INL/RPT-24-80484-Rev000

Summer Internship Report: Integrating PCTRAN with AI-Driven Host-Intrusion Detection and Secured Container Systems for Advanced Malware Analysis

August 2024

Kennedy Marsh
Intern, Thermal Fluid Systems Methods and Analysis, Idaho National Laboratory

Palash Kumar Bhowmik
Mentor, Thermal Fluid Systems Methods and Analysis, Idaho National Laboratory

Piyush Sabharwall
Manager, Thermal Fluid Systems Methods and Analysis, Idaho National Laboratory

Idaho National Laboratory

INL is a U.S. Department of Energy National Laboratory operated by Battelle Energy Alliance, LLC

INL/RPT-24-80484-Rev000

Summer Internship Report: Integrating PCTRAN with AI-Driven Host-Intrusion Detection and Secured Container Systems for Advanced Malware Analysis

Kennedy Marsh
Intern, Thermal Fluid Systems Methods and Analysis,
Idaho National Laboratory

Palash Kumar Bhowmik
Mentor, Thermal Fluid Systems Methods and Analysis,
Idaho National Laboratory

Piyush Sabharwall
Manager, Thermal Fluid Systems Methods and Analysis,
Idaho National Laboratory

August 2024

Idaho National Laboratory
Idaho Falls, Idaho 83415

http://www.inl.gov

Prepared for the
U.S. Department of Energy
Office of Nuclear Energy
Under DOE Idaho Operations Office
Contract DE-AC07-05ID14517

Page intentionally left blank

ABSTRACT

This study presents a solution for enhancing the security of the Personal Computer Transient Analyzer (PCTRAN) PC-based Nuclear Power Plant Simulator by integrating the software with an artificial intelligence (AI)-driven host-intrusion detection system (HIDS), in addition to a secured container system, for malware analysis. PCTRAN is a Windows XP-based software package that has the ability to simulate a variety of accident and transient conditions for nuclear power plants (NPPs). It offers a high-resolution replica of the Nuclear Steam Supply System (NSSS) and displays the status of important parameters allowing for operator interaction.

By including AI-driven HIDS for the NSSS, the framework can identify security threats in real-time, ensuring the integrity of the nuclear simulation environment. Additionally, the secured container system offers the ability to isolate and analyze malware, preventing potential threats from affecting core systems. The integration process involves extensive testing and validation in order to ensure accuracy, reliability, and compliance with security policies. This framework sets a new precedent for secure simulation and training in NPP operations, and offers insight for future advancements in cybersecurity.

Page intentionally left blank

ACKNOWLEDGEMENTS

The author would like to thank the Thermal Fluid Systems Methods and Analysis department in the Reactor System Design and Analysis Division at Idaho National Laboratory (INL) for their encouragement and support. The author would also like to express his gratitude to the Department of Computer Science at North Carolina A&T State University for providing their support during this INL internship. Appreciation also goes out to P. K. Bhowmik, K. L. Fossum, and P. Sabharwall at INL for their mentoring and guidance.

Page intentionally left blank

CONTENTS

FIGURES

ACRONYMS

AI	artificial intelligence
CPU	central processing unit
CSV	comma-separated values
DLL	dynamic link library
GUI	graphical user interface
HIDS	host-intrusion detection system
I/O	input and output
INL	Idaho National Laboratory
IP	Internet protocol
ML	machine learning
NIDS	network-intrusion detection system
NPP	nuclear power plant
NSSS	Nuclear Steam Supply System
OCX	ActiveX control
OS	operating system
PC	personal computer
PCTRAN	Personal Computer Transient Analyzer
PRA	probabilistic risk assessment
RMS	radiation monitoring system
SHA-256	Secure Hash Algorithm 256-bit
SVM	Support Vector Machine
VM	virtual machine
VB6	Microsoft Visual Basic 6
XVFB	X virtual framebuffer

Page intentionally left blank

Integrating PCTRAN with AI-Driven Host-Intrusion Detection and Secured Container Systems for Advanced Malware Analysis

1. INTRODUCTION

The Personal Computer Transient Analyzer (PCTRAN) is a sophisticated software program that simulates accidents and transient conditions in nuclear power plants (NPPs). The program can run on a personal computer (PC) and is designed to operate at speeds that are faster than real-time [1]. Since it is a computer program, however, its' database is only as strong as its' latest patch, which must be applied manually. This work presents a solution for enhancing PCTRAN by integrating the software with an artificial intelligence (AI)-driven host-intrusion detection system (HIDS)—as well as the use of a secured container system developed by Docker—for malware analysis. This report discusses this technology fusion and provides the results of experiments that were recently conducted at Idaho National Laboratory (INL) on the newly developed framework to test its' robustness and effectiveness.

1.1 Background and Motivation

Operating within a Windows XP-based operating system (OS), PCTRAN provides high-resolution color mimic displays of the Nuclear Steam Supply System (NSSS) and its' containment, thereby allowing users to interactively control simulations and observe important parameters in real-time. The graphical user interface (GUI) for the software can be seen in Figure 1. The purpose of using PCTRAN is multifaceted because not only does it serve as a training simulator for operators and engineers by generating comprehensive lessons and emergency drills to enhance reactor theory and transient phenomena understanding, it also provides diagnostic tools that can be used to examine the NSSS in a faster than real-time evaluation. PCTRAN also supports parametric and scoping studies and aids in system modifications, emergency procedure development, and regulatory inquiries. Finally, PCTRAN contributes to probabilistic risk assessments (PRAs) in predicting the consequences of specific event branches, such as core melt or containment failure, and the impact these issues can have on overall plant risk.

A tool named Docker was used to containerize the PCTRAN application. Docker is an open-source platform designed to automate the deployment, scaling, and management of applications within lightweight, portable containers [2]. These containers encapsulate an application along with its' dependencies, ensuring that it operates consistently across various computing environments. By providing a standardized unit of software, Docker simplifies the deployment process and enhances the portability of applications. Docker containers feature several key attributes: containerization, isolation, portability, and efficiency. Containerization ensures an application behaves the same way regardless of the environment in which it is being used, while isolation allows applications to run concurrently on the same host without interference. The portability of Docker containers means they can be deployed on any system supporting Docker, including diverse OSs and cloud platforms. Additionally, the efficiency of Docker containers, which share the kernel and resources of the host OS, leads to faster startup times and reduced resource consumption as compared with traditional virtual machines (VMs) [2].

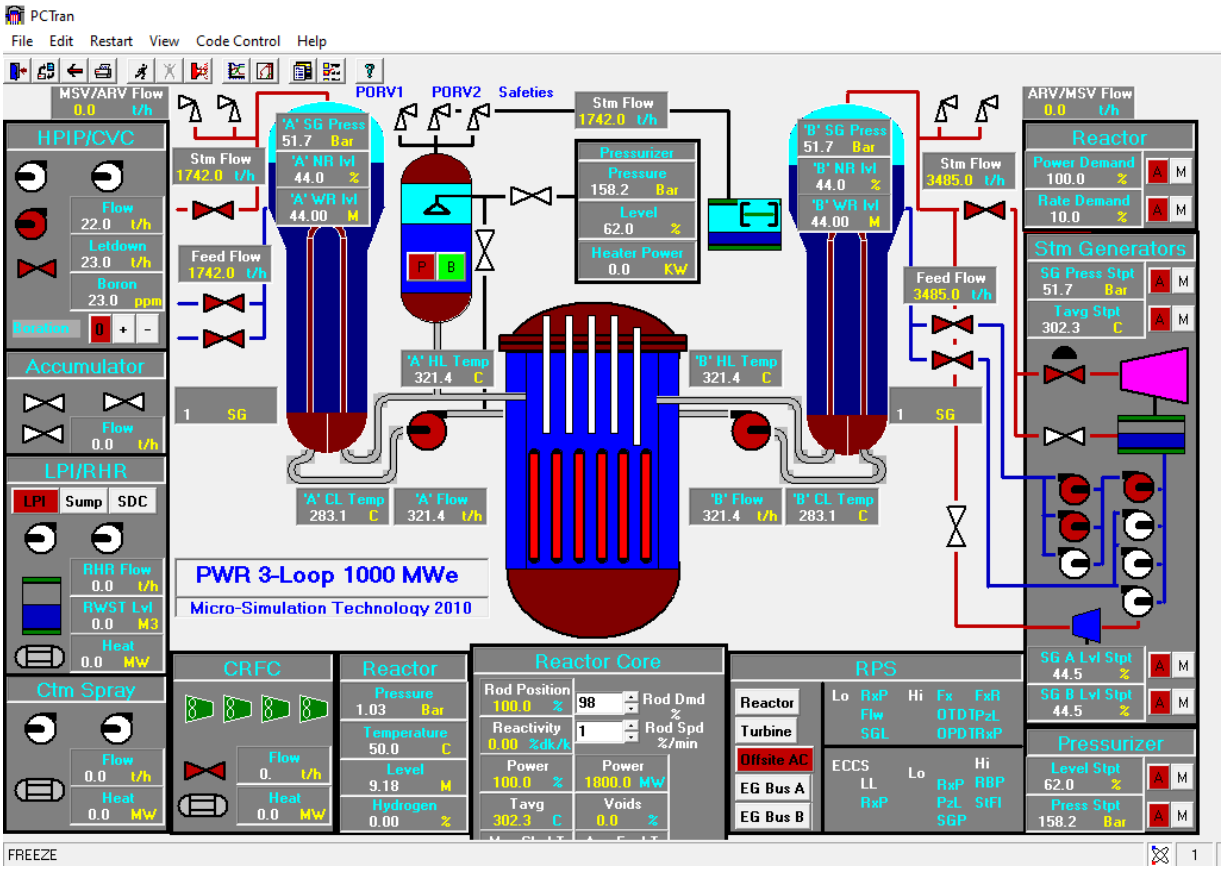

Figure 1. PCTRAN GUI.

An important component in the project is the HIDS, which is a crucial component in any security solution designed to monitor and analyze activities on individual host systems and detect potential security breaches, malicious activities, or policy violations. Unlike network-intrusion detection systems (NIDS) that only focus on network traffic, HIDS targets specific machine or server activities [3]. Common techniques employed by HIDS include file integrity monitoring, which tracks changes to critical system files and settings to detect unauthorized modifications; log analysis, which examines system logs and audit trails for suspicious behavior; and behavioral analysis, which observes deviations from normal application and process patterns. HIDS also utilizes signature-based detection, relying on a database of known attack patterns to identify recognized threats and anomaly detection, which involves establishing a baseline of normal behavior to uncover novel or previously unknown threats.

Integrating AI and machine-learning (ML) into HIDS enhances the ability of the combined framework to identify and respond to potential threats effectively. It leverages the power of advanced algorithms to analyze complex patterns, adapt to new behaviors, and continuously improve detection capabilities. Two distinct approaches in ML include classification models and anomaly detection models. The key differences between classification and anomaly detection lie in their objectives, data requirements, and outputs. Classification aims to predict predefined categories for new data based on labeled training data, while anomaly detection focuses on identifying outliers or deviations from normal patterns, often in unsupervised settings [4]. For the purpose of this project, anomaly detection models are used instead of classification models for anomaly detection in system behavior. Examples of anomaly detection algorithms include Isolation Forest, which isolates observations through random partitioning, and One-Class SVM, which trains a model to recognize the boundary of normal data. Anomaly detection is commonly used in fraud detection, network security, and industrial equipment monitoring.

Incorporating an AI-enhanced HIDS and a secured container system with PCTRAN for malware analysis holds several key implications, such as increased security in nuclear simulations. One feature of PCTRAN is the ability it has for its source code to be modified to the specific specifications of the client's power plant. The software can be custom-designed, ranging from a simple package for NSSS simulation only, to the most extensive model encompassing core-melt, radiation monitoring system (RMS), and source term [1]. Naturally, this software can include proprietary, sensitive, and potentially classified information regarding power plant operations, design, and more. Therefore, the proposed project is paramount, because it addresses the need to protect such sensitive data against malicious activity in real-time with the use of host-intrusion detection. The secured container systems, once employed, will provide isolated environments for analyzing malware, preventing any potential threats from spreading to critical systems. This is also important for improved threat understanding, as detailed analysis of malware behavior within containers enhances the understanding of new threats, which in turn contributes to the development of more effective countermeasures.

1.2 Objectives of the Study

The objective of this project is to integrate PCTRAN with an AI-driven HID and a secured container environment to enhance the malware analysis capabilities of the program within a critical infrastructure NPP context. The project aims to enhance the security posture, implement an isolated environment for running simulations, and ensure that running analyses do not pose risks to the actual OSs.

1.3 Scope of the Study

The scope of the project includes several key components. The primary component is the integration of PCTRAN with an AI-driven HIDS. This includes developing integration interfaces to enable PCTRAN to communicate with the AI-enhanced HIDS for real-time threat-monitoring and detection. Additionally, the project will implement AI models capable of analyzing and responding to suspicious activities detected by the HIDS within the simulation environment. Another crucial component is the deployment of secured container systems. This involves setting up and configuring containerized environments to run the PCTRAN simulations, ensuring these containers are secure and isolated from other systems. This approach aims to enhance the overall security and effectiveness of the simulation process.

2. PCTRAN SOFTWARE

This section provides a detailed description of the chosen software and its components.

2.1 Software Description

As previously mentioned, the PCTRAN software can simulate a variety of accident and transient conditions for NPPs. By using graphic icons and pull-down windows, plant control is conducted by an intuitive point-and-click of the mouse. As observed in Figure 2, the PCTRAN system has the ability to freeze, back-track, snap a new initial condition, change the simulation speed, trend plot selected variables, etc., on a graphic user interface (GUI) for the convenience of conducting a training session or an engineering analysis.

The source code is entirely developed using Microsoft Visual Basic 6 (VB6), and the system operates exclusively within the Microsoft Windows XP environment [5]. Data input and output are managed in a Microsoft Access database format, and the reports and data can be transferred over a network using Office Suite.

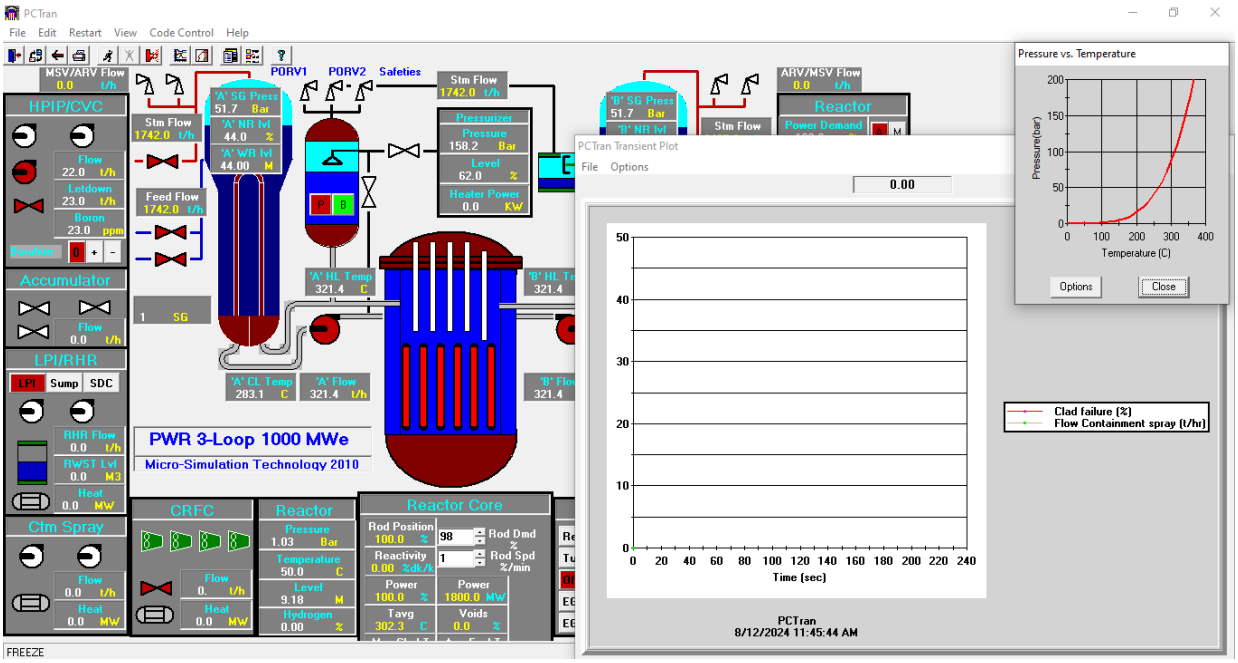

Figure 2. PCTRAN GUI with plotted charts.

2.2 Software Components

As displayed in Figure 3, PCTRAN requires a variety of files in order to run, ranging from Access files, ActiveX control files (OCX), dynamic link library (DLL) files, and more.

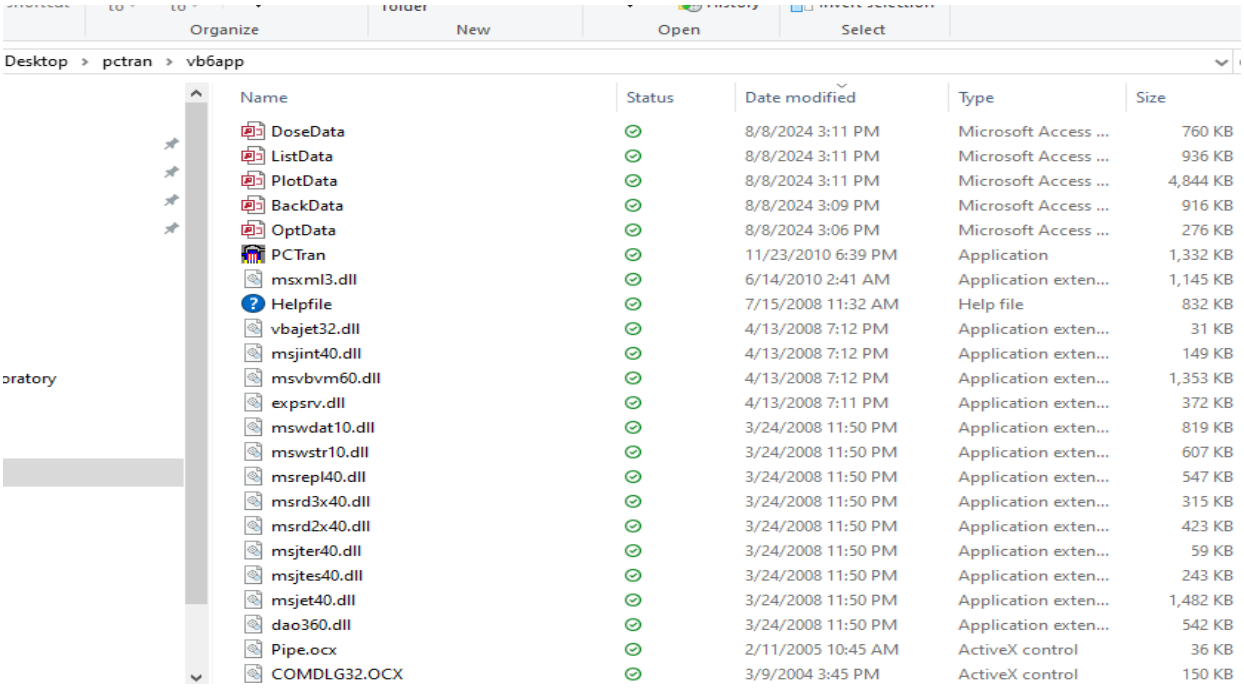

Figure 3. Snapshot of the types of files needed to run PCTRAN.

3. CONTAINERIZATION

In this section, details regarding how the containerization of the simulation software was used will be discussed. The results of the INL experiments with the framework will also be presented.

3.1 Methodology

In order to create a Docker container, one must start by accessing or creating a Docker image, which is a lightweight, standalone, and executable package that includes everything needed to run a piece of software. This package consists of the application code, runtime environment, libraries, dependencies, and configurations required for the software to operate. For this project, a Dockerfile is created so that it outlines the steps to set up a Docker container for running PCTRAN, a VB6 application, on a base Ubuntu image. An Ubuntu image is needed due to the nature of working with a Linux-based Docker container. The Linux-based container is needed because Windows containers do not have the capability of running GUI applications.

The Dockerfile used in this work begins by using “ubuntu:20.04” as the base image. This ensures that the container runs on an Ubuntu 20.04 system. The environment variables for Wine then need to be set. Wine is a compatibility layer that allows running Windows applications on Unix-like OSs. The “WINEARCH” parameter is set to both win32 and win64 to support both 32-bit and 64-bit Windows applications. The “WINEPREFIX” setting specifies the directory where Wine stores its configuration and installed applications files, while the “DISPLAY” is set to :0.0 to handle the graphical output. The Dockerfile then will proceed to install the necessary dependencies. The Dockerfile will first add support for a 32-bit architecture by running “dpkg --add-architecture i386.” The “apt-get update” command will be used to refresh the package lists, and that will be followed by the installation of several packages: “software-properties-common;” “wine64;” “wine32;” “xvfb,” which is the X virtual framebuffer that allows Wine to run without a physical display; “winbind” for the network services; and “cabextract” for extracting the Windows cabinet files. The command “rm -rf /var/lib/apt/lists/*” is necessary because it will clean up the temporary files to reduce the image size.

Further installations include “ca-certificates” and updating them to ensure secure communications, and “wine” and “winetricks” for additional Wine configuration. The “COPY” command then will be used to place the VB6 runtime installer into the container’s cache directory for “winetricks,” which is a tool needed for the installation and management of the Windows DLLs and OCX files. The Dockerfile then will copy the VB6 application and the DLL and OCX files into Wine’s system32 directory. This is necessary for running applications that rely on these files. Additional Microsoft components are also copied to “/tmp” and prepared for installation if needed. The working directory is set to “/app,” where the VB6 application is located. To set up the Wine environment, “xvfb-run” is used to run the Wine commands that initialize and configure the environment. The “winetricks” tool then is employed to install the required components and update the Wine configuration. Specifically, “VB6.0-KB290887-X86.exe” will be executed to install the VB6 runtime file, while “comctl32ocx” is installed via “winetricks” to run in that location. The script then will register the DLL and OCX files using “wine regsvr32.” This step will ensure these files are properly integrated into the Wine environment so the VB6 application can operate properly. Lastly, the “CMD instruction” will set the default command to run when the container starts by using “xvfb-run” to execute the “PCTran.exe” VB6 application under Wine, allowing it to run without a physical display. It now will be possible to use this Dockerfile to build the Docker image, by navigating to the directory containing the Dockerfile. As an example, the command “docker build -t winetest” will be used to create an image named “winetest.” Then, to run the container, navigating to the “MobaXTerm” will provide access to an X11 server so that the application can run with a GUI. The forwarding capability of the server will be needed so users can visualize PCTRAN directly on their local machine. Once in the X-11 server, the “docker run -it -e DISPLAY=host.docker.internal:0 winetest /bin/bash” command will be entered to run the program, as shown in Figure 4.

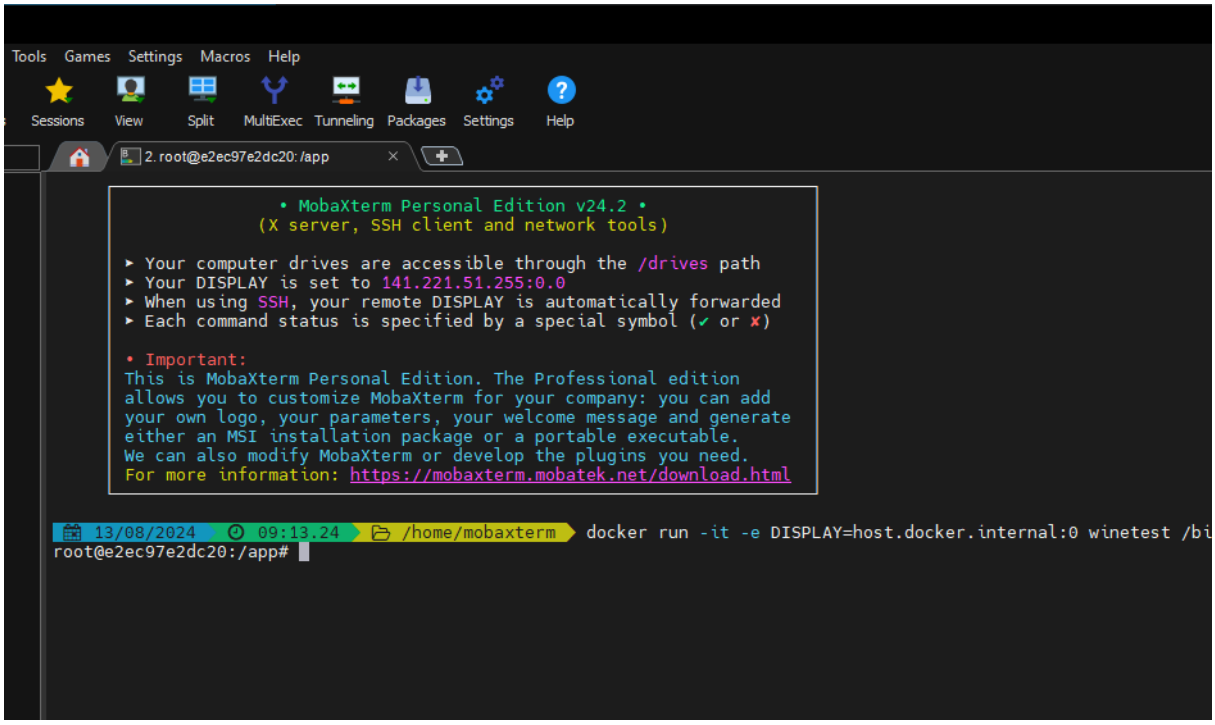

Figure 4. X11 server using MobaXTerm.

The "-it" flag makes the container interactive and attaches a terminal session, which allows access to the application. The "—rm" flag ensures the container will be removed automatically at the conclusion of the program. This command starts the container and executes the default command specified in the Dockerfile, ideally running the VB6 application with Wine. Figure 5 shows the procedure for creating the Docker container.

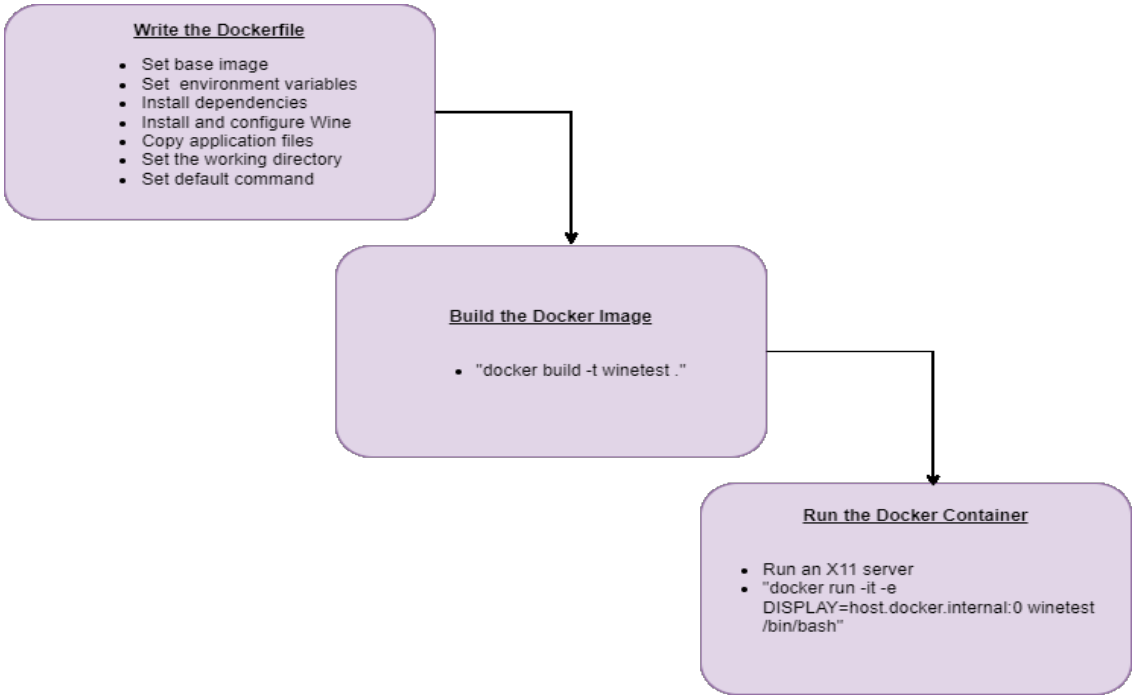

Figure 5. Docker container implementation.

3.2 Results and Discussion

The result of the previous methodology will be a running Linux container encompassing all of the required files for PCTRAN to run, as shown in Figure 6 and Figure 7. The DLL and OCX files for the application will be registered successfully to the Wine registry in Windows. However, issues can arise when attempting to run the GUI application, as shown in Figure 8.

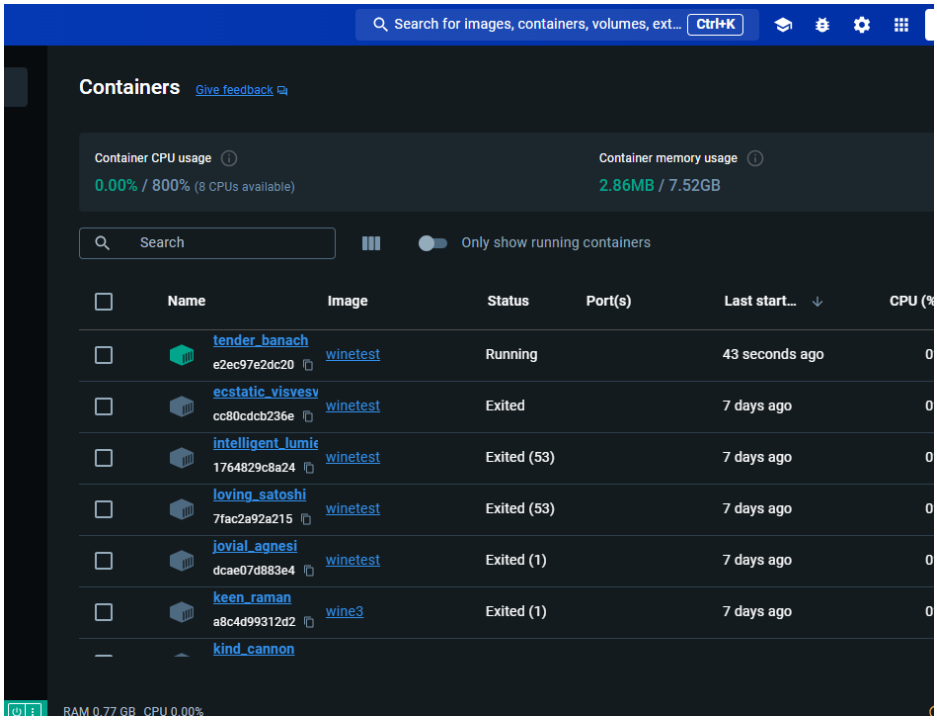

Figure 6. PCTRAN container shown in run state on the Docker desktop.

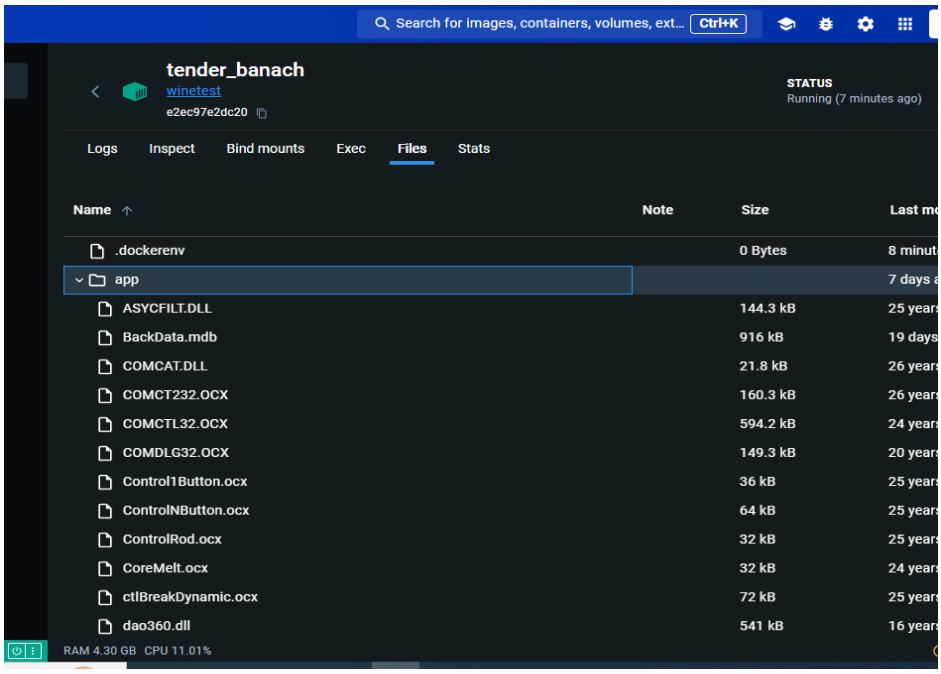

Figure 7. PCTRAN files shown in the container on the Docker desktop.

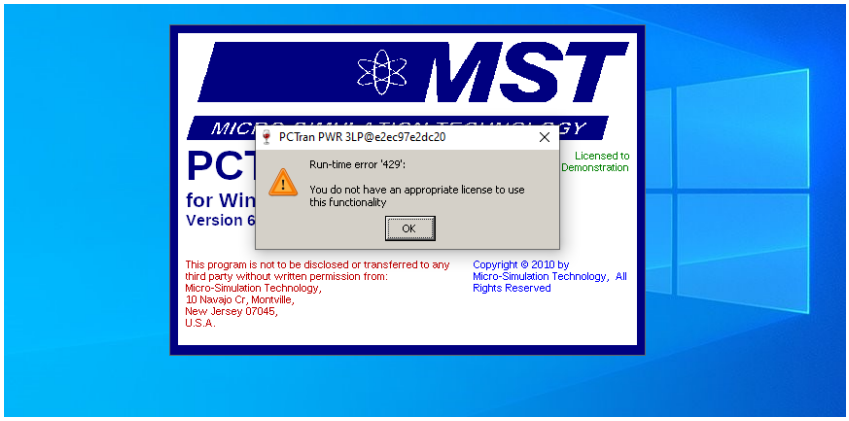

Figure 8. Error message displayed when attempting to run the GUI.

4. HOST-INTRUSION DETECTION SYSTEM

In this section, the steps taken to implement file integrity monitoring, as well as the AI-driven system behavioral analysis, will be reviewed and the corresponding results presented.

4.1 Methodology

4.1.1 File-Integrity Monitoring

File-integrity monitoring is a security technique designed to ensure that critical files and system configurations remain unchanged and free from unauthorized alterations. The primary objective of file-integrity monitoring is to detect any tampering or integrity breaches that could indicate potential security incidents or system compromises. This process begins with establishing a baseline snapshot of file attributes and content, including details such as file size, hash values, permissions, and metadata. The files are then continuously or periodically monitored, comparing their current state against this baseline. When discrepancies are detected, such as changes in file content, attributes, or permissions, the system generates alerts or reports to notify administrators of potential issues.

For this project, file integrity was checked by writing Python scripts to create Secure Hash Algorithm 256-bit (SHA-256) hashes for all important files that PCTRAN uses and input them into a text file that can later be secured by encryption. This text file serves as the baseline for what the hashes of downloaded files should look like. Then, another script was written to verify the hashes of the recently downloaded application by comparing them to the hashes in the original text file.

Two Python modules are required to create the baseline hashes. The first module was “hashlib,” which provides various hash functions used for cryptographic hashing by creating hash values or checksums for data. Hashlib was specifically needed for importing the “sha256” function, which is a cryptographic hash function designed to produce a unique, fixed-size, 256-bit hash value from the input data. The second module was the OS, whose function is to provide a way to interact with the host OS. The code sets the path to the directory containing the files to be processed, while the path is specified as a raw string literal in order to handle backslashes correctly. An empty dictionary is initialized to store the file paths and their corresponding SHA-256 hashes. The code then lists all entries in the specified directory using “os.listdir().” For each entry, it constructs the full file path and checks whether it is a file using “os.path.isfile().” If the entry is a file, a new SHA-256 hash object is created, and the contents of the file are read and used to update the hash object. The computed hash is then converted to a hexadecimal string and stored in the dictionary with the file path as the key. Finally, the code writes the collected file paths and their hashes to the “expectedHashes.txt” text file. This file is opened in write mode and iterates over the dictionary, writing each file path and its corresponding hash to the file, with each entry on a new line. This process effectively creates a record of file hashes that is to be used for verifying file integrity.

The same Python modules also are used in the script to verify the hashes. The script specifies the path to the directory containing the files to be checked. Two dictionaries are then created: “newHashes” to store the hashes of files in the directory and “expectedHashes” to hold the previously recorded hash values. To populate the “newHashes” dictionary, the script lists all entries in the directory using “os.listdir()” and constructs the full path for each file. It checks whether each path corresponds to a file (not a directory) and, if so, calculates its SHA-256 hash. The hash is computed by reading the contents of the file and updating the hash object. The computed hash is then added to the “newHashes” dictionary with the file path as the key. Next, the code populates the “expectedHashes” dictionary by reading the “expectedHashes.txt” file, which contains previously recorded hash values. For each line in this file, the script splits the line into parts using a colon as a delimiter. It then parses these parts to extract the file path and its expected hash, adding these to the “expectedHashes” dictionary. Lines that do not conform to the expected format are skipped, and a message is printed to indicate such occurrences. The final step involves comparing the “newHashes” dictionary, which contains the current hashes, with the “expectedHashes” dictionary. If the two dictionaries match, the script prints a message indicating that hash verification was successful, confirming the software is authentic. If discrepancies are found, a message will be shown that indicates the verification failed and provides detailed information on the differences. This includes hashes that differ between the two dictionaries, as well as any file paths that are unique to one dictionary but not the other, thus helping to identify and troubleshoot any issues with file integrity. Figure 9 outlines the procedure for creating the file integrity integration.

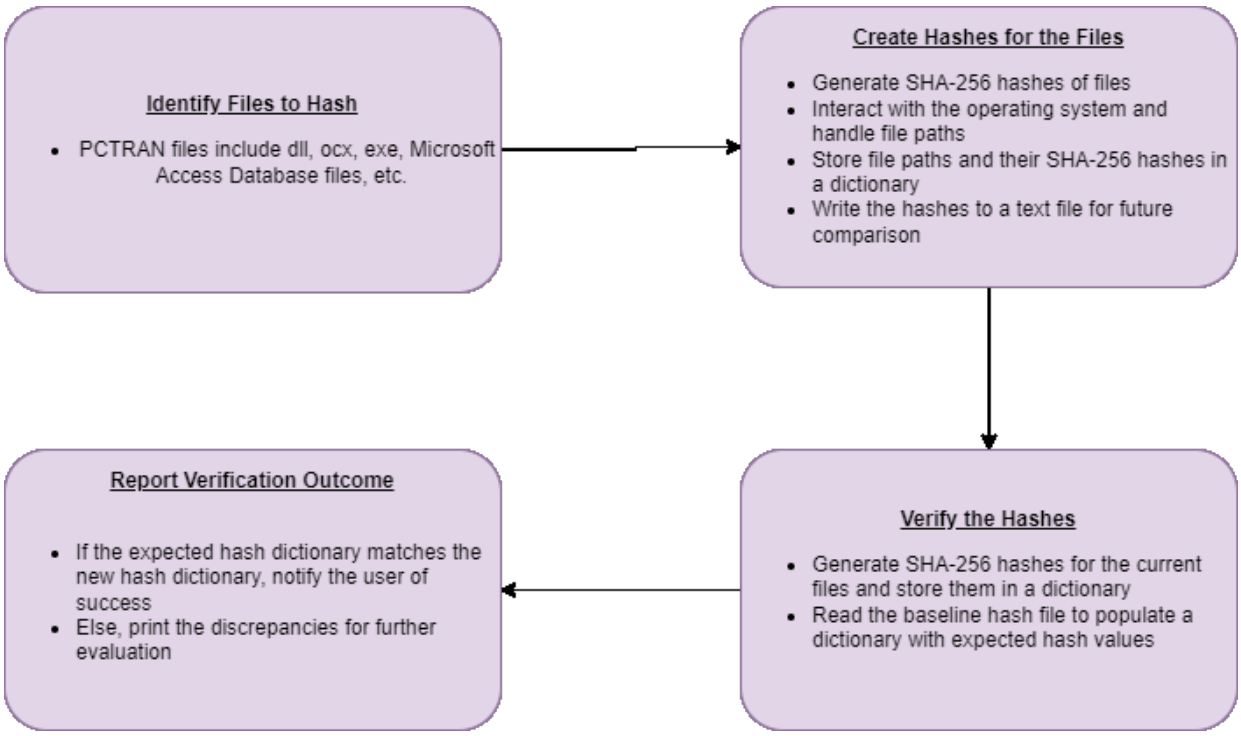

Figure 9. File integrity implementation.

4.1.2 System Behavioral Analysis

Host-intrusion detection focuses on monitoring for signs of unauthorized access or malicious activity within a host system. Detecting such intrusions often requires analyzing subtle signs and correlating information from various sources. System metrics can provide valuable clues to potential intrusions when they display unusual patterns or values. One potential sign of host intrusion is a sudden spike in central processing unit (CPU) usage. Unexpectedly high CPU usage that does not correlate with known processes or scheduled tasks may indicate unauthorized activities, such as crypto mining, resource-intensive attacks, or rogue processes running on the system. Another indicator is unusual memory usage. A significant and

persistent increase in memory consumption, especially if it is not associated with known applications or processes, might suggest an intrusion. This could be due to malware or an in-memory attack that exploits system resources. Increased network activity is also a red flag. Unexplained network traffic, particularly if it involves communication with unfamiliar Internet protocol (IP) addresses, may signal data exfiltration, command-and-control communications, or other malicious network activities. An unusual number of threads within a process is another potential sign of compromise. If a process exhibits a higher-than-normal number of threads, this may be an indication the process has been hijacked to perform unauthorized tasks. Unexpectedly open files can also be indicative of an intrusion. When processes access files that are not typically used during normal operations, it may suggest an attempt to read, modify, or exfiltrate sensitive data. Lastly, the presence of new or unrecognized processes on the system can be a sign of unauthorized access. Processes that are unfamiliar or should not be running on the system may indicate an attacker has gained control and is executing malicious software.

As a solution, an initial script is written to continuously collect baseline data for performance metrics regarding processes associated with the PCTRAN application. It utilizes the "psutil" library for retrieving system and process information, the time module for creating delays between data collection, and "pandas" for handling and storing data in a comma-separated values (CSV) file. The "find_pctran_processes()" function will scan all running processes to identify any of them associated with the "PCTran.exe" program. It will employ "psutil.process_iter()" to iterate over process details and filters for any processes using the name "PCTran.exe." It also will accumulate these processes in a list, which then is returned for further analysis. The "collect_metrics(process)" function gathers various performance metrics from each identified "PCTran.exe" process. It measures CPU usage, both resident and virtual memory consumption, and the number of threads the process is using. In addition, it collects network input and output (I/O) counters, such as bytes and packets sent and received, although these are system-wide rather than specific to the process. The function also will retrieve a list of files currently opened by the process and format this list into a semicolon-separated string. It will handle exceptions for processes that no longer exist or if access is denied, thus updating the metrics with appropriate status messages.

In the main part of the script, an infinite loop will be employed to continuously search for "PCTran.exe" processes every five seconds. When such processes are found, the script will collect their metrics using the "collect_metrics()" function. These metrics are then converted into a "pandas DataFrame" and appended to a CSV file. The use of the header argument ensures the headers are written to the file only if they do not already exist, thereby facilitating the ongoing logging of data. When running this script, 314 lines of data will be collected and used for analysis, as well as for training the ML models, as shown in Figure 10.

Next, the anomalous dataset will be generated by determining the numeric ranges for each piece of data—CPU usage, memory consumption, etc.—and altering some of the data from "normal" to surpass those parameters. This will be done by manipulating the "pandas DataFrame" by introducing random variations to specific columns. Initially, the code imports the necessary libraries, including "pandas" for data manipulation and "numpy" for numerical operations. It then will define several functions to handle different types of data transformations.

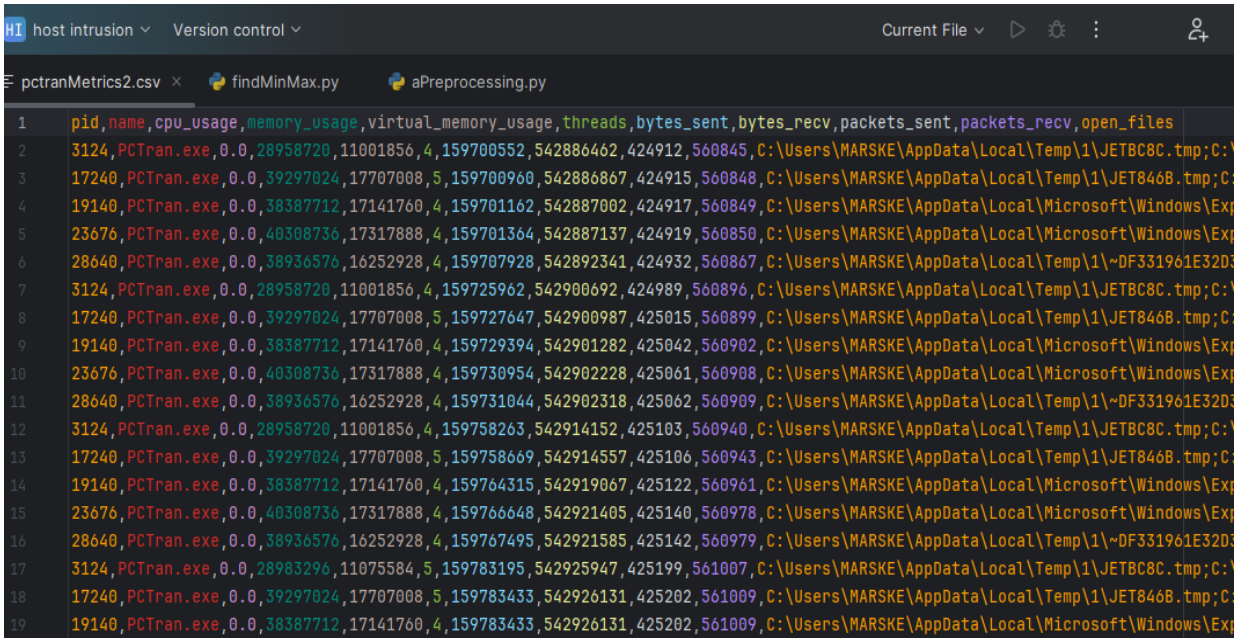

Figure 10. Snapshot of baseline system behavior metrics.

The “add_random_integers()” function will be used to add random integers to specified columns within the “DataFrame.” This will ensure the minimum value is not greater than the maximum value and will set a seed for reproducibility if provided. The function will iterate over the specified columns, verify that each column is of an integer type, and then generate and add random integers within the given range to the existing values in the column. If a column is not of an integer type or does not exist, the function will display a relevant message. Similarly, the “add_random_floats()” function introduces random floating-point numbers to the “DataFrame” and will follow a similar procedure as “add_random_integers” but will target columns of numeric types, including floats and integers. The function then will generate random floats within a specified range and append them to the existing values. If the target column is not numeric or is missing, it will display a message indicating the issue. The “load_strings_from_file()” function will read strings from a specified file and returns them as a set. Each line in the file will be treated as a distinct string, making it easy to sample and use these strings for further processing. The “add_random_strings()” function appends random strings from a provided set to a specific column in the “DataFrame.” This will ensure the target column is of a string type, and then sample random strings from the set to concatenate these strings to the existing values in the column. An error message will be displayed if the column is not of a string type or it does not exist.

After defining all functions, the script then will load data from a CSV file into a “DataFrame” that will specify different columns and ranges for adding random numerical values and perform the necessary transformations using the previously defined functions. Additionally, it will load a set of strings from a file representing malware files collected from a GitHub dataset called “DikeDataset” [6] to be appended to a designated column. The modified “DataFrame” then will be saved to a new CSV file. The script will use the append mode to add data to the file if it already exists, or create a new file if it does not. The header parameter will be adjusted to ensure the headers are written only once. This process will result in a new dataset that includes the introduced anomalies or variations. After running this script, 96 lines of data will be collected and used for analysis and training. There are 96 anomalous instances to simulate a realistic ratio of normal to anomalous occurrences, as shown in Figure 11.

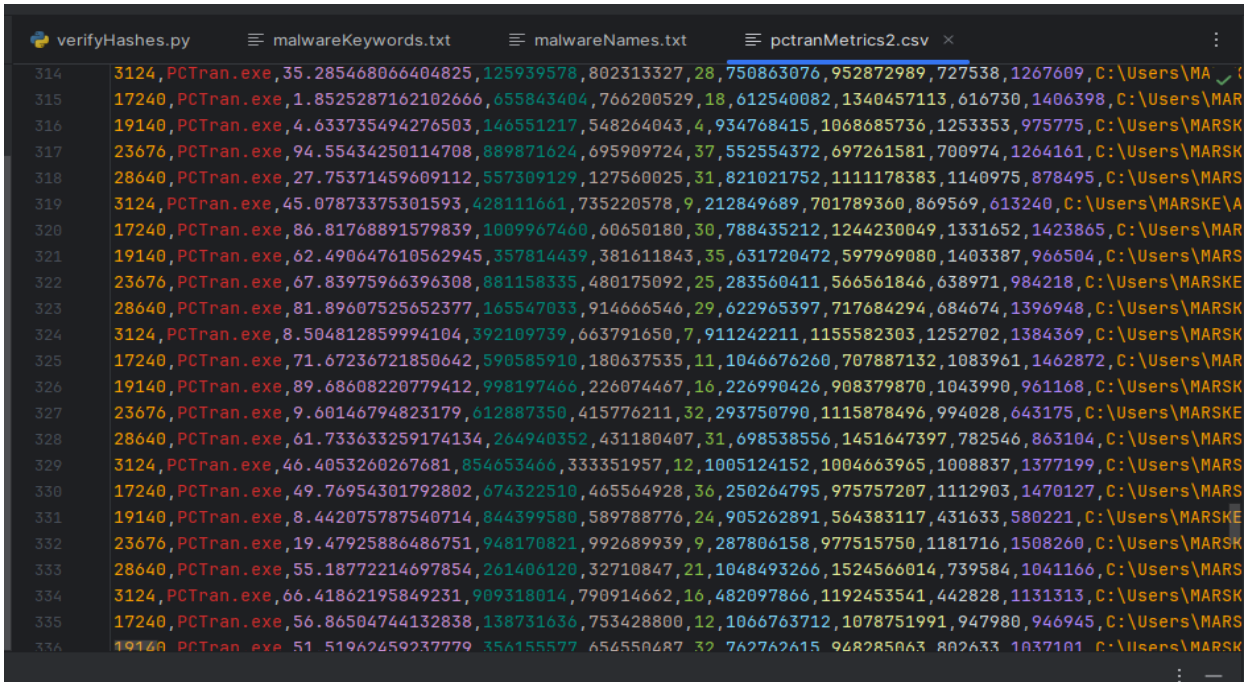

Figure 11. Snapshot of anomalous system behavior metrics.

Once the normal and anomalous data was placed into the same CSV file, the data had to be preprocessed. For this, a method that specifically handles anomaly detection tasks was used. The script utilized various libraries and techniques to clean, transform, and standardize the data, making it ready for analysis or model training. The script begins by defining a function that reads suspicious keywords from a specified text file. Then, these keywords are used to identify potentially malicious file names. The keywords are converted to lowercase and stripped of any extra spaces for consistency. Following this, the dataset is loaded into a DataFrame using pandas. The script identifies which columns are numeric and non-numeric and handles missing values accordingly. For numeric columns, missing values are filled with the mean of each column. Non-numeric columns have missing values filled with empty strings to ensure data completeness. If the DataFrame contains a “file_names” column, additional preprocessing is performed. This column is split into individual file names that then are processed to extract features such as file extensions and the presence of suspicious keywords. Features related to file extensions are encoded using one-hot encoding, and a new column indicating the presence of suspicious keywords is added. These new features then are integrated into the original DataFrame, replacing the “file_names” column. The script then selects a subset of features relevant to the analysis. If the “file_names” column was processed, its associated features are included in the selection. The selected features then are standardized using “StandardScaler” to normalize the data, which is crucial for many ML algorithms. The script includes a commented-out section for adding noise to the features, which could be used to simulate variability in the data but not with this work. The preprocessed data then is saved to a new CSV file named “preprocessedData.csv,” as shown in Figure 12.

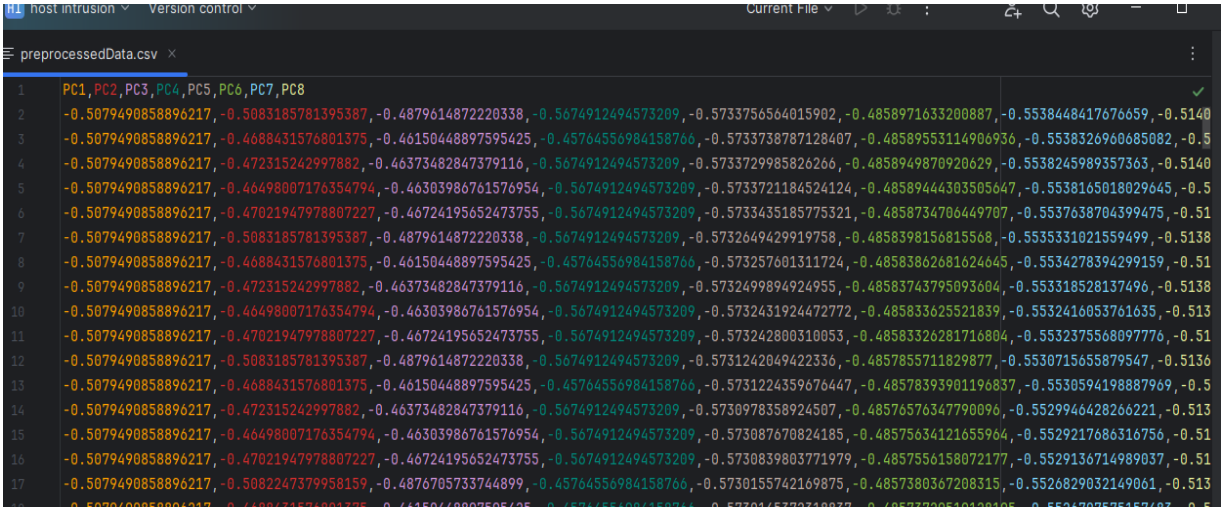

Figure 12. Snapshot of the preprocessed data.

The final step involves the evaluation and tuning of two types of anomaly detection models: (1) Isolation Forest; and (2) One-Class Support Vector Machine (SVM). This is done by utilizing several libraries to perform tasks such as data loading, model training, and evaluation with a focus on selecting the best model configuration.

First, the code imports necessary libraries including pandas and numpy for data manipulation and numerical operations, “sklearn” for ML models and metrics, “seaborn” and “matplotlib” for visualization, and “ParameterGrid” for parameter-tuning. It begins by loading a preprocessed dataset into a pandas “DataFrame” and also reads true labels from a file used for model evaluation. The true labels are mapped to integer values, where “normal” is represented by 0 and “anomaly” by 1. The code then sets up parameter grids for tuning the models. For the Isolation Forest model, a contamination parameter is specified, while for the One-Class SVM model, parameters such as Nu and Gamma are tuned. The “evaluate_anomaly_detection_model()” function is defined to fit a model, make predictions, and evaluate performance using accuracy, a classification report, and a confusion matrix. Predictions are converted from the default output format of the model to match the label format. Another function, “tune_and_evaluate_model(),” is designed to handle model tuning and evaluation that iterates over possible parameter combinations defined in the parameter grid, and evaluates each combination using the “evaluate_anomaly_detection_model()” function, and keeps track of the best-performing model and its parameters based on accuracy. The confusion matrix for each parameter set is visualized using a heatmap to help understand the performance of the model. Then, the code iterates through the defined models and their parameter grids, performs tuning and evaluation, and prints the best model parameters. Optionally, a placeholder is included to save the best-performing model using “joblib.” Figure 13 outlines the procedure for implementing the AI system behavioral analysis.

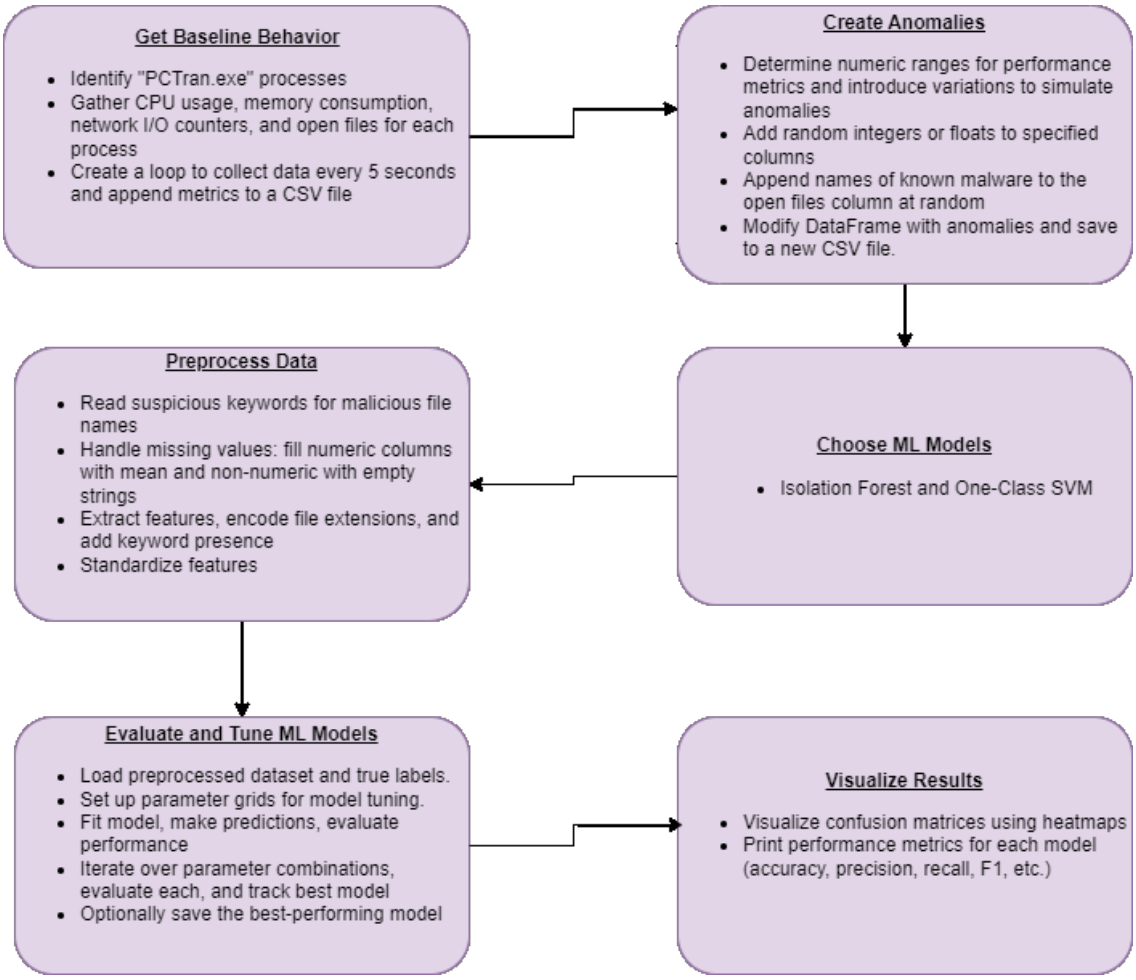

Figure 13. System behavioral analysis implementation.

4.2 Results and Discussion

4.2.1 File Integrity Results

In the final step of verifying the hashes, the “newHashes” dictionary is compared with the “expectedHashes” dictionary. If the two dictionaries match, the script will then display a message indicating that hash verification was successful, thereby confirming the software is authentic. If discrepancies are found, a message indicating that the verification failed instead is display and provides detailed information on the differences, as shown in Figure 14. This includes hashes that differ between the two dictionaries, as well as any file paths that are unique to one dictionary but not the other, thus helping to identify and troubleshoot any issues with file integrity.

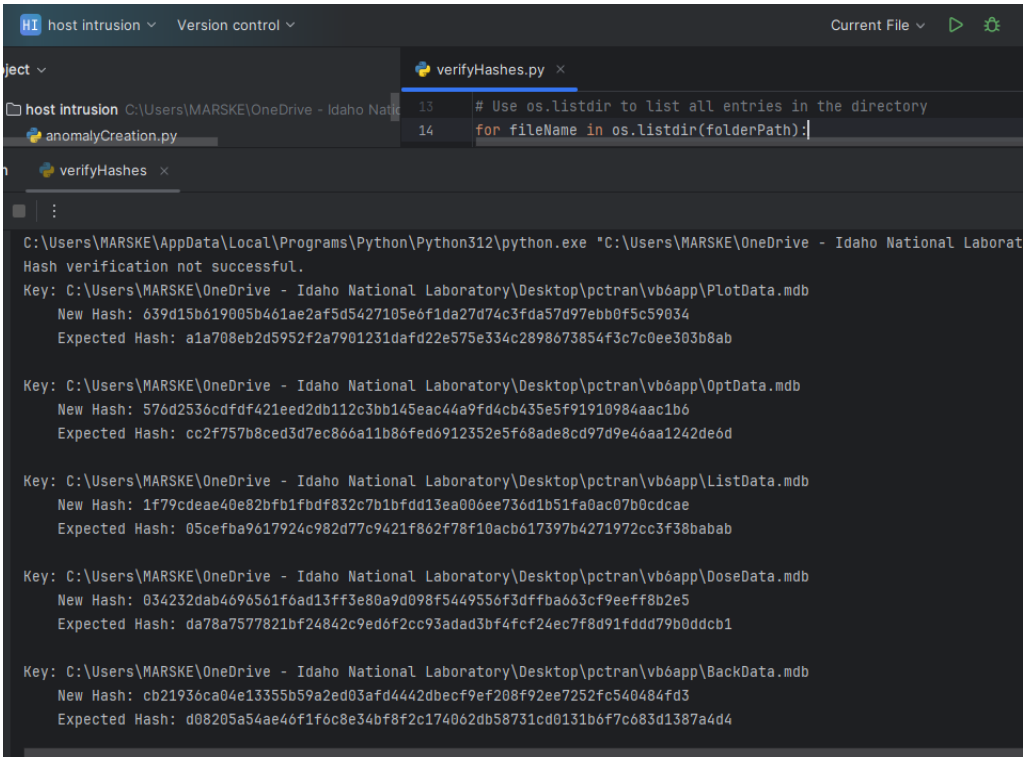

Figure 14. Example of file integrity outcome.

4.2.2 Isolation Forest Results

In order to interpret the results, it is important to understand the different metrics and how they are calculated. These calculations are detailed in Figure 15.

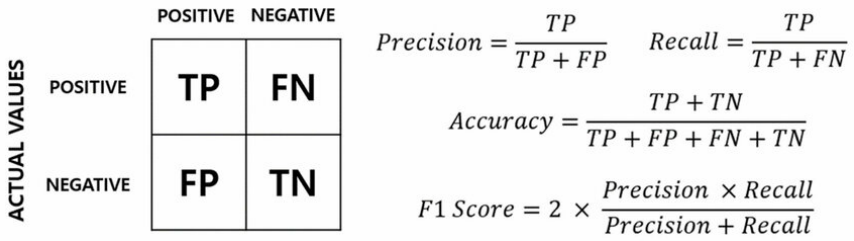

Figure 15. Formulas and interpretations for ML performance metrics.

The results of evaluating the Isolation Forest model with a contamination parameter of 0.2 indicate a high level of accuracy with a robust performance in classifying anomalies. The model achieved an accuracy score of approximately 96.33%, reflecting its effectiveness in distinguishing between normal and anomalous data points, as shown in Figure 16. The classification report further details the model's performance across different metrics. For the class labeled as "0" (normal), the model achieved a precision of 0.95, a recall of 1.00, and an F1-score of 0.98, as displayed in Figure 17, which means the model is highly accurate in identifying normal instances with a perfect recall rate. For the class labeled as "1" (anomaly), the precision was 1.00, indicating all identified anomalies are correctly classified, though the recall was slightly lower at 0.85. This suggests that while the model is effective at identifying anomalies when they are detected, there is room for improvement in capturing all possible anomalies.

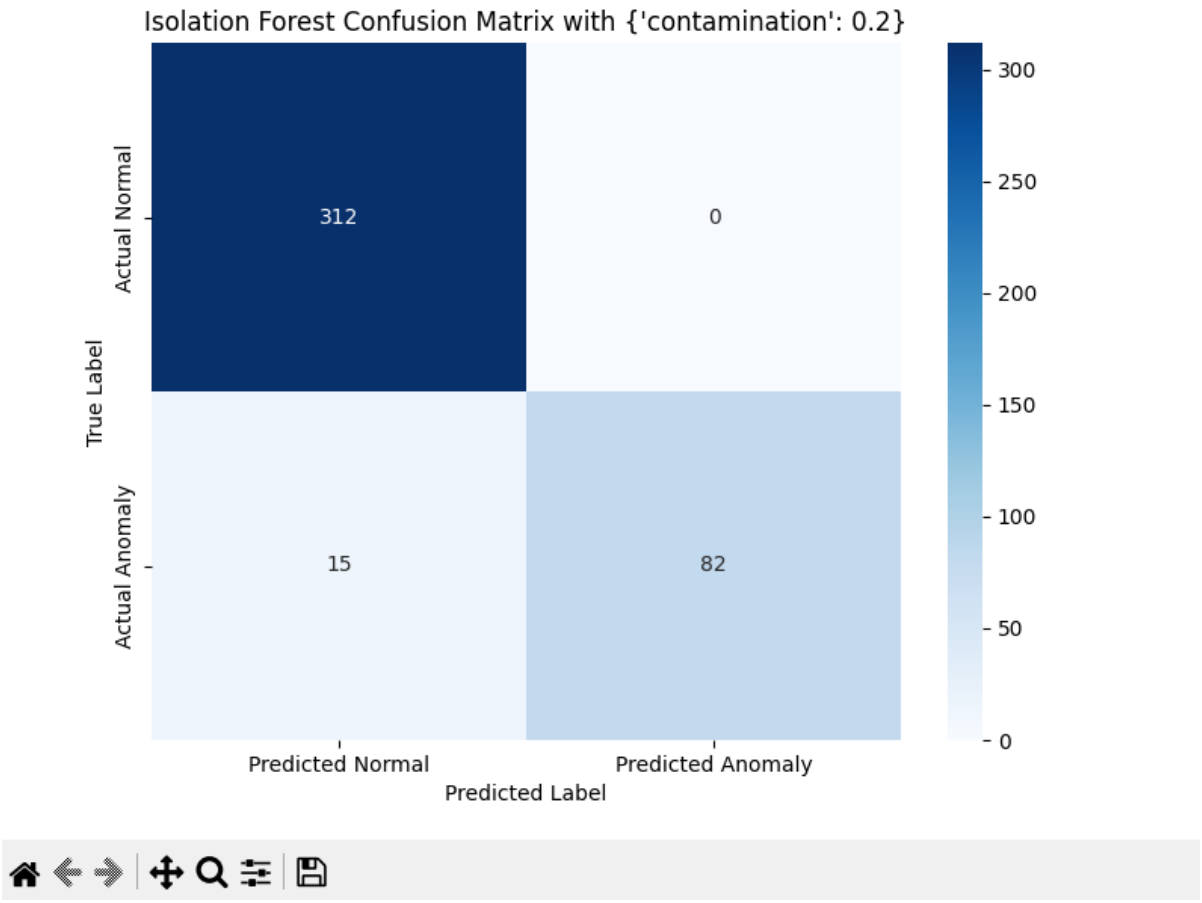

Figure 16. Isolation Forest model confusion matrix with contamination set to 0.2.

```
Evaluating Isolation Forest...
Tuning Isolation Forest with parameters: {'contamination': 0.2}

Isolation Forest Accuracy Score with {'contamination': 0.2}: 0.9633251833740831

Isolation Forest Classification Report with {'contamination': 0.2}:
              precision    recall  f1-score   support

           0       0.95      1.00      0.98       312
           1       1.00      0.85      0.92        97

    accuracy                           0.96       409
   macro avg       0.98      0.92      0.95       409
weighted avg       0.97      0.96      0.96       409
```

Figure 17. Isolation Forest model scores with contamination set to 0.2.

4.2.3 One-Class SVM Results

The results for the One-Class SVM model with the initial parameters of "gamma='scale'" and "nu=0.2" show a strong performance in anomaly detection. This configuration achieved an accuracy score of approximately 91.44%, highlighting its overall effectiveness in distinguishing between normal and anomalous data, as shown in Figure 18. The classification report for these parameters revealed that for the normal class (labeled as "0"), the model has a precision of 0.92, a recall of 0.97, and an F1-score of 0.95, as shown in Figure 19. This indicates the model is very accurate in identifying normal instances, with a high recall suggesting it captures most of the normal cases. For the anomaly class (labeled as "1"), the precision was 0.88 and recall was 0.74, resulting in an F1-score of 0.80. While precision is high, recall is somewhat lower, suggesting the model could improve in identifying all anomalies present in the dataset.

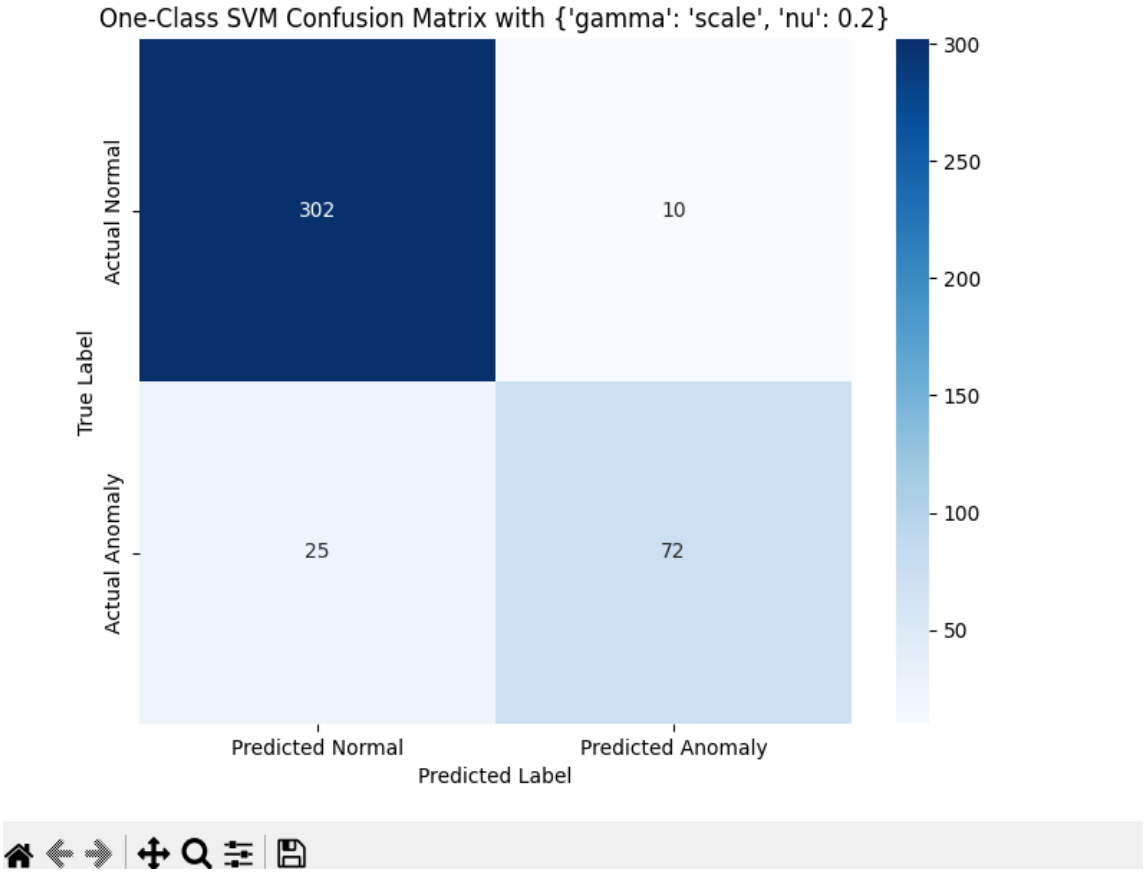

Figure 18. One-Class SVM model confusion matrix with Gamma set to scale and Nu set to 0.2.

```
Tuning One-Class SVM with parameters: {'gamma': 'scale', 'nu': 0.2}

One-Class SVM Accuracy Score with {'gamma': 'scale', 'nu': 0.2}: 0.9144254278728606

One-Class SVM Classification Report with {'gamma': 'scale', 'nu': 0.2}:
              precision    recall  f1-score   support

           0       0.92      0.97      0.95       312
           1       0.88      0.74      0.80        97

    accuracy                           0.91       409
   macro avg       0.90      0.86      0.87       409
weighted avg       0.91      0.91      0.91       409
```

Figure 19. One-Class SVM model scores with Gamma set to scale and Nu set to 0.2.

When the One-Class SVM parameters are tuned to “gamma=‘scale’” and “nu=0.3,” the performance of the model improved significantly, as shown in Figure 20. The accuracy score with these parameters rose to about 93.40%, reflecting better overall performance in anomaly detection. In the classification report with these tuned parameters, the precision for the normal class increased to 1.00, with a recall of 0.92 and an F1-score of 0.95, as shown in Figure 21. This shows the model now perfectly identifies normal instances while maintaining a high recall rate. For the anomaly class, precision scored a 0.79, with a recall of 0.99, and an F1-score of 0.88. Although precision slightly decreased, the recall improved substantially, indicating the model is more effective at detecting anomalies. The macro average scores with the tuned parameters indicate a precision of 0.89, a recall of 0.95, and an F1-score of 0.92. The weighted average scores show even better results with a precision of 0.95, a recall of 0.93, and an F1-score of 0.94. These improvements confirm the tuned One-Class SVM model is highly effective in balancing precision and recall, thus making it a robust choice for anomaly detection in this context.

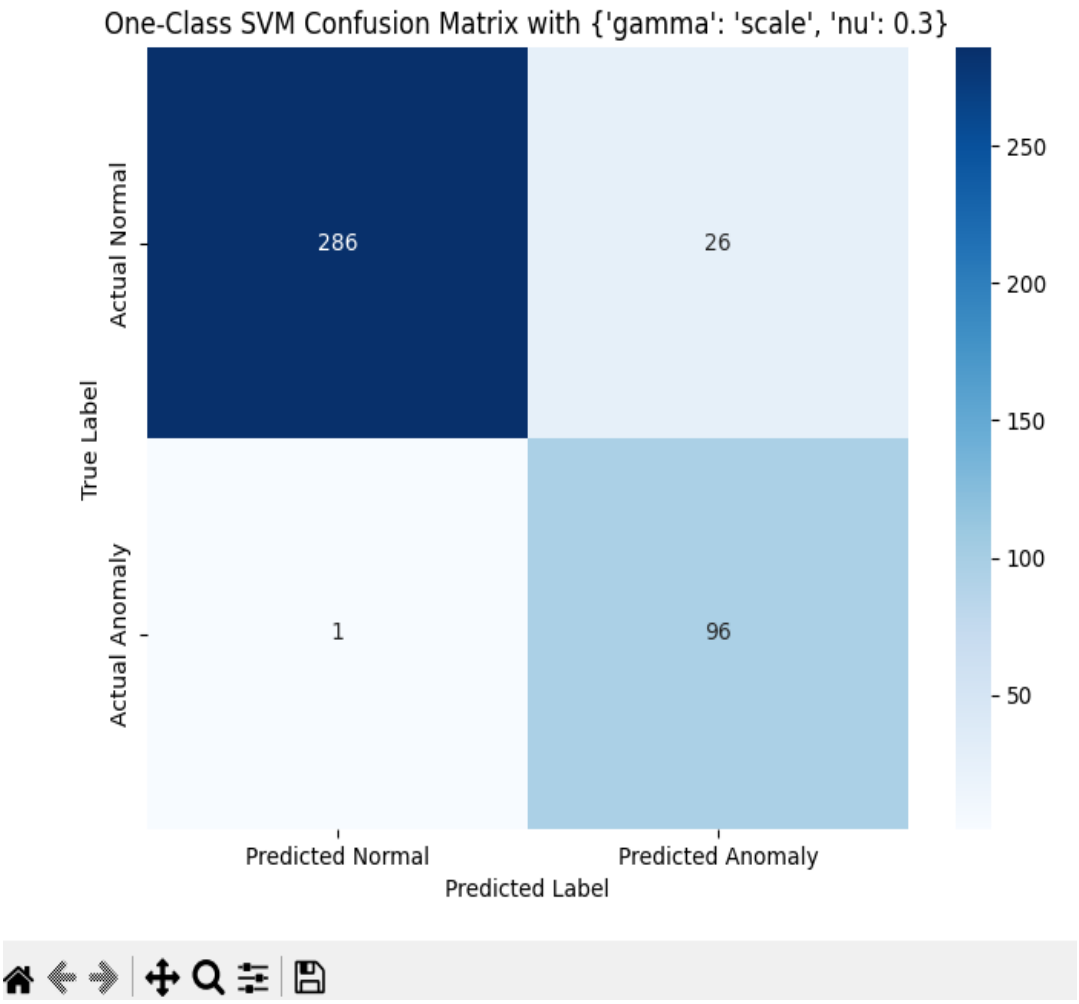

Figure 20. One-Class SVM model confusion matrix with Gamma set to scale and Nu set to 0.3.

```
Tuning One-Class SVM with parameters: {'gamma': 'scale', 'nu': 0.3}

One-Class SVM Accuracy Score with {'gamma': 'scale', 'nu': 0.3}: 0.9339853300733496

One-Class SVM Classification Report with {'gamma': 'scale', 'nu': 0.3}:
              precision    recall  f1-score   support

           0       1.00      0.92      0.95       312
           1       0.79      0.99      0.88        97

    accuracy                           0.93       409
   macro avg       0.89      0.95      0.92       409
weighted avg       0.95      0.93      0.94       409
```

Figure 21. One-Class SVM model scores with Gamma set to scale and Nu set to 0.3.

5. CONCLUSIONS

In this section, final thoughts on the key findings will be presented, as well as the future research directions and a summary of internship experiences.

5.1 Summary of Key Findings

The project successfully created a Linux container environment capable of running the PCTRAN application, which included registering the required DLL and OCX files within Wine's Windows registry. Despite this successful setup, problems arose with the GUI for the application, indicating that additional adjustments or troubleshooting might be necessary to fully support the GUI in this containerized environment.

In addition to setting up the container, a hash verification step was implemented to ensure the integrity of the software. By comparing the "newHashes" and "expectedHashes" dictionaries, the script confirmed whether the files were authentic and unchanged. When the hashes matched, the software was validated as authentic. Discrepancies were identified with detailed information on differing hashes and unique file paths, which helped in resolving any file integrity issues.

The Isolation Forest model, tested with a contamination parameter of 0.2, showed impressive performance with an accuracy of about 96.33%. This model effectively distinguished between normal and anomalous data, achieving a precision of 0.95 and a perfect recall of 1.00 for normal instances. For anomalies, the precision for the model was 1.00, though the recall was slightly lower at 0.85, indicating that while the model accurately identified anomalies, there was still room for improvement in detecting all potential anomalies.

The One-Class SVM model, initially configured with "gamma='scale'" and "nu=0.2," also performed well, with an accuracy of approximately 91.44%. It showed high precision and recall for normal instances, scoring 0.92 and 0.97, respectively. However, its performance for anomalies was less robust, with precision at 0.88 and recall at 0.74. After tuning the One-Class SVM parameters to "gamma='scale'" and "nu=0.3," the model's accuracy improved to around 93.40%. The tuned model achieved perfect precision for normal instances and a notable increase in recall for anomalies, with scores of 1.00 and 0.99, respectively. These improvements demonstrate the tuned One-Class SVM model is highly effective in balancing precision and recall, thereby making it a robust choice for anomaly detection.

5.2 Future Research Directions

Future research would address the current challenges, particularly the GUI issues within the containerized environment. With more time, alternative configurations or additional settings for Wine may resolve these problems and improve compatibility, or instead of using a container, using a VM could be explored. Further investigation into the performance of anomaly detection models is also essential. This could involve testing different algorithms and adjusting parameters to enhance detection accuracy and handle edge cases more effectively. Incorporating advanced techniques, such as deep learning or additional features, might offer better insights and improve model performance. Additionally, refining the hash verification process to include more detailed checks could enhance software integrity validation. Expanding research to include larger datasets would help evaluate the robustness and scalability of the model in practical applications.

5.3 Summary of Internship Experiences

Throughout this internship, the opportunity and pleasure was given of attending a variety of presentations and expos detailing different initiatives and projects currently being worked on at INL. One experience that stood out to me was the AI-ML Symposium. During this symposium, it was demonstrated that ML can be applied to control NPPs using two main approaches: (1) modeling the plant; or (2) employing reinforcement learning for direct control. While the dynamics required for control are less complex than those needed for modeling, regulatory requirements for determinism, simplicity,

explainability, and verifiability pose challenges. ML algorithms can be stochastic and difficult to interpret, leading to variability and complexity. A hierarchical approach addresses these issues by combining traditional high-performance control methods with ML for supervisory control and digital twins. This approach allows for the plant to be controlled by proven methods, while ML aids in adapting and optimizing control strategies without directly affecting the plant's operation, ensuring verified performance even as plant conditions change.

Another experience that was very enjoyable was the Digital Twin presentation by Dr. Mohammad Abdo. In this presentation, Dr. Abdo explained that a digital twin is a virtual replica of a physical component, system, or process using real-time sensory data and advanced analytics—including ML, deep learning, and AI—to enable adaptive learning, inference, and decision-making with minimal human intervention. Its purpose is to ensure continuous, dynamic communication between design, manufacturing, and quality processes. Digital twins require ongoing data streaming from their physical counterparts and come in various types: component/part twin, asset twin, system twin, and process twin. To be effective, a digital twin needs readiness in several areas: physical space (designs, schematics, actuators), digital space (GUIs, analytics), data space (storage, connectivity), and services space (prediction, monitoring, control).

I am very grateful to have had the opportunity to learn so much and gain so much experience through this internship.

6. REFERENCES

1. MicroSim Technology. (2007). "PCTRAN." MicroSim Technology webpage. Available at: http://www.microsimtech.com/pctran/ (accessed August 10, 2024).
2. Docker, Inc. (2024). "Get started with Docker." Docker webpage. Available at: https://www.docker.com/get-started/ (accessed August 10, 2024).
3. GeeksforGeeks. (2023). "Difference between HIDS and NIDS." August 4, 2024. GeeksforGeeks webpage. Available at: https://www.geeksforgeeks.org/difference-between-hids-and-nids/ (accessed August 10, 2024).
4. Li, V. (2019). "Differences between classification and anomaly detection." Medium website. Available at: https://medium.com/@ljyds/differences-between-classification-and-anomaly-detection-f0b4b4b990e2 (accessed August 10, 2024).
5. Po, L.-C. (2008). "PC-based simulator PCTRAN for advanced nuclear power plants." *2008 International Congress on Advances in Nuclear Power Plants (ICAPP '08)*, 8–12 June 2008, Anaheim, CA, USA. Available at: http://www.microsimtech.com/pctran/ICAPP8.html (accessed August 10, 2024).
6. GitHub. (2023). *"*DikeDataset*."* June 25, 2023. User iosifache. GitHub. Available at: https://github.com/iosifache/DikeDataset/tree/main (accessed August 10, 2024).
7. Gulati, A. (2024). "Anomaly detection with machine learning overview." April 23, 2024. KnowledgeHut. Available at: https://www.knowledgehut.com/blog/data-science/machine-learning-for-anomaly-detection. (accessed August 10, 2024).
8. Meng, Y. (2023). "What is AI (artificial intelligence)?" University of Illinois Chicago, College of Engineering. January 10, 2023. Available at: https://meng.uic.edu/news-stories/ai-artificial-intelligence-what-is-the-definition-of-ai-and-how-does-ai-work/ (accessed August 10, 2024).