Visual Malware Detection via Convolutional Neural Networks on Digram and Opcode Representations

1st Bernardo Figueiredo

2nd Leonardo Falcão

***Abstract*—Traditional signature-based malware detection increasingly fails against evolving variants. This project explores an alternative approach by converting executable files into visual representations and applying convolutional neural networks for binary classification. We present a Python-based pipeline that performs dual-modality analysis: (1) digram visualization, which captures byte-pair frequency and positional statistics from binary sequences via a 256×256 tensor, and (2) opcode fingerprinting, which maps x86-64 instruction patterns to a 1024×1024 locality-sensitive hash density map. Generated images are normalized and fed to a three-block CNN with batch normalization, dropout regularization, and aggressive data augmentation (horizontal/vertical flips, rotations, zoom, contrast, translation).**

Evaluation on a small, balanced dataset reveals mixed results. The digram model achieves 75% validation accuracy but exhibits a marked generalization gap and threshold-sensitive precision/recall trade-offs. The opcode model similarly captures instruction-level structure yet remains unstable under the small validation regime. Confusion matrices and loss curves indicate meaningful learned patterns, but conclusions are statistically fragile, with each misclassification significantly impacting reported metrics on sets of only 12 validation samples.

We conclude that CNN-based visual malware detection, while theoretically promising, requires substantially larger and more diverse labeled datasets to achieve reliable deployment. As an intermediate finding, classical statistical analysis of image similarity clustering, distributional comparisons, and heuristic-based classification emerges as a more dependable exploitation of these visualizations pending adequate data accumulation.

***Index Terms*—malware detection, binary visualization, convolutional neural networks, opcode analysis, digram analysis.**

I. Introduction

This project investigates a novel approach to malware detection using binary visualization and convolutional neural networks (CNNs). Traditional signature-based malware detection methods face increasing challenges as adversaries develop new variants at scale. We propose a technique that transforms executable files into visual representations by analyzing their binary sequences and opcodes, converting low-level program data into images suitable for deep learning analysis.

A Python-based pipeline processes executable files from an input directory, performing statistical visualization of binary data alongside opcode analysis for executable files. These generated images are subsequently used to train multiple CNN models capable of distinguishing between benign and malicious software.

Our approach leverages image classification techniques to detect anomalies in binary behavior patterns, providing an alternative perspective to traditional static and dynamic analysis methods.

II. Related Work

Several approaches have explored the application of visual analysis and machine learning to malware detection.

Veles [1], a binary visualization tool, provides statistical visualizations of binary sequences, enabling analysts to identify patterns and anomalies in binary data through entropy analysis and visual heuristics. Our analysis of Veles and its source code informed our binary visualization pipeline, establishing the foundation for converting low-level binary data into analyzable image representations.

The seminal work given by the teacher, introduced opcode analysis as a technique for malware detection, leveraging the observation that malicious executables exhibit distinctive opcode patterns compared to benign software. This paper cited in our project description provided the theoretical basis for extracting and analyzing opcode sequences from executable files as a feature extraction strategy. [2]

Our approach replicates and extends the data extraction techniques from both Veles and the opcode analysis methodology. However, rather than relying on manual similarity percentages or heuristic-based classification as in previous work, we leverage modern deep learning architectures (Convolutional Neural Networks) to automatically learn discriminative features from the generated visualizations. This shift from traditional feature engineering and similarity matching to end-to-end deep learning enables our system to discover complex patterns in binary data that may not be apparent through manual analysis.

The combination of binary visualization with CNN-based classification represents a contemporary advancement over existing approaches, allowing for automated, scalable malware detection without explicit feature engineering or predefined similarity metrics.

III. Methodology / Solution Architecture

A. Data Acquisition

Malicious samples are sourced from Malware Bazaar, while benign executables come from internally created or pre-existing clean files. This yields an inherent class imbalance. Careful creation of the dataset was undertaken to ensure that there was a 50/50 split between benign and malicious samples in both training and validation sets, despite the overall imbalance in the raw data.

B. Preprocessing

Each binary is converted to an image (Binary Sequence: 256×256; Opcode Analysis: 1024×1024) produced by the Python visualization pipeline (binary sequence + opcode analysis).

Datasets are loaded with `image_dataset_from_directory` using `normal` and `anomaly` folders, an 80/20 train/validation split (seed 42), RGB color mode, batch size 4 (to fit GPU memory with 256×256 inputs), and on-the-fly normalization via `Rescaling(1./255)`. Images are resized during loading to $(\text{img\_height}, \text{img\_width})$ (defaults 256×256 in the training script). Caching and prefetching (`AUTOTUNE`) reduce I/O overhead; the test loader mirrors this pipeline but sets `shuffle=False` for deterministic evaluation.

C. Feature Extraction

Binary bytes and opcode distributions are rendered into statistical visualizations that encode structural and semantic patterns of executables. These images serve directly as input features to the CNN, eliminating manual feature engineering and enabling the model to learn visual cues indicative of malicious behavior.

D. Model Selection and Training

A three-block CNN is used: `Conv2D` layers with 32, 64, 128 filters (kernel 3×3), each followed by batch normalization, `MaxPooling2D(2,2)`, and dropout (0.2, 0.2, 0.3). The feature map is flattened and passed through two dense layers (`Dense(256)` and `Dense(128)`) with ReLU activations, batch normalization, and dropout (0.5, 0.3), then a sigmoid output neuron. The model is trained with Adam (lr 1×10^{-3}), binary cross-entropy loss, and metrics accuracy/precision/recall/AUC.

Data augmentation is aggressive on training batches to counteract the problem of the dataset size by using random transformations on the images.

Class imbalance is mitigated via inverse-frequency class weights. Training control uses `EarlyStopping` (patience 50, restore best), `ReduceLROnPlateau` (factor 0.5, patience 7, `min_lr` 1×10^{-7}), and `ModelCheckpoint` to `best_model.keras`. Runs are capped at 300 epochs, typically halted earlier by early stopping; evaluation targets validation or a held-out `data_test` set.

IV. Implementation

The implementation splits into a file-analysis pipeline (binary/opcode visualization) and a CNN-based classifier trained on the generated images. Key dependencies: `numpy`, `Pillow`, `moderngl` for GPU rendering, `lief + capstone` for opcode disassembly, and `tensorflow/keras` for model training.

A. File Analysis

Pipeline driver (`main.py`). Recursively walks `input/`, skips hidden files, mirrors the directory structure under `output/`, and hands each file to the visualization pipeline. Executable extensions (`.exe`, `.dll`, `.elf`, `.bin`, `.elf64`, `elf32`) produce both opcode and digram images; everything else produces digrams only.

FileBinaryObject (I/O wrapper). Lightweight helper that opens files in binary mode (`rb+`), streams bytes via `readNBytes(1)`, and exposes `reset()` / `close()`.

Digram (byte-pair histogrammer). Maintains a sliding window of two bytes and updates a $256 \times 256 \times 2$ float32 tensor: channel 0 counts frequency of each ordered byte pair; channel 1 accumulates the byte position (a running counter). On `finalize()`, both channels are normalized by data size (frequency) and data size squared (position) to yield values in $[0, 1]$. Figures 1, 2, and 3 show example digram visualizations.

ShaderRenderer (GPU renderer). Creates a headless ModernGL context, uploads the $256 \times 256 \times 2$ tensor as an RG32F texture, and renders it with fragment shader that maps frequency and relative position into RGB. Off-screen rendering writes a 256×256 PNG.

OpcodeAnalyzer (opcode fingerprinting). For executables, parses with LIEF and disassembles sections using Capstone (x86_64). Mnemonics are truncated to three chars and concatenated until a marked opcode (`mov`, `ret`) triggers processing. The accumulated string is SimHashed (64-bit) to obtain (x, y) coordinates on a $2^{10} \times 2^{10}$ grid; a DJB2 hash yields RGB. A 3×3 neighborhood around (x, y) is incremented in a $1024 \times 1024 \times 3$ float32 tensor (clamped to $[0, 1]$). Figures 4 and 5 show example opcode visualizations.

B. Machine Learning

Data loading and normalization. Generated images are organized into `data_dir/normal` and `data_dir/anomaly`. Datasets are created with `image_dataset_from_directory` using an 80/20 train/validation split (seed 42), RGB color mode, batch size 4 to accommodate GPU memory constraints with larger images, and dynamic resizing to 256×256 pixels during loading. On-the-fly normalization via `Rescaling(1./255)` converts pixel values to $[0, 1]$ range. Performance optimizations include `cache()` to retain preprocessed batches in memory after the first epoch and `prefetch(buffer_size=AUTOTUNE)` to overlap data loading with model training.

Model architecture. The structure of the CNN has already been described in Section 3.D.

Imbalance handling and training control. Class weights are computed as $w_i = \frac{N}{2 \cdot N_i}$ (inversely proportional to class frequency) to offset benign/malicious imbalance. Training employs `EarlyStopping` (monitor `val_loss`, patience 50, restore best weights), `ReduceLROnPlateau` (factor 0.5, patience 7, `min_lr` 1×10^{-7}), and `ModelCheckpoint` to `best_model.keras`. Training runs up to 300 epochs, typically halted earlier by early stopping; evaluation targets validation or a held-out `data_test` set.

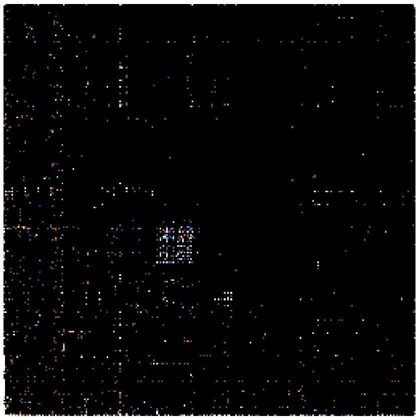

Fig. 1. Digram visualization (Benign Sample).

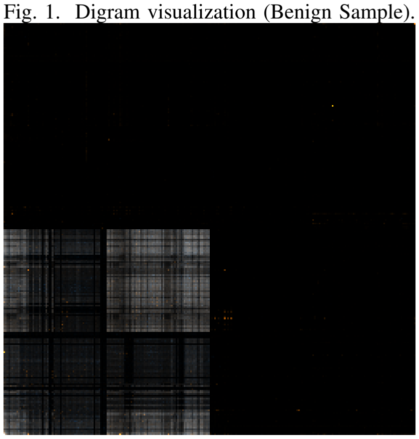

Fig. 2. Digram visualization (Malicious Sample 1).

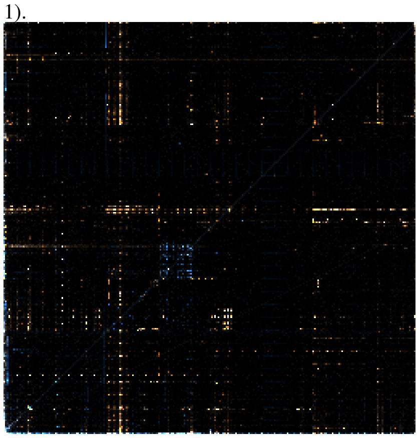

Fig. 3. Digram visualization (Malicious Sample 2).

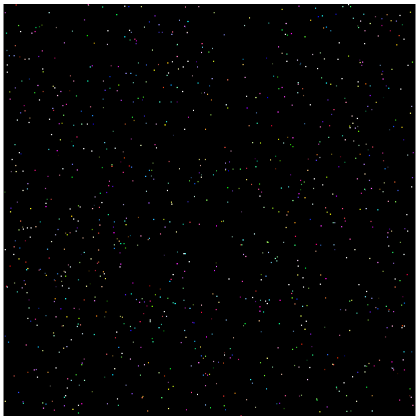

Fig. 4. Opcode visualization (Benign Sample).

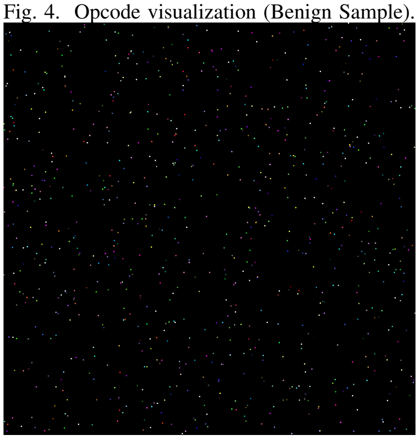

Fig. 5. Opcode visualization (Malicious Sample).

Outputs. Training history plots (loss, accuracy, precision, recall) are saved; the best-performing weights are stored for later inference via the lightweight `predict_image` helper.

V. Results and Evaluation

For the digram-based model, validation was performed on only 12 samples, making every misclassification materially affect the reported metrics. The confusion matrix (Normal/Anomalia) shows: true normal correctly classified (4), normal misclassified as anomaly (3), anomaly misclassified as normal (2), and anomaly correctly classified (3). Thus, false negatives (2) and false positives (3) are both non-trivial, indicating limited class separability under the current operating threshold.

The loss curves exhibit early instability in validation loss (spikes near 3) before a sharp drop around epoch ˜45, after which validation loss plateaus near ˜1.0 while training loss remains markedly lower (˜0.5–0.7). This persistent gap indicates mild overfitting, compounded by the small validation split.

Accuracy hovers around 70–75% for both training and validation but changes in discrete steps, reflecting the very small validation set where a single sample flip materially alters the percentage. Consequently, accuracy is not a robust indicator in this setting.

Precision and recall on validation are highly threshold-sensitive: precision begins at 0, spikes to 1.0, and stabilizes near 0.6; recall starts at 0 and then jumps to 1.0, remaining high.

Overall, the model learns meaningful patterns (recall for anomalies becomes strong), but conclusions are statistically fragile due to the tiny validation set. The observed generalization gap and the precision–recall trade-off suggest the need for more validation data, calibrated decision thresholds, or cost-sensitive tuning to reduce both missed anomalies and false alarms.

A. *Digrams*

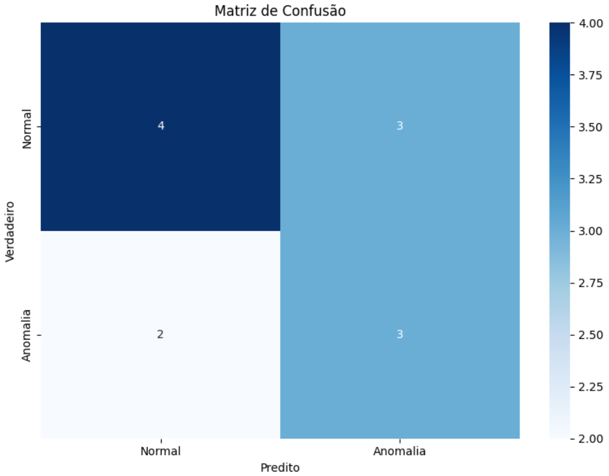

Fig. 6. Confusion matrix for the digram-based classifier.

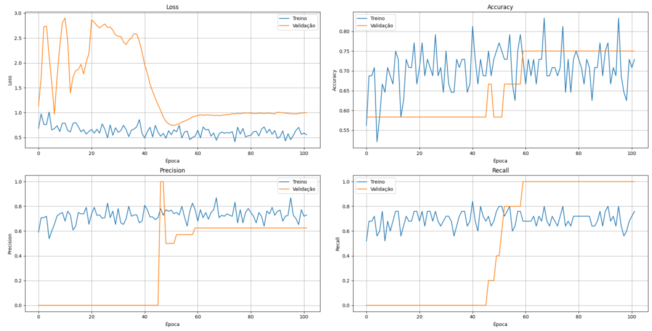

Fig. 7. Training/validation curves (loss, accuracy, precision, recall) for digram inputs.

B. *Opcodes*

For the opcode-based model, the confusion matrix in Figure 8 summarizes binary classification performance across the *Normal* and *Anomaly* classes. The distribution of true positives, false positives, false negatives, and true negatives indicates only moderate class separability under the current decision threshold (0.5). In particular, false negatives (anomalies predicted as normal) are operationally costly in security contexts, while false positives (normals predicted as anomalies) inflate the alert burden and may reduce trust in the system. Given the limited validation size, each misclassification materially perturbs the apparent rates, and care is warranted when interpreting these counts.

The training/validation curves in Figure 9 exhibit early volatility in validation loss, followed by partial stabilization at later epochs. A persistent generalization gap (validation loss remaining above training loss) suggests mild overfitting, plausibly driven by small validation splits, and variability introduced by opcode sequence hashing. Accuracy follows discrete steps consistent with small validation sets, and therefore is not a robust metric here. Precision and recall show threshold-sensitive behavior: periods of low precision coincide with aggressive anomaly detection (higher recall), whereas more conservative operating points improve precision but risk missed anomalies. This trade-off is expected for opcode-level fingerprints that are susceptible to obfuscation, packing, and section-level heterogeneity.

Overall, the opcode model does capture instruction-level structure—as evidenced by non-random confusion patterns and meaningful precision/recall dynamics—but conclusions remain statistically fragile due to the small validation set and the fixed operating threshold. Strengthening external validity will likely require (i) enlarging and balancing the validation set, (ii) calibrating the decision threshold (e.g., ROC-based selection, cost-sensitive criteria), (iii) regularization and/or augmentation tailored to opcode fingerprints, and (iv) evaluating robustness to common transformations (packing/obfuscation). These adjustments aim to reduce both missed anomalies and false alarms while preserving sensitivity to genuine opcode-level malicious behavior.

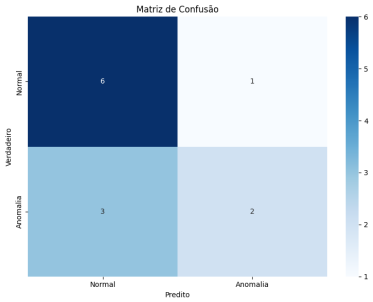

Fig. 8. Confusion matrix for the opcode-based classifier.

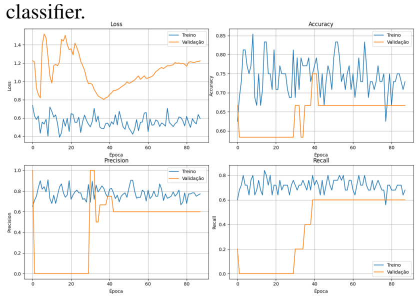

Fig. 9. Training/validation curves (loss, accuracy, precision, recall) for opcode inputs.

VI. Conclusion and Future Work

The empirical results were not sufficiently conclusive to assert that the proposed CNN reliably distinguishes between benign and malicious binaries from the generated visualizations. In particular, confusion matrices and training/validation curves indicate unstable generalization, threshold-sensitive precision/recall, and a persistent gap between training and validation losses. Under these conditions, classical statistical analysis of the visual artifacts—for example, image similarity measures, clustering, and distributional comparisons over digram/opcode features, remains a more dependable way to exploit these visualizations for exploratory analysis.

The principal bottleneck is data scarcity: with a small and imbalanced validation set, each misclassification disproportionately affects metrics, undermining the statistical reliability of conclusions. If additional time were available, among the possible improvements referenced throughout the project, the single most impactful investment would be curating a larger, more diverse, and better-balanced dataset of labeled samples (both benign and malware), including variants with different packing/obfuscation schemes. This would enable:

- Robust evaluation (e.g., stratified cross-validation, confidence intervals, and ROC/PR analysis) and calibrated decision thresholds.
- More expressive models or regularization regimes without overfitting to scarce validation data.
- Comparative baselines with statistical similarity pipelines (e.g., k-NN over feature embeddings, clustering stability tests).
- Stress-testing for common transformations (packing/obfuscation) to quantify robustness.

In summary, while the model captures meaningful structure in the visualizations, current evidence is insufficient to claim dependable deployment.

References

[1] Codilime, "Codilime/veles: Binary data analysis and visualization tool." [Online]. Available: https://github.com/codilime/veles?tab=readme-ov-file

[2] K. Han, J. H. Lim, and E. G. Im, "Malware analysis method using visualization of binary files," in *Proceedings of the 2013 Research in Adaptive and Convergent Systems, RACS 2013*, 2013, pp. 317–321.