinmyeong, Choi, Yoon-Ho, 2022. CodeNet: Code-targeted convolutional neural network architecture for smart contract vulnerability detection. IEEE Access 10, 32595–32607.

Ince, Peter, Luo, Xiapu, Yu, Jiangshan, Liu, Joseph K, Du, Xiaoning, 2024. Detect llama-finding vulnerabilities in smart contracts using large language models. In: Australasian Conference on Information Security and Privacy. Springer, pp. 424–443.

Kevin, Jun, Yugopuspito, Pujianto, 2025. Smartllm: Smart contract auditing using custom generative AI. In: 2025 International Conference on Computer Sciences, Engineering, and Technology Innovation. ICoCSETI, IEEE, pp. 260–265.

Kim, Tae Kyun, 2015. T test as a parametric statistic. Korean J. Anesthesiol. 68 (6), 540–546.

Liang, Zhehao, Cui, Baojiang, Wang, Dongbin, Xu, Jie, Liu, Huipeng, 2024. A smart contract vulnerability detection system based on BERT model and fuzz testing. In: International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing. Springer, pp. 288–295.

Liao, Jian-Wei, Tsai, Tsung-Ta, He, Chia-Kang, Tien, Chin-Wei, 2019. Soliaudit: Smart contract vulnerability assessment based on machine learning and fuzz testing. In: 2019 Sixth International Conference on Internet of Things: Systems, Management and Security. IOTSMS, IEEE, pp. 458–465.

Liu, Yinhan, Ott, Myle, Goyal, Naman, Du, Jingfei, Joshi, Mandar, Chen, Danqi, Levy, Omer, Lewis, Mike, Zettlemoyer, Luke, Stoyanov, Veselin, 2019. Roberta: A robustly optimized bert pretraining approach. arXiv preprint arXiv:1907.11692.

Loshchilov, Ilya, Hutter, Frank, 2017. Decoupled weight decay regularization. arXiv preprint arXiv:1711.05101.

Lundberg, Scott M., Lee, Su-In, 2017. A unified approach to interpreting model predictions. Adv. Neural Inf. Process. Syst. 30.

Luo, Ruijie, Luo, Feng, Wang, Bingsen, Chen, Ting, 2022. Smart contract vulnerability detection based on variant LSTM. In: Proceedings of the 2022 International Conference on Big Data, IoT, and Cloud Computing. pp. 1–4.

Lutz, Oliver, Chen, Huili, Fereidooni, Hossein, Sendner, Christoph, Dmitrienko, Alexandra, Sadeghi, Ahmad Reza, Koushanfar, Farinaz, 2021. Escort: ethereum smart contracts vulnerability detection using deep neural network and transfer learning. arXiv preprint arXiv:2103.12607.

Luu, Luu, Chu, Duc-Hiep, Olickel, Hrishi, Saxena, Prateek, Hobor, Aquinas, 2016. Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 254–269.

Mi, Feng, Zhao, Chen, Wang, Zhuoyi, Halim, Sadaf, Li, Xiaodi, Wu, Zhouxiang, Khan, Latifur, Thuraisingham, Bhavani, 2023. An automated vulnerability detection framework for smart contracts. arXiv preprint arXiv:2301.08824.

Mueller, Bernhard, 2018. Mythril: Security analysis tool for EVM bytecode. https://github.com/ConsenSys/mythril, (Accessed 15 April 2024).

Nguyen, Hoang H, Nguyen, Nhat-Minh, Doan, Hong-Phuc, Ahmadi, Zahra, Doan, Thanh-Nam, Jiang, Lingxiao, 2022. MANDO-GURU: vulnerability detection for smart contract source code by heterogeneous graph embeddings. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 1736–1740.

Permenev, Anton, Dimitrov, Dimitar, Tsankov, Petar, Drachsler-Cohen, Dana, Vechev, Martin, 2020. Verx: Safety verification of smart contracts. In: 2020 IEEE Symposium on Security and Privacy. SP, IEEE, pp. 1661–1677.

Qian, Peng, Liu, Zhenguang, He, Qinming, Zimmermann, Roger, Wang, Xun, 2020. Towards automated reentrancy detection for smart contracts based on sequential models. IEEE Access 8, 19685–19695.

Ren, Xiaojun, Wu, Yongtang, Li, Jiaqing, Hao, Dongmin, Alam, Muhammad, 2023. Smart contract vulnerability detection based on a semantic code structure and a self-designed neural network. Comput. Electr. Eng. 109, 108766.

Ressi, Dalila, Romanello, Riccardo, Piazza, Carla, Rossi, Sabina, 2024. AI-enhanced blockchain technology: A review of advancements and opportunities. J. Netw. Comput. Appl. 103858.

Ribeiro, Marco Tulio, Singh, Sameer, Guestrin, Carlos, 2016. " why should i trust you?" explaining the predictions of any classifier. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 1135–1144.

Rizzo, Matteo, Ressi, Dalila, Gasparetto, Andrea, Rossi, Sabina, 2024. A comparison of machine learning techniques for ethereum smart contract vulnerability detection.

Sanh, Victor, Debut, Lysandre, Chaumond, Julien, Wolf, Thomas, 2019. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. arXiv preprint arXiv:1910.01108.

Sendner, Christoph, Chen, Huili, Fereidooni, Hossein, Petzi, Lukas, König, Jan, Stang, Jasper, Dmitrienko, Alexandra, Sadeghi, Ahmad-Reza, Koushanfar, Farinaz, 2023. Smarter contracts: Detecting vulnerabilities in smart contracts with deep transfer learning. In: NDSS.

Singh, S. Kumar, Tiwari, Varun, Vadi, V. Rao, 2007. Smart contract using solidity (remix-ethereum IDE). Int. J. Adv. Res. Comput. Commun. Eng. ISO 3297 (2).

Sun, Xiaobing, Tu, Liangqiong, Zhang, Jiale, Cai, Jie, Li, Bin, Wang, Yu, 2023. ASSBert: Active and semi-supervised bert for smart contract vulnerability detection. J. Inf. Secur. Appl. 73, 103423.

Sun, Yuqiang, Wu, Daoyuan, Xue, Yue, Liu, Han, Wang, Haijun, Xu, Zhengzi, Xie, Xiaofei, Liu, Yang, 2024. Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. pp. 1–13.

Tong, Van, Dao, Cuong, Tran, Hai-Anh, Tran, Truong X, Souihi, Sami, 2024. Enhancing BERT-based language model for multi-label vulnerability detection of smart contract in blockchain. J. Netw. Syst. Manage. 32 (3), 63.

Tsankov, Petar, Dan, Andrei, Drachsler-Cohen, Dana, Gervais, Arthur, Buenzli, Florian, Vechev, Martin, 2018. Securify: Practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 67–82.

Ullah, Saad, Han, Mingji, Pujar, Saurabh, Pearce, Hammond, Coskun, Ayse, Stringhini, Gianluca, 2023. Can large language models identify and reason about security vulnerabilities? not yet. arXiv preprint arXiv:2312.12575.

Vargha, András, Delaney, Harold D., 2000. A critique and improvement of the CL common language effect size statistics of McGraw and wong. J. Educ. Behav. Stat. 25 (2), 101–132.

Węglarczyk, Stanisław, 2018. Kernel density estimation and its application. In: ITM Web of Conferences, vol. 23, EDP Sciences, p. 00037.

Woolson, Robert F., 2005. Wilcoxon signed-rank test. Encycl. Biostat. 8.

Wu, Hongjun, Zhang, Zhuo, Wang, Shangwen, Lei, Yan, Lin, Bo, Qin, Yihao, Zhang, Haoyu, Mao, Xiaoguang, 2021. Peculiar: Smart contract vulnerability detection based on crucial data flow graph and pre-training techniques. In: 2021 IEEE 32nd International Symposium on Software Reliability Engineering. ISSRE, IEEE, pp. 378–389.

Yu, Lei, Chen, Shiqi, Yuan, Hang, Wang, Peng, Huang, Zhirong, Zhang, Jingyuan, Shen, Chenjie, Zhang, Fengjun, Yang, Li, Ma, Jiajia, 2024. Smart-llama: Two-stage post-training of large language models for smart contract vulnerability detection and explanation. arXiv preprint arXiv:2411.06221.

Yu, Xingxin, Zhao, Haoyue, Hou, Botao, Ying, Zonghao, Wu, Bin, 2021. Deescvhunter: A deep learning-based framework for smart contract vulnerability detection. In: 2021 International Joint Conference on Neural Networks. IJCNN, IEEE, pp. 1–8.

yxliang01, 2017. Oyente - an analysis tool for smart contracts.

Zhang, Xuesen, Li, Jianhua, Wang, Xiaoqiang, 2022. Smart contract vulnerability detection method based on bi-lstm neural network. In: 2022 IEEE International Conference on Advances in Electrical Engineering and Computer Applications. AEECA, IEEE, pp. 38–41.

Zhang, Zhuo, Zhang, Brian, Xu, Wen, Lin, Zhiqiang, 2023. Demystifying exploitable bugs in smart contracts. In: 2023 IEEE ACM 45th International Conference on Software Engineering. ICSE, IEEE.

Zhao, Yu, Gong, Lina, Huang, Zhiqiu, Wang, Yongwei, Wei, Mingqiang, Wu, Fei, 2024. Coding-ptms: How to find optimal code pre-trained models for code embedding in vulnerability detection? In: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering. pp. 1732–1744.

Zhou, Liyi, Xiong, Xihan, Ernstberger, Jens, Chaliasos, Stefanos, Wang, Zhipeng, Wang, Ye, Qin, Kaihua, Wattenhofer, Roger, Song, Dawn, Gervais, Arthur, 2023. Sok: Decentralized finance (defi) attacks. In: 2023 IEEE Symposium on Security and Privacy. SP, IEEE, pp. 2444–2461.

Zhuang, Y., Liu, Z., Qian, P., Liu, Q., Wang, X., He, Q., 2021. Smart contract vulnerability detection using graph neural networks. In: Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence. pp. 3283–3290.