Student Full Name: Ghassen Jemiai
Student Id: 7195718

CPU Anomaly Detector Project Report

1. Problem Statement

Modern web applications built on React-based frameworks such as Next.js have recently been targeted by runtime vulnerabilities, notably CVE-2025-55182 (React2Shell). This exploit enables attackers to inject malicious code at runtime, allowing compromised servers to be used for unauthorized purposes such as cryptocurrency mining or uncontrolled background process execution.

A common symptom of such attacks is abnormally high and sustained CPU usage, which directly impacts service availability and system stability. If not detected early, the compromise may require costly corrective actions such as full server migration, forensic analysis, and service redeployment.

In my case, an exploitation of this vulnerability at my day job resulted in four hours of service unavailability, during which the system had to be inspected, cleaned, and redeployed. Although the attack was eventually mitigated, the absence of early detection significantly increased downtime and operational impact.

This project addresses the need for early runtime anomaly detection by monitoring CPU behavior and automatically identifying deviations from normal operating patterns. The objective is to reduce detection latency and limit the consequences of resource-abuse attacks before they escalate into critical service outages.

2. Monitoring Design and Architecture

In order to address the delayed detection of resource-abuse attacks described in the previous section, a continuous runtime monitoring mechanism is required. Since such attacks primarily manifest through abnormal CPU behavior, the proposed solution relies on system-level monitoring to capture deviations from normal operating patterns. This section presents the design and architecture of the monitoring component responsible for collecting, processing, and structuring runtime metrics for anomaly detection.

The monitoring component is designed as a lightweight, modular system responsible for continuously collecting runtime system metrics and transforming them into structured features suitable for anomaly detection. The implementation is based on the `psutil` Python library and encapsulated within a dedicated `SystemMonitor` class.

2.1 Architecture Overview

At the operating system level, the monitor interfaces with core system resources, including **CPU, memory, disk, network, and process statistics**. These raw metrics are accessed through `psutil`, which provides a platform-independent and low-overhead interface to system counters.

The `SystemMonitor` class acts as the central abstraction layer. During initialization, it captures baseline counter values required for rate-based calculations (e.g., I/O throughput, context switches per second). At runtime, the monitor periodically executes a sampling routine that collects a snapshot of system activity.

The collected data flows through three main outputs:

- a **feature vector** used by the anomaly detection model at runtime,
- a **CSV logging mechanism** used to build labeled datasets for offline training,
- an optional **background thread** enabling continuous, non-blocking data collection.

This design ensures a clear separation between data acquisition, feature extraction, and machine learning components.

2.2 Collected Metrics and Feature Construction

Each sampling cycle produces a structured set of metrics, including:

- **CPU metrics**: overall utilization percentage, per-core usage
- **Memory metrics**: percentage usage and available memory
- **Disk I/O**: read and write rates
- **Network I/O**: sent and received rates
- **Process statistics**: process count and thread count
- **System activity rates**: context switches per second and interrupts per second

Why collect everything?

1. **CPU stress affects other metrics** - context switch spike, memory may fluctuate
2. **Richer feature set** - ML model can learn correlations we might not expect
3. **Future extensibility** - same data can detect memory/disk/network anomalies later, as we wanted to keep things simple for this project
4. **Minimal overhead** - `psutil` calls are very fast (~1-2ms total)

To ensure temporal consistency, raw counters are converted into rates using a delta-over-time approach. The final output of each sampling cycle is a **fixed-size feature vector (24 features)**, making it suitable for time-series–based anomaly detection algorithms.

2.3 Data Flow and Sampling Strategy

The monitoring process operates at a fixed sampling interval of one second. During each interval, raw system counters are collected, transformed into rate-based values, and aggregated into the final feature vector. This fixed-rate sampling ensures uniform temporal spacing, which is required by several of the anomaly detection algorithms evaluated in this project.

2.4 Design Rationale

The monitor was designed with the following constraints in mind:

- **Low runtime overhead**, avoiding additional system stress
- **Real-time compatibility**, enabling immediate anomaly detection
- **Extensibility**, allowing future inclusion of additional metrics
- **Framework independence**, ensuring applicability beyond a specific application stack

By combining multiple system-level indicators into a single feature vector, the monitor captures both instantaneous load and dynamic behavior, which is essential for detecting resource-abuse attacks that may not manifest as simple CPU spikes.

3. Selection of Monitored Indicators

The selection of monitored indicators was performed using a **domain-driven approach**, guided by the characteristics of the target anomaly, namely **CPU stress caused by resource-abuse attacks**. Rather than relying on automated feature selection techniques, the process focused on identifying system behaviors that are known to change during abnormal CPU activity.

3.1 Definition of the Target Anomaly

The target anomaly addressed in this project is **CPU stress**, which may result from cryptomining activity or uncontrolled process execution following a system compromise. The selection process therefore begins by identifying which system behaviors are expected to change when CPU stress occurs.

This question guided the entire indicator selection process:
“What system-level metrics exhibit abnormal behavior during sustained or irregular CPU load?”

3.2 Primary Indicators (Direct Effects)

CPU stress directly impacts computational activity and process scheduling. Based on this observation, the following **primary indicators** were selected:

- **Overall CPU usage (`cpu_percent`)**, which increases sharply under stress conditions
- **Per-core CPU usage (`cpu_core_0..11_percent`)**, allowing detection of whether stress affects all cores or only a subset
- **Context switches per second**, reflecting increased thread scheduling
- **Interrupts per second**, capturing elevated hardware and timer activity
- **Thread count**, as stress workloads often spawn multiple worker threads

These indicators represent the most direct and reliable signals of CPU stress and form the core detection features of the system.

3.3 Secondary Indicators (Indirect Effects)

In addition to direct CPU effects, stress workloads may produce **secondary impacts** on other system resources. The following indicators were included to capture these indirect effects:

- **Memory usage**, as worker processes allocate additional memory
- **Number of running processes**, reflecting process spawning behavior
- **CPU frequency**, which may increase due to turbo boost under sustained load

While these indicators are not primary detection signals, they provide complementary information that helps distinguish malicious stress from short-lived or benign CPU spikes.

3.4 Baseline and Context Indicators

To improve robustness and provide contextual baselines, additional indicators related to disk and network activity were retained:

- Disk read/write rates
- Network sent/received rates
- Available memory in absolute terms

These metrics are not expected to correlate strongly with CPU stress but help establish a baseline and support future extension of the detection system to other anomaly types.

3.5 Feature Engineering Strategy

Many system metrics are exposed as cumulative counters. To make them suitable for anomaly detection, raw counters were transformed into **rate-based features** using a delta-over-time approach:

$$\text{rate} = \text{current value} - \text{previous value}/\Delta t$$

This transformation was applied to indicators such as context switches, interrupts, disk I/O, and network traffic. The final outcome of this process is a **fixed-size feature vector of 24 features**, combining instantaneous values and rate-based metrics.

3.6 Rationale for Per-Core CPU Metrics

Per-core CPU usage was explicitly included to distinguish between different stress patterns. Under normal conditions, CPU usage varies across cores. In contrast, CPU stress typically results in **simultaneous saturation of all cores**, while single-threaded anomalies affect only a limited subset. Including per-core metrics therefore improves the system's ability to discriminate between benign and anomalous workloads.

3.7 Absence of Automated Feature Selection

Automated feature selection methods such as PCA or correlation-based reduction were not applied during initial feature selection. This decision was justified by:

- The limited and manageable feature set size
- Strong domain knowledge of CPU stress behavior
- The ability to validate feature relevance post-training using model-based importance measures

4. Training Dataset Generation and Anomaly Injection

The training dataset was generated using a controlled and reproducible pipeline that combines recordings of normal system behavior with CPU stress–based anomaly injection. The objective of this process is to create labeled time-series data that reflects both normal operation and abnormal CPU usage patterns caused by resource-abuse scenarios.

4.1 Normal Behavior Collection

Normal CPU behavior was collected using the monitoring system without introducing any artificial load. Data collection was performed by running the script `collect_normal_data.py` for a fixed duration (typically 10 minutes), during which the system was used under normal conditions such as idle periods, web browsing, typing, and file access.

System metrics were sampled once per second, resulting in approximately 600 samples per collection session. All collected samples were labeled as *normal (label = 0)*. The resulting data was stored in CSV format and represents baseline CPU and system behavior, including natural fluctuations caused by operating system scheduling and benign background processes.

4.2 Anomaly Simulation

Anomalous behavior was generated through explicit CPU stress injection rather than reproducing real exploits. This approach was chosen because real attack traces are difficult to obtain and unsafe to reproduce, while CPU stress accurately reflects the observable effects of attacks such as cryptomining or uncontrolled process execution.

CPU stress was generated using a multiprocessing-based injector to bypass Python's Global Interpreter Lock (GIL). Multiple worker processes were spawned, each executing CPU-intensive operations in parallel. The number of worker processes was matched to the number of logical CPU cores, resulting in near-saturation of all cores.

Several CPU-intensive operations were used to introduce variability in the stress patterns, including arithmetic loops, matrix multiplication, and prime number computation. This allowed the generation of both sustained and computationally diverse stress workloads.

4.3 Anomalous Data Collection Process

Anomaly data collection followed a structured three-phase timeline:

1. **Warm-up phase**: normal behavior recorded before stress injection
2. **Stress phase**: CPU stress applied for a fixed duration
3. **Cooldown phase**: system recovery after stress termination

During the warm-up and cooldown phases, samples were labeled as *normal (label = 0)*, while samples collected during the stress phase were labeled as *anomalous (label = 1)*. This design captures realistic transitions between normal and abnormal states, allowing the models to learn boundary behavior rather than relying on abrupt on/off patterns.

4.4 Training Set Preparation

Normal and anomalous CSV files were combined using the training preparation script. All data files were concatenated, shuffled, and class-balanced to avoid bias toward the majority class. Undersampling was applied when necessary to ensure an equal number of normal and anomalous samples.

The final training dataset consists of a balanced set of labeled feature vectors stored in a single CSV file, which is then used for training, testing, and comparing anomaly detection algorithms.

5. Algorithms, Training, and Comparison

This section describes the selection, training, and evaluation of the anomaly detection algorithms. Multiple algorithm families were deliberately explored to assess how different learning paradigms behave when applied to CPU-based anomaly detection. The objective was not only to identify the best-performing model, but also to understand the trade-offs between accuracy, robustness, and suitability for runtime deployment.

5.1 Algorithm Selection Motivation

Two main categories of algorithms were evaluated: **supervised classification models** and **unsupervised anomaly detection models**.

Supervised models were included because the dataset generated in this project is explicitly labeled, enabling direct learning of the boundary between normal and anomalous behavior. These models are expected to perform well when anomaly patterns are known and representative.

The supervised algorithms evaluated are:

- **Random Forest**
- **Support Vector Machine (SVM)**
- **Gradient Boosting**
- **XGBoost**
- **Logistic Regression**
- **Multilayer Perceptron (MLP)**

In parallel, **unsupervised models** were also evaluated to simulate a more realistic deployment scenario where anomaly labels may not be available:

- **Isolation Forest**
- **One-Class SVM**

This comparison allows the evaluation of whether supervised learning significantly outperforms unsupervised approaches when labels are available, and whether unsupervised models remain viable alternatives.

Each algorithm was selected for specific reasons. Tree-based models (Random Forest, Gradient Boosting, XGBoost) are well suited for tabular data and non-linear relationships. SVMs are effective in moderate-dimensional spaces and can model complex decision boundaries. Logistic Regression serves as a simple and interpretable baseline, while MLPs are capable of learning more complex non-linear patterns. Isolation Forest and One-Class SVM are classical anomaly detection techniques designed to learn the structure of normal data without relying on anomaly labels.

5.2 Training and Testing Procedure

All models were trained using a consistent pipeline to ensure fair comparison. The combined dataset was split into **70% training data and 30% test data**, using a stratified split to preserve the balance between normal and anomalous samples.

Before training, feature scaling was applied using standard normalization. This step is essential for algorithms such as SVM and neural networks, which are sensitive to feature magnitude. The

scaler was fitted exclusively on the training set and then applied to the test set to prevent data leakage.

Model training was performed only on the training set. The test set was strictly reserved for final evaluation and was never used during training or parameter tuning.

5.3 Hyperparameter Tuning and Cross-Validation

To avoid biased results due to arbitrary parameter choices, **hyperparameter tuning was performed using GridSearchCV**. For each supervised model, a predefined parameter grid was explored, covering key parameters such as the number of estimators, tree depth, learning rate, kernel type, and regularization strength.

Each parameter combination was evaluated using **5-fold cross-validation** on the training set. The training data was split into five folds, with four folds used for training and one fold for validation at each iteration. The final cross-validation score corresponds to the average performance across all folds.

This approach ensures that model selection is robust to variations in the training data and reduces the risk of overfitting.

5.4 Evaluation Metrics and Selection Criterion

Model performance was evaluated on the test set using several metrics derived from the confusion matrix: accuracy, precision, recall, F1-score, and ROC-AUC.

Among these metrics, the **F1-score was selected as the primary selection criterion**. In anomaly detection, both false positives and false negatives are costly. False positives generate alert fatigue and reduce trust in the monitoring system, while false negatives allow attacks to remain undetected. The F1-score provides a balanced measure by jointly optimizing precision and recall, making it more suitable than accuracy alone.

ROC-AUC was used as a complementary metric to assess the overall discriminative ability of each model across different thresholds.

5.5 Comparative Results and Discussion

The experimental results show a clear distinction between supervised and unsupervised approaches. Supervised models consistently outperform unsupervised models across all evaluation metrics. This confirms that when labeled anomaly data is available, supervised learning provides a significant advantage in detecting CPU stress patterns.

Among the supervised models, **Random Forest achieved the highest overall performance**, combining strong F1-score, high recall, and stable precision. Gradient Boosting and XGBoost also performed well but showed slightly higher sensitivity to parameter tuning. Logistic

Regression and MLP provided reasonable results but were less robust to complex non-linear patterns.

Unsupervised models such as Isolation Forest and One-Class SVM achieved lower performance, particularly in recall. While these models remain useful in scenarios where labeled anomalies are unavailable, their results indicate that they are less effective at capturing the full range of CPU stress behaviors present in the dataset.

Based on this comparative analysis, **Random Forest was selected for runtime deployment**, as it offers the best balance between detection performance, robustness to noise, interpretability through feature importance, and low inference cost.

6. Runtime Integration and Final Anomaly Detector

This section describes the integration of the trained anomaly detection model into a real-time monitoring system. The objective of this final step is to demonstrate how offline-trained models can be reliably deployed for **continuous runtime anomaly detection** with low overhead and consistent behavior.

6.1 Runtime Detector Initialization

At startup, the runtime detector loads all components required to ensure consistency between training and deployment. The trained machine learning model and the feature scaler are both loaded from disk using serialized artifacts produced during training.

The runtime detector is composed of three main elements:

- the **trained anomaly detection model**,
- the **feature scaler** fitted during training,
- the **system monitor**, responsible for collecting live system metrics.

Loading the same scaler used during training is a critical design decision. Without this step, runtime feature distributions would differ from training data, leading to incorrect predictions even for valid inputs.

6.2 Online Detection Loop

The runtime detection process operates in a continuous loop with a fixed sampling interval of one second. Each iteration of the loop follows a deterministic sequence of operations.

First, the system monitor collects the current system metrics, producing a structured snapshot of CPU and system state. These raw measurements are then converted into a fixed-order feature vector matching the feature layout used during training.

Next, the feature vector is normalized using the previously loaded scaler. This guarantees that runtime data is transformed into the same feature space as the training data.

The scaled features are then passed to the trained model to generate a prediction. For supervised models, the output is directly interpreted as *normal* or *anomalous*. For unsupervised models, whose outputs differ in convention, a conversion step is applied to map model-specific outputs to a unified binary anomaly label.

6.3 Output and Logging Strategy

Detection results are immediately communicated through two output channels.

The first channel is a **real-time console output**, which displays the system status at each sampling step. Color-coded messages are used to improve readability and situational awareness, with normal states displayed in green and anomalous states displayed in red. This design allows operators to visually identify abnormal behavior as soon as it occurs.

The second channel is a **persistent CSV log file**, where each detection is recorded together with its timestamp and associated system metrics. This log serves multiple purposes: post-analysis of detector behavior, performance auditing, and traceability for security investigations.

6.4 Handling Practical Runtime Constraints

Several practical considerations were addressed to ensure robustness during continuous execution. The feature extraction process enforces a strict and consistent feature order to prevent misalignment between runtime data and model expectations. Missing or invalid values are sanitized to avoid runtime crashes.

The detection loop was designed to be lightweight, with inference performed on a single feature vector per second. This ensures that the detector introduces minimal overhead and does not interfere with the system it is monitoring.

6.5 Contribution to the Overall System Goal

By integrating the trained model into a real-time monitoring loop, the project achieves its primary objective: **early detection of abnormal CPU behavior at runtime**. The detector continuously observes system activity, applies learned decision boundaries, and reports anomalies as soon as they emerge.

This integration demonstrates how offline machine learning techniques can be effectively transformed into a practical runtime monitoring solution, capable of reducing detection latency and limiting the impact of resource-abuse attacks.

7. Conclusion and System Limitations

This project presented the design and implementation of a **CPU-based runtime anomaly detection system** aimed at early identification of resource-abuse attacks such as cryptomining and CPU exhaustion. Starting from a real-world incident, the work followed a complete methodology covering monitoring design, feature selection, dataset generation, algorithm evaluation, and runtime deployment.

The results demonstrate that CPU behavior, when carefully monitored and modeled, provides a strong signal for detecting abnormal system activity. By comparing multiple supervised and unsupervised learning algorithms, the study showed that supervised models—particularly tree-based approaches—offer superior detection performance when labeled data is available. The final system successfully integrates offline-trained models into a real-time detection loop with low overhead, enabling continuous monitoring and immediate feedback.

Overall, the project validates the feasibility of deploying machine learning–based anomaly detection for runtime monitoring and highlights the importance of methodological rigor at each stage of the pipeline.

7.2 System Limitations

Despite its effectiveness, the proposed system has several limitations that should be acknowledged.

Single anomaly type.
The detector is trained exclusively to identify CPU stress anomalies. As a result, it is effective for detecting cryptomining, CPU-based denial-of-service attacks, and similar resource-exhaustion scenarios, but it does not detect other attack types such as memory leaks, disk I/O attacks, network-based intrusions, privilege escalation, or data exfiltration. This limitation is a direct consequence of the training data and feature scope.

Lack of contextual awareness.
The system cannot reliably distinguish between malicious CPU stress and legitimate high CPU usage caused by benign workloads such as software compilation, video rendering, system updates, or antivirus scans. Both scenarios may produce similar system-level patterns, potentially leading to false positives and alert fatigue.

Static model behavior.
The deployed model is static and does not adapt to changes in system behavior over time. As software, hardware, or usage patterns evolve, the learned baseline may become outdated. This can result in false alarms or missed detections unless the model is periodically retrained or extended with online learning capabilities.

Training data dependency.
Model performance depends heavily on the quality and diversity of the training dataset. Normal behavior was collected over a limited time window, and anomalies were synthetically generated. While this approach is controlled and reproducible, it may not fully capture the variability of real-world workloads or stealthy attack patterns, limiting generalization.

Sampling rate constraints.
The monitoring system samples metrics once per second. While this choice balances detection capability and overhead, very short-lived anomalies occurring between samples may go undetected. Increasing the sampling rate would improve sensitivity but at the cost of higher system overhead.

Binary classification output.
The detector produces a binary output indicating only whether behavior is normal or anomalous. It does not classify the type of anomaly or provide semantic explanations, limiting its usefulness for automated incident response.

Cold-start effects.
Rate-based features require previous values for accurate computation. During the initial samples after startup, feature values may be inaccurate, potentially triggering false detections.

Runtime overhead and platform dependence.
Although lightweight, the monitor itself consumes system resources, which may slightly influence measurements on low-resource systems. Additionally, the system was primarily tested on Windows, and metric behavior may differ across operating systems despite the use of cross-platform libraries.

Lack of integrated alerting.
The current implementation reports anomalies through console output and log files only. Without persistent alerting mechanisms such as email notifications, dashboards, or automated responses, anomalies may go unnoticed if the system is not actively monitored.

7.3 Future Work

Future improvements could address these limitations by extending the monitoring scope to multiple anomaly types, incorporating contextual information, introducing adaptive or online learning techniques, increasing sampling flexibility, and adding alerting and response mechanisms. Cross-platform validation and deployment would further enhance the robustness and applicability of the system.