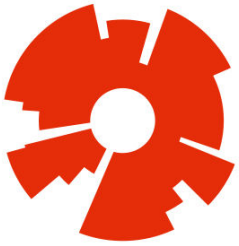

HÁSKÓLINN Í REYKJAVÍK
REYKJAVIK UNIVERSITY

Computer Science department

Machine Learning in Cybersecurity

T-710-MLCS

Report 4 - Malware classification with API call traces

Saldana Giorgio
Email: giorgio24@ru.is

12th October 2024

Contents

Introduction

Malware classification is a crucial task in cybersecurity, where distinguishing between benign and malicious software can prevent significant system damage. This task leverages API call traces—sequences of system calls made by programs during execution—to classify software behavior. In this assignment, a dataset of pre-recorded API call traces, along with corresponding labels, is used to train machine learning models to differentiate between different malware families.

The primary challenge is converting variable-length traces into a fixed-size representation suitable for machine learning models. This is addressed using methods such as Bag-of-Words and N-grams, which capture the presence and frequency of API calls. These methods enable the application of various classifiers, such as Logistic Regression, Random Forest, and LightGBM.

Additionally, more advanced techniques like Long Short-Term Memory (LSTM) networks are explored. LSTMs are particularly effective for sequential data, capturing temporal patterns in API traces that traditional models may overlook. The ultimate goal is to determine the most effective model for this classification task and export it for independent evaluation. The report is detailed as follows:

- Preprocessing Steps;
- Train and Test Simple Classifier;
- Export Best Model for an Independent Classifier;
- Advanced Techniques (LSTM);
- N-grams;

For reproducibility, the code can be found here.

To run the experiments and find the best model, execute the following command:

```
python pipeline.py
```

Note that running the experiments requires using both the `scikit-learn` and `lightgbm` packages. You can install `lightgbm` by running:

```
pip3 install lightgbm
```

Once the best model is determined, you can classify a new test set by running:

```
python classify-trace.py <file_name>
```

1 Data Preprocessing and Splitting

The raw dataset comprises API call traces, represented as comma-separated lists of numerical values, where each trace is associated with a label categorizing it as malware family. Preprocessing was handled by a dedicated class structure to ensure modularity and scalability, making it adaptable to future enhancements, such as N-gram extraction.

Removing Duplicate API Calls

Consecutive repeated API calls were removed from the traces, effectively reducing noise without losing critical sequence information. Redundant API calls often occur in rapid succession, and their removal allows for a more concise representation of the trace, which in turn simplifies the feature extraction process. By eliminating these consecutive repetitions, the model can focus on the unique API call patterns that are more likely to define a particular malware family. The `Preprocessor` class automates this process, making it easy to apply this cleanup step before further transformations.

Bag-of-Words

After cleaning the traces, feature extraction was carried out using a Bag-of-Words (BoW) approach. BoW converts the API traces into fixed-size feature vectors, where each vector element represents the occurrence of a specific API call in the trace, regardless of its position in the sequence. This approach is effective for capturing the presence of individual API calls that may be characteristic of specific malware families. The `CountVectorizer` was used to implement this method, enabling the extraction of features based on API call frequencies.

Splitting the Data

Once the API traces were converted into feature vectors, the dataset was split into training and testing sets. A typical 80/20 split was employed, reserving 20% of the data for testing purposes. This ensures that the models are trained on a portion of the data and then evaluated on unseen samples, allowing for an unbiased estimate of the model's generalization performance.

Modular and Scalable Design

The modular design of the preprocessing pipeline is crucial in maintaining flexibility for future developments. For instance, the current structure allows for easy integration of more sophisticated feature extraction techniques, such as N-grams or deep learning-based embeddings. By using a class-based approach, the codebase remains maintainable and adaptable, ensuring that new preprocessing strategies can be incorporated with minimal changes to the existing framework.

2 Train and Test Simple Classifier

To evaluate the performance of different classifiers on the preprocessed dataset, several machine learning models were implemented. The evaluation process was handled by the MalwareClassifierEvaluator class, which allowed the models to be tested and compared consistently.

Logistic Regression

Logistic Regression was employed as a baseline classifier, known for its simplicity and effectiveness on linearly separable data. To handle the class imbalance present in the dataset, the class_weight='balanced' parameter was used. The Newton-CG solver was selected for optimization, providing better performance for this dataset. Feature scaling was applied using the StandardScaler class from the scikit-learn library, ensuring that the logistic regression model could converge efficiently by normalizing feature values.

K-Nearest Neighbors (KNN)

The K-Nearest Neighbors (KNN) algorithm was also tested as a distance-based classifier, relying on the proximity between data points to assign labels. This model was straightforward to implement using the KNeighborsClassifier class from the scikit-learn library. Despite being simple, KNN can perform well for certain types of feature spaces but tends to struggle with high-dimensional data like the API traces. However, it provided a useful comparison point in the performance evaluations.

Random Forest

Random Forest, an ensemble learning method, was selected due to its robustness in handling complex feature interactions and its ability to deal with high-dimensional data. The model was implemented using the RandomForestClassifier class, with hyperparameters tuned for optimal performance. A total of 1000 trees were used (n_estimators=1000), and max_depth=30 was set to control the depth of individual trees. Balanced class weights were again applied to address the uneven distribution of malware families. This model was chosen because of its interpretability and its capacity to avoid overfitting by averaging over many trees.

LightGBM

LightGBM, a gradient boosting framework, was identified as a highly efficient model for large-scale datasets and was included in the experiment after researching its superior performance in various text and sequence-based classification tasks. It was implemented using the LGBMClassifier from the lightgbm library. Hyperparameters such as the number of leaves (num_leaves=64), maximum tree depth (max_depth=20), and learning rate (learning_rate=0.01) were tuned based on the characteristics of the dataset.

LightGBM's ability to handle large feature spaces and its efficient memory usage made it particularly suitable for this task, outperforming classical models in terms of speed and accuracy. It also handles class imbalance effectively with the class_weight='balanced' parameter.

Evaluation Metrics

Each classifier was evaluated using confusion matrices to visualize the model's classification accuracy across different malware families. The macro F1 score, a balanced metric that accounts for precision and recall across all classes, was used to compare the performance of the models. The generated confusion matrices are presented below.

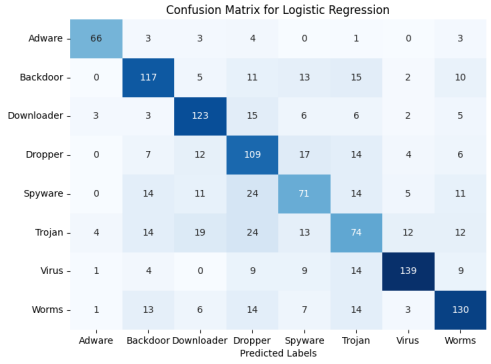

Figure 1: LR Confusion Matrix

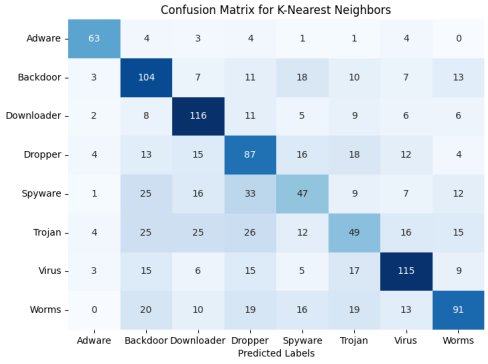

Figure 2: KNN Confusion Matrix

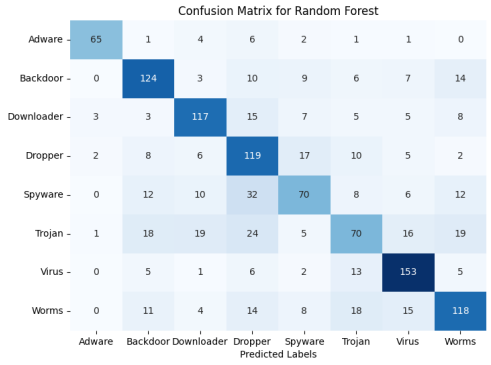

Figure 3: RF Confusion Matrix

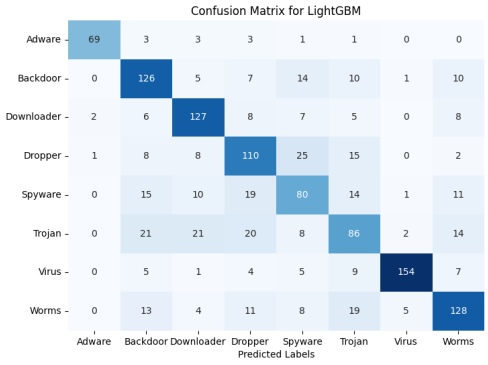

Figure 4: LGBM Confusion Matrix

The testing revealed that while Logistic Regression and KNN provided reasonable baselines, the Random Forest and LightGBM classifiers performed significantly better in terms of both accuracy and F1 score. The final model selected for persistence and independent testing was LightGBM, as it consistently achieved the highest F1 scores and handled the complexity of the data most efficiently.

3 Exporting the Best Model

The highest-performing model, based on the macro F1 score, was saved for future use using the joblib library. The selected model, saved as best_model.joblib, was integrated into a standalone Python script, classify-trace.py, which processes new API traces for classification. This script accepts an input file containing API traces, which are preprocessed using the same vectorization technique employed during training. The script loads a pre-fitted vectorizer (vectorizer.joblib), which transforms the input traces into feature vectors, avoiding the need for repeated preprocessing or fitting. Specifically, the vectorizer handles the feature extraction using the N-grams (ngram_range=(1, 3)) technique. The transformed input is then classified by the saved model, and the script outputs the predicted malware family for each trace.

4 Optional Enhancements

As part of the optional enhancements, two advanced techniques were implemented to further improve the classification performance: Long Short-Term Memory (LSTM) networks and N-grams. Both approaches were seamlessly integrated into the existing modular structure, utilizing the Preprocessor class for data preparation and the MalwareClassifierEvaluator class.

4.1 LSTM

Long Short-Term Memory (LSTM) networks were implemented to handle the sequential nature of API call traces, capturing temporal dependencies that simpler models might overlook. To prepare the data for LSTM input, a specialized preprocessing routine was created. This involved tokenizing and padding the API call sequences to ensure uniform input length, as required by the LSTM architecture. The Preprocessor class was extended to handle this tokenization and padding, ensuring that the sequential relationships between API calls were preserved.

The LSTM model itself was built using multiple bidirectional LSTM layers, which allowed for better context retention by processing sequences in both forward and backward directions. An embedding layer was used to transform the API calls into dense vectors, while dropout layers were added to prevent overfitting. Dense layers were included for the final classification output. This architecture was specifically designed to capture long-term dependencies within the API call sequences.

Despite its advanced architecture, the LSTM model underperformed compared to more traditional models like Random Forest and LightGBM. This may be attributed to the relatively small dataset, which may not have been sufficient to extract meaningful temporal patterns. The LSTM model, while promising for larger sequential datasets, did not capture the necessary feature relationships as effectively in this context.

4.2 N-grams

In addition to LSTM, N-grams were also implemented as a feature extraction technique, leveraging the modular structure of the Preprocessor class. N-grams capture consecutive sequences of API calls, providing more contextual information compared to the basic Bag-of-Words approach. For this task, the N-gram range was set to (1, 3), meaning that unigrams, bigrams, and trigrams were extracted from the API traces.

This approach allowed classifiers like Random Forest and LightGBM to make use of richer features, as N-grams can capture the relationships between consecutive API calls. The Preprocessor class handled this feature extraction seamlessly by allowing for flexible N-gram configurations. The inclusion of N-grams significantly improved the ability of the models to differentiate between malware families, particularly when used in combination with the LightGBM classifier, which excels at handling high-dimensional feature spaces.

Conclusion

In this assignment, multiple models were evaluated for classifying malware families using API call traces. The preprocessing phase, organized through a modular class structure, played a crucial role in converting variable-length traces into fixed-size feature vectors, facilitating the application of various classifiers. The use of N-grams (with an N-gram range of 1 to 3) significantly enriched the feature space, improving the effectiveness of models like Random Forest and LightGBM.

Among the classifiers tested, LightGBM demonstrated superior performance, offering both high accuracy and efficiency, particularly in managing the high-dimensional feature space generated by N-grams. Although the LSTM model showed potential for capturing temporal dependencies, it underperformed in this case, likely due to the dataset's size and its inability to effectively extract sequential patterns. Ultimately, LightGBM proved to be the most effective model for this classification task, fulfilling the assignment's objectives.