SCALABLE APT MALWARE CLASSIFICATION VIA PARALLEL FEATURE EXTRACTION AND GPU-ACCELERATED LEARNING

● **Noah Subedar**
Master of Cybersecurity and Threat Intelligence
University of Guelph
HBSc, Computer Science Specialist
University of Toronto
nsubedar@uoguelph.ca

● Taeui Kim
Master of Cybersecurity and Threat Intelligence
University of Guelph
BSc, Software Engineering
University of New Brunswick
taeui@uoguelph.ca

● Saathwick Venkataramalingam
Master of Cybersecurity and Threat Intelligence
University of Guelph
Post-Graduate Certificate in Offensive Cyber Security
York University
venkatas@uoguelph.ca

ABSTRACT

This paper presents an underlying framework for both automating and accelerating malware classification, more specifically, mapping malicious executables to known Advanced Persistent Threat (APT) groups. The main feature of this analysis is the assembly-level instructions present in executables which are also known as opcodes. The collection of such opcodes on many malicious samples is a lengthy process; hence, open-source reverse engineering tools are used in tandem with scripts that leverage parallel computing to analyze multiple files at once. Traditional and deep learning models are applied to create models capable of classifying malware samples. One-gram and two-gram datasets are constructed and used to train models such as SVM, KNN, and Decision Tree; however, they struggle to provide adequate results without relying on metadata to support n-gram sequences. The computational limitations of such models are overcome with convolutional neural networks (CNNs) and heavily accelerated using graphical compute unit (GPU) resources.

Keywords Malware Classification · Automation · Parallel Processing · Machine Learning · Neural Network Models · GPU-Accelerated Computing

1 Introduction

The rapid evolution of malware, particularly those associated with APT groups, present a significant challenge to cybersecurity researchers and analysts [1, 2]. Traditional signature-based detection techniques often fail to recognize novel or obfuscated threats, necessitating more robust, data-driven approaches to malware classification. One promising avenue involves the analysis of opcode sequences which represent low-level machine instructions extracted from malicious binaries. These sequences capture behavioural patterns that are difficult to mask, even under obfuscation or packing; however, extracting and analyzing large-scale opcode datasets remains computationally intensive and prone to bottlenecks when conducted sequentially or with manual intervention [3, 4].

To address these limitations, this study introduces a fully automated, parallelized pipeline for malware classification based on opcode analysis, encompassing traditional machine learning and deep learning techniques. By leveraging open-source tools like Ghidra in headless mode, combined with Python scripts, thousands of malicious executables are efficiently decompiled and transformed into structured opcode datasets [5, 6]. These are further processed into n-gram representations and fed into models such as Support Vector Machines (SVM), K-Nearest Neighbors (KNN),

Decision Trees, and GPU-accelerated Convolutional Neural Networks (CNNs) [7]. This comprehensive approach not only streamlines the data extraction and training process but also highlights the performance trade-offs across various classifiers, metadata usage, and dataset configurations including one-to-one and one-to-many label mappings [8–10].

2 Scripts & Environment Setup

2.1 Container Environment

To facilitate ease of use and reproducibility, a Podman container comprising the scripts and tools discussed in this paper has been published on Docker Hub. It is recommended to use Podman along with a NVIDIA GPU to run the container with hardware acceleration. The following instructions demonstrate how to run the container on a Fedora Linux system which includes Podman by default. They also outline the steps required to enable support for NVIDIA GPUs, assuming the necessary drivers have been installed correctly.

2.1.1 Enabling NVIDIA GPU Support

To enable the container to interface with NVIDIA GPUs, the NVIDIA Container Toolkit must be installed, and a Container Device Interface (CDI) specification file must be generated to expose the GPUs to the container environment [11].

```
sudo dnf install nvidia-container-toolkit
sudo nvidia-ctk cdi generate --output=/etc/cdi/nvidia.yaml
```

Fedora Linux enables Security-Enhanced Linux (SELinux) by default which restricts containers from accessing host devices for security reasons. To permit containers to use such devices, SELinux must be configured accordingly using the following command [11]:

```
sudo setsebool -P container_use_devices true
```

2.1.2 Data Format

Before running the container, it is crucial that the data provided to it is formatted correctly. The scripts discussed in this paper require a specific file structure and naming scheme. The organizational structure is outlined below: the left side shows the required directory structure, and the right side presents an example of the data formatted correctly.

```
data                                    data
|-- APT Group                           |-- G0001
    |-- Software Name                   |   |-- Software A
        |-- Malicious Executable        |   |   |-- G0001_Malware1.exe
                                        |   |   '-- G0001_Malware2.dll
                                        |   '-- Software B
                                        |       '-- G0001_Malware3.exe
                                        |-- G0002
                                        |   |-- G0002_Malware4.exe
                                        |   '-- Software C
                                        |       '-- G0002_Malware5.exe
                                        '-- G0003
                                            '-- G0003_Malware6.exe
```

2.1.3 Running the Container

To run the container, first pull it from Docker Hub and execute it in interactive mode with a bash shell. Within the container, the shell script named `run.sh` can be run to automatically execute all methods outlined in the paper with results saved to `/app/results` directory within the container. By default, all scripts are executed with the arguments specified in the paper; however, these can be customized by modifying the environment variables, editing `run.sh` or manually running the individual scripts. To provide malware samples to the container, mount a host directory containing the structure defined above to the container's `/data` directory.

```
# Host
podman run --rm -it \
  --device nvidia.com/gpu=all \
```

```
  -v /path/to/host/directory:/data:z \
  docker.io/noahsub/scalable-apt-malware-classification

# Container
./run.sh
```

2.2 Scripts

Although the container offers an automated method for analyzing malware samples, the scripts are also available on the paper's associated GitHub page [12], specifically in the releases section, along with the collected malware samples. The functionality and usage of each script are defined below.

2.2.1 Ghidra Manager

The `ghidra_manager.py` script is responsible for managing the extraction of opcodes from malicious executables using Ghidra headless mode.

```
python3 ghidra_manager.py \
    --GHIDRA <path_to_Ghidra_headless_analyzer> \
    --DIRECTORY <path_to_malware_root_folder> \
    --THREADS <number_of_threads> \
    --SKIP <true|false> \
    --TIMEOUT 1200
```

Table 1: Ghidra Manager Command Line Flags Description

Flag	Description	Required	Default Value
–GHIDRA	The path to Ghidra's headless analyzer.	Yes	N/A
–DIRECTORY	The path to the directory containing malware.	Yes	N/A
–THREADS	The number of threads to use, i.e., how much malware to analyze at once.	No	4
–SKIP	If a malicious executable already has had its opcodes extracted then skip analyzing it.	No	No
–TIMEOUT	The maximum amount of time given to analyze a single malicious executable.	No	1200

2.2.2 Preprocess

The `preprocess.py` script creates an optimized dataset suitable for machine learning models using opcodes extracted from collected malware samples.

```
python3 preprocess.py \
    --opcodes <path_to_directory_containing_opcodes> \
    -n 2 \
    --percentiles <percentile_1>,<percentile_n>
```

Table 2: Preprocess Command Line Flags Description

Argument	Usage	Required	Default Value
-opcodes	Specifies the path to the directory containing the .opcode files.	Yes	N/A
-n	Sets the maximum n-gram size	Yes	2
-percentiles	Accepts a comma-separated list of percentiles.	Yes	N/A

2.2.3 Classifier

The `classifier.py` script classifies malicious samples by using opcodes and metadata with three types of classifiers: SVM, KNN, and Decision Tree.

```
python3 classifier.py --dataset <path_to_csv_or_pkl_dataset>
```

Table 3: Classifier Command Line Flags Description

Argument	Usage	Required
-dataset	The dataset to use for training and performance evaluation. Can be in `.csv` or `.plk` (serialized pandas dataframe) format.	Yes

2.2.4 CNN Preprocess

The `preprocess-cnn.py` script intelligently removes samples from the dataset that cause one-to-many mappings.

```
python3 preprocess.py --dataset <dataset_name>
```

Table 4: CNN Preprocessing Command Line Flags Description

Argument	Usage
-dataset	The path to the dataset of opcodes.

2.2.5 CNN Model

The `model.py` script trains a CNN model to classify malware based on opcode sequences inspired by the paper *Deep Android Malware Detection* [13].

```
python3 model.py \
    --directory <data_directory> \
    --percentile <percentile_value> \
    --k <k_value> \
    --epochs <num_epochs> \
    --batch_size <batch_size> \
    --validation_split <validation_fraction>
```

Table 5: Model Script Arguments

Argument	Usage	Required	Default Value
-directory	The directory containing the opcode files.	Yes	N/A
-percentile	The percentile to determine the maximum sequence length.	No	50
-k	The output dimension of the embedding layer.	No	8
-epochs	The number of epochs to train the model.	No	16
-batch_size	The batch size to train the model.	No	32
-validation_split	The validation split to use during training.	No	0.1

3   Methodology: Opcode Extraction

The extraction of machine-level instructions (opcodes) from malicious executables were automated using Ghidra's headless mode in conjunction with Python scripting. This approach optimizes the creation and management of Ghidra projects while enabling concurrent opcode extraction through multiple processes, significantly enhancing efficiency in malware analysis and reverse engineering.

3.1   Sample Collection

Malicious executable samples linked to various APT groups and their associated software were gathered from multiple malware databases. The collected samples have been made available through the paper's GitHub repository [12].

3.2   Engine Selection

Before extracting the opcodes from collected malicious executable samples, the engine to be used as the base for the scripts had to be determined. Various options were explored including IDA Pro, Ghidra, and Binary Ninja. Each option had its advantages and disadvantages. Both IDA Pro and Binary Ninja were relatively easy to work with with Binary Ninja being heavily focused on scripting which made it ideal for the analysis of executable malware [14, 15]. Ultimately, Ghidra was selected due to its open-source nature and ability to operate in a headless environment [16].

3.3   Automated Scripts

Ghidra's headless analyzer allowed an executable and a `Python2` script to be passed in as arguments. This enabled Ghidra to automatically analyze the provided executable and then run the provided script [17]. In this case, the script's purpose was to extract opcodes.

The automation framework was structured into two primary components:

- **Opcode Extraction Component** – Analyzed each binary, retrieved machine-level instructions, and extracted the corresponding opcode sequences.
- **Manager Component** – Managed process spawning, file processing, and orchestrated Ghidra's functionality, ensuring seamless execution of the analysis workflow.

Given the large volume of malicious executables requiring analysis, efficiency was identified as a critical priority. To address this, parallel computing was leveraged, enabling simultaneous extraction of opcodes from multiple samples. This approach was essential, as the dataset comprised nearly ten gigabytes of executable malware—totaling 4,630 individual samples [18, 19].

The automated opcode extraction process followed these structured steps:

1. **Identification of Executables** – Located all executable files within the specified directory containing malware samples. This included malware in subdirectories of the specified directory.
2. **Project Initialization** – Created a separate Ghidra project for each binary to ensure an isolated and organized analysis environment.
3. **Binary Importation** – Loaded the malware executable into its corresponding Ghidra project.
4. **Opcode Extraction** – Executed `oopcode_extractor.py` within Ghidra's headless mode to extract machine-level opcodes.
5. **Storage and Organization** – Saved extracted opcode sequences in a designated output directory for further analysis.
6. **Performance Optimizations** – Implemented safeguards such as timeout handling and file tracking to enhance efficiency and prevent redundant processing.
7. **Cleanup Operations** – Removed unnecessary files to maintain a streamlined and organized workspace.

By automating opcode extraction using Ghidra and Python, this methodology improved scalability, reduced manual intervention, and enhanced the overall efficiency of malware analysis workflows.

3.4 Hardware Utilized

To run the scripts, an isolated Linux machine was utilized with the following configuration:

Table 6: System Hardware Specifications Used for Opcode Extraction

Hardware Type	Model	Specifications
CPU	AMD Ryzen 7 3700X	8 cores / 16 threads, Base Clock: 3.6GHz, Boost Clock: up to 4.4GHz
GPU	NVIDIA GeForce RTX 3060	12GB GDDR6 VRAM, 3584 CUDA cores, Boost Clock: 1.78GHz
Memory	DDR4 Non-ECC	16GB, 3200MHz
Operating System	Pop!_OS	Version 22.04 LTS (64-bit)

4 Methodology: N-Gram Dataset & Training

4.1 Preprocessing

To utilize the raw extracted opcodes, they first had to be preprocessed into a usable format. The preprocessing pipeline begins by recursively discovering all .opcode files in the provided directory. This process stores the relative path from the specified directory for each of the files which contains metadata such as the associated group and malware name; the metadata was based on the diligent organization structure utilized during opcode extraction [20, 21]. To ensure consistency in machine learning models, all opcodes were converted to uppercase, preventing variations in case from being misinterpreted as distinct instructions. Before generating n-gram sequences, the extracted opcode lines had to be adjusted to ensure their length was divisible by the selected n-gram size. This was achieved by padding the extracted opcode lines with "PAD" tokens until they reached the required length.

4.2 N-Gram Sequence Generation

The opcodes were then grouped into n-gram sequences:

- **1-Gram (Unigram)**: Each opcode was treated individually capturing its frequency across the sample.
- **2-Gram (Bigram)**: Pairs of consecutive opcodes were generated capturing local context and instruction transitions.

4.3 Vocabulary Generation

Although each opcode file contained a different set of instructions (unless they were the same malware), the datasets used for training and testing must contain each unique opcode or n-gram opcode pair so that they can store their frequencies across all files. To achieve this, all opcode files were iterated over the generated n-gram sequences which were then converted to sets to remove duplicates. This effectively allowed the creation of headers (features) for the dataset.

4.4 Dataset Construction

4.4.1 Extracting Metadata

In addition to opcode features stored in the dataset, relevant metadata was extracted to serve as labels. This metadata included the associated APT group, the name of the malware, and the type of the executable. The metadata was extracted from the relative path of the opcode file which contained such information. This step was carefully planned during the opcode extraction phase to ensure that metadata was retained even after converting the malware into `.opcode` files. If any given piece of metadata was missing for an opcode file, it was simply listed as unknown in the dataset.

4.4.2 Generating Feature Data

For each malware sample, a feature vector was created, where each feature represented the normalized frequency of a unique n-gram. This was achieved by counting the occurrences of a specific opcode instruction in the opcode file and

then dividing this count by the total number of opcode instructions in the file. The acquired data was then inserted into the dataset corresponding to the columns generated by the associated n-gram vocabulary.

4.5 Dataset Optimization

To improve the efficiency and computational performance of subsequent classification, the dataset was optimized by removing features with low variance as determined by a threshold based on predefined percentiles. This step ensured that only informative features, which provided meaningful distinctions, contributed to the model training process. The optimized dataset was then stored in `.csv` and `.pkl` formats and was ready for use in machine learning models. This optimization helped particularly with the 2-gram dataset, where the number of columns neared sixty thousand, which would have made training take far longer. Below, a table showing the sizes of the original and optimized datasets is provided. For the 1-gram dataset, the 10th percentile variance selection was used, while for the 2-gram dataset, the 80th percentile variance selection was used.

Table 7: Comparison of Dataset Dimensions Before and After Optimization

Dataset	Original Size (Rows × Features)	Optimized Size (Rows × Features)
1-Gram	1930 × 3658	1687 × 3658
2-Gram	59,276 × 3658	11,847 × 3685

4.6 Serialization

Serialization was extensively used to reduce computation time, trading off increased memory usage for faster performance. When handling large datasets, even seemingly quick computations could accumulate significant overhead, leading to inefficiencies. To avoid redundant recalculations, data was serialized using the pickle format, as it allowed easy serialization and deserialization of Python objects including Pandas DataFrames [22].

4.7 Classification

4.7.1 Classification Algorithms

We utilized three types of classification algorithms on the optimized dataset:

- **Support Vector Machine (SVM)**: SVM constructed an optimal hyperplane that maximized the margin between classes. For non-linear separability, kernel functions (such as Radial Basis Function (RBF) or polynomial kernels) were available to transform the feature space [23].

- **K-Nearest Neighbors (KNN)**: KNN classified a sample by analyzing the classes of its k-nearest neighbors (using Euclidean distance). Although the provided configuration used $k = 3$, the methodology could be adjusted based on empirical validation [24].

- **Decision Tree**: This classifier recursively partitioned the dataset based on feature thresholds, yielding an interpretable tree structure that highlighted the most significant opcode features influencing the decision process [25].

4.7.2 Classification Modes

To evaluate the influence of opcode data and associated metadata, three modes were adopted:

- **Single Mode**: Classification was performed solely on opcode features to predict a target label with metadata (such as APT group, malware name, and malware type) excluded.

- **Multi Mode**: The target label was predicted using opcode features supplemented by metadata from the remaining labels.

- **All Mode**: All label metadata were incorporated into the classification process with the opcode filename serving as the target.

For each mode, the classifiers were trained, and predictions were made on the same dataset so that performance metrics could be computed and compared. Since multiple classification algorithms and modes were used, it was determined that the most efficient approach was to create a specialized class to manage the results. The `ClassifierResult`

class encapsulated key performance metrics and stored essential details including the classifier type and classification mode. To ensure both human readability and seamless reusability in Python, the results were serialized in JSON format allowing for both human readability and further computer analysis.

The classification function trained and evaluated a machine learning model using a given dataset, classifier, mode, and target label. It first processed the dataset based on the specified mode (single, multi, or all); for instance, it converted label features to numerical values using ordinal encoding for modes that used metadata as features. Depending on the chosen classifier (SVM, KNN, or Decision Tree), the function trained the model and then made predictions to evaluate performance. The performance metrics included accuracy, recall, precision, F1-score, and confusion matrix. The results, including the trained model and computed metrics, were returned as a `ClassifierResult` object.

Once the model had been trained and evaluated using the classification function, the results were then visualized in the form of bar charts for accuracy, recall, precision, and F1-score. In terms of the confusion matrix, it was visualized using a heatmap.

4.8   Hardware Utilized

Given the large volume of data involved, a high-performance CPU-based system was utilized to minimize computation time.

Table 8: System Hardware Specifications Used for N-Gram Dataset and Training

Hardware Type	Model	Specifications
CPU 1	Intel Xeon E5-2697 v3	14 cores / 28 threads, Base Clock: 2.6GHz, Turbo: 3.6GHz, 35MB Cache
CPU 2	Intel Xeon E5-2697 v3	14 cores / 28 threads, Base Clock: 2.6GHz, Turbo: 3.6GHz, 35MB Cache
Memory	DDR4 RECC	256GB, 2133MHz
GPU 1	NVIDIA GeForce RTX 3060	12GB GDDR6 VRAM, 3584 CUDA cores, Boost Clock: 1.78GHz
GPU 2	NVIDIA GeForce RTX 2080 Super	8GB GDDR6 VRAM, 3072 CUDA cores, Boost Clock: 1.81GHz
Operating System	AlmaLinux	Version 9 (64-bit)

5   Methodology: Convolutional Neural Network

5.1   Preprocessing

As seen in the results for training using n-grams below, the confusion matrices indicated that traditional machine learning models struggle with one-to-many mapping, particularly when they lack metadata to rely on. In the context of malware analysis, these one-to-many relationships occur when the same malicious executable was associated with multiple APT groups. To examine how this impacted CNN performance, the opcode dataset was preprocessed to remove such overlaps, creating a subset of the dataset with strictly one-to-one mappings to reduce confusion. This filtering process resulted in a 63.75% reduction in opcode files from our dataset, though many of these were duplicates as the data was organized by APT groups. The reasoning was that carefully curating the dataset to remove ambiguous samples would lead to a more accurate model, even if it meant training on fewer files. Additionally, our initial dataset contained over 3,500 opcode files; therefore, filtering still left a considerable amount of data to work with.

Figure 1: One-to-Many Relationship Between Software and APT Groups

Figure 2: One-to-One Relationship Between Software and APT Groups

More specifically, the preprocessing script worked by creating a copy of the original dataset and systematically identifying and removing any duplicate opcode subdirectories pertaining to software belonging to one-to-many relationships. This process ensured that only unique opcodes remained, resulting in a clean, well-structured opcode database that respected a one-to-one mapping. Furthermore, opcode files were refined by stripping whitespace, removing empty lines, and converting opcodes to uppercase to ensure consistency. These processed sequences were then used to generate a structured dataset, where opcodes were grouped based on a selected target label (e.g., APT group, malware name, or malware type).

5.2 Generating Vocabulary

Following the methodology used in the paper titled *Deep Android Malware Detection*, the opcodes were encoded as one-hot vectors, where each opcode was represented by a binary vector with all values set to zero except for a single one at the index corresponding to that opcode. This process initially involved identifying all unique opcodes in the dataset and assigning each one an integer index to create a vocabulary. Additionally, labels had to be encoded to be compatible with the model which was done automatically using the label encoding functionality of scikit-learn [13].

Figure 3: Opcode Vocabulary Mapping

5.3   Implementing the CNN Model

The architecture was inspired by the model presented in the research paper *Deep Android Malware Detection* and was adapted specifically for this case. Unlike the original model, which focused on binary malware detection (benign vs. malicious), this implementation was tailored to classify malware according to specific target labels such as APT group, malware name, or malware type [13].

5.3.1   Embedding Layer

The embedding layer transformed encoded opcode sequences into dense vector representations making them suitable for processing by convolutional layers.

5.3.2   Convolutional Layers

In this implementation, two convolutional layers were applied after the embedding layer enabling the model to hierarchically learn and select important features. The first convolutional layer captured low-level patterns, while the deeper layer extracted higher-level semantic features that were crucial for malware classification. This automated feature extraction process enhanced the deep learning model's ability to distinguish effectively between different malware categories [26].

5.3.3   Pooling Layer

Pooling is a downsampling technique used in CNNs to reduce the spatial dimensions of feature maps while preserving the most important information. In this implementation, `MaxPooling1D` was used after each convolutional layer. The pool size of two reduced the sequence length by half, ensuring that only the most important features were retained while minimizing redundancy.

5.3.4   Flatten Layer

In this implementation, an additional layer called a flatten layer was added. This served to take the multi-dimensional data generated by the pooling layers and transform it back into one-dimensional data so that the fully connected layer could process the data.

5.3.5   Fully Connected (MLP) Layer

These models worked by forming neurons or nodes throughout each layer. The fully connected layer was responsible for creating connections between these layers combining all the learned features so they could be used to classify the input according to the associated labels. The `Dense` layer enhanced feature representation by introducing non-linearity, while the `Dropout` layer prevented overfitting by randomly deactivating neurons during training [27].

5.3.6 Softmax Classification Layer

The softmax classification layer was the final layer that converted the model's output into probabilities for each of our labels allowing for multi-class classification.

Figure 4: CNN Sequence

5.4 Training the Model

In order to train the model, the inputs had to be padded such that their lengths were uniform. To do this, the length of the sequences was selected based on a percentile, as simply choosing the length of the largest sequence would have created much unnecessary padding which could have interfered with the performance of the model. Additionally, malware labels were encoded into a categorical format. The CNN model was trained using the Adam optimizer which adjusted learning rates for faster learning, and categorical cross-entropy loss which was suited for multi-class classification. Together, they helped the model learn patterns in opcode sequences while reducing the chance of overfitting [28]. The model was trained using the following parameters:

Table 9: Parameters Used During CNN Training

Parameter	Value
k	8
percentile	50
epochs	16
batch_size	32
validation_split	0.1

5.5 Hardware Utilized

Given the large volume of data to be processed and the model's capability of leveraging GPU acceleration, the strongest available GPU compute device was selected for use.

Table 10: System Hardware Specifications Used for CNN Training

Hardware Type	Model	Specifications
CPU	AMD Ryzen 9 5950X	16 cores / 32 threads, Base Clock: 3.4GHz, Boost Clock: up to 4.9GHz
Memory	DDR4 Non-ECC	128GB, 3600MHz
GPU	NVIDIA RTX 4090	24GB GDDR6X VRAM, Boost Clock: up to 2.52GHz, 16384 CUDA cores
Operating System	Fedora	Version 41 (64-bit)

6 Results: Opcode Extraction

The script took a total of 18 hours to run was was set to analyze five items at a time. Without parallel processing, it was estimated that this process would have taken at least 90 hours. It should be noted that the large amount of time required was due to limitations in computer hardware as well as the number of malicious executables to be analyzed. Once the script had terminated, a total of 3789 .opcode files organized in the same structure as the original malware dataset remained. This number was lower than the total number of files because non-binary files such as Python scripts were included in the dataset but could not be decompiled by Ghidra, as they are not technically executables but executed with separate applications such as Python.

7 Results: N-Gram Dataset & Training

7.1 Performance Evaluation

To enhance understanding of the model's performance both bar charts and heat maps were used. These included all combinations of each mode against each classifier, analyzed for both 1-gram and 2-gram feature extraction techniques. All visualizations can be found in the Appendix of the paper.

7.2 Preprocessing

Generating the large datasets from the .opcode files took ten minutes to complete.

7.3 Training

Training the models on large datasets was a time-intensive process even after optimizing the datasets. In total, training all combinations of classifiers on the 1-gram dataset took 5.3 minutes, and on the 2-gram dataset it took 12.4 hours.

7.4 Metrics

Performance is quantitatively evaluated using:

- **Accuracy**: The proportion of correct predictions (both positive and negative) out of all predictions made [29].
- **Recall**: The ability of the model to correctly identify all positive instances while minimizing missed positives [29].
- **Precision**: The ability of the model to only predict positives when they are truly positive while minimizing false positives [29].
- **F-Measure**: The harmonic mean of precision and recall balancing the trade-off between them for overall performance [29].
- **Confusion Matrix**: A summary table showing the counts of true positives, true negatives, false positives, and false negatives revealing the types of errors made by the model [29].

This portion of the paper evaluates the performance of three machine learning (ML) classifiers—Support Vector Machine (SVM), K-Nearest Neighbors (KNN), and Decision Tree—for the classification of malware based on opcode sequences extracted from executables associated with APT groups. The classifiers are systematically assessed on their

ability to predict three primary targets: malware name, malware type, and APT group attribution, utilizing both 1-gram and 2-gram opcode feature representations to capture the sequential patterns of instruction codes.

7.5 Classifier Comparison

7.5.1 Decision Tree

The Decision Tree classifier consistently outperformed the other classifier models across both 1-gram and 2-gram datasets demonstrating the highest scores in accuracy, precision, recall, and F-Measure. Notably, the Decision Tree achieved remarkable performance in malware type and malware name classification attaining an accuracy of 99.69% with an F-Measure of 0.85 for malware type and 97.37% accuracy with an F-Measure of 0.88 for malware name on the 2-gram dataset using the multi-mode approach. Even when challenged by APT group attribution, it fell only to a moderate F-Measure of around 0.63 still outperforming other models.

7.5.2 KNN

KNN demonstrated moderate classification performance, producing acceptable results in both malware name and type classification task; however, it consistently lagged behind the Decision Tree classifier in terms of accuracy and F-Measure.

7.5.3 SVM

The SVM classifier demonstrated limited effectiveness across all classification tasks particularly in APT group attribution. The F-Measure for SVM consistently remained below 0.20, suggesting that SVM was not well-suited for capturing the subtle patterns in the opcode data making it less reliable for this task.

7.6 Mode Impact

7.6.1 Single Mode (Opcode Features Only)

When only opcode-derived features were used, all classifiers struggled to some extent, particularly for APT group attribution. The limited performance in this mode implied that opcode frequencies alone did not provide sufficient context to differentiate between groups especially for threat actors with overlapping characteristics.

7.6.2 Multi Mode (Opcode Features & Metadata)

Incorporating additional metadata significantly improved the classifiers' performance. This mode showed that adding contextual information such as the malware name and type, could compensate for the limitations of the opcode features. The Decision Tree, in particular, showed remarkable gains, indicating that the combination of opcode data and metadata provided a more complete feature set for accurate classification.

7.6.3 All Mode (All Labels for File Name Prediction)

In this mode, where all metadata was included and the file name served as the target, SVM and Decision Tree achieved similar performance (70% accuracy). This suggested that when more contextual labels were available, even a weaker classifier like SVM could catch up to a more robust model in terms of overall accuracy; however, KNN remained significantly behind highlighting its vulnerability to the complexity introduced by multiple label dependencies.

7.7 Raw Performance Data

7.7.1 1-Gram Performance Evaluation Table

A colour-coded table displaying the performance results of the 1-gram models is shown below:

Table 11: 1-Gram Classifier Performance

Classifier	Mode	Target	Accuracy	Recall	Precision	F-Measure
SVM	Single	Group	0.2797	0.0741	0.0728	0.0587
SVM	Single	Name	0.5295	0.0661	0.1039	0.0686
SVM	Single	Type	0.7438	0.1377	0.2419	0.143
KNN	Single	Group	0.4248	0.4789	0.5098	0.4028
KNN	Single	Name	0.8576	0.3693	0.3885	0.3612
KNN	Single	Type	0.9584	0.6202	0.6376	0.6164
Decision Tree	Single	Group	0.5197	0.6482	0.7194	0.5972
Decision Tree	Single	Name	0.9535	0.8925	0.8258	0.8415
Decision Tree	Single	Type	0.9896	0.9022	0.8318	0.8563
SVM	Multi	Group	0.3751	0.1405	0.0964	0.1099
SVM	Multi	Name	0.6107	0.0716	0.0631	0.0614
SVM	Multi	Type	0.6996	0.1007	0.2092	0.0936
KNN	Multi	Group	0.5082	0.508	0.5611	0.4458
KNN	Multi	Name	0.8666	0.4002	0.4072	0.3918
KNN	Multi	Type	0.9751	0.6852	0.7537	0.6977
Decision Tree	Multi	Group	0.5287	0.6925	0.761	0.6359
Decision Tree	Multi	Name	0.9738	0.9435	0.8718	0.8898
Decision Tree	Multi	Type	0.997	0.9066	0.833	0.8592
SVM	All	File Name	0.7094	0.7094	0.6603	0.6728
KNN	All	File Name	0.2586	0.2586	0.1208	0.1525
Decision Tree	All	File Name	0.7094	0.7094	0.6603	0.6728

7.7.2 2-Gram Performance Evaluation Table

A colour-coded table displaying the performance results of the 2-gram models is shown below:

Table 12: 2-Gram Classifier Performance

Classifier	Mode	Target	Accuracy	Recall	Precision	F-Measure
SVM	Single	Group	0.2914	0.0807	0.1599	0.0684
SVM	Single	Name	0.573	0.0757	0.1217	0.0791
SVM	Single	Type	0.7963	0.1686	0.3353	0.1907
KNN	Single	Group	0.4276	0.4755	0.5415	0.4118
KNN	Single	Name	0.8609	0.4023	0.4084	0.3907
KNN	Single	Type	0.962	0.6353	0.6675	0.639
Decision Tree	Single	Group	0.5197	0.6482	0.7194	0.5972
Decision Tree	Single	Name	0.9535	0.8925	0.8258	0.8415
Decision Tree	Single	Type	0.9896	0.9022	0.8318	0.8563
SVM	Multi	Group	0.3751	0.1405	0.0964	0.1099
SVM	Multi	Name	0.611	0.0716	0.0633	0.0615
SVM	Multi	Type	0.6996	0.1007	0.2092	0.0936
KNN	Multi	Group	0.5085	0.5059	0.5399	0.436
KNN	Multi	Name	0.8723	0.4281	0.4399	0.4192
KNN	Multi	Type	0.974	0.6956	0.7441	0.6939
Decision Tree	Multi	Group	0.5287	0.6925	0.761	0.6359
Decision Tree	Multi	Name	0.9738	0.9435	0.8718	0.8898
Decision Tree	Multi	Type	0.997	0.9066	0.833	0.8592
SVM	All	File Name	0.7094	0.7094	0.6603	0.6728
KNN	All	File Name	0.2542	0.2542	0.118	0.1492
Decision Tree	All	File Name	0.7094	0.7094	0.6603	0.6728

7.8 Performance Summary

From this, it was understood that Decision Trees excelled in opcode-based malware classification especially when enriched with metadata, while SVM underperformed and KNN showed moderate results. Nevertheless, accurately attributing APT groups remained challenging, indicating that additional contextual data was needed.

8 Results: Convolutional Neural Network

Experiments evaluated the CNN model on two distinct datasets: one with a one-to-one mapping (cleaned opcode sequences) and one with a one-to-many mapping (raw opcode sequences). For each dataset, the model was assessed on three target labels—APT group, malware name, and malware type—using standard performance metrics (accuracy, precision, recall, and F1-score).

Compared to traditional classifiers, the implementation of a deep learning model significantly accelerated the process of feature extraction and training by eliminating the need for manual feature selection. By leveraging GPU acceleration, training time was orders of magnitude faster enabling efficient model construction and optimization [27].

8.1 Preprocessing

Cleaning the dataset to create a one-to-one mapping took the system 1.53 minutes. This step drastically reduced confusion in the data particularly for the APT group target.

8.2 Training

The training process was highly resource-intensive with the workstation drawing approximately 600W of power, fully utilizing the GPU's 24GB of VRAM, and consuming 50GB of system memory. The substantial resource demand enabled the model to complete training in 3.65 minutes on all targets. This performance significantly outperformed the training times reported in *Deep Android Malware Detection*, highlighting the advantages of modern hardware in efficiently training complex models.

8.3 Dataset Comparison

As expected, the one-to-one dataset significantly outperformed the one-to-many dataset in results for the group target, with the one-to-many relationship achieving only 35% accuracy, while the one-to-one relationship achieved 92% accuracy. This improvement was attributed to the removal of ambiguity in the data allowing the CNN to accurately categorize samples into APT groups. Meanwhile, name and type classification showed relatively stable performance across both datasets achieving 84.64% and 89.80% in the one-to-one dataset compared to 85.69% and 96.50% in the one-to-many dataset. These labels typically did not involve overlapping mappings and were therefore less susceptible to the confusion that affected the group target.

8.3.1 One-To-Many Dataset

Table 13: CNN Performance With One-to-Many Dataset

Classifier	Mode	Target	Accuracy	Recall	Precision	F-Measure
CNN	Single	Group	0.35	0.2428	0.35	0.2335
CNN	Single	Name	0.8569	0.7985	0.8569	0.8177
CNN	Single	Type	0.965	0.9536	0.965	0.9588

Figure 5: Metric Comparison With One-To-Many Dataset

8.3.2 One-To-One Dataset

Table 14: CNN Performance With One-to-One Dataset

Classifier	Mode	Target	Accuracy	Recall	Precision	F-Measure
CNN	Single	Group	0.9215	0.9214	0.9215	0.9162
CNN	Single	Name	0.8464	0.828	0.8464	0.8188
CNN	Single	Type	0.898	0.8779	0.898	0.8748

Figure 6: Metric Comparison With One-To-One Dataset

16

9 Discussion: Opcode Extraction

This paper utilized open-source decompiler technologies and leveraged parallel computing to efficiently and effectively extract opcodes from malicious executables. The results demonstrated that, if scripts are designed to utilize and effectively manage hardware resources then opcode extraction of large datasets is achievable within a reasonable time frame. This opcode extraction provides a strong foundation for further malware analysis using classification algorithms and advanced machine learning techniques.

10 Discussion: N-Gram Dataset & Training

The results from the experiment demonstrated that the Decision Tree classifier consistently outperformed both SVM and KNN for malware classification using opcode features. In particular, when using 2-gram features along with metadata in a specified target (multi-mode) approach, the Decision Tree model achieved exceptional accuracy (up to 99.69% for malware type and 97.37% for malware name) and strong F-measures. While KNN produced moderate results, SVM struggled across all tasks with F-Measure values below 0.20 especially for APT group attribution.

Integrating metadata with opcode features significantly improved classifier performance compared to using opcode features alone; however, classifying APT groups remained a challenge, indicating that opcode patterns alone may not fully capture the nuances required for accurate threat actor identification. This difficulty stemmed primarily from the fact that multiple APT groups often used the same software which was included in the dataset to ensure realism. Consequently, it was expected that the models would struggle to accurately predict the specific APT group, as there was not a one-to-one mapping; several APT groups could be associated with the same set of malware. This was further demonstrated by inspecting the confusion matrix heatmaps in the Appendix associated with the group target label.

11 Discussion: Convolutional Neutral Networks

The results of the experiment showed that the CNN drastically outperformed traditional machine learning classifiers such as SVM, KNN, and Decision Tree when considering pure opcode analysis without the use of metadata.

11.1 Metric Comparison

When attempting to classify malware using traditional methods solely based on pure opcodes without the help of metadata, these models struggled considerably. This is not to say that the models themselves are flawed; rather, they were provided with lower-quality data compared to the CNN. First, the data fed to such models was not cleaned to the extent that the CNN datasets were, and numerous one-to-many relationships were present. In addition, the datasets were limited to 1-gram and 2-gram structures restricting each model's view to very short opcode windows. Extending n-grams beyond 2-gram was not feasible due to the immense resource overhead it would require, especially since moving from 1-gram to 2-gram alone already increased the feature space into the tens of thousands of features [26].

Regarding CNN performance, it vastly outperformed the traditional models across multiple targets. For APT group classification specifically, the CNN achieved a 77.65% higher accuracy compared to the best Decision Tree on uncleaned datasets (from 52% to 92%). In terms of precision and recall, the CNN consistently scored above 0.90 on one-to-one data for this target, whereas Decision Tree scores ranged between 0.64 and 0.72. For other labels such as name and type, the CNN similarly maintained a higher F1-score, in some cases surpassing 0.95, while the best traditional models hovered around the mid-0.80 range. These improvements highlight the importance of extended sequence analysis for malware detection, as CNNs are not limited to small, rigid n-gram windows.

In real-world scenarios, metadata such as file origin or known associations may not be accessible. Traditional models have shown that without metadata, their accuracy can drop to near chance levels for complex tasks like APT group attribution. By contrast, the CNN was able to learn from purely sequential opcode data showcasing its robustness and adaptability. This underscores the significance of leveraging models capable of capturing deeper, long-range dependencies—particularly in scenarios where additional context or metadata is unavailable [30].

11.2 Computational Comparison

Leveraging GPU acceleration for training the CNN models resulted in a substantial reduction in computational time compared to traditional classifiers. As detailed in the results section, the CNN completed training in mere minutes, although it consumed significant amounts of power, while the traditional models required a total of 12.4 hours to train. This contrast in training times was simply incomparable, especially considering that the CNN demonstrated superior

performance in pure opcode analysis further emphasizing the efficiency and effectiveness of deep learning approaches over conventional methods.

11.3 Concluding Remarks

This paper, along with the broader scope of the project, demonstrated that real-world malware collected from the internet could be effectively classified using only raw opcodes. The implementation of such machine learning methods holds significant potential for practical application in malware detection systems; however, training these models remains computationally demanding often limiting their accessibility to researchers and large organizations. Continued research is therefore essential to develop more efficient algorithms, and reduce the computational cost of training and deploying models in real-world environments.

Future work will explore other deep learning architectures particularly recurrent neural networks (RNNs) and transformers. While RNNs (such as LSTMs or GRUs) can learn sequence patterns with fewer resources, transformers excel at capturing long-range dependencies in a highly parallelizable manner. Moreover, future efforts will focus on incorporating multi-label classification techniques such as using a sigmoid output layer with threshold tuning or leveraging classifier chains to better address real-world scenarios where malware may overlap across multiple threat actor categories. By experimenting with these architectures and broader labeling strategies, the goal is to refine the balance between accuracy and computational overhead paving the way for broader deployment of advanced malware detection solutions.

References

[1] A. Yazdinejad, R. M. Parizi, A. Dehghantanha, Q. Zhang, and K.-K. R. Choo, "An energy-efficient sdn controller architecture for iot networks with blockchain-based security," *IEEE Transactions on Services Computing*, vol. 13, no. 4, pp. 625–638, 2020.

[2] A. Yazdinejad, R. M. Parizi, A. Dehghantanha, H. Karimipour, G. Srivastava, and M. Aledhari, "Enabling drones in the internet of things with decentralized blockchain-based security," *IEEE Internet of Things Journal*, vol. 8, no. 8, pp. 6406–6415, 2020.

[3] J. Sakhnini, H. Karimipour, A. Dehghantanha, A. Yazdinejad, T. R. Gadekallu, N. Victor, and A. Islam, "A generalizable deep neural network method for detecting attacks in industrial cyber-physical systems," *IEEE Systems Journal*, vol. 17, no. 4, pp. 5152–5160, 2023.

[4] A. Yazdinejad, R. M. Parizi, A. Dehghantanha, and K.-K. R. Choo, "P4-to-blockchain: A secure blockchain-enabled packet parser for software defined networking," *Computers & Security*, vol. 88, p. 101629, 2020.

[5] D. Namakshenas, A. Yazdinejad, A. Dehghantanha, and G. Srivastava, "Federated quantum-based privacy-preserving threat detection model for consumer internet of things," *IEEE Transactions on Consumer Electronics*, 2024.

[6] A. Yazdinejad, A. Dehghantanha, R. M. Parizi, G. Srivastava, and H. Karimipour, "Secure intelligent fuzzy blockchain framework: Effective threat detection in iot networks," *Computers in Industry*, vol. 144, p. 103801, 2023.

[7] A. Yazdinejad, A. Dehghantanha, R. M. Parizi, and G. Epiphaniou, "An optimized fuzzy deep learning model for data classification based on nsga-ii," *Neurocomputing*, vol. 522, pp. 116–128, 2023.

[8] B. Zolfaghari, A. Yazdinejad, A. Dehghantanha, J. Krzciok, and K. Bibak, "The dichotomy of cloud and iot: Cloud-assisted iot from a security perspective," *arXiv preprint arXiv:2207.01590*, 2022.

[9] A. Yazdinejad, A. Dehghantanha, R. M. Parizi, M. Hammoudeh, H. Karimipour, and G. Srivastava, "Block hunter: Federated learning for cyber threat hunting in blockchain-based iiot networks," *IEEE Transactions on Industrial Informatics*, vol. 18, no. 11, pp. 8356–8366, 2022.

[10] A. Yazdinejad, A. Bohlooli, and K. Jamshidi, "Efficient design and hardware implementation of the openflow v1. 3 switch on the virtex-6 fpga ml605," *The Journal of Supercomputing*, vol. 74, pp. 1299–1320, 2018.

[11] Podman Desktop Contributors, "GPU Container Access - Podman Desktop," https://podman-desktop.io/docs/podman/gpu, 2024, accessed: 2025-04-16.

[12] N. Subedar, T. Kim, and S. Venkataramalingam, "Scalable apt malware classification via parallel feature extraction and gpu-accelerated learning," 2025. [Online]. Available: https://github.com/chrkis7/cis6530-project

[13] N. McLaughlin, J. Martinez del Rincon, B. Kang, S. Yerima, P. Miller, S. Sezer, Y. Safaei, E. Trickel, Z. Zhao, A. Doupé, and G. Joon Ahn, "Deep android malware detection," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, ser. CODASPY '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 301–308. [Online]. Available: https://doi.org/10.1145/3029806.3029823

[14] A. Yazdinejad, R. M. Parizi, A. Bohlooli, A. Dehghantanha, and K.-K. R. Choo, "A high-performance framework for a network programmable packet processor using p4 and fpga," *Journal of Network and Computer Applications*, vol. 156, p. 102564, 2020.

[15] A. Yazdinejad, E. Rabieinejad, A. Dehghantanha, R. M. Parizi, and G. Srivastava, "A machine learning-based sdn controller framework for drone management," in *2021 IEEE Globecom Workshops (GC Wkshps)*. IEEE, 2021, pp. 1–6.

[16] R. Rohleder, "Hands-on ghidra - a tutorial about the software reverse engineering framework," in *Proceedings of the 3rd ACM Workshop on Software Protection*, 2019, pp. 77–78. [Online]. Available: https://doi.org/10.1145/3338503.3357725

[17] H. Markarian, "Function identification threats in embedded systems," 2023.

[18] A. Yazdinejad, A. Dehghantanha, and G. Srivastava, "Ap2fl: Auditable privacy-preserving federated learning framework for electronics in healthcare," *IEEE Transactions on Consumer Electronics*, 2023.

[19] A. Yazdinejad, A. Dehghantanha, G. Srivastava, H. Karimipour, and R. M. Parizi, "Hybrid privacy preserving federated learning against irregular users in next-generation internet of things," *Journal of Systems Architecture*, vol. 148, p. 103088, 2024.

[20] A. Yazdinejad, A. Dehghantanha, H. Karimipour, G. Srivastava, and R. M. Parizi, "A robust privacy-preserving federated learning model against model poisoning attacks," *IEEE Transactions on Information Forensics and Security*, 2024.

[21] A. Yazdinejad, "Secure and private ml-based cybersecurity framework for industrial internet of things (iiot)," Ph.D. dissertation, University of Guelph, 2024.

[22] H. Temiz, "Recording performances of some file types for pandas data," *Avrupa Bilim ve Teknoloji Dergisi*, no. 36, pp. 55–60, 2022.

[23] S. Suthaharan and S. Suthaharan, "Support vector machine," *Machine learning models and algorithms for big data classification: thinking with examples for effective learning*, pp. 207–235, 2016.

[24] L. E. Peterson, "K-nearest neighbor," *Scholarpedia*, vol. 4, no. 2, p. 1883, 2009.

[25] B. De Ville, "Decision trees," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 5, no. 6, pp. 448–455, 2013.

[26] D. Gibert, C. Mateu, and J. Planes, "The rise of machine learning for detection and classification of malware: Research developments, trends and challenges," *Journal of Network and Computer Applications*, vol. 122, pp. 1–21, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804519303868

[27] X. Li *et al.*, "An efficient convolutional neural network with transfer learning for malware classification," *Security and Communication Networks*, vol. 2022, 2022. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1155/2022/4841741

[28] S. Venkatraman *et al.*, "An opcode-based malware detection model using supervised learning algorithms," *International Journal of Information Security Research*, vol. 12, no. 1, pp. 1–12, 2022. [Online]. Available: https://www.igi-global.com/gateway/article/289818

[29] S. Swaminathan and B. R. Tantri, "Confusion matrix-based performance evaluation metrics," *African Journal of Biomedical Research*, vol. 27, pp. 4023–4031, 11 2024.

[30] J. Li *et al.*, "Data augmentation for opcode sequence based malware detection," https://pureadmin.qub.ac.uk/ws/portalfiles/portal/302309338/CRCI_2022_paper_17.pdf, 2022, conference on Recent Advances in Cybersecurity and Informatics (CRCI).

A   Appendix: 1-Gram Performance Visualizations

Classifier Performance Visualizations

Figure 7: SVM – Single Target

Figure 8: KNN – Single Target

Figure 9: Decision Tree – Single Target

Figure 10: SVM – Multi-Label

Figure 11: KNN – Multi-Label

Figure 12: Decision Tree – Multi-Label

Figure 13: SVM – All Labels

Figure 14: KNN – All Labels

Figure 15: Decision Tree – All Labels

Confusion Matrices

Figure 16: SVM – Group

Figure 17: SVM – Name

Figure 18: SVM – Type

Figure 19: KNN – Group

Figure 20: KNN – Name

Figure 21: KNN – Type

Figure 22: Decision Tree – Group

Figure 23: Decision Tree – Name

Figure 24: Decision Tree – Type

Figure 25: KNN – Group

Figure 26: KNN – Name

Figure 27: KNN – Type

Figure 28: Decision Tree – Group

Figure 29: Decision Tree – Name

Figure 30: Decision Tree – Type

B   Appendix: 1-Gram Performance Visualizations

Classifier Performance Visualizations

Figure 31: SVM – Single Target

Figure 32: KNN – Single Target

Figure 33: Decision Tree – Single Target

Figure 34: SVM – Multi-Label

Figure 35: KNN – Multi-Label

Figure 36: Decision Tree – Multi-Label

Figure 37: SVM – All Labels

Figure 38: KNN – All Labels

Figure 39: Decision Tree – All Labels

Confusion Matrices

Figure 40: SVM – Group

Figure 41: SVM – Name

Figure 42: SVM – Type

Figure 43: KNN – Group

Figure 44: KNN – Name

Figure 45: KNN – Type

Figure 46: Decision Tree – Group

Figure 47: Decision Tree – Name

Figure 48: Decision Tree – Type

Figure 49: KNN – Group

Figure 50: KNN – Name

Figure 51: KNN – Type

Figure 52: Decision Tree – Group

Figure 53: Decision Tree – Name

Figure 54: Decision Tree – Type