Malware Defense Model - Comprehensive Report

1. Source Code

Repository Information

- **Git Repository**: https://github.com/gamdhameet/MLSEC.competition.git
- **Branch**: Main branch containing the defender implementation
- **Directory Structure**: The defender model implementation is located in the `defender/` directory

Key Components

- `defender/__main__.py` - Application entry point
- `defender/models/advanced_model.py` - Advanced LightGBM malware detection model
- `defender/models/enhanced_feature_extractor.py` - Comprehensive PE feature extraction
- `defender/apps.py` - Flask web application framework
- `Dockerfile` - Container configuration for deployment
- `docker-requirements.txt` - Python dependencies for Docker environment

---

2. Methodology

2.1 Model Design Choices

Our defense model employs an **Advanced LightGBM-based classifier** combined with heuristic fallback mechanisms. This design choice was motivated by several key considerations:

2.1.1 LightGBM Selection Rationale

LightGBM (Light Gradient Boosting Machine) was chosen as the primary machine learning algorithm for the following reasons:

1. **Performance Efficiency**: LightGBM provides excellent accuracy with faster training and inference times compared to traditional gradient boosting methods, making it ideal for real-time malware detection scenarios.
2. **Handling High-Dimensional Features**: Our feature extraction process generates a comprehensive set of features (80+ dimensions) from PE files. LightGBM's leaf-wise tree growth strategy efficiently handles high-dimensional feature spaces.
3. **Memory Efficiency**: LightGBM's histogram-based algorithm reduces memory consumption, crucial for deployment in resource-constrained Docker environments (1.5GB memory limit).
4. **Robustness**: LightGBM handles missing values and outliers gracefully, which is important when processing diverse PE file formats that may have incomplete or malformed headers.
5. **Interpretability**: Feature importance scores from LightGBM help identify which PE characteristics are most indicative of malicious behavior.

2.1.2 Hybrid Approach: ML + Heuristics

The model implements a **two-tier detection strategy**:

1. **Primary Detection**: LightGBM classifier analyzes extracted features to make predictions
2. **Fallback Detection**: YARA-style heuristic patterns activate when:
    - Model components fail to load
    - Feature extraction encounters parsing errors
    - Edge cases require additional validation

This hybrid approach ensures reliability and maintains detection capability even when ML components encounter unexpected inputs.

2.1.3 Feature Engineering Philosophy The feature extraction methodology focuses on **static analysis** of PE files, extracting:

- **Structural Features**: PE header information, section characteristics, optional header fields
- **Behavioral Indicators**: Import/export analysis, API call patterns, suspicious function detection
- **Statistical Features**: Entropy calculations, string analysis, opcode patterns
- **Metadata Analysis**: Resource information, debug information, TLS data

This comprehensive feature set captures multiple aspects of PE file structure and content that correlate with malicious behavior.

2.2 Model Architecture

The `AdvancedMalwareModel` class implements the following architecture:

```
Input: PE File Bytes
    ↓
Enhanced Feature Extraction (EnhancedPEFeatureExtractor)
    ↓
Feature Vector (80+ features)
    ↓
StandardScaler (Normalization)
    ↓
LightGBM Classifier
    ↓
Probability Score
    ↓
Threshold Comparison (default: 0.80)
    ↓
Binary Decision (0=Benign, 1=Malicious)
```

2.3 Threshold Optimization

The detection threshold (default: 0.80) was optimized to balance: - **False Positive Rate (FPR)**: Target $< 1\%$ to minimize benign file misclassification - **False Negative Rate (FNR)**: Target $< 10\%$ to ensure malware detection - **Real-world Performance**: Threshold tuned on validation datasets to achieve optimal trade-off

The threshold can be adjusted via the `DF_MODEL_THRESH` environment variable in Docker deployment.

---

3. Training Process

3.1 Training Data Sources

The model was trained on a combination of datasets to ensure robust generalization:

1. **EMBER Dataset**:
    - Pre-extracted features from 100,000+ PE samples
    - Includes both benign and malicious samples
    - Provides standardized feature representations
2. **DikeDataset**:
    - Raw PE files requiring feature extraction
    - Additional diversity in malware families
    - Complements EMBER dataset coverage

3.2 Feature Extraction Pipeline

During training, features are extracted using the `EnhancedPEFeatureExtractor` class, which implements:

3.2.1 General PE Features

- File size, virtual size
- Debug information presence
- Relocation table presence
- Resource section presence
- Digital signature presence
- TLS (Thread Local Storage) presence

3.2.2 Header Features

- Timestamp
- Machine type
- Number of sections
- Number of symbols
- Characteristics flags

3.2.3 Optional Header Features

- Base addresses (code, data)
- DLL characteristics
- File alignment
- Image base address
- Version information (image, linker, OS, subsystem)
- Size information (code, headers, image, initialized/uninitialized data)

3.2.4 Section Analysis

- Section count

- Average, maximum, minimum entropy per section
- Text section entropy
- Data section entropy
- Resource section entropy
- Executable section count
- Writable section count

3.2.5 String Analysis

- IP address patterns
- URL patterns
- Domain name patterns
- Registry key patterns
- File path patterns
- MZ header occurrences
- Command-line tool references
- API function name patterns

3.2.6 Entropy Features

- Overall file entropy (Shannon entropy)
- Block-wise entropy distribution
- High-entropy block count (entropy > 7.0)
- Entropy standard deviation

3.2.7 Import/Export Analysis

- DLL import count
- Function import count
- Suspicious API function detection (weighted scoring)
- Suspicious DLL detection
- Export function count
- Presence of specific DLLs (kernel32.dll, ntdll.dll, wininet.dll, ws2_32.dll)

3.2.8 Resource Features

- Resource count
- Resource total size
- Resource average entropy

3.2.9 Opcode Features

- CALL instruction count
- JMP instruction count
- PUSH instruction count
- MOV instruction count
- NOP instruction count
- INT (interrupt) instruction count

3.3 Training Algorithm and Framework

3.3.1 LightGBM Configuration The LightGBM classifier is configured with the following approach:

- **Objective**: Binary classification (malware vs. benign)
- **Boosting Type**: Gradient Boosting Decision Tree (GBDT)
- **Tree Construction**: Leaf-wise (best-first) growth
- **Feature Sampling**: Random feature selection for tree construction
- **Regularization**: L1 and L2 regularization to prevent overfitting

3.3.2 Hyperparameter Tuning Hyperparameters were optimized through: - Cross-validation on training data - Grid search over parameter space - Focus on F1 score optimization - Balance between accuracy and inference speed

Key hyperparameters include: - Number of boosting rounds - Learning rate - Maximum tree depth - Minimum samples per leaf - Feature fraction - Regularization parameters

3.3.3 Data Preprocessing

1. **Feature Alignment**: Features from different datasets are aligned to ensure consistency
2. **Missing Value Handling**: Missing features are filled with zeros (default values)
3. **Normalization**: StandardScaler applied to normalize feature distributions
4. **Train-Test Split**: Stratified split (80% train, 20% test) maintaining class balance

3.3.4 Training Framework

- **Framework**: Python 3.9 with LightGBM library
- **Data Processing**: Pandas, NumPy for data manipulation
- **Feature Extraction**: LIEF library for PE parsing
- **Model Persistence**: Pickle for saving trained models and preprocessors

3.4 Model Evaluation

The model is evaluated using standard classification metrics:

- **Accuracy**: Overall classification correctness
- **Precision**: Malware detection precision
- **Recall**: Malware detection recall
- **F1 Score**: Harmonic mean of precision and recall
- **False Positive Rate (FPR)**: Rate of benign files misclassified as malware
- **False Negative Rate (FNR)**: Rate of malware files misclassified as benign

Validation is performed on held-out test sets and challenge datasets to ensure generalization.

---

4. Implementation Details

4.1 Code Structure

The implementation follows a modular architecture:

```
defender/
  __main__.py                 # Entry point, initializes model and Flask app
  apps.py                     # Flask application with POST endpoint
  models/
      advanced_model.py       # AdvancedMalwareModel class
      enhanced_feature_extractor.py  # EnhancedPEFeatureExtractor class
      advanced_model.pkl      # Trained LightGBM model
      advanced_scaler.pkl    # StandardScaler for feature normalization
      advanced_features.pkl  # Feature names for alignment
```

4.2 Model Parameters and Configuration

4.2.1 Model Initialization

```
model = AdvancedMalwareModel(
    model_path="models/advanced_model.pkl",
    thresh=0.80,  # Detection threshold
    name="Advanced-LightGBM-Detector"
)
```

4.2.2 Environment Variables The Docker container supports the following environment variables:

- `DF_MODEL_THRESH`: Detection threshold (default: 0.80)
- `DF_MODEL_GZ_PATH`: Path to model file (default: models/advanced_model.pkl)
- `DF_MODEL_NAME`: Model name identifier (default: advanced-lightgbm)

4.2.3 Feature Extraction Parameters The `EnhancedPEFeatureExtractor` extracts features with the following characteristics:

- **Total Features**: 80+ numerical features
- **Feature Types**: Mixed (counts, ratios, flags, entropies)
- **Extraction Time**: ~0.2-0.3 seconds per file (average)
- **Fallback Handling**: Basic features extracted if LIEF parsing fails

4.2.4 Suspicious Pattern Detection The model includes YARA-style pattern matching for heuristic fallback:

Suspicious API Categories (with weights): - Process/Thread manipulation: CreateRemoteThread (5), WriteProcessMemory (5), VirtualAllocEx (4) - Network operations: URLDownloadToFile (5), InternetOpen (2), send/recv (1-2) - Registry manipulation: RegCreateKey (2), RegSetValue (2) - Cryptography: CryptEncrypt (3), CryptDecrypt (3) - Anti-debugging: IsDebuggerPresent (3), CheckRemoteDebuggerPresent (3)

Packing Indicators: UPX, MPRESS, ASPack, PECompact, Themida, VMProtect, Armadillo

Ransomware Indicators: encrypt, decrypt, ransom, bitcoin, wallet, payment

Keylogger Indicators: keylog, GetAsyncKeyState, GetForegroundWindow

4.3 API Endpoint Implementation

The Flask application exposes a single POST endpoint:

Endpoint: `POST /` - **Content-Type**: `application/octet-stream` - **Request Body**: Raw PE file bytes - **Response**: JSON `{"result": 0}` (benign) or `{"result": 1}` (malicious) - **Response Time**: < 5 seconds (requirement: < 5s for files up to 2MB)

Model Info Endpoint: `GET /model` - Returns model metadata and configuration

4.4 Docker Configuration

4.4.1 Base Image

- **Base**: `python:3.9-slim`
- **Multi-stage Build**: Yes (optimize image size)

4.4.2 Dependencies Installation

- Python virtual environment (`/opt/venv`)
- Core dependencies from `docker-requirements.txt`:
  - flask>=1.1.2
  - gevent>=1.4.0
  - lightgbm
  - scikit-learn
  - lief (PE parsing)
  - numpy, pandas
  - pefile, annoy, tqdm
- EMBER library from GitHub

4.4.3 Container Configuration

- **Port**: 8080 (exposed)
- **User**: Non-root user (`defender`)
- **Working Directory**: `/opt/defender/`
- **Memory Limit**: 1.5GB (as per competition requirements)
- **CPU Limit**: 1 core (as per competition requirements)

4.4.4 Model Files

- Model files are copied into the container at build time
- Model loading occurs at container startup
- Models are stored in `defender/models/` directory

4.5 Error Handling and Robustness

The implementation includes comprehensive error handling:

1. **PE Parsing Errors**: Fallback to basic feature extraction if LIEF fails
2. **Model Loading Errors**: YARA heuristic fallback if model components missing
3. **Feature Extraction Errors**: Return default feature values
4. **Prediction Errors**: Default to malicious classification (fail-safe)

4.6 Performance Optimizations

1. **Feature Caching**: Feature extraction results could be cached for repeated files
2. **Lazy Loading**: Model components loaded only when needed
3. **Efficient Parsing**: LIEF library optimized for PE file parsing
4. **LightGBM Inference**: Fast prediction times ($<$ 0.1s typically)

---

5. Contributions

[This section will be completed with team member contributions]

---

6. Documentation

6.1 Code Documentation

All major components include: - **Docstrings**: Class and method documentation - **Type Hints**: Type annotations for parameters and return values - **Logging**: Comprehensive logging for debugging and monitoring - **Comments**: Inline comments explaining complex logic

6.2 Technical Documentation

- **PROJECT_STATUS.md**: Current project status, performance metrics, and known issues
- **README.md**: Competition instructions and setup guide
- **FAQ.md**: Troubleshooting guide for common issues

6.3 Deployment Documentation

The Docker deployment process is documented with: - Build instructions - Run commands with various configurations - Environment variable descriptions - Troubleshooting steps

6.4 Model Documentation

The model architecture and training process are documented through: - Code comments explaining design decisions - Feature extraction methodology documentation - Training script documentation - Model evaluation metrics and results

---

Conclusion

This defense model implements a robust, production-ready malware detection system using LightGBM machine learning combined with heuristic fallback mechanisms. The comprehensive feature extraction pipeline captures multiple aspects of PE file structure and behavior, enabling accurate classification while maintaining fast inference times suitable for real-time deployment.

The modular architecture ensures maintainability and extensibility, while the Docker containerization provides consistent deployment across different environments. The hybrid ML+heuristic approach ensures reliability even in edge cases, making the system robust for production use in malware defense scenarios.