Lab Report – Vulnerability Detection
Group A

(Gharenazifam, Barkeshloomansouri, Kovac)

1 Literature Recap

1.1 Vulnerability Detection with Code Language Models: How Far Are We?

This paper evaluates the effectiveness of code language models like GitHub Copilot and Amazon CodeWhisperer in real-world vulnerability detection. The study highlights critical dataset issues, such as incorrect labeling and code duplication, which artificially inflate model performance. To address this, the authors introduce PRIMEVUL, a rigorously curated dataset using real CVE data, significantly enhancing the reliability and fairness of evaluations.

1.2 An Empirical Study of Deep Learning Models for Vulnerability Detection

Steenhoek et al. (2023) conducted a comprehensive empirical study comparing nine deep learning models (GNNs, Transformers, RNNs, MLPs) for vulnerability detection. They discovered that model predictions were unstable, relied heavily on superficial patterns, and demonstrated limited generalization to unseen projects. Crucially, the study revealed that additional training data did not always improve performance and that models performed better when test data originated from projects seen during training.

1.3 Vulnerability Detection with Fine-Grained Interpretations

IVDetect represents a significant advancement in vulnerability detection, utilizing Program Dependency Graphs (PDGs) and interpretable graph neural networks to offer precise explanations at the line level. This fine-grained interpretation enables developers to better understand and address the detected vulnerabilities.

1.4 LineVD: Statement-level Vulnerability Detection using Graph Neural Networks

Hin et al. (2022) introduced LineVD, a model combining transformer-based embeddings and graph neural networks to achieve statement-level vulnerability detection. By leveraging both structural and contextual information, LineVD significantly improves the accuracy and interpretability of detection, achieving up to a 105% increase in F1-score over prior approaches.

1.5 LineVul: A Transformer-based Line-Level Vulnerability Prediction

LineVul utilizes the transformer architecture of CodeBERT, employing its self-attention mechanism to pinpoint vulnerabilities at the line level accurately. This approach significantly surpasses previous methods like IVDetect, enhancing precision and interpretability, and providing developers with actionable insights for rapid remediation.

2 Methodology

Our prototype follows the same philosophy as LineVul but is engineered to satisfy the three milestones defined in the assignment. All components are packaged in a Jupyter notebook so it can be easily deployed on vast.ai after Google Colab's free tier proved insufficient.

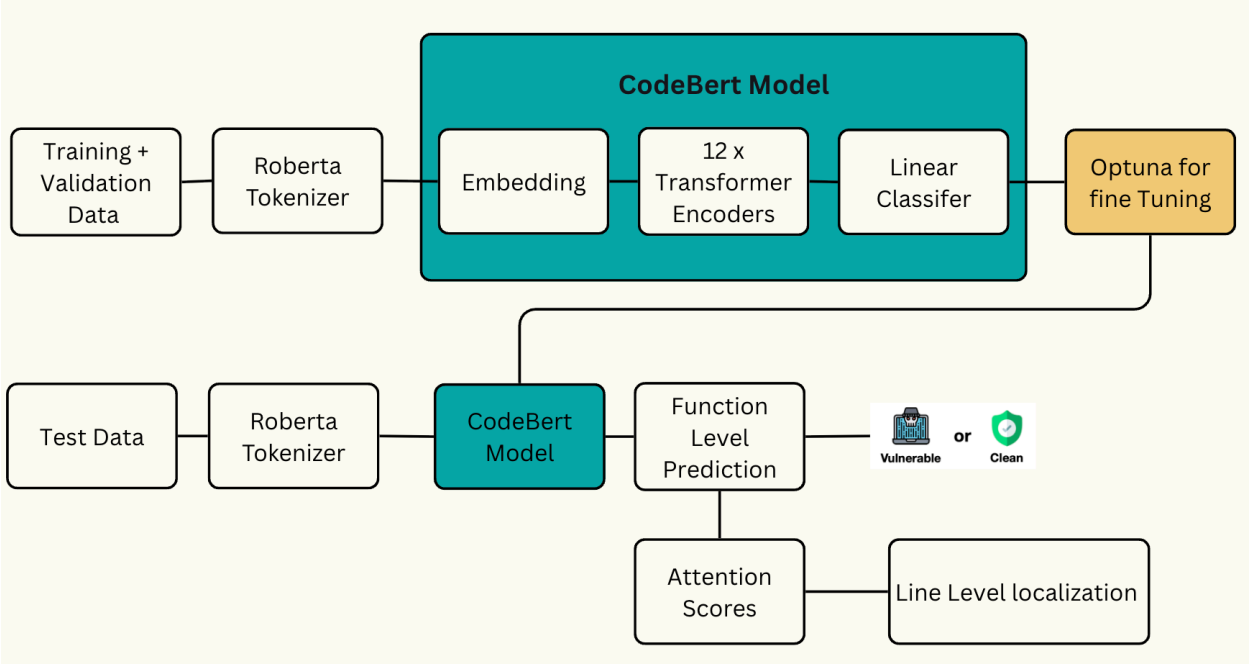
The task involved building a predictive model using Big-Vul dataset to detect vulnerabilities at both the function and line levels in C/C++ code snippets. Our approach was structured into three milestones:

- **M1:** Obtain embeddings of functions using a tokenizer (CodeBERT).
- **M2:** Predict vulnerability at the function level.
- **M3:** Extend the model to line-level prediction, identifying precisely which lines are vulnerable.

Our Novelties compared to Linevul paper are:

- Using Optuna,
- Using Roberta Tokenizer,
- Evaluating different aggregations on the line level detection (sum, avg, max),
- Defining new custom metrics for the line level

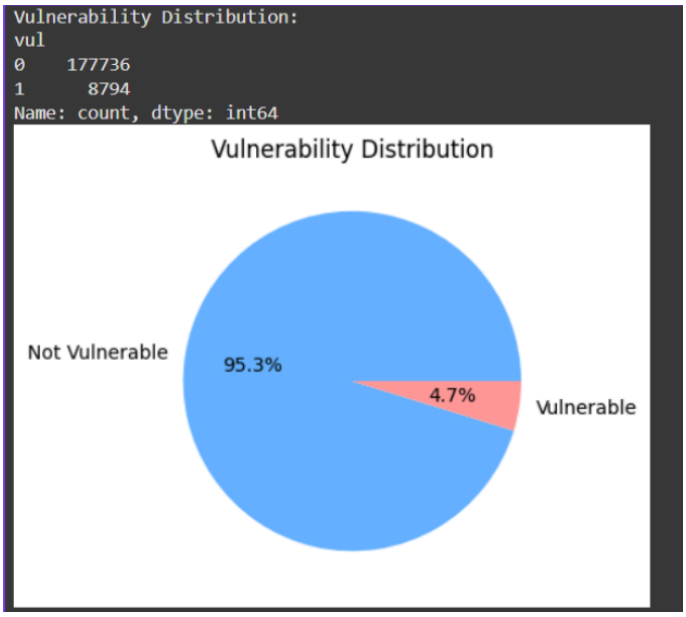
Here is the general flow of our model:



2.1 Dataset Preparation

The Big-Vul dataset was loaded into a Google Colab environment directly from Google Drive to facilitate easier data management and efficient access

For dataset preparation, we used the Big-Vul dataset, which initially contained 186530 samples. Vulnerable and non-vulnerable functions were distributed with a ratio of about 4.7% vulnerable to 95.3% non-vulnerable. To manage computational resources, we created a smaller dataset comprising 10% of the original data (~18,000 samples), preserving the original distribution between vulnerable and safe functions. This reduced dataset was further split into training, validation, and testing subsets with an 80%-10%-10% ratio, respectively. While the original distribution was maintained for the final experiment, various distributions were tested in other experiments to evaluate their effect on results.



train	validation	test
80%	10%	10%

2.2 Roberta Tokenizer

We began by feeding the code into the publicly available CodeBERT byte-pair-encoding tokenizer, specifically using RobertaTokenizerFast from HuggingFace (`microsoft/codebert-base`). This tokenizer converts each token and its position into integer IDs and returns offset mappings that help link tokens back to their original lines of code. These sequences were passed through the frozen layers of CodeBERT to obtain a dense representation of size 512 × 768. This embedding step was crucial as it converted textual data into a numeric format suitable for deep learning.

2.3 Training Model

We fine-tuned a **RobertaForSequenceClassification** model from HuggingFace Transformers (CodeBERT). During the forward pass, the model generated 12×12 attention maps and a 768-dimensional pooled output vector (pooler_output). For function-level vulnerability prediction, we used this 768-dimensional vector.

The training process employed backpropagation with the AdamW optimizer. We set up a warmup period for the initial 10% of the total training steps, during which the learning rate linearly increased from 0 to 2e-5. We invoked `scheduler.step()` after each `optimizer.step()` within the training loop to implement a linear warmup followed by linear decay, ensuring the model did not become trapped in suboptimal solutions or oscillate during early training phases

2.4 Hyperparameter Optimization with Optuna

Optuna, a Python library designed for automated black-box hyperparameter optimization, was employed to efficiently tune hyperparameters:

- **Objective function**: We sampled hyperparameters such as:
  - earning rate (lr ∈ [2e-6,1e-5,2e-5,3e-5,5e-5,2e-4]),
  - batch size (∈ [8,16,32]}),
  - weight decay (wd ∈ [0.0,0.01,0.1]),
  - max_epochs (∈ [2-8]),
  - max sequence length (∈ {512}).
- **Training and validation**: Each trial trained the model using these parameters, evaluated the validation F1 score each epoch, and pruned underperforming trials early based on their relative performance.
- **Sampler and Pruner**: We used the Tree-structured Parzen Estimator (TPESampler) for efficient exploration and MedianPruner to halt poorly performing trials early. For example:

```
Starting trial: 12
```

```
Some weights of RobertaForSequenceClassification were not initialized from the
model checkpoint at microsoft/codebert-base and are newly initialized:
['classifier.dense.bias', 'classifier.dense.weight',
'classifier.out_proj.bias', 'classifier.out_proj.weight']

You should probably TRAIN this model on a down-stream task to be able to use it
for predictions and inference.

[I 2025-06-01 12:27:58,911] Trial 12 pruned.
```

The best model from 15 trials was subsequently employed for function-level vulnerability predictions.

```
study.optimize(objective, n_trials=15)
```

2.5 Line-Level Localisation

For line-level localization, we followed these detailed steps:

1. Functions predicted as vulnerable at the function level were selected for further line-level analysis.
2. Token-to-Line Mapping:
    - Obtained the start and end character positions of each subtoken from the tokenizer.
    - Created a character-to-line mapping by scanning through the code, incrementing the line number at each newline character.
    - Assigned each subtoken to a source-code line by matching its starting character index to the character-to-line mapping.
3. Calculating Token Scores Using Self-Attention:
    - Extracted all self-attention weights (12 layers × multiple heads) from CodeBERT.
    - Summed and averaged the self-attention weights across all heads in each layer and across all layers.
    - Extracted attention from the [CLS] token to each subtoken, assigning a single importance score per subtoken.
4. Calculating Line Scores:
    - **Summed/Aggregated/Max** subtoken scores by line, summing or averaging to obtain a single attention-based score per line.
    - Sorted lines based on these scores, identifying top-ranked lines likely containing vulnerabilities.

3 Experimental Results & Discussion

3.1 Hyperparameter Optimization

Optuna trials identified the following best parameters:

- Learning Rate: 2e-5
- Batch Size: 16
- Weight Decay: 0.1
- Max Sequence Length: 512
- Max epoch: 10

3.2 Training and Validation Performance

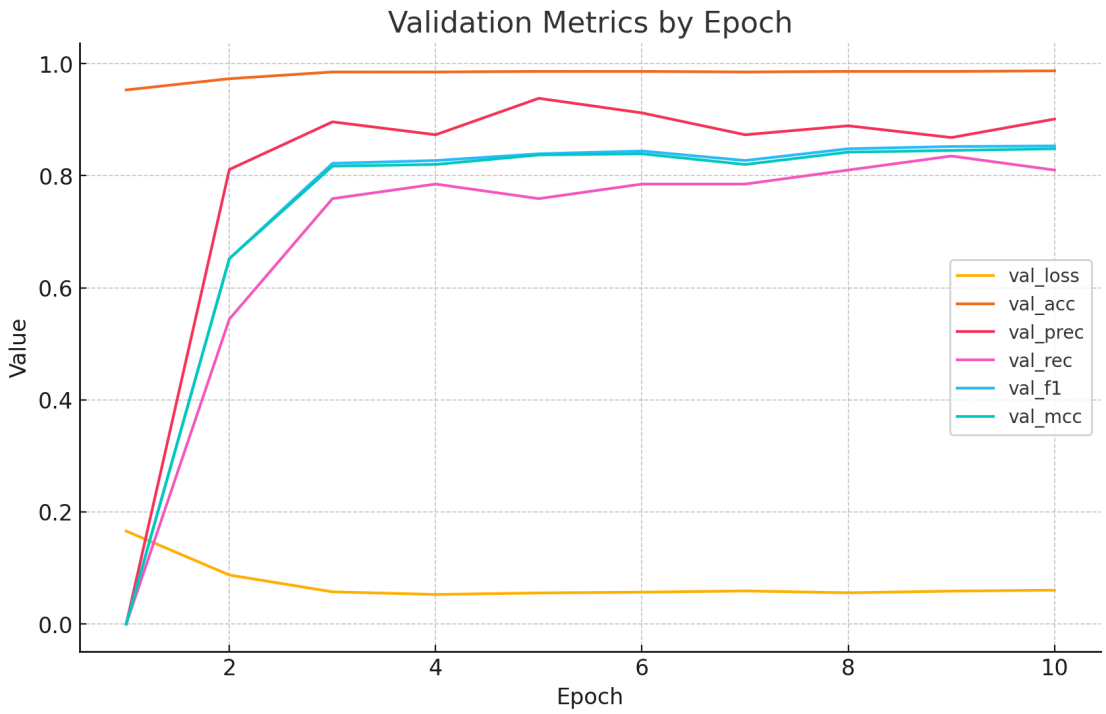
Due to limited compute and storage resources, we fine-tuned our Transformer on a subset of just 17,000 training examples. We trained for 10 epochs using the best Optuna-found hyperparameters (learning rate = 1×10^{-5}, batch size = 16, max. sequence length = 512, weight decay = 0.1). Over those 10 epochs, validation loss fell from 0.1659 to 0.0603 and validation accuracy climbed from 0.953 to 0.987. Our precision, recall, F1-score, and Matthews correlation coefficient (MCC) all showed strong gains by epoch 3 and then plateaued, ending at 0.901, 0.810, 0.853, and 0.848, respectively. Despite using only 17k samples, the model achieved stable, high performance by epoch 5, with only minor metric fluctuations, which demonstrates that our chosen hyperparameters and early-stopping pruner effectively guided training under resource constraints.

Table 1 lists the full set of validation metrics by epoch, and Figure 2 visualizes their progression. These results suggest that the model converges reliably and achieves high performance under our chosen hyperparameter settings.

Table 1. Validation Metrics by Epoch
Validation metrics for each of the 10 fine-tuning epochs (17,000 samples; lr = 2×10^{-5}, bs = 16, msl = 512, wd = 0.1).

Epoch	Val. Loss	Val. Acc	Precision	Recall	F1-Score	MCC
1	0.1659	0.953	0.000	0.000	0.000	0.000
2	0.0874	0.973	0.811	0.544	0.652	0.652
3	0.0574	0.985	0.896	0.759	0.822	0.817
4	0.0527	0.985	0.873	0.785	0.827	0.820
5	0.0554	0.986	0.938	0.759	0.839	0.837
6	0.0568	0.986	0.912	0.785	0.844	0.839
7	0.0590	0.985	0.873	0.785	0.827	0.820
8	0.0557	0.986	0.889	0.810	0.848	0.842
9	0.0587	0.986	0.868	0.835	0.852	0.845
10	0.0603	0.987	0.901	0.810	0.853	0.848

Figure 1. Progression of validation loss, accuracy, precision, recall, F1-score, and Matthews correlation coefficient (MCC) over 10 fine-tuning epochs (17,000 samples; lr = 2×10^{-5}, bs = 16, msl = 512, wd = 0.1).



3.3 Test-Evaluation

Function-level

On the **function-level** detection task, our fine-tuned Transformer achieved an accuracy of 0.983, precision of 0.895, recall of 0.739, F1-score of 0.810, and Matthews correlation coefficient (MCC) of 0.805 on held-out functions.

Line-Level Metrics

For our line-level vulnerability detection, we report four metrics: **Top-K Precision**, **Top-K Recall**, **IFA-Average**, and **IFA-Mean**.

1. Top-K Precision

Top-K Precision measures, across all functions predicted as vulnerable, the fraction of the top K lines that are truly vulnerable. In other words, if you take every function flagged as vulnerable and look at its K highest-scoring lines, Top-K Precision tells you what proportion of those K lines are true positives.

2. Top-K Recal

Top-K Recall captures, over all truly vulnerable functions, the fraction whose single flawed line appears anywhere among its top K ranked lines. For example, with K = 10, Top-10 Recall is the percentage of vulnerable functions for which the real vulnerable line is included in that function's top 10 predicted lines.

3. Initial False Alarm (IFA)

The remaining two metrics quantify how many non-vulnerable lines you encounter before hitting the true positive line:

- **IFA-Average** is the average number of false alarms (non-vulnerable lines) inspected before the first correct detection.

- **IFA-Mean** reports the median number of false alarms before the first true positive.

Together, these metrics give a fuller picture of both precision-focused performance (Top-K) and the early-detection efficiency (IFA) of our line-level model.

Line-Level

The model achieved a Top-10-precision of 0.632, Top-10-recall of 0.552, IFA-average of 8.51 lines, and IFA-median of 5.5 lines. The drop in recall relative to the function-level task reflects the greater difficulty of pinpointing individual vulnerable lines, while the high precision shows a conservative detection strategy that minimizes false alarms. Together, these results demonstrate that our approach generalizes well from coarse- to fine-grained vulnerability detection.
Here is a comparison of different approaches for line-level detection:

Aggregation Method	Top-10 Precision	Top-10 Recall	Initial False Alarm (Avg.)	Initial False Alarm (Med.)
Sum	0.632	0.522	8.51	5.50
Max	0.447	0.370	10.87	8.50
Average	0.408	0.337	13.04	10.00

4 Summary on Techniques and Resources

- **Model:** RobertaForSequenceClassification (CodeBERT)
- **Tokenization:** RobertaTokenizerFast
- **Optimization:** Optuna
- **Datasets:** Big-Vul dataset,
- **Libraries:** PyTorch, HuggingFace Transformers, Scikit-learn, Pandas, Matplotlib

5 Source Code

The complete implementation and experiments can be found in the following repository:

https://github.com/rokkovach/623.828-group-a-final

6 Resource & Hosting Management

We initially experimented with Google Colab to access free GPUs, but encountered several limitations, such as unpredictable session timeouts, restricted RAM (around 12 GB), and GPU availability fluctuations that interrupted long-running training jobs. Local training on our personal machines avoided these cloud constraints but imposed its own bottleneck; only one team member could work at a time, making collaboration and parallel experiments impractical.

To overcome these challenges, we turned to rented on-demand GPUs and selected Vast.ai for its flexible pricing and hardware options. Throughout this project, we provisioned a single “1× RTX 5080” instance (53.4 TFLOPS, 15.8 GB VRAM) on an Intel Core™ i5-14600K host with 32 GB RAM and a 500 GB SSD (Figure 3). This setup ran our PyTorch workloads smoothly (CUDA 12.8.1) at roughly $0.20 per hour, allowing multiple team members to schedule jobs concurrently without worrying about session limits or local hardware conflicts

7 References

- Vaswani, A. et al. (2017). "Attention Is All You Need." *Advances in Neural Information Processing Systems.*
- Fan, Y. et al. (2022). "LineVul: A Transformer-based Line-Level Vulnerability Prediction." *MSR 2022.*
- Lab Assignment Presentation (2025).
- https://optuna.org/
- Vast.AI QuickStart Guide Walkthrough, https://www.youtube.com/watch?v=iz8Tzt9UN3k&ab_channel=VastAI (2023)