UNIVERSITY OF HEIDELBERG

Project of the Practical AI Methods and Tools for Programming

**BinaryML: Classifying Binaries for Malware**

**and Vulnerability Detection**

Anusha Chattopadhyay, Soumili Samanta and Sri Pavan Sesha Sai Rallapalli

Prof. Dr. Arthur Andrzejak

September 2023

**Table of Contents**

**0 Preliminaries** **3**

**1 Literature Search** **3**

**2 Malware detection model** **3**

2.1 Introduction3

2.2 Repository Structure4

2.3 Main Pipeline4

2.4 Project Roadmap4

2.5 Key Components5

2.5.1 Classes5

2.5.2 Functions5

2.5.3 Enhancement- Individual Predictions5

2.5.4 Predictive analysis based on individual predictions 6

2.5.5 Training Challenges6

2.6 Future Extensions6

**3 Vulnerability detection model7**

3.1 Romeo7

3.1.1 Project Roadmap7

3.1.2 Introduction Romeo7

3.1.3 Repository Structure7

3.1.4 Main Pipeline8

3.1.5 Key Components of the Pipeline8

3.1.6 Research Specifics9

3.1.7 Creating Data loader for Malware dataset 9

3.1.8 Predictive Analysis10

3.1.9 Future Extensions10

3.2Illuminati11

3.2.1 Introduction Illuminati11

3.2.2 Repository Structure 11

3.2.3 Data Loader11

3.2.4 Challenges12

3.2.5 Future Extensions 12

1. **Preliminaries**

This developer documentation provides an overview of the BinaryML project's structure, key components, and guidelines for extending and maintaining the codebase. It aims to help developers understand the project and facilitate further development.

The code for our project can be found at <https://github.com/pvs-hd-tea/23ss-BinaryML>

As the project is graded individually, we mark who is primarily responsible for the respective sections. The colour notations we used to identify each person's primary contribution are given below.

1. **Anusha Chattopadhyay:** coloured in Green
2. **Soumili Samanta**: in Brown
3. **Sri Pavan Sesha Sai Rallapalli**: in Purple
4. **Equally Responsible**: in Black
5. **Literature Search**

We spent the first section of the project working on a Literature search keeping the following parameters in mind:

1. The paper must have already existing code or very specific parameters defined
2. The libraries in the code base must be as recent as possible (projects with outdated libraries were rejected eventually)
3. Preferably work on Binary files or Source Code, should be interchangeable. This led us to reject many approaches that visualized Malware binaries into images and then classified, as this would prove difficult for using the same approach for Vulnerability source code files
4. Link to the comparison table of our literature search: [Vulnerability and Malware - Google Tabellen](https://docs.google.com/spreadsheets/d/18S8JuEuAODwQkR4lEggQR33j6HUOi8ETXkKnT6neS9A/edit#gid=0)
5. **Malware detection model**
   1. **Introduction**

In the ever-evolving landscape of cybersecurity, the detection of malware within binary files and the identification of vulnerabilities are paramount challenges. To address these issues, we present a cutting-edge solution—an adaptable model that leverages the power of Holographic Reduced Representations (HRRs) and excels in cross-domain transfer learning. Originating as a formidable contender in malware detection, the HRRformer achieves near state-of-the-art accuracy at remarkable speeds. What sets it apart is its seamless extension into the realm of vulnerabilities—a starkly different domain. Through meticulous training, it transcends its origins, showcasing its remarkable ability to generalize across distinct data domains. This project introduces the HRRformer, a deep learning model tailored for the detection of both malware and vulnerabilities, offering a versatile and efficient tool for addressing critical cybersecurity challenges.

* 1. **Repository Structure**

The repository has the following structure:

**23ss-BinaryML-main**  
├── **MalwareDetectionHRR**

│ ├── binary\_loader.py ── Handles loading and preprocessing of binary data for the model.  
│ ├── dataset.py ── Provides functions to create data loaders for training, validation, and testing.  
│ ├── hrrformer\_mgpu.py ── Contains the transformer model implementation and prediction code.

│ ├── initializations.py ── Initializes data paths, flags, and settings for the model.

│ ├── malware\_utils.py ── Defines utility functions for model training, evaluation, and serialization.

│ ├── README.md ── Documentation and information about the project and its usage.  
│ ├── **weights** ── Directory containing pre-trained model weights for different datasets.  
│ │ ├── **vulnerability\_dataset**  
│ │ │ ├── hrrformer\_multi\_1\_<seq-len>.csv  
│ │ │ ├── hrrformer\_multi\_1\_<seq-len>.h5

│ │ ├── **malware\_dataset**  
│ │ │ ├── hrrformer\_multi\_1\_<seq-len>.csv  
│ │ │ ├── hrrformer\_multi\_1\_<seq-len>.h5

**Essential Paths:**

To work effectively with this codebase, you should be aware of essential paths defined in initializations.py. These paths specify the locations of training, validation, and test data, as well as the base directory for storing model weights. Make sure to update these paths according to your dataset and file structure.

* 1. **Main Pipeline**

The hrrformer\_mgpu.py file serves as the primary script, responsible for model training, testing, and predictions.

* Usage: `python hrrformer\_mgpu.py <malware/vulnerability> <train/test/predict>`
  1. **Project Roadmap**
* Initial Research: Researched 15 papers on malware detection, selected top 10, and filtered to 3 projects based on published year, code and data availability. Link for the comparison matrix is provided in preliminaries.
* Project Setup Challenges: However, numerous setup issues were encountered, preventing successful execution of any of the three projects.
* Transformer Implementation: Studied the "Recasting Self Attention" paper, crafted a basic transformer model from scratch before the code for the paper became publicly available, enabling us to leverage it for our project.
* **Challenges:**
* Setup Issues: Struggled with setup including difficulties in installing essential packages, discovered code limitations that could only be setup on a Linux.
* Data Loading: Data loader issues with paper's Ember dataset, adapted script, and faced compatibility issues.
* **Cross-Domain Transfer Exploration**: Explored the feasibility of code transfer for vulnerability detection. Extensive research was conducted to identify a suitable vulnerability dataset that aligned with their project goals. Multiple datasets discovered and discarded. Finally, landed upon the right dataset with state-of-the-art results.
  1. **Key Components**
     1. **Classes** (All of them present inhrrformer\_mgpu.py except for **BinaryDataset**)
* **MHAttention**: This class defines a multi-head attention mechanism. It's used within the `**Encoder**` class to perform attention calculations.
* **FeedForwardLayer**: This class defines a feedforward layer used within the `Encoder` class. It performs feedforward transformations on the input data.
* **Encoder**: This class represents the encoder layer of the Transformer model. It uses multi-head attention and feedforward layers for processing sequences.
* **Embedding**: This class defines the embedding layer that combines word embeddings and positional embeddings for input sequences.
* **Network**: This class defines the main neural network model that stacks multiple encoder layers to process sequences. It includes methods for training, testing, and making predictions.
* **BinaryDataset**: This class represents a loader for binary files which is present in binary\_loader.py. It loads and stores binary files along with their labels and sizes. Files can be sorted by size if needed.
  + 1. **Functions** inhrrformer\_mgpu.py
* **predict\_ind ()**: This function is used for individual predictions on a single data point. It computes the loss and accuracy for a single input.
* **individual\_predict ()**: This function is used for individual file predictions. It loads a single test file and predicts its class. Location: Defined in the script, used for individual predictions.
* **train\_step ()**: This function represents a single training step in the model. It computes the loss and gradients for a batch of data and updates the model's parameters.
* **Test ()**: This function is used to evaluate the model's performance on a test dataset. It computes test loss and accuracy.
  + 1. **Enhancement- Individual Predictions**

Designed the code to test and analyse individual prediction on Transformer-based model, primarily for malware detection.

Functionality introduced in our individual prediction function:

* Predicted the nature (benign or malicious) of individual files.
* Set up the model and loads pre-trained weights.
* Iterated through individual files in "good" and "bad" directories.
* Predicted each file’s nature using the model.
* Recorded whether each prediction is "Correct" or "Wrong."
* Printed the prediction results for each file, indicating if the model's prediction matches the true nature of the file.

The code leverages various libraries and custom utility functions for deep learning, data processing, and evaluation. It's a comprehensive tool for analysing predictions of machine learning models for malware detection.

* + 1. **Predictive analysis based on individual predictions:**

Interpretation of prediction results on 5496 samples:

* The model demonstrates a reasonably good balance between precision (78.87%) and recall (89.76%), as indicated by the F1-Score (84.01%). This means it's capable of making accurate predictions while maintaining a good ability to capture actual vulnerabilities.
* The high recall value indicates that the model effectively identifies most vulnerable files, reducing the risk of missing actual vulnerabilities.

Key insights per CWE:

* CWE546 stands out with the highest percentage of incorrect predictions (58.62%) indicating a critical area for model improvement.
* CWE789 also has a notable percentage of incorrect predictions (44.64%).
  + 1. **Training Challenges**

The project is designed to specifically run on Linux. Though we would have preferred to run our final model on a GPU, we faced issues with JAX and CUDA compatibility issues. In the interest of time, we settled for training this one on CPU.

* 1. **Future Extensions:**
* **Feature Engineering:** Enhance the model's ability to detect malware by creating new features from raw data sources (e.g., file headers, API calls, byte patterns).
* **Model Architecture:** Customize neural network architectures (e.g., CNNs, RNNs) to improve malware detection. Adjust depth, width, and complexity based on your dataset and resources.
* **Data Augmentation:** Implement data augmentation techniques to increase the diversity of training samples, which can enhance the model's robustness.
* **Hyperparameter Tuning:** Optimize model performance by fine-tuning hyperparameters like learning rate, batch size, dropout rates, and weight decay.

1. **Vulnerability Detection Model**
   1. **Romeo: Vulnerability Detection Model**
      1. **Project Roadmap**

* **Initial Research:** Researched 10 potential papers, initially narrowed it down to 2 potential projects: Illuminati and VulDeePecker. Decided on Illuminati due to VulDeePecker having unavailable Libraries
* **Finding ROMEO:** Reinitiated search for a better research work on our topic. ROMEO was published much after our initial literature search, so we found it later. Research and verification for its suitability for our project was evaluated. It was on latest technology and packages and is coherent to our requirements.
* **Final Project Decision:** Illuminati was explored for a significant time; however, it was dropped in the end as we found ROMEO. Illuminati used PYG 2.0.0 which led us to have to “reinvent the wheel” for many aspects, and as it didn’t use binaries for source code, we collectively made the decision to prefer ROMEO. However, we have uploaded our initial work on Illuminati in the GitHub for any potential future exploration.
* **Cross-Domain Transfer Exploration**: Explored the feasibility of code transfer for Malware Detection.
* **Challenges:**
* Needing to Drop Illuminati: We specify our progress in this paper later on in the document.
* Setup Issues: Romeo Model was quite large with initial configurations, needed to change batch sizes and dataset sizes to deal with CUDA out of memory and Long Runtime errors
* Data Loading: The entire malware dataset proved to be too large for our training system, so we settled for 50000 randomly chosen from over 100000’s files.

**3.1.2 Introduction Romeo:**

Diving into cybersecurity through advanced vulnerability detection, our journey led us from traditional source code to the intriguing world of binary artifacts. We've shifted from source code to binary for vulnerability detection, harnessing binary's unique benefits—realistic code execution, compatibility with dynamic analysis, and closed-source software assessment. ROMEO, our project, explores assembly language's potential versus traditional source code for vulnerability detection. This emphasizes the value of detecting vulnerabilities in compiled programs and binary's advantages over source code.

Simultaneously, we've ventured into applying Vulnerability Detection Models to malware datasets. Our research developed a custom data loader tailored to malware dataset intricacies. This loader seamlessly integrates with ROMEO, enabling advanced vulnerability detection and malware vulnerability identification. Our goal is to strengthen vulnerability detection models and enhance cybersecurity.

**3.1.3 Repository Structure**

* pipelines/**:** Directory for defining and orchestrating data processing.
* intermediate/: Directory for storing saved models and test and train logs, after each training run.
* logs/: Location for logging training and evaluation information.
* python/: Source code directory containing the main project files.
* README.md: Project overview and instructions.
* requirements.txt: List of required Python packages.

**Essential paths:**

To work effectively with this codebase, you should be aware of essential paths defined in 23ss-BinaryML/VulnerabilityDetectionRomeo/python/juliet\_access/\_\_init\_\_.py. These paths specify the locations of training, validation, and test data, as well as the base directory for storing model weights. Make sure to update these paths according to your dataset and file structure.

The classifier specified in VulnerabilityDetectionRomeo/python/classifier/\_\_init\_\_.py here is where we suggest any changes in batch sizes be done to tackle any CUDA out of Memory issue.

**3.1.4 Main Pipeline**

The python/run\_pipeline.py file serves as the primary script, responsible for running model training, testing, and predictions. Include the --malware flag if you are working with the cross-domain malware model.

* Usage: `python -m run\_pipeline ../pipelines/fulldataset.yaml fulldataset <--malware> `

**3.1.5 Key components of the pipeline**

1. **Preparing Juliet Suite for Extraction:**

* Classes used: EnumerateCWEsPipelineStep and EnumerateTestcasesPipelineStep
* Modifications: Removed empty functions (good1-good9 and bad1-bad9) from io.c support file in Juliet test suite.
* Compilation and Linking: Individually compiled Juliet testcases with GCC 11.2.0, creating single object files per unit, and linked them with io.c support, resulting in 41,812 object files covering 91 CWEs, excluding Windows-specific cases.

1. **Obtaining Binaries:**

* Classes used: LabelStrategy, FilterTestcasesPipelineStep and FilterCWEsPipelineStep
* Compilation to Machine Code: Transformed C/C++ sources into machine code binaries.
* Disassembly: Convert binaries into Intel syntax assembly code with addresses and symbols.

1. **Extracting Examples:**
   * Classes used: ExtractExamplesPipelineStep and LabelExamplesPipelineStep
   * Symbol Representation: Create symbol scrambling tables to prevent label leakage.
   * Function Selection and Representation: Convert object file functions into text with unique markers.
   * Context Inclusion: Include referenced functions recursively, filtering based on label definitions.
2. **Transformer-based Model for Vulnerability Detection:**
   * Classes used: TrainTokenizerPipelineStep and TrainClassifierPipelineStep
   * Utilized a deep learning model, "Transformer," for vulnerability detection, implemented using PyTorch and HuggingFace Transformers.
   * Initialized with "CodeBERT," a pretrained model trained on extensive code data.
   * Customized tokenization and encoding using "byte-pair encoding" optimized for the ROMEO dataset and efficient representation of assembly mnemonics.
3. **Prediction**

* Classes used: EvaluatePredictionsPipelineStep and PrintEvalulationTablePipelineStep
* Applied the trained model to make predictions on new data, identifying potential vulnerabilities.
* Evaluated the model's performance using various metrics, including accuracy, precision, recall, F1-score, and confusion matrices.
* Generated structured tables summarizing the model's performance metrics for easy analysis and interpretation.

**3.1.6 Research Specifics**

**For Building our Data Loader:**

* **Dataset Understanding:** We began by thoroughly analysing a unique malware dataset. Its size, diverse malware types, and varied file formats posed distinctive challenges.
* **Custom Data Loader:** To tackle these challenges, we developed a specialized data loader. This loader efficiently handles diverse and large files, preparing the data for ROMEO vulnerability detection.
* **Seamless Integration:** Our custom data loader seamlessly integrates with the ROMEO model, unlocking the full potential of the malware dataset. This integration enhances the model's accuracy and efficiency in identifying vulnerabilities and malware.
* **Performance Validation:** Throughout our research, we rigorously validated the loader's performance. Through experiments and fine-tuning, we ensured optimal results. We used multiple combinations of batch sizes and epochs to get a result.

In conclusion, our efforts have significantly advanced the ROMEO vulnerability detection model. Our custom data loader empowers ROMEO to effectively detect malware vulnerabilities, reinforcing cybersecurity practices and defences against malware threats.

**3.1.7 Creating Data loader for Malware dataset:**

Initially built a data loader for malware dataset for cross domain predictions (code pushed to branch: `custom\_data\_loader\_malware\_ROMEO`)

* It reads binary files, labels them as 'good' or 'bad', and splits them into training, validation, and test sets. Initialized a data structure with information about the dataset, including features of malware data, label encoding.
* For each loaded binary file, it generated file paths, labels, and sizes.

Our updated data loader is as follows.

To incorporate the data loading for malware data set, we updated the data loader as follows:

* The `is\_malware` parameter is a crucial addition to the data loader, designed to enable flexibility in loading and processing datasets for either malware or vulnerability detection.
* `enumerate\_cwes\_malware()` and `enumerate\_testcases\_malware()` functions are integral to the data loader's ability to handle malware-specific datasets effectively. They provide the necessary flexibility and structure for loading and processing data in the context of malware detection, enhancing the overall functionality of the data loader.
* `extract\_examples\_malware()` is responsible for converting malware test cases into structured examples suitable for machine learning.
  1. Disassembly: The function begins by disassembling the malware code contained within the test case. It extracts information about the code, including its sections, functions, and text representations.
  2. Labelling: The function assigns labels to the examples based on the presence of functions in specific categories. Examples are labelled as "Good" or "Bad" accordingly.
  3. Example Structuring: The resulting examples are structured as dictionaries, including the text representation of the code, information about the associated test case, and the assigned label.

Overall, it disassembles and structures the code, ensuring that it is ready for subsequent stages of data processing, feature extraction, and model training. It provides a structured representation of the malware data that can be used effectively in machine learning models for malware detection.

**3.1.8 Predictive analysis**

We have analysed the predictions of each file in the training, validation and test run. Below given are the average statistics of failed and successful predictions.

|  |  |
| --- | --- |
| **Vulnerability detection:** | **Cross domain detection:** |
| FalseNegativeRate: 0.13277034951911798 FalsePositiveRate: 0.025675891857302832  TrueNegativeRate: 0.9743241081426972 TruePositiveRate: 0.867229650480882 | FalseNegativeRate: 0.05368035786905246 FalsePositiveRate: 0.043700787401574806  TrueNegativeRate: 0.9562992125984252 TruePositiveRate: 0.9463196421309475 |

It can be concluded that the model predicts more results correctly and very few incorrectly. Hence, the model displays a great accuracy as well.

**3.1.9 Future Extensions**

* **Custom Dataset Loading:** Customized dataset loading by modifying `EnumerateCWEsPipelineStep` and `EnumerateTestcasesPipelineStep`. Implement custom loading functions compatible with another dataset.
* **Filter Testcases:** Use of `FilterTestcasesPipelineStep` to filter test cases based on specific criteria, like object files. Modifying the `filter\_testcases` function to include custom filtering logic.
* **Tokenization and Model Configuration:** Tailoring the tokenizer and model during training by adjusting `TrainTokenizerPipelineStep` and `TrainClassifierPipelineStep`.
* **Evaluation and Metrics:** Customizing evaluation in `EvaluatePredictionsPipelineStep`. Adjusting metrics and criteria by modifying the `\_evaluate\_subset` function to suit malware/vulnerability detection requirements.
* **Memory Optimization:** We faced numerous memory issues while trying to train this dataset. We suggest some future code optimizations where more memory is released throughout the process.

**3.2 Illuminati: Vulnerability Detection Model**

**3.2.1 Introduction Illuminati**

For our Vulnerability to Malware Detection repository we had initially settled for a Paper titled ILLUMINATI: Towards Explaining Graph Neural Networks for Cybersecurity Analysis. This paper seemed promising in the beginning due to it being relatively recent and official code being available online. It focuses on a comprehensive and accurate explanation framework for cybersecurity applications using GNN models.

**3.2.2 Repository Structure**

**The project has two main parts: The Graphical Dataset Classifier and the Node based Classifier.**

* **Datasets:**This contains the original load\_datasets.py that is accessed while running parts of the repository. It also holds the raw and processed versions of the pytorch datasets when loaded.
* **configs.json**: This is the config file that controls the project. The graph and node datasets are defined here
* train\_{TASK}.py - Task here is either Node / Graph. These are two separate files, the train\_node.py file is a separate classifier for node training, and train\_graph.py is a classifier for graph training. Saves results of validation and testing.
* explain\_{TASK}.py - similar to the previous files, these are two different files. It explains the entire dataset used.
* evaluate\_{TASK}.py - This is the final aim of the paper, which consists of Essentialness Percentage (EP) and probability reduction for explained subgraphs and remaining subgraphs.
* models.py: The main classifier GCN, imported into the previously mentioned train files.
* explainer.py: Explainer functions, imported into the previously mentioned explain files
* PGMexplainer.py and PGExplainer: Explainer functions, imported into the previously mentioned explain files
* README : Instructions on how to run the project. does mention pytorch geometric is used for the project. However no specific versions or environment specified.

**3.2.3 Dataloader:**

The data loaders for illuminati were defined to be different based on the dataset specified in config.json. The project itself uses multiple datasets to train, not just Vulnerability datasets. Most of these were directly loaded from pytorch geometric 2.0.0 ‘s datasets. There were a few custom dataloaders taken from DiveIntoGraphs.

We tried implementing a new data loader for loading EMBER Vulnerability dataset. We explored BBBP dataset, which tested the accuracy of model only, without making any predictions. Inclusion of EMBER dataset for our data loader introduced us to some challenges. Integrating your EMBER dataset loader with the model of Illuminati introduced additional challenges and considerations.

* Model Compatibility: Illuminati had specific requirements and expectations regarding the format and structure of input data.
* Feature Engineering: EMBER dataset features were predefined in .json files.
* Data Augmentation: Incorporating these into data loader's preprocessing pipeline was researched upon to fit the parts coherently.

While we were pursuing porting this project for Malware detection, we considered using the MalNetTiny dataset from pytorch geometric’s later dataset libraries. This library is however not available in pytorch geometric’s 2.0.0 so we had to add the loader separately instead of importing from libraries.

**3.2.4 Challenges:**

There were various challenges while working with the Illuminati project, which led us to eventually drop it. However the bit of progress we achieved we have also documented here.

* The readme never specified the version of pytorch geometric used, after contacting the original authors we learned that it uses PYG 2.0.0. Using later versions would lead to a large influx of errors of unidentified functions.
* Though the paper documents their training on Vulnerability Library Juliet, the code base does not consist of the data loader that trains on it.
* We initially attempted to use the graphical library MalNetTiny for training, based on later versions of pyg’s datasets we recreated the data loader for this. However for further calculations of data.x for MalNetTiny we felt as if we were reinventing the wheel for multiple functions.
* Finally, Illuminati’s vulnerability training is on Source code, however not binaries, and we eventually felt ROMEO was a better fit for our comparison metrics.

**3.2.5 Future Extensions**

* Illuminati originally used custom loader functions from the Dive into Graphs’ library to make up for pyg 2.0.0 lack of data processing functions. We had begun a similar approach for the data loader for MalNetTiny.
* As illuminati is designed to explain any dataset it trains, more data loaders with different structured datasets would lead to more defined results.