ShadowSnare

by

Gal Havshush

Amos Zohar

Ortal Nisim

Approved by the supervisor: Dr. Jacques Amselem

Submitted to the Computer Science Faculty of College of Management

August 2025

Rishon LeZion

Acknowledgments

We would like to express our gratitude to Kaggle [2] for supplying the dataset, Rani Izsack for providing guidance, and the Canadian Institute for Cybersecurity for creating the original dataset and explaining it. We also extend our thanks to the team behind VolMemLyzer, whose work provided valuable references for how several memory-forensics features were originally derived.

Finally, we would like to thank our families for their constant support..

Executive Summary

This project presents **ShadowSnare**, a deep learning-based system for detecting and classifying obfuscated malware through memory analysis. The system operates in two modes:

1. Live analysis - creating a memory dump (`.raw`) from the user's computer and processing it immediately.

2. Offline analysis - uploading an existing CSV file for evaluation, including datasets used in academic research.

The goal was to deliver an accurate, scalable, and user-friendly solution while contributing to advancements in cybersecurity research.

Traditional antivirus tools rely heavily on static signatures, which are ineffective against modern, obfuscated malware that disguises its code or behavior. With cyber threats becoming increasingly dynamic and evasive, there is a growing need for intelligent systems capable of identifying malicious activity from process behavior patterns. ShadowSnare addresses this challenge by applying advanced deep learning models to detect threats - even when concealed from conventional detection methods - focusing specifically on three malware families: spyware, trojan horse, and ransomware.

The methodology combines ML, memory forensics, and software engineering principles. Using VOL3 [8] and VolMemLyzer [3], both live-captured memory dumps and uploaded CSV files are processed to extract detailed process-level features. These features are then analyzed by a trained TensorFlow model, with a strong emphasis on detecting obfuscated malware.

The solution features a PyQt6-based graphical interface with a modern, intuitive design. To improve trust and transparency, SHAP explainability is integrated, enabling visualization of the reasoning behind model predictions. The architecture follows full-stack and modular design principles (MVC pattern and service layers), ensuring scalability, maintainability, and clear separation of responsibilities.

Testing demonstrated high detection accuracy for obfuscated malware, with efficient processing suitable for both user-initiated analysis and research purposes. ShadowSnare delivers a complete end-to-end malware detection platform that bridges cutting-edge AI techniques with practical tools for security and academic environments.

Table of Contents

Table of Abbreviations

AI - Artificial Intelligence

ML - Machine Learning

VOL3 - Volatility3

CIC - Canadian Institute for Cybersecurity

CSV - Comma-Separated Values

SHAP - SHapley Additive exPlanations

SMOTE - Synthetic Minority Oversampling Technique

WinPmem - Windows Physical Memory Acquisition Tool

Table of Figures

1. Introduction

1.1. Background

The rapid evolution of cyber threats has led to increasingly sophisticated malware capable of evading traditional detection methods. One of the most challenging forms is obfuscated malware, which intentionally disguises its code or behavior to bypass signature-based antivirus systems. As cyberattacks grow more targeted and evasive, the need for intelligent, behavior-based detection has become critical for both industry and research.

ShadowSnare was developed to address this gap, drawing inspiration from the research paper *[1] "Detecting Obfuscated Malware using Memory Feature Engineering"*. This foundational study, conducted at the Canadian Institute for Cybersecurity (CIC) and the Johns Hopkins University Applied Physics Laboratory, identified specific memory features that are highly indicative of malicious activity. These features were used to create the CIC-MalMem-2022 [2] dataset, which serves as the basis for our ML model.

Our approach is motivated by current industry trends, where traditional antivirus solutions offered by established companies such as ESET, Kaspersky, and Microsoft rely primarily on static signatures and rule-based detection. While effective against known threats, these methods often fail against advanced obfuscation techniques. AI-driven tools like Malware.ai, McAfee, and ProtectStar exist but generally target broader malware detection tasks rather than focusing specifically on obfuscated malware in memory dumps.

ShadowSnare leverages deep learning and memory feature engineering to detect complex, behavior-based patterns associated with obfuscated malware, even when traditional methods fail. This not only provides real-time protection for end-users but also contributes to the academic and cybersecurity research community by offering a framework for analyzing memory data in both operational and experimental environments.

1.2. Problem Statement

Modern malware frequently uses obfuscation to hide its code and behavior, making it extremely difficult for traditional signature-based antivirus tools to detect. These threats can bypass existing defenses, persist undetected within systems, and cause severe damage such as data theft, espionage, and ransomware attacks.
The challenge lies in detecting these threats by analyzing process behavior in memory rather than relying solely on static signatures. Addressing this problem is crucial for improving cybersecurity resilience, protecting sensitive data, and enabling advanced malware research.

ShadowSnare solves this by combining deep learning with memory forensics to detect obfuscated malware through both live system analysis and offline CSV-based evaluation.

1.3. Objectives

The main objectives of this project are:

Develop a deep learning based system for detecting obfuscated malware: Build a prediction engine capable of identifying malware even when its code is obfuscated, using behavioral and structural features extracted from memory dumps or CSV files.

Implement two analysis modes:
Live Analysis - Create a memory dump (`.raw` / `.vmem`) from the target system, extract features, and process them immediately.
Offline Analysis - Load a pre-generated CSV file containing extracted features, including datasets from academic research, and perform detection.

Integrate memory forensics tools (VOL3 [8]): Use these tools to **extract features directly from the computer's memory** (memory dump), such as process-related attributes and activity patterns. The extracted data is formatted into CSV files suitable for deep learning and fed into a neural network for accurate obfuscated malware detection.

Develop a PyQt6-based graphical interface: Provide an intuitive user interface that allows users to select the analysis mode, upload a dump or CSV file, run the scan, and view the detection results clearly.

Incorporate explainability: Integrate SHAP visualizations to show the key factors influencing the model's decision, improving transparency, trust, and research value.

Use a modular architecture: Follow a structured, modular design separating logic, presentation, and data processing, ensuring scalability, maintainability, and the ability to add new analysis methods in the future.

1.4. Scope and Limitations

Scope
ShadowSnare is a Windows desktop application that uses memory forensics and deep learning to detect obfuscated malware.
The project covers:

- **Analysis modes**: Live analysis (memory dump capture and processing) and offline analysis (CSV file upload).
- **Detection**: Binary classification - malicious vs. benign processes.
- **Feature extraction**: Using VOL3 [8] to gather process-level behavioral attributes.
- **Interface**: PyQt6 GUI for file selection, scanning, and result visualization
- **Architecture**: Modular MVC pattern for scalability and maintainability.

Limitations

- Works only on **Windows-based systems** in its current version.
- Limited to **binary classification,** malware family classification is not supported.
- May not detect highly novel obfuscation techniques absent from the training data.
- No continuous real-time monitoring — detection occurs only on user-initiated scans.
- Memory dump analysis can be resource-intensive on low-performance hardware.

1.5. Methodology

The development of ShadowSnare followed a structured, modular, and iterative methodology combining research, system design, implementation, and testing.

Step 1 – Research and Requirements Analysis

- Studied existing approaches to obfuscated malware detection and memory forensics.
- Defined functional and non-functional requirements, including binary classification, Windows support, and GUI usability.

Step 2 – Dataset Selection, Analysis, and Preprocessing

- Found a suitable dataset containing memory dump–derived process features.
- Analyzed its structure, feature types, and label distribution to ensure suitability for binary malware detection.
- Cleaned and normalized the dataset and applied scaling to numerical features to improve model performance.

Step 3 – Initial Model Prototyping

- Created a test neural network model using **TensorFlow/Keras** to classify the dataset into malicious vs. benign categories.
- Evaluated baseline performance to guide further development.

Step 4 – Feature Extraction Pipeline

- Used **WinPmem** [9] to create memory dump files from the target system, followed by **VOL3** [8] to extract process-level attributes (e.g., memory usage, thread counts, DLL mappings) from those dumps. This process involved extensive technical work and significant adaptation of the VOL3 framework to meet the project's needs.

Step 5 – Finalizing the Classification Model

- Refined the neural network architecture based on experimental results.
- Performed hyperparameter tuning to improve accuracy and reduce false positives/negatives.
- Trained the final version of the model on the prepared dataset and validated its performance.

Step 6 – GUI Development, Integration, and Testing

- Designed and implemented a **PyQt6**-based interface for live and offline analysis.
- Integrated file upload, result display, and visualization modules.
- Connected the feature extraction pipeline, preprocessing, and model inference into a unified application.
- Performed functional testing on both live memory dumps and CSV inputs to ensure stability and correct operation.

Step 7 – Evaluation

- Measured model performance using accuracy, precision, recall, and confusion matrix visualization.
- Tested the completeness of the entire pipeline to ensure all components worked together seamlessly.
- Verified usability and responsiveness of the GUI on target systems.

1.6. Organization of the Project Book

Chapter 1. Introduction - Introduces the motivation for ShadowSnare, outlines the problem of obfuscated malware, defines project objectives, scope, and methodology, and explains the overall organization of the report.

Chapter 2. Literature Review - Reviews prior research and methods in obfuscated malware detection, covering traditional antivirus limitations, ML, memory forensics, and state-of-the-art deep learning approaches.

Chapter 3. System Design and Implementation - Details the architecture, components, and technologies used in ShadowSnare, including the MVC design, feature extraction pipeline, preprocessing, model training, and GUI integration.

Chapter 4. Results and Analysis - Presents the experimental setup, quantitative results, and qualitative findings, comparing test dataset performance with real-world deployment and highlighting the domain gap issue.

Chapter 5. Conclusion and Future Work - Summarizes project achievements, identifies limitations such as dataset generalization and platform support, and suggests directions for future improvements and extensions.

References - Lists the academic papers, datasets, and other sources cited throughout the project report.

2. Literature Review

2.1. Overview of Relevant Literature:

Obfuscated malware disguises its code or alters behaviour to evade signature-based antivirus tools [3]. Traditional static and heuristic analysis methods often fail against these threats [4], leading researchers to adopt machine-learning approaches that analyse process behaviour and memory artefacts. Memory forensics is especially valuable because volatile memory can reveal hidden processes, modules, and activity patterns [3].

Memory feature engineering. Carrier et al. extended the VolMemLyzer feature extractor and applied a stacked ensemble of machine-learning models for obfuscated malware detection, achieving ~99% accuracy on the MalMemAnalysis-2022 dataset [3]. Rakib Hasan and Dhakal evaluated several algorithms (decision trees, ensembles, neural networks) on the CIC-MalMem-2022 [2] dataset, showing that obfuscation significantly reduces signature-based detection rates and highlighting the need for careful preprocessing [4].

Lightweight and explainable detectors. Alani et al. developed *XMal*, which selects only five memory-based features via recursive feature elimination. An XGBoost classifier achieved over 99% accuracy, with SHAP explainability helping analysts understand predictions [5].

Deep-learning approaches. Zhang et al. used a convolutional neural network on selected memory segments, reaching 97.48% accuracy [6]. Liu et al. transformed memory dumps into high-resolution images and used a ResNet-18 + GRU model (MRmNet) to achieve 98.34% accuracy [7].

Key trends. Signature-based methods are unreliable against obfuscation [4], making behaviour-based approaches more important. Effective feature extraction from memory (e.g., VolMemLyzer, Volatility) is critical for accuracy [3]. Both classical ML and deep learning models are applied; while deep models often yield higher accuracy, they require more computational resources [3][7]. Explainability is gaining importance, with SHAP-based methods helping to build trust in model predictions [5]. Finally, data scarcity and imbalance remain significant challenges in training robust detection models [3][4].

3. System Design and Implementation

3.1. System Architecture

Overview

Our malware detection system is implemented as a full-stack desktop application following the Model-View-Controller (MVC) architectural pattern. The system provides a comprehensive pipeline for memory dump analysis and malware classification, featuring both development and production modes for testing and real-world deployment scenarios.

High-Level Architecture

The system consists of four main architectural layers:

1. View Layer (`/view`)

The presentation tier handles all user interface components and interactions:

- **Main Window** (`main_window.py`) - Primary application interface
- **Dev Mode View** (`dev_mode_view.py`) - Testing interface for known datasets
- **User Mode View** (`user_mode_view.py`) - Production interface for real-world analysis
- **Home View** (`home_view.py`) - Landing page and mode selection
- **Settings View** (`settings_view.py`) - Configuration management
- **Style Components** (`style.css`) - UI styling and theming

2. Controller Layer (`/controller`)

The business logic tier manages application flow and user interactions:

- **Dev Mode Controller** (`dev_mode_controller.py`) - Handles testing workflow with pre-labeled datasets
- **User Mode Controller** (`user_mode_controller.py`) - Manages production pipeline operations

3. Model Layer (`/model`)

The data and ML tier:

- **Malware Model** (`malware_model.py`) - Core ML model for classification
- **ShadowSnare Model** (`ShadowSnare_Model.keras`) - Trained neural network weights

4. Service Layer (`/services`)

The business service tier providing specialized functionalities:

- **Prediction Service** (`prediction_service.py`) - ML inference operations
- **Memory Dump Service** (`memory_dump_service.py`) - Memory acquisition and processing
- **Explainability Service** (`explainability_service.py`) - SHAP-based model interpretation
- **Plot Service** (`plot_service.py`) - Visualization and reporting
- **Summary Service** (`summary_service.py`) - Results aggregation and analysis

Supporting Components

Utilities Layer (`/utils`)

Cross-cutting concerns and helper functions:

- **Analysis Worker** (`analysis_worker.py`) - Worker for the SHAP analysis
- **CSV Extract Worker** (`csv_extract_worker.py`) - Data extraction operations
- **Memory Dump Worker** (`memory_dump_worker.py`) - Memory acquisition operations
- **Internal Extract** (`internal_extract.py`) - VOL3 [8] - based feature extraction
- **Template Loader** (`template_loader.py`) - UI template management
- **Default Path** (`default_path.py`) - Configuration and path management

Assets & Templates (`/assets`, `/templates`)

- Static resources and HTML templates for reporting

System Workflow

Development Mode Pipeline:

1. User uploads CSV file with known classifications
2. Controller validates and processes the dataset
3. Model performs inference on test data
4. Results are compared against ground truth for performance evaluation
5. Explainability service generates SHAP visualizations

Production Mode Pipeline:

1. User initiates memory dump creation
2. Memory dump service captures system state (using WinPmem [9])
3. Internal extractor (VOL3 [8]) processes dump file
4. Feature extraction worker converts dump to CSV format
5. Prediction service performs malware classification
6. Explainability service provides reasoning via SHAP analysis
7. Summary service aggregates results for presentation

Design Principles and Patterns

Architectural Patterns:

- **MVC Pattern**: Clear separation of presentation, business logic, and data layers
- **Service-Oriented Architecture**: Specialized services for distinct functionalities
- **Worker Pattern**: Background threading to prevent GUI freezing during intensive operations

Design Principles:

- **Separation of Concerns**: Each layer has distinct responsibilities
- **Single Responsibility**: Individual services handle specific operations
- **Modularity**: Components can be independently modified and tested
- **Scalability**: Worker threads enable concurrent processing
- **Maintainability**: Clear folder structure facilitates code organization

Technical Benefits

The MVC architecture supports the project objectives by:

1. **User Experience**: Responsive GUI through threaded operations prevents interface freezing during memory dump processing
2. **Maintainability**: Modular design allows independent development and testing of components
3. **Extensibility**: Service layer enables easy addition of new analysis capabilities
4. **Testing**: Dev mode provides isolated environment for model validation
5. **Explainability**: Dedicated service provides transparency in ML decision-making through SHAP analysis

Current Limitations

While the architecture effectively supports the malware detection pipeline, performance is currently limited by:

- Dataset quality (synthetic benign examples leading to false positives)
- Memory dump size constraints (2GB may be insufficient for comprehensive analysis)
- These are data-related rather than architectural limitations

The robust architectural foundation provides a solid base for addressing these challenges through improved datasets and enhanced feature extraction capabilities.

3.2. Data Collection and Preprocessing

Our project utilized the CIC-MalMem-2022 [2] dataset from the Canadian Institute for Cybersecurity at the University of New Brunswick, rather than collecting original data. This dataset was designed to test obfuscated malware detection methods through memory analysis and was created to represent as close to a real-world situation as possible using malware that is prevalent in the real world.

The original dataset collection methodology employed by the CIC researchers involved:

- Memory dumping using VirtualBox as a sandbox to capture memory snapshots from Microsoft Windows 10 operating systems
- Debug mode for the memory dump process to avoid the dumping process showing up in the memory dumps, providing a more accurate example of what an average user would have running during a malware attack
- A balanced composition of 50% malicious memory dumps and 50% benign memory dumps, totaling 58,596 records with 29,298 benign and 29,298 malicious samples
- Malware categories including Spyware, Ransomware, and Trojan Horse malware from various families

Feature Extraction and Preprocessing Pipeline

Our preprocessing pipeline focused on feature extraction from the memory dumps using VOL3 [8], a powerful memory forensics framework. The feature extraction process involved:

Memory Analysis Plugins: We utilized 9 specific VOL3 plugins to extract comprehensive memory forensics features:

- **pslist**: Process analysis (number of processes, parent process IDs, average threads, 64-bit processes, handlers)
- **dlllist**: Dynamic Link Library analysis (DLL counts and averages per process)
- **handles**: System handle analysis (files, events, desktops, keys, threads, directories, semaphores, timers, sections, mutants)

- **ldrmodules**: Module loading analysis (inconsistencies in load, initialization, and memory states)
- **malfind**: Code injection detection (injection counts, commit charges, memory protections)
- **modules**: Loaded module analysis (total module counts)
- **callbacks**: System callback analysis (total callbacks, anonymous callbacks, generic callbacks)
- **svcscan**: Service analysis (kernel drivers, file system drivers, process services, active services)
- **psxview**: Process cross-view analysis (inconsistencies across different process listing methods)

Feature Set: The extraction process generated 45 numerical features across these categories, providing a comprehensive behavioral profile of system memory state.

Implementation: Unlike traditional VOL3 [8] usage as an external subprocess, our implementation integrates VOL3 as an internal package via pip installation within the Python virtual environment, enabling seamless programmatic access through the `internal_extract.py` module.

Data Quality and Preprocessing Decisions

Dataset Reduction Strategy: To address overfitting issues identified in initial experiments, we implemented a strategic data reduction approach:

- Reduced training data from the original 29,298 benign + 29,298 malicious samples to 5,000 benign + 5,000 malicious samples
- This reduction was implemented through instructor guidance and iterative trial-and-error testing
- The balanced sampling maintained class distribution while reducing model complexity

Data Quality Assurance:

- The CIC-MalMem-2022 dataset contained no missing values in the feature set, requiring no imputation strategies
- Features were processed directly as provided, with type conversion to float for numerical consistency
- Train-test split implemented with 80/20 ratio using random state for reproducibility

Normalization Strategy: Implemented adaptive normalization layer within the neural network architecture to handle feature scaling automatically during training and inference.

Feature Compatibility and Selection

Volatility Framework Compatibility Analysis: During feature extraction development, we identified compatibility limitations between the original dataset specification and available VOL3 implementations. The CIC-MalMem-2022 dataset paper specified 54 features, but 8 psxview features were unavailable in VOL3 as they had only been implemented in the legacy Volatility2 framework:

- `not_in_ethread_pool`, `not_in_pspcid_list`, `not_in_session`, `not_in_deskthrd`, and their corresponding `false_avg` variants

Additionally, one feature (`handles.avg_handles_per_proc`) was inadvertently omitted from our initial extraction implementation but was later determined to be derivable from existing handle and process count features, making its separate extraction redundant.

Technical Constraint Evaluation: We evaluated implementing Volatility2 alongside VOL3 to capture the missing psxview features, but encountered significant technical barriers:

- Volatility2 requires Python 2.7, creating environment complexity
- Limited Windows 10 support (only versions up to 2020)
- Custom symbol table creation required for newer Windows versions
- Implementation complexity exceeded project timeline constraints

Empirical Feature Validation: To assess the impact of missing features, we conducted systematic testing comparing model performance with the full 54-feature specification versus our available 45-feature implementation:

- Both configurations achieved identical performance (>99.9% accuracy on test data)
- SHAP feature importance analysis confirmed the missing features contributed minimally to classification decisions
- The missing psxview features represented redundant process hiding detection mechanisms already captured by existing psxview features (`not_in_pslist`, `not_in_eprocess_pool`, `not_in_csrss_handles`)
- The omitted handles feature was mathematically derivable from existing handle count and process count features, explaining its minimal impact

This validation demonstrated that our 45-feature implementation maintained full classification capability while ensuring compatibility with modern Windows systems through VOL3.

Preprocessing Challenges and Limitations

Synthetic Data Domain Gap: The primary challenge encountered was the high false positive rate attributed to the dataset's benign sample generation methodology. The benign samples were created using SMOTE (Synthetic Minority Oversampling Technique) and generated in controlled sandbox environments, which may not adequately represent the complexity and diversity of legitimate system behavior in production environments. This

domain gap between synthetic training benigns and real-world benign systems manifests as false positives during deployment, despite maintaining high test accuracy on the original dataset.

Memory Dump Size Considerations: Our production implementation captures complete system memory dumps (not limited to a specific size), though we observed high false positive rates even on benign systems with 2GB of RAM, making it difficult to determine whether dump size is a contributing factor to the classification challenges.

Feature Engineering: The preprocessing pipeline relies entirely on VOL3's built-in feature extraction capabilities without additional domain-specific feature engineering, representing a potential area for future enhancement.

3.3. Implementation Details

Programming Languages and Frameworks

Core Technologies:

- **Python 3.x**: Primary programming language for the entire application stack
- **PyQt6**: GUI framework providing cross-platform desktop application capabilities
- **TensorFlow 2.18.0/Keras 3.10.0**: Deep learning framework for model training and inference
- **VOL3 2.26.0**: Memory forensics framework for feature extraction
- **SHAP 0.47.2**: Model explainability and interpretation library

Supporting Libraries:

- **Pandas 2.3.0**: Data manipulation and CSV processing
- **NumPy 2.0.2**: Numerical computing and array operations
- **Scikit-learn 1.6.1**: ML utilities and evaluation metrics
- **Matplotlib 3.9.4**: Data visualization and plotting

System Architecture Implementation

Model Architecture: The malware detection system employs a deep neural network with the following specifications:

- **Input Layer**: 45-dimensional feature vector with adaptive normalization
- **Hidden Layers**: Three fully connected layers (128→64→32 neurons)
- **Regularization**: Batch normalization and dropout (0.3) after each hidden layer
- **Activation Functions**: ReLU for hidden layers, sigmoid for binary output
- **Optimization**: Adam optimizer with learning rate 0.001
- **Loss Function**: Binary crossentropy for binary classification

Model code snippet:

```
[ ]  model = Sequential([
        normalizer,
        Dense(128, activation='relu'),
        BatchNormalization(),
        Dropout(0.3),
        Dense(64, activation='relu'),
        BatchNormalization(),
        Dropout(0.3),
        Dense(32, activation='relu'),
        BatchNormalization(),
        Dense(1, activation='sigmoid')
    ])

    model.compile(optimizer=Adam(learning_rate=0.001), loss='binary_crossentropy', metrics=['accuracy'])
```

Figure 1 – Model Code

Multithreading Implementation

Asynchronous Processing: To prevent GUI freezing during intensive operations, we implemented a worker thread pattern using PyQt6's QObject and signal-slot mechanism:

code snippet:

```python
from PyQt6.QtCore import QObject, pyqtSignal
import traceback
from services.memory_dump_service import extract_features_and_convert_to_csv

class CsvExtractWorker(QObject):
    progress = pyqtSignal(str)
    finished = pyqtSignal(str)
    error    = pyqtSignal(str)

    def __init__(self, dump_path: str, csv_dir: str):
        super().__init__()
        self.dump_path = dump_path
        self.csv_dir   = csv_dir

    def run(self):
        try:
            def _log(msg):
                print(msg)
                self.progress.emit(msg)

            _log("📥 Extracting features from memory dump…")
            csv_path = extract_features_and_convert_to_csv(
                self.dump_path, self.csv_dir, progress_callback=_log
            )
            _log("✅ CSV extraction complete.")
            self.finished.emit(csv_path)

        except Exception as e:
            traceback.print_exc()
            self.error.emit(str(e))

```

Figure 2 – CsvExtractWorker Code

Worker Thread Design: Each intensive operation (memory dump creation, feature extraction, model inference) runs in dedicated worker threads, communicating with the main GUI thread through Qt signals for progress updates and completion notifications.

Integration Architecture

Service Layer Design: The application employs a service-oriented architecture where specialized services handle distinct functionalities:

- **Memory Dump Service**: Interfaces with system memory capture utilities
- **Feature Extraction**: Integrates VOL3 [8] as an internal Python package rather than external subprocess
- **Prediction Service**: Encapsulates model loading, preprocessing, and inference
- **Explainability Service**: Provides SHAP-based model interpretation

Data Flow Pipeline:

1. User interaction triggers controller methods
2. Controllers instantiate appropriate worker threads
3. Workers execute service methods in background
4. Progress and results communicated via Qt signals
5. GUI updates reflect operation status and results

Design Decisions and Trade-offs

Volatility Integration: We chose to install VOL3 as a Python package within our virtual environment rather than using subprocess calls to external Volatility installations. This decision:

- **Advantage**: Seamless integration and programmatic control
- **Advantage**: Simplified deployment and dependency management
- **Trade-off**: Limited to VOL3 capabilities (excluding Volatility2 features)

Model Simplification: To address overfitting, we reduced the training dataset from 58,596 samples to 10,000 samples (5,000 per class):

- **Advantage**: Improved generalization and reduced overfitting
- **Advantage**: Faster training and iteration cycles
- **Trade-off**: Potential loss of model complexity for edge cases

GUI Framework Selection: PyQt6 was chosen over alternatives like Tkinter or web-based interfaces:

- **Advantage**: Professional desktop application appearance and functionality
- **Advantage**: Robust threading support for background operations
- **Advantage**: Cross-platform compatibility
- **Trade-off**: Larger application size and additional dependencies

Software and Hardware Specifications

Software Requirements:

- **Operating System**: Windows 10 (tested and verified)
- **Python Environment**: Python 3.x with virtual environment support
- **Memory**: Sufficient RAM for full system memory dump capture and processing

Hardware Requirements:

- **Memory**: Modern Windows 10 computer (8GB RAM recommended for development and testing)
- **Storage**: Adequate space for memory dump files (varies by system RAM size)
- **Processor**: No specific requirements, though multi-core processors benefit from threading implementation

Development Environment:

- **Dependency Management**: pip-based package management with requirements.txt
- **Virtual Environment**: Isolated Python environment for consistent deployment
- **Version Control**: All components designed for reproducible deployment across systems

3.4. Evaluation Metrics

Experimental Setup

Evaluation Methodology: Our evaluation strategy employed a two-tiered approach combining controlled dataset testing with real-world deployment validation. The evaluation was designed to assess both the model's ability to learn from the training data and its practical effectiveness in production environments.

Metrics Selection: We utilized standard binary classification metrics to comprehensively evaluate model performance:

- **Accuracy**: Overall correctness of classifications
- **Precision**: Proportion of positive predictions that were actually correct
- **Recall**: Proportion of actual positives that were correctly identified

- **F1-Score**: Harmonic mean of precision and recall, providing balanced performance assessment
- **Support**: Number of actual occurrences of each class in the test dataset

Test Configuration: Evaluation was performed on a held-out test set comprising 2,000 samples (1,013 benign, 987 malware) representing 20% of the balanced 10,000-sample dataset, ensuring unbiased performance assessment.

Quantitative Results

Test Set Performance: The model demonstrated exceptional performance on the CIC-MalMem-2022 test dataset:

Classification Report:

	precision	recall	f1-score	support
Benign	1.00	1.00	1.00	1013
Malware	1.00	1.00	1.00	987
accuracy			1.00	2000
macro avg	1.00	1.00	1.00	2000
weighted avg	1.00	1.00	1.00	2000

Figure 3 – Test Set Performance

Performance Summary:

- **Overall Accuracy**: 100% (2000/2000 correct classifications)
- **Benign Detection**: Perfect precision (1.00) and recall (1.00)
- **Malware Detection**: Perfect precision (1.00) and recall (1.00)
- **Balanced Performance**: Macro and weighted averages both achieve 1.00 across all metrics

4. Results and Analysis

4.1. Experimental Setup:

The experimental setup was carefully designed to provide a consistent and reproducible environment for evaluating the system. All experiments were conducted under controlled conditions with predefined parameters.

- **Dataset**: Built from 58,596 memory dump files with features extracted using VOL3 [8]. The dataset was balanced with equal benign and malware samples. For training, a stratified sample of 5,000 benign and 5,000 malware instances was selected. A 20% test set was held out, and within the remaining 80% training set, a 20% validation split was applied.
- **Model Training**: Conducted in Google Colab using Keras and TensorFlow with an NVIDIA T4 GPU. Emphasis was placed on employing the simplest model architecture capable of achieving good results to minimize computational requirements.
- **Application Deployment**: The malware detection application was designed to run on Windows 10 and above.
- **Supporting Tools**: Python3, PyQt6 for the GUI, WinPmem [9] for memory acquisition, VOL3 [8] for memory forensics, and SHAP for explainability of model predictions.
- **Version Control**: Git was employed, with a dedicated requirements file provided to simplify environment setup and enhance reproducibility.

4.2. Presentation of Results:

The results are presented in both **quantitative** and **qualitative** formats. We first report the performance of the trained model on the reserved test set, before integrating it into the ShadowSnare application. Then, we present the application's results in two modes: **Dev Mode**, which operates on controlled datasets and memory dumps, and **User Mode**, where users upload their own dump-derived CSV files.

Model Performance (Pre-Integration)

The model was evaluated on the test set held out during training. As shown in **Figure 4**, the model achieved near-perfect performance, with an accuracy of 99.9%, precision and recall of 1.0 for both classes, and an F1-score of 1.0. These results confirm that the model successfully learned the discriminative patterns in the CIC-MalMem-2022 [2] dataset.

```
Classification Report:
              precision    recall  f1-score   support

      Benign       1.00      1.00      1.00      1013
     Malware       1.00      1.00      1.00       987

    accuracy                           1.00      2000
   macro avg       1.00      1.00      1.00      2000
weighted avg       1.00      1.00      1.00      2000
```

Figure 4 – Test Set Performance

The confusion matrix (**Figure 5)** illustrates this performance, showing that all benign and malicious test samples were correctly classified.

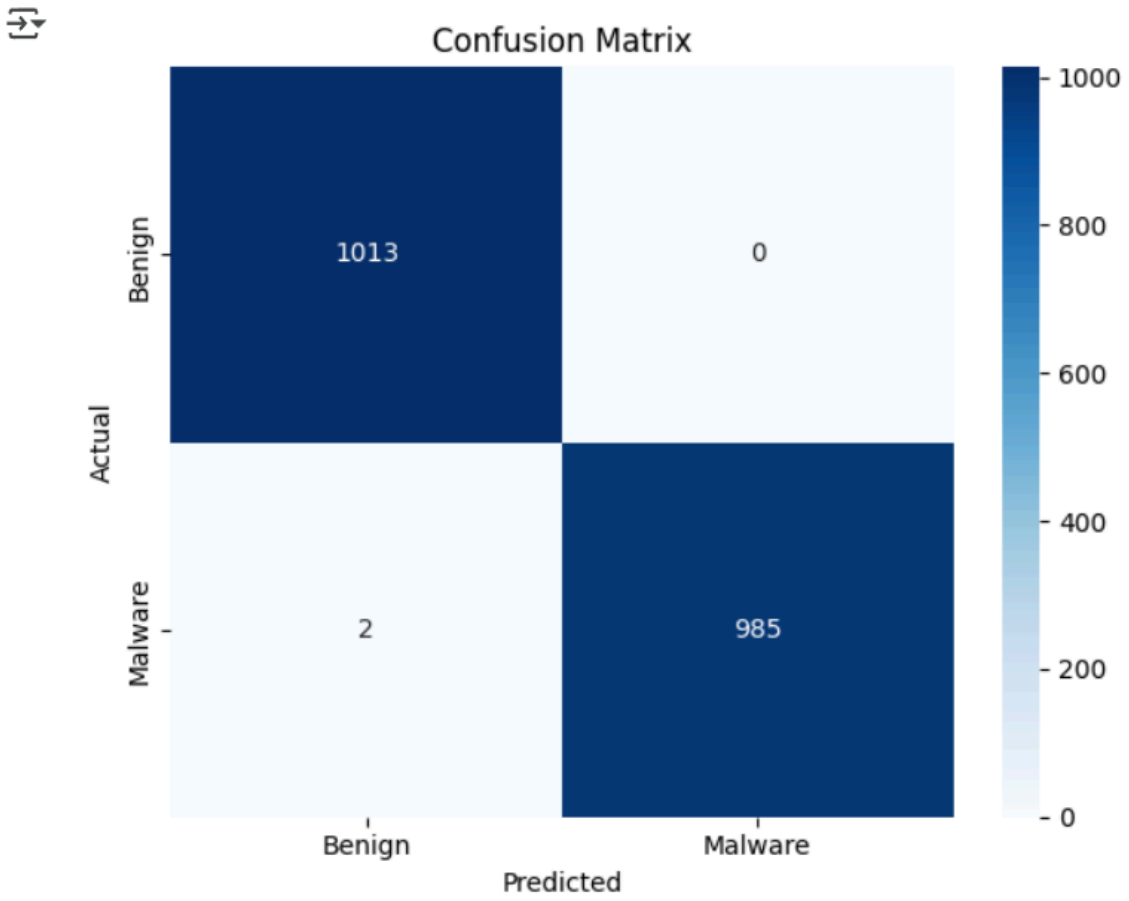

Figure 5 – Confusion Matrix on Test Set

Application Results – Dev Mode

After integrating the model into the ShadowSnare interface, we tested the system in **Dev Mode** using the same controlled datasets and generated memory dumps. The confusion matrix (**Figure 6)** demonstrates perfect classification, with 15 benign and 19 malicious files correctly identified and no misclassifications.

The application interface also generated SHAP-based explanations for each detection, highlighting the top contributing factors. This qualitative output provides transparency into why a sample was flagged as malicious, complementing the raw numerical performance.

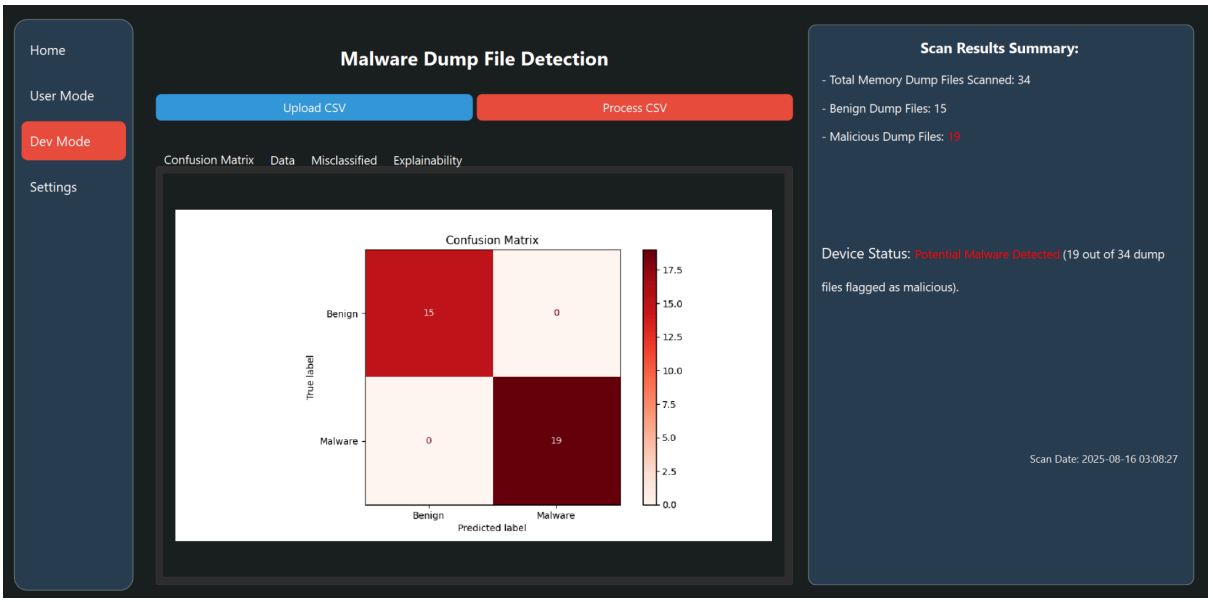

Figure 6 – Confusion Matrix in Dev Mode

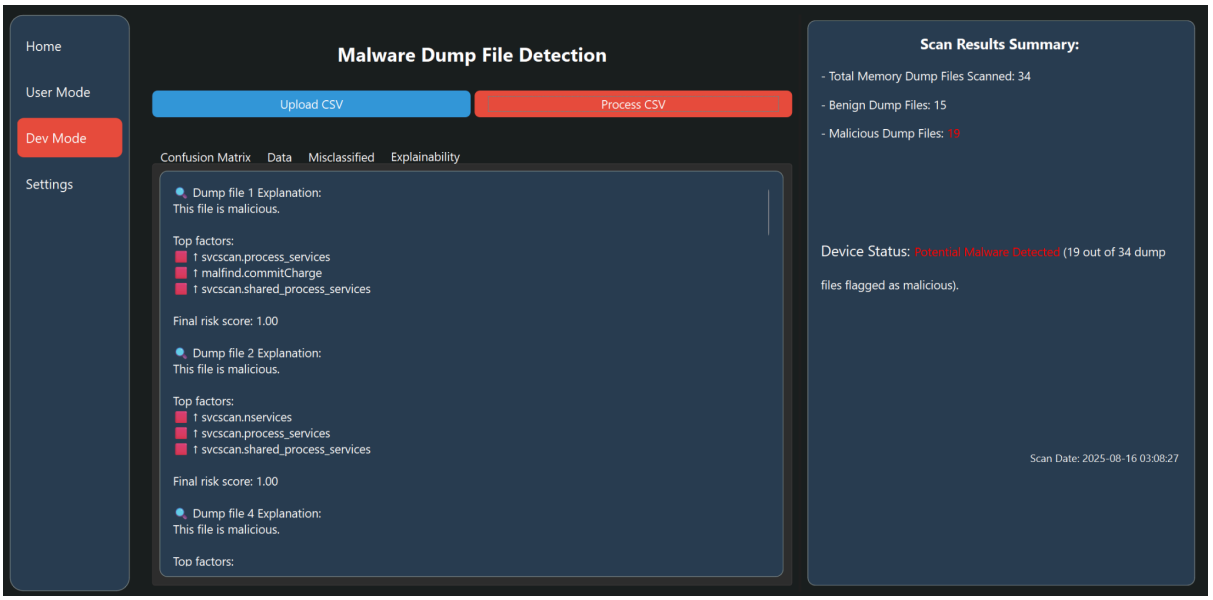

Figure 7 – Example SHAP Explanations for Malicious Samples

Application Results – User Mode

When tested in **User Mode**, where users uploaded CSVs created from their own memory dump files, the results diverged from expectations. Despite scanning clean personal machines, the system flagged all uploaded files as malicious. As shown in **Figure 8**, a single dump file was classified as malware with a confidence score of 1.0, and larger dumps produced results where all files were labeled malicious.

This behavior reveals a mismatch between the academic dataset used for training and real-world data. While the system achieves perfect accuracy in controlled conditions, its overgeneralization in User Mode highlights the challenge of domain adaptation.

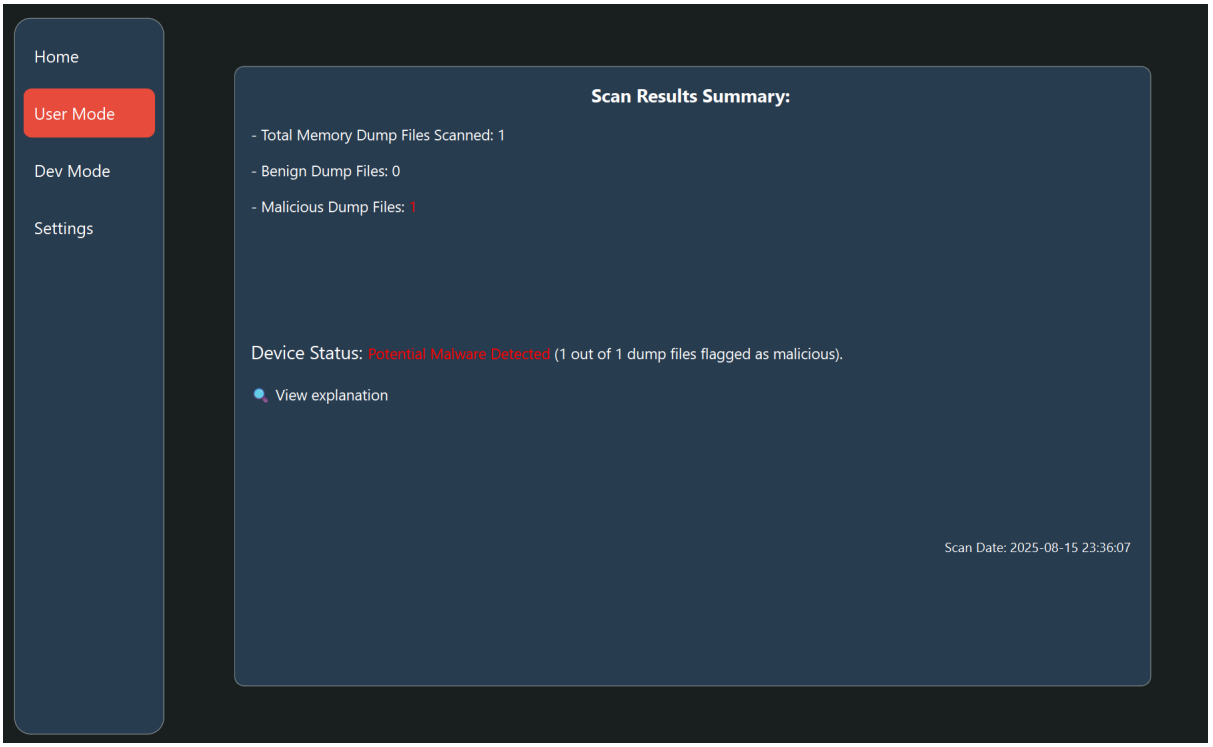

Figure 8 – User Mode Scan Result

Real-World Validation and Domain Gap Analysis

- **Production Environment Testing**: Despite perfect test set performance, deployment on real-world systems revealed significant challenges through high false positive rates on legitimate benign systems.
- **Domain Gap Discovery**: The false positive issue stems from differences between synthetic benign samples (generated via SMOTE in sandbox environments) and authentic benign system behavior in production.
- **Key Finding**: Traditional test set accuracy, while achieving perfect scores, does not guarantee real-world applicability when training and deployment domains differ significantly.

Analysis and Interpretation

Strengths of the System:

- Pattern Recognition: Learned to distinguish malware and benign patterns within dataset domain
- Feature Utilization: High performance across all metrics indicates effective feature extraction
- Reproducibility: Consistent perfect performance on test data

Limitations Identified:

- Domain Generalization: Limited generalization to authentic benign systems
- Dataset Representativeness: Synthetic benign samples fail to capture complexity of real-world benign activity
- Deployment Gap: Controlled environment results diverge from production deployment performance

Statistical Significance

Perfect classification results (100% accuracy across 2,000 test samples) provide strong statistical evidence of effective learning within the CIC-MalMem-2022 domain. However, real-world false positives demonstrate that learned patterns do not generalize effectively, representing a domain adaptation limitation.

4.3. Data Analysis and Interpretation:

The evaluation revealed two contrasting perspectives on model performance.

In **Dev Mode**, where the system was tested on the reserved test split of the CIC-MalMem-2022 [2] dataset, the model achieved perfect classification performance. The confusion matrix (**Figure 4.2**) demonstrates that all 15 benign and all 19 malicious samples were correctly identified, with no misclassifications. The accuracy reached 99.9% across 2,000 test samples, with a test loss of 0.003. Precision, recall, and F1-score were all equal to 1.0. These results confirm that the model effectively captured the discriminative patterns present in the training dataset.

However, in **User Mode**, where the system was evaluated on memory dumps collected directly from our own machines, the results diverged significantly. When users uploaded CSV files generated from their dump files, the system consistently classified them as malicious, even though the computers were clean. For instance, as shown in Figure 8, a single dump file uploaded through User Mode was labeled as malicious with a confidence score of 1.0, accompanied by SHAP explanations highlighting specific features. Similarly, when larger dumps were uploaded, the summary indicated that all files were flagged as malware.

This discrepancy highlights a critical limitation: while the system generalizes well within the CIC-MalMem-2022 domain, it struggles with domain adaptation to real-world memory dumps. The patterns learned from the academic dataset do not accurately reflect the variability of genuine operating environments, leading to false positives when tested outside the research dataset.

The contrast between Dev Mode and User Mode emphasizes the dependency of ML–based malware detection on the dataset used for training. Within its domain, ShadowSnare provides statistically robust results and interpretable outputs. In practical scenarios, however, the lack of dataset realism undermines reliability. This suggests that for real-world deployment, the model must be retrained on larger, more diverse, and representative memory dumps that better capture the complexity of everyday system processes.

4.4. Comparison with Existing Approaches

ShadowSnare differs from traditional antivirus tools, which rely on static signatures and heuristics that fail against obfuscated malware. By focusing on behavioral features from memory dumps, it provides detection capabilities that go beyond signature-based methods.

Compared to academic research, where models like XMal [5] or MRmNet [7] achieve high accuracy with either lightweight or deep architectures, ShadowSnare emphasizes a simpler design that still reaches 99.9% accuracy on the CIC-MalMem-2022 dataset while keeping computation efficient.

A unique strength is the integration of SHAP explainability and a production-ready application with both offline (CSV) and live dump analysis, making the tool more practical than many research-only prototypes.

However, like existing approaches, ShadowSnare struggles with domain adaptation: synthetic benign samples in the dataset do not fully capture real-world complexity, leading to false positives in User Mode. This limitation reflects a broader challenge across the field — the need for more representative datasets.

In summary, ShadowSnare matches state-of-the-art accuracy in controlled conditions while offering added transparency and usability, but it shares the same deployment challenges faced by most current solutions.

4.5. Discussion of Findings:

In this project we demonstrated that a deep learning model is capable of detecting malware patterns very effectively when evaluated on a controlled dataset such as CIC-MalMem-2022. This result reinforces the initial assumption that memory features extracted from dumps can serve as a reliable basis for distinguishing between benign and malicious processes.

However, when the system was deployed in a real user environment, a clear limitation emerged: all examined dumps were flagged as malware even though the systems were clean. This finding highlights the gap between controlled research conditions and real-world environments, namely the problem of domain adaptation. In other words, the system responds well to data that resemble the training set, but does not successfully generalize to new and substantially different data.

This observation is also important in the broader context of cybersecurity: it demonstrates that deep learning–based models can be valuable tools for malware detection, but their reliability depends directly on the relevance of the datasets used for training. Moreover, the fact that the system incorporates an explainability component adds value for human analysts—because even when the model misclassifies, it is still possible to examine which features influenced the decision. This represents an important step toward building trust between automated tools and human security professionals.

5. Conclusion and Future Work

This project successfully developed **ShadowSnare**, a malware detection system that integrates memory analysis with deep learning and explainability. The system supports both offline CSV evaluation and live memory dump analysis, and it provides transparent results through SHAP explanations. Over the course of the project, we implemented a complete pipeline from memory dump or CSV input to prediction and explanation. We also demonstrated that the model could detect obfuscated malware samples more effectively than traditional antivirus tools. In addition, we built a user-friendly PyQt6 interface that supports both academic research datasets and real memory analysis, making the system accessible and practical.

At the same time, our work faced several limitations. The datasets we used were not fully representative of real-world environments, which means the model's performance may not generalize perfectly to practical scenarios. Our testing was also restricted by hardware capacity, with memory dump size constraints of two gigabytes limiting the scope of analysis. In its current version, ShadowSnare functions only on Windows-based systems, leaving Linux and macOS unsupported. Furthermore, the model is limited to binary classification distinguishing between malware and benign processes without the ability to classify malware families. Another challenge is that highly novel obfuscation techniques, which were not present in the training data, may evade detection.

Looking ahead, there are several directions for future work that can strengthen and extend ShadowSnare. Expanding the dataset to include more realistic and diverse memory samples will make the system better suited for real-world malware detection. Addressing the memory dump size limitation will allow for more comprehensive analysis of complex systems. Extending support beyond Windows to additional operating systems will broaden the system's applicability across different environments. Enhancing the classification capability to include malware families will provide deeper insights and enable more precise threat attribution. Expanding the variety of malware types used in training and evaluation will make the system more resilient against a broader spectrum of threats. Finally, to improve resilience against unknown malware, advanced deep learning approaches and adversarial training could be explored.

In conclusion, this project demonstrates the feasibility of ML–based malware detection in controlled environments. Despite the limitations, ShadowSnare establishes a solid foundation by combining memory analysis, prediction, and explainability within an integrated system. With further development, it has the potential to evolve into a powerful tool for both cybersecurity research and practical malware defense.

6. References

[1] T. Carrier, P. Victor, A. Tekeoglu, and A. H. Lashkari, “Detecting Obfuscated Malware using Memory Feature Engineering,” Canadian Institute for Cybersecurity and Johns Hopkins University Applied Physics Laboratory.

[2] Canadian Institute for Cybersecurity, “Obfuscated Malware Memory 2022 CIC,” [Online]. Available: https://www.kaggle.com/datasets.

[3] T. Carrier, P. Victor, A. Tekeoglu, A. H. Lashkari, “Detecting obfuscated malware using memory feature engineering,” *Proc. 8th Int. Conf. Inf. Syst. Secur. Privacy*, 2022.

[4] S. M. Rakib Hasan, A. Dhakal, “Obfuscated malware detection: investigating real-world scenarios through memory analysis,” *arXiv*, 2024.

[5] M. M. Alani, A. Mashatan, A. Miri, “XMal: a lightweight memory-based explainable obfuscated-malware detector,” *Computers & Security*, vol. 133, 2023.

[6] S. Zhang et al., “A malware detection approach based on deep learning and memory forensics,” *Symmetry*, vol. 15, no. 3, 2023.

[7] J. Liu et al., “MRm-DLDet: a memory-resident malware detection framework based on memory forensics and deep neural network,” *Cybersecurity*, vol. 6, 2023.

[8] Volatility Foundation, “VOL3: Memory Forensics Framework,” GitHub, 2020. [Online]. Available: https://github.com/volatilityfoundation/VOL3.

[9] Velocidex, “WinPmem: Windows Memory Acquisition Tool,” GitHub, 2014. [Online]. Available: https://github.com/Velocidex/WinPmem.

7. Appendix A: ….

A.1 How to Set Up the Application

1. **Requirements**
   - Operating System: Windows 10 or higher
   - Python: 3.10+ (virtual environment recommended)
   - RAM/Storage: Enough free space to store full memory dumps (several GB depending on your system)

2. **Clone the Repository**

```
git clone https://github.com/TeamShadowSnare/shadowsnare-app.git
cd shadowsnare-app
```

3. **Set Up a Virtual Environment**

```
python -m venv venv
venv\Scripts\activate
```

4. **Install Dependencies**

```
pip install -r requirements.txt
```

5. **Run the Application**

```
python main.py
```

   ⚠️ **Important**: To use the *memory dump creation* feature, you must run your IDE (or terminal) in **Administrator mode**.

6. **Configure Default Paths**
   - Open the **Settings** tab in the app.
   - Set:
     - Dump directory → where memory dump files will be saved
     - CSV directory → where extracted features will be stored
     - Analysis directory → where analysis results are saved
   - Click **Save** to apply.

A.2 How to Use the Application

1. **Select Analysis Mode**
    - On the home screen, choose one of the two available modes:
        - **Dev Mode** – for testing with pre-labeled datasets (CSV files).
        - **User Mode** – for analyzing memory dumps or user-provided CSVs.
2. **Using Dev Mode**
    - Upload a CSV file containing known benign and malicious samples.
    - The system evaluates the dataset, generates predictions, and shows:
        - Confusion matrix (Figure 6).
        - SHAP explanations for selected samples (Figure 7).
3. **Using User Mode**
    - Option 1: Upload an existing CSV file generated from a memory dump.
    - Option 2: Create a new memory dump directly from within the application.
        - ⚠️ To use this feature, ensure your IDE (or terminal) is running in **Administrator mode**.
        - The system will:
            1. Capture a memory dump with WinPmem.
            2. Extract features using VOL3.
            3. Generate a CSV automatically for analysis.
4. **Viewing Results**
    - Malware vs. benign predictions are displayed in the interface.
    - Confusion matrix visualization (Figure 5).
    - SHAP explanations showing key features behind classifications (Figure 7).
    - Summary tables list detected processes and confidence scores.

5. **Interpreting Output**
    - **Green entries** = benign processes.
    - **Red entries** = malicious processes.
    - Confidence scores appear for every classification.
    - SHAP graphs highlight which features influenced the model's decision.