Conference Paper Title*

*Note: Sub-titles are not captured in Xplore and should not be used

1st Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

2nd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

3rd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

4th Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

5th Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

6th Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

***Abstract*—**

- **Word count: Approx. 150-250 words.**
- **Content: A concise summary of the entire project. Mention the specific security problem addressed, the method proposed (e.g., "Support Vector Machines on OpCodes"), and the final result (e.g., "Achieved 95% F1-score").**

***Index Terms*—component, formatting, style, styling, insert**

I. Abstract

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. Experimental results demonstrate that this method achieves promising performance in malware detection, validating the effectiveness of binary visualization as a feature extraction strategy. The findings suggest that deep learning-based visual analysis can be a valid and efficient complement to existing malware detection systems, particularly for identifying novel or obfuscated threats.

II. Related Work

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

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

Identify applicable funding agency here. If none, delete this.

(more malware than benign), which is explicitly addressed during training via class weights.

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).

TODO: CHECK THIS Datasets are loaded with `image_dataset_from_directory` using `normal` and `anomaly` folders, an 80/20 train/validation split, grayscale mode, batch size 8, and on-the-fly normalization (`Rescaling(1./255)`). Caching and prefetching (`AUTOTUNE`) reduce I/O overhead.

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

TODO: CHECK THIS A compact CNN is used to suit the small dataset: one `Conv2D(16, 3x3)` with ReLU, `MaxPooling2D`, and dropout, followed by `Flatten`, a `Dense(32)` layer with ReLU and dropout, and a sigmoid output neuron. The model is trained with Adam (lr 1×10^{-4}), binary cross-entropy loss, and metrics accuracy/precision/recall/AUC. Aggressive data augmentation (random flips, rotations up to 0.4 rad, zoom 0.3, contrast 0.3, translations 0.2) combats overfitting. Class imbalance is mitigated via inverse-frequency class weights computed from the training set. Early stopping (patience 15, restoring best weights), learning-rate reduction on plateau (factor 0.5, patience 7, `min_lr` 1×10^{-7}), and checkpointing to `best_model.keras` are enabled. Training runs up to 50 epochs; evaluation is reported on validation or held-out test data when available.

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

Input discovery and outputs. `main.py` recursively walks `input/`, mirrors the folder structure under `output/`, and processes every non-hidden file. Executable types (`.exe`, `.dll`, `.elf`, `.bin`, `.elf64`, `.elf32`) get both opcode and digram visualizations; other files get digram-only images.

Binary digram images. Each file is read byte-by-byte via `FileBinaryObject`; `Digram` keeps a $256 \times 256 \times 2$ tensor where channel 0 counts occurrences of consecutive byte pairs and channel 1 accumulates positional offsets. After normalization (divide by data size), the tensor is uploaded to the GPU with `ShaderRenderer.upload_texture` and rendered off-screen to a 256×256 RGB image saved under `output/.../digram_images/`. The OpenGL renderer uses either bundled GLSL shaders or inline fallbacks; off-screen rendering avoids window dependencies.

Opcode images. For executables, `OpcodeAnalyzer` disassembles code sections using `lief` and `capstone` (x86_64). Opcodes are truncated to three characters and streamed; on marked opcodes (`mov`, `ret`), the current opcode string is hashed twice (SimHash and DJB2) to derive $2^{10} \times 2^{10}$ pixel coordinates and RGB values. A 3×3 splash updates neighboring pixels in `texture_array`; the result is written to `output/.../opcode_images/`. This encodes opcode distribution and locality as color density maps.

Technical considerations. The pipeline guards against hidden files, preserves directory layout, and releases GPU resources after each render. Image sizes differ by modality (digram: 256×256; opcode: 1024×1024) but both are normalized float buffers converted to PNG.

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` (80/20 split, grayscale, 64×64, batch size 8), followed by on-the-fly `Rescaling(1./255)`, caching, and prefetching with `AUTOTUNE`.

Model architecture. A compact CNN suits the small dataset: one convolution (16 filters, 3×3, ReLU), max pooling, dropout, then `Flatten`, a `Dense(32)` with ReLU and dropout, and a sigmoid output. Optimizer: Adam (learning rate 1×10^{-4}); loss: binary cross-entropy; metrics: accuracy, precision, recall, AUC. Aggressive augmentation (random flip, rotation up to 0.4, zoom 0.3, contrast 0.3, translation 0.2) is applied to training batches.

Imbalance handling and training control. Class weights are computed inversely to class frequency to offset the benign/malicious imbalance. Training uses early stopping (patience 15, restore best), learning-rate reduction on plateau (factor 0.5, patience 7, `min_lr` 1×10^{-7}), and model checkpointing to `best_model.keras`. Default run: 50 epochs; evaluation can target validation or a held-out `data_test` set.

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

- **Metrics:** Do not just show Accuracy. You must include:
  - Precision, Recall, F1-Score.
  - Confusion Matrix.

- **Comparisons:** Compare different models (e.g., Random Forest vs. Naive Bayes) or different feature sets.
- **Security Analysis:** Discuss False Positives vs. False Negatives. In your specific context, which is worse? (e.g., Blocking a CEO's legitimate email vs. letting a virus through).

VI. Conclusion and Future Work

- Summarize the main achievements.
- What would you do if you had 6 more months? (e.g., "Test against adversarial attacks", "Optimize for real-time inference").