**MSc course on “Cybersecurity Data Science”**

*PBL Project Report*

GROUP MEMBERS:

* *Astha, Gupta | 640204*
* *Mayank, Rawat | 640192*
* *Akshita, Badola | 642493*

CODE REPO:

[**Link to the Git Repository**](https://github.com/mayankr225/CDS_2025.git)

*\*Describe the following aspects of your solution (PBL parts 2 and 3).*

1. **Dataset:**
   1. **Sources**:We built our dataset using information from ProjectKB, a collaborative platform that collects and organizes data about security vulnerabilities in open-source Java software. ProjectKB provides detailed records of Common Vulnerabilities and Exposures (CVEs) along with the specific code changes (commits) that addressed these issues.
   2. **Filtering criteria**:

To build our dataset, we carefully reviewed the list of CVEs and their corresponding fixing commits available in ProjectKB. For each vulnerability, we identified the specific code changes that were made to fix the issue. Our approach was as follows:

* First, we selected relevant CVEs from the ProjectKB repository and located the commits that resolved these vulnerabilities.
* Next, we focused on the methods within those commits that were actually modified as part of the fix.
* For each method, we extracted two versions:

▫ The **“vulnerable”** version, which is the code before the fix was applied

▫ The **“fixed”** version, which is the code after the fix

By comparing these two versions, we were able to put together a dataset that clearly shows both the vulnerable and patched states of the code, making it well-suited for machine learning analysis.

* 1. **Size**:The dataset consists of 38,255 entries, each including code samples along with labels indicating whether they are vulnerable or not.
  2. **Distribution**:The dataset is made up of 21,428 samples labeled as non-vulnerable (0) and 16,827 samples labeled as vulnerable (1), covering a total of 698 unique CVEs.

1. **Architecture**:
2. **Feature representation:**

For feature representation, we used **automatically extracted code embeddings**. The raw code text was tokenized using tf.keras.preprocessing.text.Tokenizer with a vocabulary size of 10,000 words. These tokens were then converted into integer sequences and **padded** to a fixed max\_length of 200 using pad\_sequences. These padded sequences of integers served as the input to the embedding layer of our deep learning models.

1. **Model choice**:

We experimented with three different deep learning architectures:

1. **CNN-only Model**: This model utilized a Convolutional Neural Network (CNN) to capture local patterns within the code sequences.
2. **BiLSTM-only Model**: This model employed a Bidirectional Long Short-Term Memory (BiLSTM) network to learn long-range dependencies and sequential information in the code.
3. **Hybrid CNN + BiLSTM Models (Two Variations)**: These models combined the strengths of both CNNs and BiLSTMs.
   * **Hybrid Model 1**: Consisted of a CNN layer followed by a BiLSTM layer.
   * **Hybrid Model 2**: Featured two CNN layers (with a max-pooling layer after each) followed by a BiLSTM layer, designed to extract more hierarchical features before processing the sequence.
4. **Model configuration/parameters**:

All models shared an initial **Embedding layer** with input\_dim=10000 (vocab size) and output\_dim=128 (embedding dimension). They all concluded with a **Dense layer** with 1 unit and a **sigmoid activation** function for binary classification. The **loss function** used for all models was binary\_crossentropy, and the **optimizer** was Adam (with a learning\_rate=1e-4 for the BiLSTM and Hybrid models). We used a batch\_size of 64 and trained for up to 20 epochs, with an EarlyStopping callback to prevent overfitting.

Here's a detailed breakdown of each model's configuration:

* **CNN-only Model**:
  + layers.Embedding(input\_dim=vocab\_size, output\_dim=embedding\_dim)
  + layers.Conv1D(filters=64, kernel\_size=5, activation='relu')
  + layers.GlobalMaxPooling1D()
  + layers.Dense(64, activation='relu')
  + layers.Dropout(0.5)
  + layers.Dense(1, activation='sigmoid')
  + Optimizer: adam (default learning rate)
* **BiLSTM-only Model**:
  + layers.Embedding(input\_dim=vocab\_size, output\_dim=embedding\_dim)
  + layers.Bidirectional(layers.LSTM(64, return\_sequences=True))
  + layers.GlobalMaxPooling1D()
  + layers.Dense(64, activation='relu')
  + layers.Dropout(0.3)
  + layers.Dense(1, activation='sigmoid')
  + Optimizer: Adam(learning\_rate=1e-4)
* **Hybrid CNN + BiLSTM Model 1**:
  + layers.Embedding(input\_dim=vocab\_size, output\_dim=embedding\_dim)
  + layers.Conv1D(filters=64, kernel\_size=5, activation='relu')
  + layers.MaxPooling1D(pool\_size=2)
  + layers.Bidirectional(layers.LSTM(64, return\_sequences=True))
  + layers.GlobalMaxPooling1D()
  + layers.Dense(64, activation='relu')
  + layers.Dropout(0.5)
  + layers.Dense(1, activation='sigmoid')
  + Optimizer: Adam(learning\_rate=1e-4)
* **Hybrid CNN + BiLSTM Model 2**:
  + layers.Embedding(input\_dim=vocab\_size, output\_dim=embedding\_dim, input\_length=max\_length)
  + layers.Conv1D(128, 5, activation='relu')
  + layers.MaxPooling1D(pool\_size=2)
  + layers.Conv1D(64, 3, activation='relu')
  + layers.MaxPooling1D(pool\_size=2)
  + layers.Bidirectional(layers.LSTM(64, return\_sequences=True))
  + layers.GlobalMaxPooling1D()
  + layers.Dense(64, activation='relu', kernel\_regularizer=regularizers.l2(0.001)) (L2 regularization added)
  + layers.Dropout(0.5)
  + layers.Dense(1, activation='sigmoid')
  + Optimizer: Adam(learning\_rate=1e-4)

1. **Training procedure**:

### **Dataset Split**

We split our dataset into **training, validation, and test sets** using a 70/15/15 ratio. Specifically, we first split the data into 70% for training and 30% for a temporary set. Then, the temporary set was further split equally into 15% for validation and 15% for testing. This yielded the following distribution:

* **Train set**: 26778 samples (70%)
* **Validation set**: 5738 samples (15%)
* **Test set**: 5739 samples (15%)

This split ensures that the model is trained on a substantial portion of the data, evaluated for hyperparameter tuning and early stopping on a separate validation set, and finally assessed on a completely unseen test set to provide an unbiased estimate of its generalization performance. We used random\_state=42 for reproducibility and stratify=y during the train\_test\_split calls to maintain the original class distribution in each subset, which is crucial given potential class imbalance in the dataset.

### **Class Imbalance**

To address potential class imbalance, we applied **class weighting** during the training of the BiLSTM and Hybrid CNN + BiLSTM models. We assigned a higher weight to the minority class (class 1) to ensure the model paid more attention to correctly classifying these samples. For the BiLSTM-only and Hybrid CNN + BiLSTM Model 1, we used class\_weight={0: 1.0, 1: 2.0}. For Hybrid CNN + BiLSTM Model 2, we fine-tuned this to class\_weights={0: 1.0, 1: 1.2}.

### **Early Stopping**

All models utilized an EarlyStopping callback. This technique monitors the val\_loss and stops training if the validation loss does not improve for a specified number of epochs (patience=5 for the first three models, and patience=3 for the Hybrid CNN + BiLSTM Model 2). The restore\_best\_weights=True parameter ensures that the model weights from the epoch with the best validation loss are restored. This prevents overfitting and reduces unnecessary training time. For Hybrid CNN + BiLSTM Model 2, we also included ModelCheckpoint to save the best performing model.

1. **Performance evaluation**:

We used the following metrics to evaluate the performance of our models:

* **Accuracy**: The proportion of correctly classified instances.
* **Precision**: The proportion of true positive predictions among all positive predictions. It is crucial when the cost of false positives is high.
* **Recall**: The proportion of true positive predictions among all actual positive instances. It is important when the cost of false negatives is high.
* **F1-Score**: The harmonic mean of precision and recall, providing a balanced measure that is particularly useful for imbalanced datasets.
* **Loss**: Binary Crossentropy, which quantifies the error between predicted probabilities and true labels.

Here are the test set results for each model:

| Model | Test Accuracy | Test Precision | Test Recall | Test F1-Score |
| --- | --- | --- | --- | --- |
| **CNN-only Model** | 0.6738 | 0.6538 | 0.5491 | 0.5969 |
| **BiLSTM-only Model** | 0.6332 | 0.5611 | 0.7627 | 0.6465 |
| **Hybrid CNN + BiLSTM 1** | 0.6175 | 0.5444 | 0.7983 | 0.6474 |
| **Hybrid CNN + BiLSTM 2** | 0.6430 | 0.5803 | 0.6799 | 0.6262 |

1. **Main takeaways**:
2. **The importance of sequential context (BiLSTM) for this task is evident, even with potential trade-offs in precision:** Both BiLSTM-only and Hybrid CNN + BiLSTM models, which incorporate recurrent layers, achieved higher F1-scores and significantly better recall compared to the CNN-only model. The BiLSTM-only model achieved an F1-score of 0.6465 and a recall of 0.7627, while Hybrid CNN + BiLSTM 1 achieved a very similar F1-score of 0.6474 with the highest recall of 0.7983. This indicates that understanding the sequential flow and dependencies within the code is more critical for identifying the "positive" class (label 1) than relying solely on local features (as in the CNN-only model). While the precision for these models might be slightly lower than that of the CNN-only model, the improved recall suggests they are better at identifying actual positive cases, which is often crucial in tasks like vulnerability detection or code classification where missing positive instances can have severe consequences.
3. **Hybrid architectures offer a balanced approach but require careful tuning:** The hybrid models aimed to leverage the strengths of both CNNs (local feature extraction) and BiLSTMs (sequential pattern learning). Hybrid CNN + BiLSTM 1 performed very similarly to the BiLSTM-only model in terms of F1-score. Hybrid CNN + BiLSTM 2, with its deeper CNN layers and L2 regularization, saw a slight drop in F1-score compared to the first hybrid model (0.6262 vs 0.6474). This suggests that while combining architectures is a good idea, simply adding more layers doesn't guarantee better performance. The subtle differences in layer configuration, such as the number of CNN layers and regularization, can significantly impact the final results. This indicates that optimal hybrid model design requires extensive hyperparameter tuning and potentially more complex architectural choices to truly realize the benefits of combining these networks.
4. **Performance did not reach extremely high levels, suggesting the need for more sophisticated approaches:** The highest F1-score achieved was around 0.6474. While this is better than random chance, it's not exceptionally high for a binary classification task. This performance might be influenced by several factors:

* **Data Complexity**: Code data can be highly complex and diverse, making it challenging for models to generalize.
* **Feature Limitations**: While tokenization and embedding are a good start, they might not fully capture the nuanced semantic and syntactic properties of code that differentiate between classes. More advanced code representations (e.g., AST-based embeddings, graph neural networks) could be explored.
* **Model Capacity**: Even with deep learning, the current model architectures might not be sufficiently complex to learn all the intricate patterns.
* **Class Imbalance Handling**: While we used class weighting, the imbalance might still pose a challenge. Further techniques like oversampling (SMOTE) or undersampling could be investigated.

We would do several things differently next time. We would investigate more advanced code representation techniques beyond simple tokenization and padding, perhaps by incorporating Abstract Syntax Trees (ASTs) or Control Flow Graphs (CFGs). We would also explore more sophisticated deep learning architectures, such as transformer-based models, which have shown great promise in natural language processing tasks and could be adapted for code. Finally, a more rigorous hyperparameter optimization process using techniques like grid search or Bayesian optimization would be beneficial to fine-tune model parameters and potentially unlock better performance.

**INDIVIDUAL CONTRIBUTIONS**

*\*Indicate the individual efforts of each team member across the different PBL activities. Note: The tasks in the Table are indicative (feel free to adjust them as needed by adding/removing tasks). All rows must add 100%!*

| **PBL Task** | *Astha, Gupta* | *Mayank, Rawat* | *Akshita, Badola* |
| --- | --- | --- | --- |
| ***Data collection and preprocessing*** (e.g., identification and mining of relevant data) | 35% | 45% | 20% |
| ***Model development*** (e.g., selected, implemented and trained the models) | 40% | 30% | 30% |
| ***Evaluation and analysis*** (e.g., compute and interpret performance metrics) | 33% | 33% | 33% |
| ***Tooling and infrastructure*** (e.g., set up repo, containers, etc.) | 35% | 35% | 30% |

**DECLARATION OF AGREEMENT**

**We, the undersigned, confirm that we have reviewed the contribution table above and agree that it fairly represents the distribution of effort in this project.**

IMPORTANT: This declaration must be signed by **at least half** of the group members!

To sign, add your names below.

*Astha, Gupta | 640204*

*Mayank, Rawat | 640192*

*Akshita, Badola | 642493*

*Hamburg, 17.07.2025*