General process detection through side channel characterization

Michael A. Taylor, Aviraj Sinha, Eric C. Larson & Mitchell A. Thornton

To cite this article: Michael A. Taylor, Aviraj Sinha, Eric C. Larson & Mitchell A. Thornton (05 Aug 2024): General process detection through side channel characterization, Journal of Cyber Security Technology, DOI: 10.1080/23742917.2024.2384172

To link to this article: https://doi.org/10.1080/23742917.2024.2384172

Published online: 05 Aug 2024.

JOURNAL OF CYBER SECURITY TECHNOLOGY
https://doi.org/10.1080/23742917.2024.2384172

General process detection through side channel characterization

Michael A. Taylor, Aviraj Sinha, Eric C. Larson and Mitchell A. Thornton

Lyle School of Engineering, Southern Methodist University, Dallas, TX, USA

ABSTRACT
In this study, we demonstrate the feasibility of using physical sensors in System-on-a-Chip (SoC) for real-time security monitoring by detecting and characterizing different process types, such as file I/O, CPU/ALU-intensive tasks, network I/O and virtualization. We present models that use sensor data for binary classification to determine whether a specific process category is active or inactive. Analyzing the detection results, we determine the importance of each sensor in identifying process types and providing insights into process behaviors and sensor impacts. We developed adaptive ensemble classifiers to accommodate varying load conditions, enhancing detection accuracy across diverse operational scenarios, including regular background activities that simulate real-world conditions. Our results show effective detection of file I/O and CPU/ALU-intensive processes under various loads. Virtualization processes are accurately detected under light loads but show a moderate accuracy decline under heavier conditions. Network I/O detection faces challenges due to fewer relevant sensors. Our ability to predict process categories consistently and with high performance allows us to discover behaviors of various malicious activities. This research underlines the efficacy of sensor-based analysis for reliable and adaptable real-time process monitoring, demonstrating the feasibility of using sensor data for security purposes in various environmental conditions.

ARTICLE HISTORY
Received 30 April 2024
Accepted 17 July 2024

KEYWORDS
Side channel; physical sensors; malware detection and characterization

1. Introduction

Computer systems have become extremely complex in order to keep up with the demands of modern businesses and consumers. This complexity can be seen in the number of functions and operations a system must perform at a given time while also having to account for such things as large-scale timing and concurrent data usage. Given some knowledge of how one of these complex systems works, it has been shown that it is possible to determine the actions being performed by a system and the sensitive information being

CONTACT Aviraj Sinha avirajs@smu.edu Lyle School of Engineering, Southern Methodist University, 6425 Boaz Lane, Dallas, TX 75205, USA

used [1–3]. Often, this type of information is overlooked as innocuous because it is a product of the regular operation of the system rather than a specific flaw that exists in the algorithms being performed by the system. However, as systems become more complex and data becomes transmitted at higher rates, a specific fingerprint begins to form in the physical system, which is even easier to detect when the processes being performed operate in a predictable and repeating manner.

The concept of measuring the performance of system hardware to exploit physically unique operating characteristics is well established. Physically Unclonable Functions (PUFs) act as digital fingerprints by leveraging the difference in performance due to the semiconductor manufacturing processes [4]. Characteristics such as timing can be used to reliably differentiate between and uniquely identify semiconductor devices manufactured from the same die [5]. PUFs are very effective at the task of authentication despite only measuring simple operations in a single component. Even the most complex computer systems are essentially large collections of physical components carrying out their own simple tasks in a very organized and cooperative effort to perform a given task. If we were to measure the physical state of one of the individual components during the execution of a specific process, it is unlikely that we would be able to use that knowledge and say with any real degree of certainty when the process is present on a system in the future. However, if we also measure a large number of other system components simultaneously and combine that knowledge, the unique physical state of the system as a side effect of performing a specific process comes into focus. The extremely sensitive operational nature of computational components, such as the semiconductor devices that allow for PUFs, can also be leveraged to create a multimodal approach to process detection on a system by exploiting the unique physical state of the system required for operation.

Characterizing process information through multimodal physical sensor behavior, rather than relying solely on either hardware counters or performance logs, offers invaluable advantages in enhancing security. One key benefit lies in the necessity for redundant information, which is essential for countering potential spoofing attempts. Side channel information provides defenders with real-time, authentic data that is far less prone to attackers' attempts at manipulation. When restricted to and managed by a defender rather than the user, sensor data remain resistant to spoofing. Additionally, employing side channel information offers users the freedom to withhold sensitive privacy details, which might be disclosed in performance logs and hardware counters. Furthermore, if both side channel information and log data are available, the synergy between the two proves advantageous. When these sources are obtained together, matching sensor data with logs and hardware information enhances security [6]. In particular, using side-channel sensor detectors together with traditional signature-based detectors is advantageous as a twotiered detection implementation since the attacker would then have to perform the complex task of spoofing both data sources simultaneously.

In previous works, specific sensors within modern System-on-Chip (SoC) with multiple CPU cores were employed to collect real-time data. This data was then input into a machine learning classifier to identify the presence of ransomware instances [7,8]. The previous approaches involved training the classifier with processes that exhibited ransomware behavior. The ransomware behavior is simulated by operations that identify victim files for encryption followed by CPU-intensive encryption operations [9–11]. This previous work shows potential for the side-channel data to be used for the detection of ransomware. In contrast, this study focuses on utilizing physical sensor side channels to detect arbitrary target processes. A binary output indicates whether the target process of a specific category is running or not. Thus, rather than training models to detect a specific anomaly such as ransomware, this work focuses on general process detection for different categories, which can then be utilized for the characterization of anomalous activity. It is worth noting that we believe this general characterization of activity can then be used for ransomware detection since, from a broader perspective, ransomware activities can be categorized as initial file I/O activity followed by CPU-intensive operations during encryption. Other previous work involves foundational general process detection in Taylor et al. [3]. However, the research on general process detection focuses on the accuracy testing of a single model without the domain-specific detection rate and sensor analysis.

This research includes a comprehensive evaluation of the effectiveness of utilizing Physical Sensor Side Channels (PSSC) for detecting various general processes. The success of PSSC-based detection and characterization hinges greatly on the presence or absence of appropriate sensors within the host architecture. For instance, if a specific host architecture lacks sensors inside or in proximity to the network interface circuitry (NIC), detecting a process that involves extensive communication with the network becomes considerably more challenging using PSSC. This study involves feature analysis of the different sensors and the effectiveness of their usage to detect different processes. In addition, real-time monitoring of specific statistical detection metrics is obtained and compared for a wide range of efficient machine learning models. By integrating these elements, the research advances the field of malware detection and other critical anomalous activities by utilizing sensor-driven insights for real-time monitoring. In our work, we extend previous general process detection by first providing a more robust methodology for testing the detection four process categories under various realistic loads, then incorporating testing of multiple models, including eight different ML models and their ensemble variants with five different metrics. After obtaining test results, we improve feature analysis that attributes process detection to specific sensors using a feature importance score. We then examine the implications of such general process detection performance by visualizing real-time process classifications. To perform this experiment, we collect physical sensor data from multiple integrated system sensors at regular intervals in order to create a vector that represents the physical state of the system for each sampling period. Recording these data and labeling periods in which a system is executing a specific process and also periods where it is not executing the specific process allows for the creation of a binary prediction model using machine learning. This study conducts detailed process detection at a granular level for each set of sensor samples, leveraging sensor data to distinguish between processes heavily skewed towards file I/O activity, network I/O, or CPU/ALU-intensive tasks. By identifying specific sequences of process activities through sensors, this study establishes fundamental building blocks for detecting malicious applications through physical side channels. Specifically, these building blocks consist of the binary detection outputs of the multiple process categories, which can then be used in the second stage of detecting specific anomalies.

This approach of process detection is useful to detect both malware and other hardware anomalies since it allows a defender to characterize desired normal behavior and anomalous behavior based on dominant sensor activities. For instance, detecting a sequence starting with file I/O intensive operations followed by CPU/ALU-intensive tasks could signify processes searching for target files and processing them through intensive computations, such as encryption phases seen in many ransomware instances [8]. There are many other types of attacks that can be linked to different process category behaviors. Distributed Denial of Service (DDOS) attacks can be identified by processes using network IO and CPU utilization spikes, which can be identified through our detection models. Especially if the high CPU usage correlates with network IO, then there may be an attempt to deny access to the server [12]. Other attacks include data exfiltration. In this case, file/IO intensive processes would be used to download a target file, and then possibly network IO processes would be heavily utilized to export the data elsewhere. Privilege escalation attacks often involve VM processes to work around typical security methods to get higher privileges. Many other attacks, such as trojans, botnets and general misuse of resources of any kind, can be found through a certain pattern of file, network and CPU utilization. To avoid learning every possible process pattern for every attack, we can use an unsupervised approach that uses the process category prediction vectors from our process detector models to learn normal behavior. A higher-level model trained on normal behavior should be able to detect whether the next set of sensor data is likely to be an anomaly [13]. This approach eliminates the need for training and customizing process classifiers for specific anomalies. Instead, system administrators can define specific process sequences of interest. Upon detecting deviations from such sequences, the system can issue alerts indicating the identification of particular process sequences, complete with associated probability values that indicate confidence levels. The process category data are robust to spoofing and avoid relying on knowledge of all processes running on user devices and access to user permissions. This extra information allows the system administrator to monitor the system not only for malware but also for possible slowness of processing power and issues with connectivity related to the corresponding CPU and I/O sensors.

The experimental results generated from our model tests demonstrate the practicality of identifying diverse target processes by analyzing the physical state of a system through existing sensors. Unlike prior methods that necessitate intricate behavioral analysis, our approach provides a swift initial indication of a target process required to avoid compromising performance or incurring significant delays [7,14]. This rapid process detection capability allows for the prompt activation of safety measures, acting as a preliminary defense until more exhaustive behavioral analyses can be conducted. In the realm of malware detection, our novel detection model serves as a crucial tool. It offers preliminary information about a target process, enabling the implementation of immediate safety protocols. This method is designed to complement existing detection techniques, enhancing their performance and acting as an additional layer of defense. Moreover, our approach has the unique ability to model threats based on sequences of fundamental and common process tasks derived from sensor data patterns rather than relying on predefined malware databases. This adaptability makes it particularly effective in detecting zero-day attacks and other instances of malware that may only be identifiable through sensors.

2. Experimental setup and data collection

Our methodology introduces significant contributions in data collection methods in order to generate high-resolution data for training and testing general process detection models. Utilizing specific benchmarking tools, we create and evaluate four distinct process types: file I/O, CPU/ALU intensive, network I/O and virtualization. The training of our models incorporates advanced ensemble methods, which enhances robustness and performance. Additionally, we employ a thorough testing approach using a realistic evaluation dataset, ensuring our models' effectiveness and generalizability in diverse and dynamic environments.

2.1. Hardware and software environment setup

Throughout our experiments, we use sensor data that is available on our specified hardware. The hardware experiments were done on an Apple Mac Mini MGEM2LL/A. The hardware specifications include a 1.4 GHz Intel Core i5 processor, 4 GB of LPDDR3 RAM and a 500 GB HDD. On this hardware are 50 systems sensors. These included sensors for temperature (16 units), voltage (6 units), current (12 units), power (15 units) and one sensor dedicated to monitoring the rotations per minute (RPM) of the exhaust fan. To gather sensor data in real-time, we use the software tool called 'Hardware Monitor' [15].

The 'Hardware Monitor' application enables us to sample data from all sensors every 100 milliseconds (0.1 seconds), providing a high sampling rate capable of capturing rapid changes in hardware sensors. Each sensor reading obtained at this rate is combined into one vector as a singular input for the machine learning model. This input is then processed by the model, resulting in a binary output representing the process being detected. As mentioned in the next sections, our classifier is trained using a total of 2 hours' worth of data, equating to 72,000 classifications made within this time frame considering the given sampling rate. The output forms a binary vector of the same length, which is then compared to the actual behavior of the process during the corresponding sensor reading. We subsequently assess and compare the outcomes of our classifications against the real-time process behavior.

Our data gathering and model development is primarily implemented in Python 3 with packages managed in the Continuum Anaconda package Manager [16,17]. The Scikit-learn library is used for data transformation, preprocessing, model training and testing [18,19]. Our experiments focus on detecting file I/O, CPU, network I/O and Virtual machine (VM) intensive processes. The I/O intensive case is designed using the Iozone filesystem benchmarking tool [20]. The CPU-intensive case is created using the FFmpeg multimedia framework since it is characterized by many ALU-intensive applications [21]. The network I/O intensive process is created using the Nmap network for network discovery and security auditing [22]. The VM process is created by launching and operating a virtual machine (VM), specifically VMware Fusion 10 Pro desktop hypervisor on the host system [23]. Each of these individual tools has multiple commands that we use to create generalizable behaviors for each process category.

We want to emphasize that the four process types originate from distinct applications but can be generalized to common categories: file I/O, CPU/ALU intensive, network I/O and virtualization. Their generalization to similar process categories is rooted in their shared use of specific computer hardware components, such as the disk, CPU or NIC. In real-world scenarios, multiple processes of these types run simultaneously. Our experiments demonstrate that our process detection can accurately identify specific process types even when other processes run concurrently under varying loads. Our model's process detection output can be integrated into more intricate models, representing heterogeneous workloads with multiple processes operating concurrently. While this study emphasizes coarse-grained process detection, these general probabilities and confidence factors can also be combined for models requiring finer control over false positives.

While running each of the four target processes, we add realistic CPU loads during both the training and testing of our models. For generating data specifically for training our models, the Go programming language is used to generate CPU loads using the script '*go-cpu-load*' script described in Vikyd [24]. It adjusts CPU loads in a loop over time and allows us to modify the delay periods between loads. This more basic system load generator is used to create the training data. For more advanced methods and patterns of CPU loads for the testing of the model, we applied the Stress-ng package, which is used to create more diverse system stressors that include over 70 distinct techniques for creating CPU loads alongside the target process [25]. The difference between the training and testing CPU load generation is used to test the generalization of our process detection to unseen conditions.

2.2. Data collection

In practice, a single Python script is used to collect data for DM training and testing in one cycle for a specified process category. A train and test cycle are performed for the specified process type along with the previously described script '*go-cpu-load*' for training and Stress-ng for testing. The process is switched to another process category for a new train and test data collection cycle. After switching, the system is powercycled to free all resources and avoid any contaminating effects of the previous process. The training and testing data for each process type are stored in a comma-separated data file in the system.

After all data processing, we train the corresponding DM models for each process type and experiment with multiple machine learning algorithms for each DM, including ensemble models that are discussed in Section 3.1. The train and test datasets are kept constant for the various DM models we experiment with. After training, testing involves finding the binary prediction for each time interval, resulting in the binary prediction vector. The binary prediction vector is used to compute performance metrics and analyze which processes can be detected adequately and which DM should be used for each method.

It is important to note that the target processes collected from the training data originate from the same applications used in the test set. However, as discussed in Section 2.1, the sensor behavior of the four process categories for detection is broad enough to represent similar processes of the same type from various applications. The available sensors provide information about the process type, which can still be generalized to similar processes within the previously defined categories. This reliance on process categories rather than specific applications means that our test set can be broadly applicable for detecting processes of the same category type, enhancing model generalization.

2.2.1. Collecting training data

For training data collection, we decided that a 2-hour time frame is sufficient to train a classification model since, as found in Taylor et al. [8], training for 24 hours was near the same level of effectiveness as the 2-hour training. Consequently, since any additional training hours have limited performance gain, we stay consistent with using 2 hours of sensor data for training. Three threads need to be established for every 120-minute training session. The threads include one thread to oversee the start and stop of the process category, one thread for controlling the CPU load, and another for data recording. The controller thread first starts the process and times the execution time. A pause of 120 seconds follows, giving system sensors the essential recalibration time. Once recalibrated, the controller thread activates a Boolean marker, which serves as a cue for data recording to commence. What follows is a repeating sequence in order to obtain both binary classes of process being on and off through the 2-hour period: the thread first observes an idle wait time equivalent to the previously logged process completion, activates the process and then waits once again for an equivalent duration. After another recalibration pause of 120 seconds, during which the process starts or stops, this sequence concludes only when the accumulated duration satisfies or surpasses the predetermined training duration. A design aspect of this data collection approach is that it yields a dataset over 2 hours that features a 2:1 inactivity to activity ratio of the test process. While this ratio does not accurately mirror real-world scenarios, it serves a specific purpose in the context of training the detection models. The intentional skew towards a higher duration without the test process running helps the models learn patterns and behaviors associated with the system in an idle state, providing a more robust training foundation for the detection algorithms.

For the ensemble models we will discuss in Section 3.1, we will make improvements upon previous work by attempting to account for the impact of different system loads on the detected process by training the classifier at different system loads. In theory, this method should allow the model to perform better on the complex evaluation data, which consists of a variety of process loads. The training data is used to train ensemble DMs consisting of 11 ML models, each model training under different system loads ranging from 0% to 100%, with one model trained at each 10% interval. Each additional load was achieved by initiating processes on all hyperthreads and running an empty loop at a dynamic rate to sustain the desired CPU load. Ensuring the balance of this additional load across the system's total CPU resources requires running the empty loop at a dynamic rate and is needed to avoid overloading any specific resource. For ensemble detector models utilizing 11 ML models trained on the full range of system loads, each of the four process category models demanded 22 hours of training.

2.2.2. Collecting evaluation data

We employ new advanced methods for collecting evaluation data, which mirrors the approach used for training data but with a few key distinctions that make it more realistic to the real-world behavior of processes. The primary aim here is to accentuate the effectiveness of our models through adequate testing, which has not been seen in previous works. Instead of initiating a timed target process and implementing a continuous loop until reaching a set duration, the evaluation procedure introduces unpredictability. Specifically, the control thread for the target process undergoes random intervals ranging from 10 to 40 minutes. In addition to a random interval, a randomly chosen extra CPU load is added to the system, adding to the load generated by regular background processes. Prior to each evaluation run, a specific CPU load level is randomly selected from a predefined range and applied to the system, utilizing the same method employed during training. Four values are chosen from each range of ten values before progressing to the next range, starting from 1 to 10 and concluding at 91 through 100. Consequently, the dataset comprises 40 evaluations, ensuring an even distribution of random additional load levels within intervals where training data were collected with known additional CPU loads. This dataset accommodates the possibility of random intervals and randomly selected additional load levels that do not include the known load levels employed during training, thus preventing load memorization for classification during the training process. The sporadic nature of this testing helps minimize any biases that could arise from fixed testing intervals. In our experimental evaluation, this testing method is labeled ‘Simple Random’ load test since the CPU load is balanced.

We can make our evaluation data even more realistic by including diverse random CPU load sources different from the training set, mimicking a production system handling multiple heterogeneous workloads. This increased data complexity enhances our ability to generalize test results for broader process detection and is an improvement upon other testing methods in previous works. Evaluation data is collected with an additional CPU load deliberately introduced onto the system, supplementing the load generated by regular background processes. Each specific process under scrutiny is executed once within a span of 1 hour, the timing chosen at random. Before each evaluation run, a distinct CPU load level is randomly chosen from a specified range and applied to the system using a method distinct from those employed during DM training. This chosen load level is then unevenly distributed across hyper-threads in a randomized manner, collectively imposing the desired additional system CPU load level. The technique employed to distribute the load across each hyper-thread is selected randomly from a pool of 70 distinct CPU stress methods, excluding the empty loop method used in training. In our experimental results in Section 5, we test all of our DMs with this ‘Advanced Random’ load test.

3. Detectors model design and evaluation

Our new general process detection methods include a variety of machine learning models and higher-complexity ensemble models to address the variability in process loads and timing described in the previous section. We then plan to test the models with a diverse and comprehensive set of performance metrics to address the complexity of real-world process detection scenarios.

3.1. Detection model implementations

Within our research framework, a 'Detection Model' (DM) functions as a high-level representation of a machine learning model. This is an important abstraction since one of the key contributions of our work is experimenting with a variety of models, including Decision Trees, K-Nearest Neighbors (KNN), Logistic Regression, Support Vector Classifier (SVC), Random Forest, Extra Trees, Multi-Layer Perceptron (MLP) and Naive Bayes. The DM also encapsulates the ensemble versions of these models to detect processes of the specific type under multiple load conditions.

The DMs are all trained to classify five distinct process categories: File I/O Process, CPU Intensive Process, Network I/O Process, Virtual Machine Running and Simulated User process on virtual machines. When given a set of current physical sensor readings, the DM produces a binary output indicating whether the target process is present or absent. This binary value, as outlined in the research in Picek et al. [26], determines the current status of a specific target process within the system. Along with the binary prediction value, the probability value generated by the machine learning algorithm before the binary prediction can be used as a confidence level. This additional factor offers valuable insights, assisting in establishing acceptable levels of false positives for process detection and guiding appropriate actions accordingly. Once a DM has been trained and implemented, the resources required for predictions are minimal.

This efficiency is attributed to the absence of complex behavioral or computational analyses. Additionally, the data utilized for process detection are sourced from side-channel data, ensuring minimal risk to user privacy and data leakage. Our approach represents a holistic and adaptable methodology underpinned by DM abstraction, facilitating robust decision-making in the realm of process detection systems.

Section 2.2.1 details how ML models are trained, and this section details how a collection of ML models is used to build an ensemble detector whose purpose is to select ML models that are optimal for the system's current physical state. Five different

DMs are trained – Single, Ens 6 Single, Ens 11 Single, Ens 6 Dual and Ens 11 Dual. Each of the five methods was tested using simple and advanced unknown additional load tests for each target process as described in Section 2.2.2. We then evaluate the prediction vectors of five DMs under a variety of performance metrics to accurately assess whether the increased complexity of models results in greater improvements in performance.

The simplest detection method utilizes a single ML model, trained with all the training data collected at the various known additional load levels. The simplicity of this method is illustrated in Figure 1. This detection method is labeled as 'Single' in the experimental results.

CPU load serves as a confounding factor in our side channel method. This is because CPU load competes for resources, leading to delays in executing processes and subsequent behavioral changes. At the sensor level, CPU load directly influences sensor readings; for instance, high CPU load translates to increased power consumption and higher heat levels. This variability in CPU load significantly influences the features of different process categories, thereby influencing the modeling of process detection. To address this challenge, we created an ensemble of classifiers trained at various CPU loads, enabling a more robust and accurate approach to our analysis.

As discussed in Section 2.2.2, CPU loads are randomly introduced with random CPU load levels in our test data to mimic real process detection situations. Considering the impact of CPU load on our model, we develop an advanced ensemble-based DM. This ensemble method selects a classifier trained on specific loads, enhancing our detection approach. In theory, training multiple models that account for the CPU load should improve classification by accounting for sensor fluctuations due to factors not related to the process we are running for detection. During the ensemble model's testing, a load-based selection of the trained ML models is made. The selected model that was trained under similar conditions of CPU utilization should perform better than any single model at detecting whether the process is on or off. As we discussed in the previous Section 2.2.1, the ensemble detectors are created from multiple versions of the highest-performing ML model for the specified process category. One ensemble uses 11 versions of the model that are each trained at different CPU load levels varying at intervals of 10% from 0% to 100%. Another ensemble uses six versions of the model trained on CPU load intervals increasing in intervals of 15% from 0% to 100%. The model receives the current system CPU load utilization and uses this information to select the model trained on

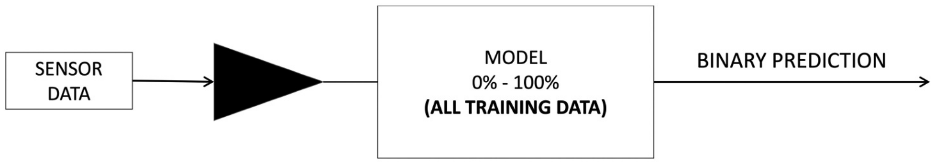

Figure 1. Single detection model (single).

the load level closest to that found in the current test data. These detectors are shown in Figures 2 and 3 and will be labeled as 'Ens 6S' and 'Ens 11 S', respectively, in the experimental results.

CPU load can also be influenced by the operation of the process we are detecting. For example, when the process is not running, the CPU load is lower; thus, the ensemble detector chosen will be trained on the lower CPU load. To address this complexity, we divide the task into two distinct components, both reliant on CPU usage, by creating dual models. Initially, one ensemble model assesses whether a process category is active based on the current CPU load. However, sensor data without an active task should yield a lower CPU load. As a result, another ML model calculates CPU usage, subtracting the average load of that specific process category. This lower CPU usage is employed to run the model that is trained at a reduced level, generating a confidence score indicating that the process is inactive. Consequently, the confidence outputs from the dual models, one trained on higher CPU and one trained on lower CPU loads, result in a more robust classification of the process type. Utilizing the confidence scores of these binary predictors, we generate the final binary prediction vector based on averaged confidence factors, ensuring a nuanced process detection outcome that accounts for the CPU load of the process itself. In the following experiment, dual-

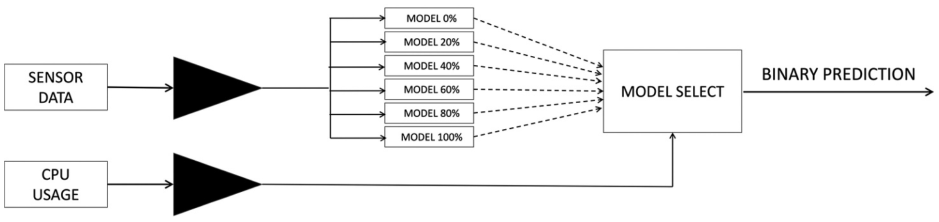

Figure 2. Ensemble detector – single model selection – 6 models (ens 6 S).

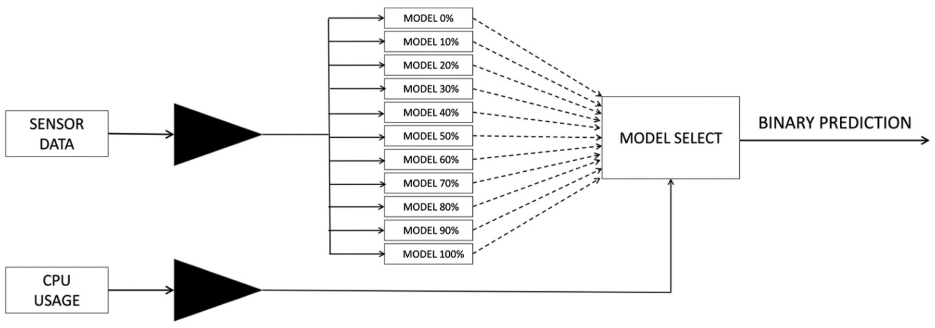

Figure 3. Ensemble detector – single model selection – 11 models (ens 11 S).

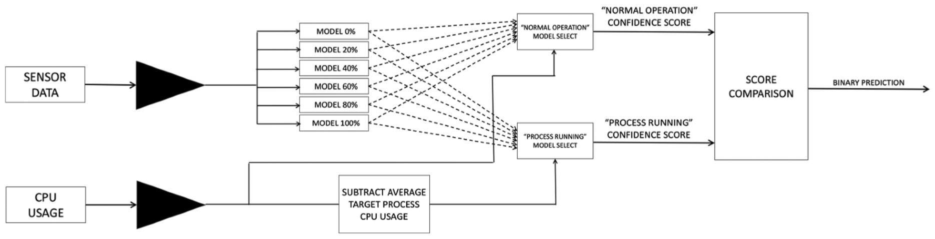
SENSOR DATA
CPU USAGE
MODEL 0%
MODEL 20%
MODEL 40%
MODEL 60%
MODEL 80%
MODEL 100%
"NORMAL OPERATION" MODEL SELECT
"PROCESS RUNNING" MODEL SELECT
SUBTRACT AVERAGE TARGET PROCESS CPU USAGE
"NORMAL OPERATION" CONFIDENCE SCORE
"PROCESS RUNNING" CONFIDENCE SCORE
SCORE COMPARISON
BINARY PREDICTION

Figure 4. Ensemble detector – dual model selection – 6 models (ens 6 D).

selection ensemble detectors with 6 ML models and 11 ML models are evaluated. These detection models (DMs) are shown in Figures 4 and 5 and will be labeled as 'Ens 6 D' and 'Ens 11 D', respectively, in the experimental results.

3.2. Performance evaluation methodology

3.2.1. Binary classification evaluation

The evaluation step of our experiment requires running our DMs to obtain observed binary prediction vectors. We then compare our prediction vector to the expected or actual system state vector in order to generate four possible classification states at every time period. These states, denoting correctly identified active periods (*true positives*), incorrectly identified inactive periods (*false negatives*), accurately identified inactive periods (*true negatives*) and erroneously identified active periods (*false positives*), form the basis for computing five crucial metrics: *sensitivity*, *precision*, *specificity*, *fallout* and *accuracy* [27]. It is important to emphasize that evaluating these metrics collectively is essential, as no single criterion provides a comprehensive performance measure in isolation. We examine the performance of our classifiers with these five statistical metrics, which utilize the four previous classification states: DPS, MCC, RPD, Fallout and TPD.

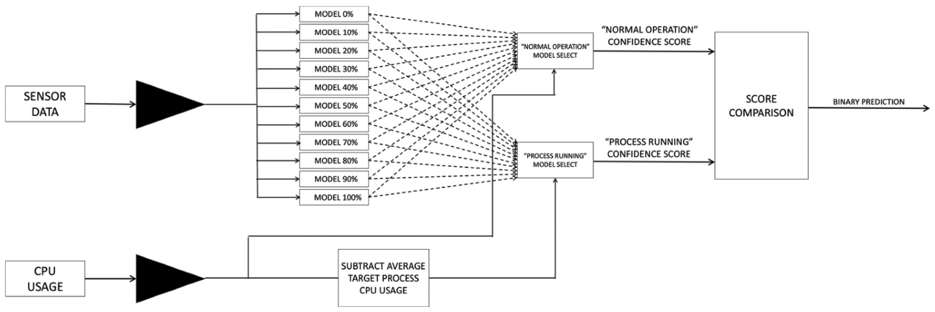
SENSOR DATA
CPU USAGE
MODEL 0%
MODEL 10%
MODEL 20%
MODEL 30%
MODEL 40%
MODEL 50%
MODEL 60%
MODEL 70%
MODEL 80%
MODEL 90%
MODEL 100%
"NORMAL OPERATION" MODEL SELECT
"PROCESS RUNNING" MODEL SELECT
SUBTRACT AVERAGE TARGET PROCESS CPU USAGE
"NORMAL OPERATION" CONFIDENCE SCORE
"PROCESS RUNNING" CONFIDENCE SCORE
SCORE COMPARISON
BINARY PREDICTION

Figure 5. Ensemble detector – dual model selection – 11 models (ens 11 D).

3.2.2. *Matthews correlation coefficient (MCC)*

The Matthews correlation coefficient (MCC) as shown in Equation 1 evaluates the relationship between actual and expected outcomes [28]. Since it equally takes into account true and false positives as well as true and false negatives, this metric is robust to class imbalances. MCC scores span from −1 to +1. A score of +1 denotes perfect prediction, while 0 is equivalent to arbitrary guessing, and −1 signifies perfect discord. Importantly, the MCC is balanced, meaning that if one interchanges the positive and negative binary evaluations, the MCC remains unchanged. This characteristic becomes especially pertinent in scenarios with a pronounced disparity between positive and negative outcomes during assessment, much like the data analyzed in this study, which has a 2:1 ratio. The MCC figures presented in the research findings act as benchmarks to gauge the efficiency of the Detection Models (DMs) strictly in their capacity as binary categorizers without considering the specific context or rationale of their application.

$$p\frac{(TP*TN)-(FP*FN)}{(TP+FP)(TP+FN)(TN+FP)(TN+FN)} \tag{1}$$

3.2.3. *Rate of process detection (RPD)*

During the evaluation of process detection, we need a metric to represent when we have an interval of time in which the process is on, and there is at least one positive prediction that marks that interval as having a process detected at that interval. This can also be thought of as the time-independent process recall rate. The area of the time period in which the process is detected is denoted by the locations of the label vector where the system transitions from a labeled state of 'normal operation' to 'process running', concluding when the state shifts from 'process running' back to 'normal operation'. Only one positive prediction is needed for each of these intervals to obtain a 100% rate of process detection.

In the testing phase, a time series defines the periods of an actual process occurrence. The analysis involves checking each time interval within an attack period for positive predictions in the prediction time series. If positive predictions coincide, the process running instance is considered detected. The goal is to have at least one positive prediction during each process run instance, achieving a perfect recognition rate of 1.0. Unlike traditional binary classification metrics like sensitivity, the rate of detection focuses not on the volume of correct positive predictions but on the application of the binary classifier. One positive prediction is as effective as numerous positive predictions since the process run only needs to be flagged once. It is essential to consider the rate of process recognition in tandem with the binary classifier's ability to make accurate positive predictions, such as precision. Note that the rate of attack recognition attains a perfect score even if all predictions are positive, highlighting the importance of a balanced assessment with other metrics.

3.2.4. Time to process detection (TPD)

Not only do we need to measure our ability to recognize individual intervals in which the process is running using the RPD, but we also need to measure how much time is taken between the start of the 'on' interval and the point where the model detects the first true positive. A delay is expected since the rising edge signifies the start of the process, but the physical sensors will be slower to change, and the model will be slower to recognize the first positive prediction. To determine the time to process detection (TPD), we find the difference between the rising edge of the interval, which is the initial state of 'process running', from the time of the first positive prediction within the interval of process labels being positive. The first positive prediction will be after the first label of 'process running' due to sensor delays.

To quantify this delay, the number of time intervals, until the first positive prediction is recorded, is counted for every recognized target process run and process instance. This metric is then averaged to determine the time to process detection or recognition. It is essential to note that instances not successfully recognized do not adversely affect this metric. Consequently, the mean time to process recognition should be evaluated alongside the rate of process recognition to assess the efficiency of detecting processes swiftly and consistently.

Moreover, it is crucial to consider the precision of binary classifiers when making correct positive predictions. If all predictions are positive, the mean time to process recognition becomes zero. Therefore, a balanced evaluation involves weighing the mean time to process recognition against the classifier's precision, providing a comprehensive understanding of the system's ability to promptly and accurately detect various process instances.

3.2.5. Detector performance score (DPS)

In order for our detection model to be useful, it must account for the previous factors of RPD, TPD and fallout. The fallout is the overall false-positive rate, which is obtained from all the labeled time periods that are marked with no process running (0), and the model falsely detects a process (1). The fallout rate is critical since a high fallout rate would result in many false-positive detection alerts, making the model unusable due to operational effects. Based on the 2020 report by Neustar Security Solutions, it is found that, on average, 26% of security alerts experienced by organizations were false positives [29]. As a result, one goal of our detection model is to have a fallout rate well below the average of 0.26. Based on previous work, we also should note that for our model to be effective, it must have a TPD of no more than 60 seconds [14]. In addition, we know that if the model misses any interval in which the process is running, it should be heavily penalized since it missed many opportunities to detect the process. So, we choose to combine these three metrics of fallout, TPD and RPD into a single 'Detection Performance Score' (DPS) using Equation 2.

$$DPS = 1 - \frac{\frac{TPD}{60} + \frac{fallout}{0.26}}{2} \tag{2}$$

4. Experimental results

We analyze the detection results for the four process categories using the methodology described in Section 3.2. The detection results are obtained for both the simple and advanced random additional load tests for each process category. Additionally, we discuss the system commands used to run each process type. For initial analysis, the primary focus is on the single model detector, offering valuable insight into the basic performance of the process detection. Through this analysis, we identify the top performing machine learning algorithm for each detection method. It is essential to note that we do not showcase the zero additional load results, as their main purpose is to assess whether further testing is necessary for a specific target process. We show simple additional load and advanced additional load to understand performance under realistic testing conditions.

With the best single model obtained, we start a time-based testing phase to closely monitor its detection capabilities. This involves illustrating metrics such as the DPS, MCC and Fallout as they evolve over time. We also provide a visual representation of the binary detection results in the form of a time series juxtaposed against the actual process state values. This visual also contains the progressively increasing process load over the indicated period.

Our research also encompasses an in-depth analysis of the importance of features. We utilize pie plots to display the feature importance percentage scores specific to random forest models. These scores rely on Scikit-Learn's 'feature importance' attribute. The determination of these importance values hinges on methodologies like gini importance or the mean decrease in impurity (MDI). To elaborate, MDI evaluates each feature's relevance by tallying the number of splits it influences across trees, weighting them proportionally to the samples it segments [18,30].

For each process type, we then use the identified best single model to formulate multiple ensemble models, with details discussed in Section 3.1. We then begin a comprehensive evaluation of all our DMs using the metrics in 3.

4.1. General process detection

4.1.1. File I/O process

To create a representative pattern for the file I/O process category type, we applied a significant amount of file I/O to the host system alongside the random amount of CPU load. The file I/O was generated using our IOzone, a filesystem benchmarking tool. IOzone includes a variety of write tests: Write, Re-write, Random Write, Record Re-Write, Fwrite and Frewrite. In addition, there are a variety of read tests: Read, Reread, Random Read, Backward Read, Stride Read, Fread and Freread. The command run to start the process is in Appendix A. When a variety of these commands are combined, these tests are representative of processes that, in general, have high file I/O.

The results of the unknown additional load tests for single detection models for file I/O process detection can be seen in Table 1. For the simple random load tests, the best-performing prediction algorithm is the support vector machine (SVC) according to the DPS score. However, for the more thorough advanced load testing, the single detector with the best prediction results according to the DPS score is shown to be the logistic regression. Since the advanced load testing is more realistic to real-world varying process load, we will use the logistic regression model for further analysis.

Various performance metrics shown over time specifically for the logistic regression model are visualized as plots in Figure 6. The selected model has high DPS and MCC scores throughout the testing period, even with varying additional random loads. The rightmost subfigure, in particular, highlights that there are very low false positives. This result indicates that the additional CPU load does not impact the sensors that are used to detect I/O processes. In particular, the MCC being high signifies that the true positive and true negative are both high and the false positive and false negatives are both low.

Figure 7 shows the three time-series plots, CPU load, the actual state of the target process and the predicted state of the target process, all over a 40-hour testing block. In the topmost figure, we show the variable additional random CPU load being added to realistically simulate the actual state of the target process. The actual state of the target process is shown in the middle figure, which reveals that the areas under the curve, between the rising edge and

Table 1. File I/O process unknown additional load test results.

Algorithm	DPS	MCC	RPD	Fallout	TPD
Simple Random					
SVC	0.996	0.9955	1	0.0013	0.175
Ran For	0.9951	0.9943	1	0.0017	0.2
E Tree	0.9944	0.9919	1	0.0019	0.25
Log Reg	0.9926	0.9899	1	0.0032	0.15
N Bayes	0.9897	0.9848	1	0.0042	0.275
KNN	0.9865	0.9809	1	0.0064	0.15
MLP	0.9857	0.9777	1	0.0069	0.125
Dec Tree	0.9001	0.8785	1	0.051	0.225
Advanced Random					
Log Reg	0.9968	0.9982	1	0.0005	0.275
Ran For	0.9964	0.9984	1	0.0004	0.35
KNN	0.9956	0.9972	1	0.0008	0.35
E Tree	0.9952	0.995	1	0.001	0.35
SVC	0.9949	0.9948	1	0.0014	0.275
N Bayes	0.9948	0.9946	1	0.0011	0.375
Dec Tree	0.9921	0.9963	1	0.0007	0.775
MLP	0.538	0.1692	1	0.8151	0.0667

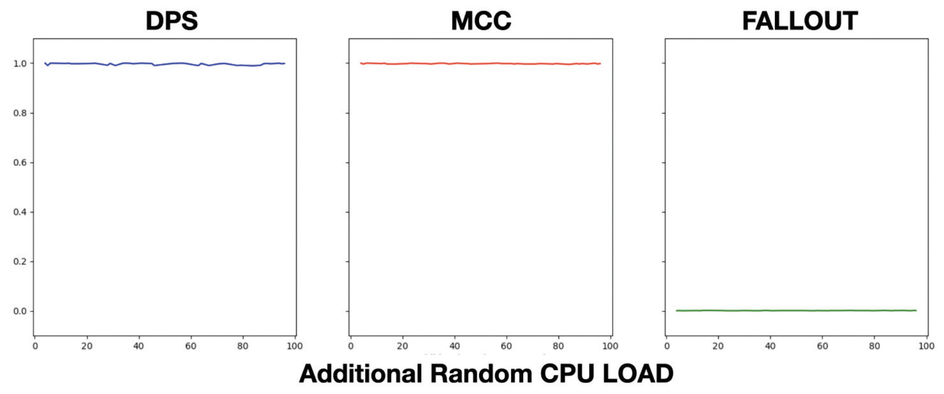

Figure 6. File I/O process binary classification metrics – single detector trained with logistic regression.

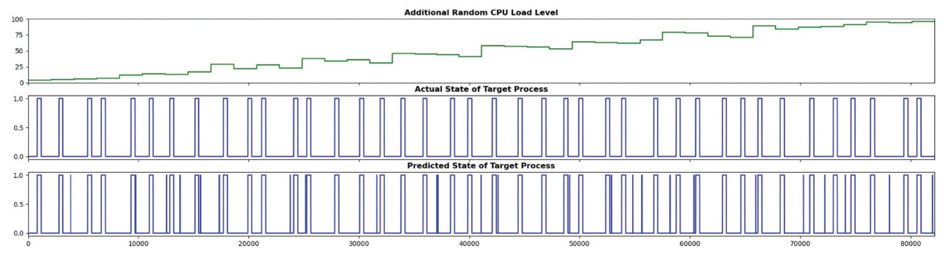

Figure 7. File I/O process prediction time series – single detector trained with logistic regression.

falling edge, are the time interval in which the target process is active. The bottom plot shows the resulting binary prediction vector in a similar form to the output of single logic regression over the time periods. When the detection rate is perfect, we will see an exact match between the actual state and the predicted binary state. We show a slight disparity caused by false positives, meaning that these results confirm that our detection method for file I/O tasks is effective.

In order to look deeper at what sensor features are more likely to be used during the classification, we create the pie plot in Figure 8. The pie plot reveals the importance values by system components for the I/O process detection. Using the figure, we can see that CPU component sensors are the most useful at 48.7%. This is surprising considering the fact that the random additional CPU loads are expected to interfere with the usage of CPU-related features. The remaining system components of the power supply, hard drive, memory and platform controller hub (PCH) are equally split in feature importance.

Figure 9 shows the percentage of the total feature importance values based on what types of measurements are being taken. The majority of the feature

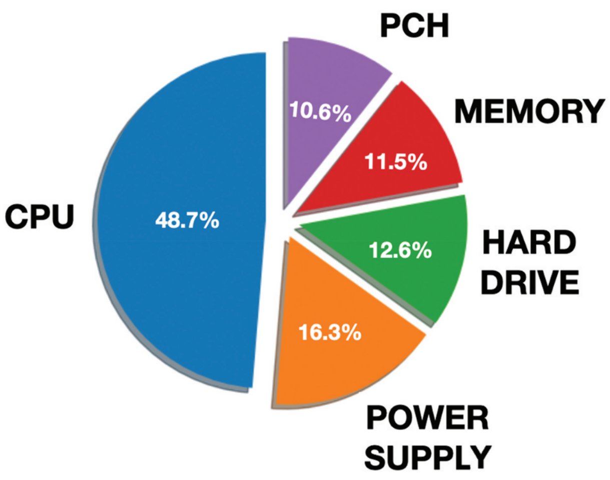

Figure 8. File I/O process feature importance by system component.

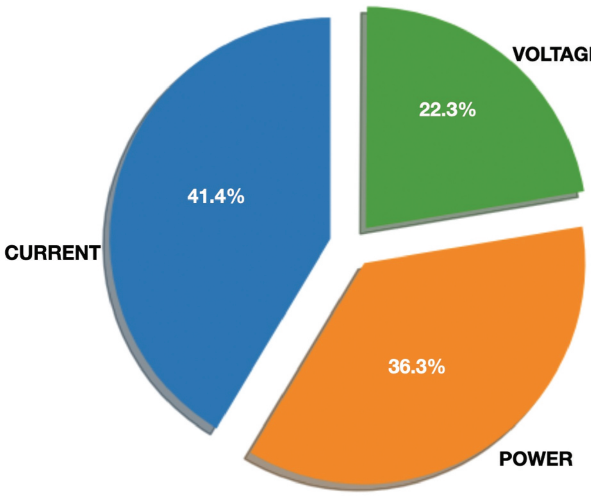

Figure 9. File I/O process feature importance by sensor type.

importance, around 41%, comes from the current sensors, and a large amount of feature importance also comes from power sensors, around 36.3%. From these statistics and knowledge of the behavior of file I/O, we know that, most likely, a complex pattern arising from all feature sensors is used by the detector to detect the process. This is because even during an I/O specific file process, the PCU is used to read and write from the hard drive, and the PCH is used to move and store data in memory. As a result, we expect to see an equally spread distribution. Though the detection process is completely automated by the detector, having these metrics simplified for our understanding can help us in the future in sensor placement and sensor-specific modeling.

Table 2. File I/O process detector type comparison.

Detector	DPS	MCC	RPD	Fallout	TPD
Single	0.9968	0.9982	1.0	0.0005	0.275
Ens 6 Single	0.9967	0.998	1.0	0.0005	0.275
Ens 11 Single	0.9944	0.9937	1.0	0.0014	0.35
Ens 6 Dual	0.9967	0.998	1.0	0.0005	0.275
Ens 11 Dual	0.9944	0.9937	1.0	0.0014	0.35

In Table 2, we see that the highest-performing detector is the single detector. Note that we create the ensemble detectors from the multiple of the best-performing ML model for the process category, in this case, the logistic regression. Though we have tested and designed our ensemble models to account for process load and also deal they all seem to perform slightly worse that the single detector based on the DPS score. Given the fact that all detectors are performing very well, the extra model complexity most likely is not necessary. Looking at the other metrics, we can see that false positives are low, with a fallout of less than 0.0005, and the time to detect the true positives is low, taking only 0.275 seconds. The overall detection rate is high.

4.1.2. CPU intensive process

The CPU-intensive process for detection testing is run alongside the additional random CPU usage. The generation of the CPU-intensive process itself is done using the multimedia FFmpeg tool. The specific task assigned that we believe generalizes to most processes in the CPU-intensive category is converting a video in.mov to an.mp4 format. This process is done on an HD video of 100MB, which is 3 minutes in length. The command to start the process is in Appendix B.

Table 3 contains the machine learning single model comparison for CPU-intensive process detection. Overall, the random forest performed best on the

Table 3. CPU intensive process unknown additional load test results.

Algorithm	DPS	MCC	RPD	Fallout	TPD
Simple Random					
Dec Tree	0.9988	0.9987	1.0	0.0002	0.1
KNN	0.9987	0.999	1.0	0.0001	0.125
Log Reg	0.9986	0.9991	1.0	0.0002	0.125
SVC	0.9983	0.9989	1.0	0.0002	0.15
Ran For	0.9983	0.9983	1.0	0.0004	0.1
E Tree	0.9974	0.9951	1.0	0.0007	0.15
MLP	0.9972	0.9944	1.0	0.001	0.1
N Bayes	0.9267	0.8958	1.0	0.0844	0.25
Advanced Random					
Ran For	0.989	0.7878	1.0	0.0002	1.275
Dec Tree	0.9875	0.7613	1.0	0.0004	1.4
E Tree	0.9861	0.5901	1.0	0.0005	1.55
KNN	0.9776	0.6876	1.0	0.0008	2.5
N Bayes	0.7896	0.6577	1.0	0.3329	0.6
MLP	0.5484	0.4717	0.55	0.0003	0.2273
Log Reg	0.4988	0.4756	0.5	0.0001	0.25
SVC	0.4738	0.4738	0.475	0.0001	0.2632

advanced random load with a DPS of 0.989. It seems to be slower to detect the process at almost 1.25 seconds compared to the File I/O process category detectors, which would take around 0.27 seconds. Similar to the previous process, the category results in a low fallout and low TPD. Resulting in a high DPS score and, thus, an overall high detection score.

However, we do notice a lower-performing MCC, which we discuss in depth.

The resulting metrics over 40 hours of testing are shown for the decision tree in Figure 10. The results for DPS and MCC are high for the first half of testing, but as the additional load increases past 75%, we see a degradation of the MCC to 0.5 and a slight reduction of the DPS to 0.8 at the end when the CPU load reaches 90%. We show the fallout is still close to zero, which is a good sign that though the true positive detection is reduced, we still have low false positives, and the intervals in which our process is running are still being detected, as shown by the DPS. The fact that the MCC is low signifies that over the interval the process is on, we may not detect all of the true positives, but as shown by the DPS, we detect the overall locations in which the process is on in general. This degradation of detection caused by the higher additional CPU loads is expected since we are specifically trying to identify the CPU-intensive process category.

The behavior of having low false positives but an increase in false negatives over time is shown in the time-series plots in Figure 11. Since the DPS remained high and the fallout very low, the overall detection rate is very good.

The one issue in the zoomed-in version of the previous time-series plot in Figure 12 is the fluctuating detection of false negatives within the areas where the process is running. The fluctuation represents the rapid changes in our model's ability to consistently determine true positives and false negatives. Though this seems like a problem initially, in a realistic system monitoring scenario, the detection can simply be averaged over larger windows of time.

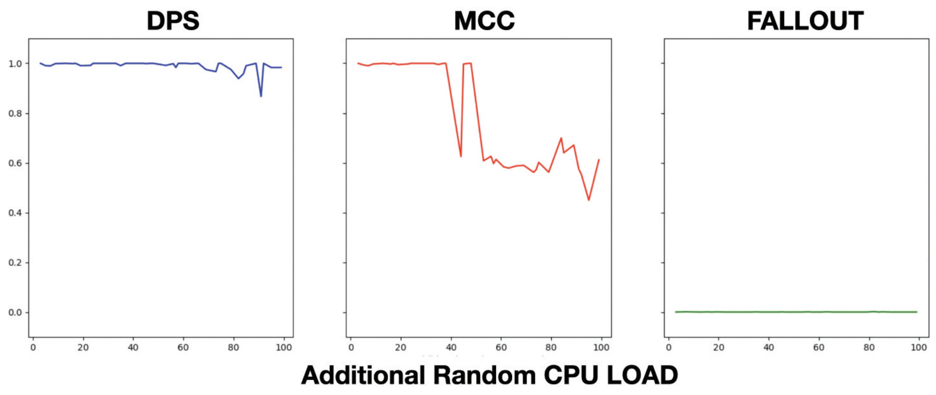

Figure 10. CPU intensive process binary classification metrics – single detector trained with random forest.

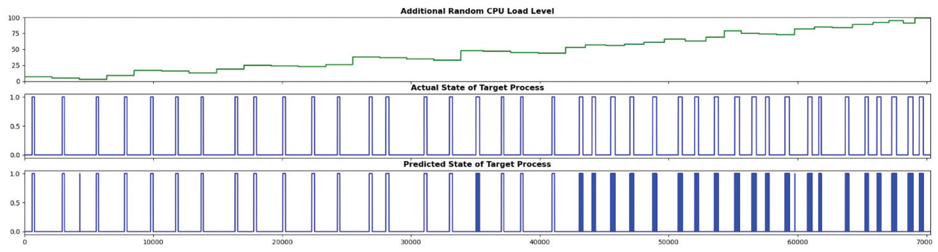

Figure 11. CPU intensive process prediction time series – single detector trained with random forest.

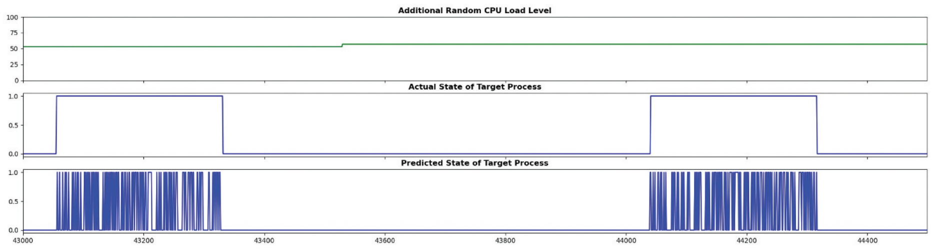

Figure 12. CPU intensive process prediction time series zoomed.

Running a moving average could aggregate multiple positive detections and avoid the effects of any false negatives. For this reason, DPS can be considered a better metric for our goal of general process detection, with the goal of triggering behavioral analysis or anomaly detection since we want to detect the process at the interval at least once and quickly. In addition, the low fallout is also a strong indicator that this model can be deployed without extraneous false positives.

The MCC, on the other hand, is more useful at indicating true positive and true negative rates but without knowing the categorization of the intervals themselves. Overall, though the results are not near perfect as a binary classifier, it is high-performing as a process detector using process detection-specific metrics.

The feature importance of different system components for CPU-intensive process detection is shown in Figure 13. The most important sensor features are related to system memory system components since 48.6% of the feature importance is assigned to this segment. The second most important feature for process detection is power supply-related features, with a feature importance of 26.6%. Surprisingly, only 24.5% of the feature importance is assigned to CPU-related sensors despite the fact that we are training a model to detect CPU-intensive processes. This does align with the idea, however, that the additional random CPU load requires the model to learn how to estimate CPU-related processes without directly relying on the obfuscated CPU load.

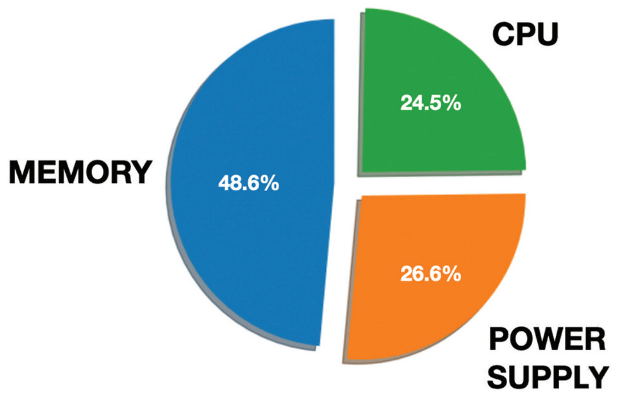

Figure 13. CPU intensive process feature importance by system component.

We also look at the feature importances categorized by sensor types rather than system components in Figure 14. Unlike the previous File I/O intensive process, which has great importance attributed to the voltage sensors at 22%, the detection of the CPU-intensive process requires more focus on the power and current-related sensors and less on the voltage sensors at 12.8%. These results are related to the active power being supplied to the system memory since the process running is a video processor and would require heavy utilization of the CPU memory with very small and rapid fluctuations in power in multiple components.

We use a table of results comparing the five detector types, this time for CPU intensive processes in Table 4. Similarly, we find that the single detector has better overall performance across most metrics with a DPS of 0.989.

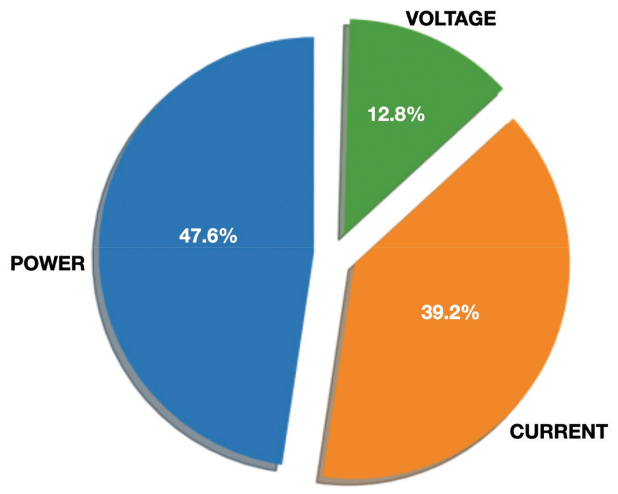

Figure 14. CPU intensive process feature importance by sensor type.

Table 4. CPU intensive process detector type comparison.

Detector	DPS	MCC	RPD	Fallout	TPD
Single	0.989	0.7878	1.0	0.0002	1.275
Ens 6 Single	0.9781	0.7271	1.0	0.002	2.15
Ens 11 Single	0.9836	0.7835	1.0	0.004	1.05
Ens 6 Dual	0.988	0.7524	1.0	0.0002	1.4
Ens 11 Dual	0.9854	0.5758	1.0	0.0004	1.65

4.1.3. Network I/O process

In order to simulate the Network I/O process category for process detection, we focus on the generation of network traffic alongside the additional random amount of CPU usage. The network traffic is generated by using the NMAP (network mapper) tool, which was utilized to conduct comprehensive scans on each port across all IP addresses within the subdomain of the environments. Though this is not the typical network behavior, the overall usage of similar sensors on a NIC would be generalizable. The actual process that is run four consecutive times each test cycle is shown in Appendix C.

Table 5 contains network I/O intensive process detection results for the single classifiers. The result is that even the best-performing model for the advanced random load has a DPS of only 0.5484. Though according to the RPD, all intervals of process running are detected, the fallout rate is very high at 0.472 and the MCC is extremely low at 0.017. As a binary classifier, the result is barely above chance since it is close to zero. As a process detector, in general, there are too many false positives to be useful. Since advanced random load results seem to show the model is not training, we can then analyze the more simple load testing. Simple load testing adds much less variety to different loads, resulting in a much higher DPS. The support vector classifier performs at a DPS of 0.836, with

Table 5. Network I/O process unknown additional load test results.

Algorithm	DPS	MCC	RPD	Fallout	TPD
Simple Random					
SVC	0.8363	0.3751	0.95	0.046	3.6842
MLP	0.7829	0.2976	1.0	0.1874	0.975
KNN	0.7641	0.1906	1.0	0.1234	0.875
Log Reg	0.7568	0.2759	0.875	0.0492	4.0571
E Tree	0.7296	0.1318	0.975	0.1662	3.3333
Dec Tree	0.6966	0.1754	1.0	0.1725	1.8
N Bayes	0.4965	0.0479	0.775	0.4243	2.8065
Ran For	0.4318	0.1693	0.5	0.0327	2.9
Advanced Random					
KNN	0.5484	0.017	1.0	0.472	1.95
Dec Tree	0.5114	0.0114	1.0	0.4957	3.7
Log Reg	0.5103	0.0094	1.0	0.8753	0.2
MLP	0.4888	0.0636	1.0	0.8323	1.65
SVC	0.4811	0.015	0.975	0.906	1.1795
E Tree	0.3904	0.0756	0.775	0.5175	5.129
N Bayes	0.3625	0.0337	0.725	0.7077	0.0
Ran For	0.3354	0.0135	0.575	0.3946	2.7826

a lower fallout rate of 0.046. However, these results indicate that the advanced random CPU loads have an effect on the sensors that make the model unusable for more realistic scenarios.

Figure 15 aligns with the previous statement that the overall performance is low since the DPS over time is around 0.5, the MCC is near 0, and the fallout sporadically increases beyond 0. The erratic nature of the fallout impacts both the DPS and MCC, resulting in overall low performance.

The time-series plots in Figure 16 would indicate that the detector is making a positive prediction almost every time interval, and despite detecting all instances of the test process, it has no real value as either a binary classification model or a rapid process detector.

Figure 16 shows how the time periods are being correctly and incorrectly classified as periods where the process is running. From what we visualize, almost every time period is classified as the process running.

A zoomed-in section Figure 17 further visualizes the lack of accurate detection since the process detection is still always detecting positive that the process is running with random fluctuations in detection throughout.

For the current set of results, the analysis of feature importance is not relevant due to the low performance of the model classification. Since feature

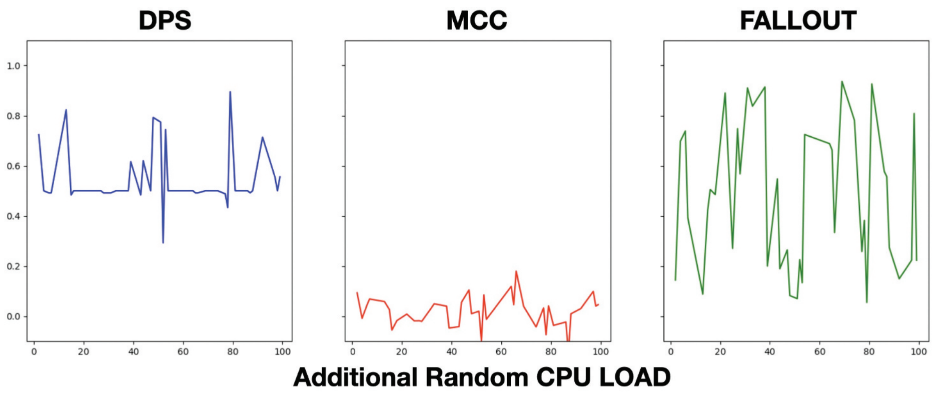

Figure 15. Network I/O process binary classification metrics – single detector trained with KNN.

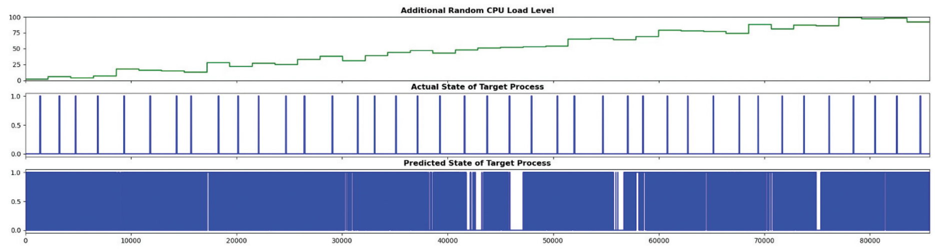

Figure 16. Network I/O process prediction time series – single detector trained with KNN.

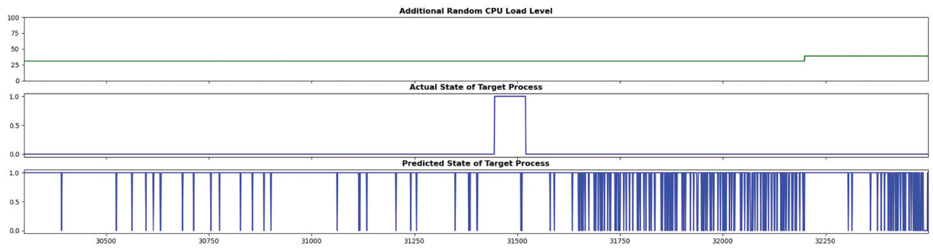

Figure 17. Network I/O process prediction time series zoomed – single detector trained with KNN.

importance is the relative importance of features for the classification of the random forest model, and the random forest model and all classifiers perform poorly, we know that the feature importance obtained will not necessarily be the features important to the classification.

Though our single model is not accurate from testing on the random additional loads, we did see previously that our testing on simple additional loads was much more effective. One hypothesis could be that if we combined multiple single classifiers trained on different CPU load levels, there could be better adaptation to the variation. However, as shown in Table 6, the DPS for all detectors is low, and there is no benefit from the more complex models. Overall, our modeling ability has certain limitations in determining the existence of the network I/O process category.

4.2. *Virtualization detection*

The fourth process category to detect is virtualized processes or processes running on a virtual machine. Launching a virtual machine using a type 2 hypervisor with a realistic user scenario and the additional random CPU load can generate this process category. The specific virtualization software used was VMware Fusion 10 Pro running on Mac Minis as described in Section 2.1. The command line tool from VMware fusion, ‘vmrun’ is used to run commands on the virtual host [23]. The commands we run in a script to simulate user behavior are to start the VM, display the desktop, run a Bash script, shut down and close the VM. The VM instance runs

Table 6. Network I/O process detector type comparison.

Detector	DPS	MCC	RPD	Fallout	TPD
Single	0.5484	0.017	1.0	0.472	1.95
Ens 6 Single	0.506	0.0119	0.975	0.8284	0.2308
Ens 11 Single	0.5444	0.0076	0.95	0.5489	4.9211
Ens 6 Dual	0.5062	0.0116	0.975	0.8375	0.2051
Ens 11 Dual	0.5401	0.0053	1.0	0.8598	0.3

Xubuntu, a lightweight version of Ubuntu that requires only 2 GB of memory and 100 GB of hard disk space. The commands to launch, display the virtual machine, run a bash script and shut down the virtual machines are shown in Appendix D.

This virtualization process detection is divided into two subsections since we can detect two different virtualization process behaviors. First, we can detect whether the virtual machine is running on the machine. For the second level of detection, we can detect if there is a process running on the virtual machine itself. We can create the dataset for training and testing using various flags during the running of our virtualization steps. To create the labels for the first virtualization detection, the time period in which the virtual machine is turned on with a flag. This flag is turned off when the 'stop' command terminates the virtual machine. To create the label for the second task of whether a process is running on the virtual machine, we wait several minutes after the virtual machine begins to run the simulated user commands. We turn on this second flag to mark the periods of time when the commands are running on the virtual machine. This second flag turns off when the commands are complete, not when the virtual machine is terminated.

From these two sets of labels, we can train two detection models, one for detecting the virtual machine running and one for detecting the user commands running on the virtual machine. Ideally, the first detection model for a virtual machine running will not be impacted by the fact that a simulated process is running for only part of the time; this is a realistic scenario since a virtual machine is not always running a process and may be idle. The second detection model will be used to detect the simulated user running virtual processes. There may be an error in the model caused by confounding variables, where the model learns to detect whether the virtual machine is on instead of learning when the virtualized user process is running, resulting in a high false-positive rate. Overall, the separation of the classification of virtualization into these two subtasks allows us to examine and separate the two behaviors more closely.

4.2.1. Virtual machine running on host

Table 7 contains the single detector results for the detection of whether the virtual machine is started. The best-performing model is the KNN with a DPS of 0.9839 with the advanced random CPU load tests. For this detection model, we have a high

RPD of 1 while having a low fallout rate of 0.0065 and a low delay of 0.425 seconds. However, we have a less-than-perfect MCC score of around 0.6658. As a result, we find that similar to the CPU-intensive process detector, this classifier is not very high performance as a binary classifier but also that the model performance for use as a process detection tool is high.

Table 7. Virtual machine running with unknown additional load test results.

Algorithm	DPS	MCC	RPD	Fallout	TPD
Simple Random					
Log Reg	0.9896	0.9402	1	0.004	0.325
KNN	0.9865	0.9375	1	0.006	0.225
SVC	0.9863	0.9365	1	0.0061	0.225
Ran For	0.9753	0.884	1	0.0106	0.525
E Tree	0.9721	0.8756	1	0.0136	0.2
MLP	0.9609	0.8591	1	0.0194	0.225
Dec Tree	0.9481	0.8228	1	0.0259	0.25
N Bayes	0.8566	0.6393	1	0.1339	0.8
Advanced Random					
KNN	0.9839	0.6658	1	0.0065	0.425
SVC	0.9836	0.6735	1	0.0071	0.325
Ran For	0.9822	0.6697	1	0.0081	0.275
MLP	0.9753	0.5257	1	0.0073	1.275
E Tree	0.9566	0.5926	1	0.021	0.375
Log Reg	0.9386	0.6506	1	0.037	0.425
N Bayes	0.9029	0.5785	1	0.0772	0.3
Dec Tree	0.7266	0.2298	1	0.2829	0.2

Figure 18 mirrors what we expect with the DPS, fallout and MCC statistics over time. These metrics are similar to what we have seen for CPU intensive process detection, however the main difference is that previous the MCC slowly degraded over time, instead the MCC stars off around 0.6 and hovers around it. This leads us to believe that the increasing CPU load is not impacting the MCC and, thus, that the error in prediction is not caused by CPU-related sensors. Overall, the high DPS score leads us to believe that detecting the processes that keep the virtual machine running is effective using our model.

In Figure 19, we can see the binary prediction vector and actual label vector side by side. We show that there is some degree of correlation between where the process detection is detected and where the first true positive prediction occurs. However, we can also see places where there are false positives, spikes where there should not be anything, and true

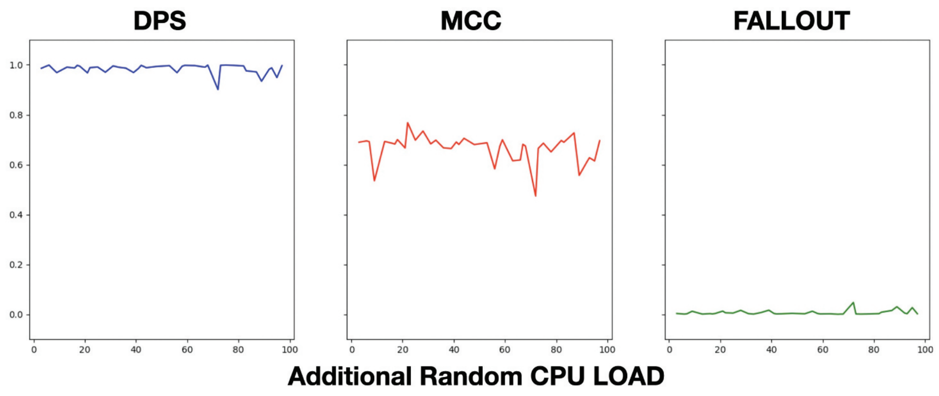

Figure 18. Virtual machine running binary classification metrics – single detector trained with KNN.

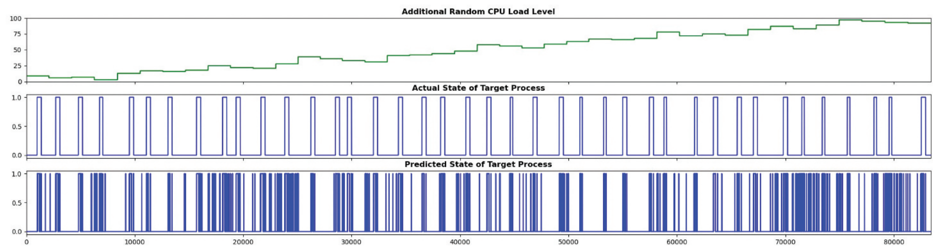

Figure 19. Virtual machine running prediction time series – single detector trained with KNN.

negatives, where the binary prediction fluctuates despite needing to be held to 1 since the process is still running on the interval. Still, from the previous discussions, we know that the random spikes can be accounted for through moving averages and that metrics of DPS and fallout rate are still high. The visual itself highlights the random spikes due to the resolution of the line widths, from needing to represent 40 hours of data. There are not as many false positives as there might be seen, and the fallout rate is a better indicator of false-positive rates than the visual.

Since model performances of decision trees are high, we can look at the feature importance to gain insight into what system component's features are important. From the pie plot in Figure 20, we see that the system component with the majority feature importance of 51.7% is the CPU. The second most important feature component is memory, capturing 30.0% of feature importance. Overall, these results make sense since elevated CPU usage and memory

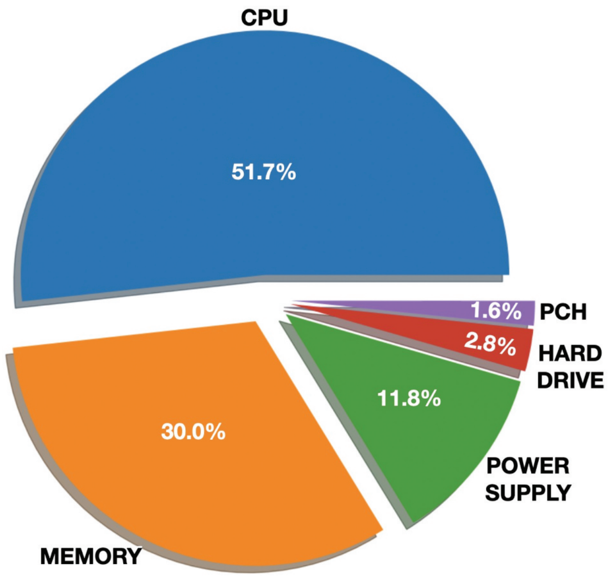

Figure 20. Virtual machine running feature importance by system component.

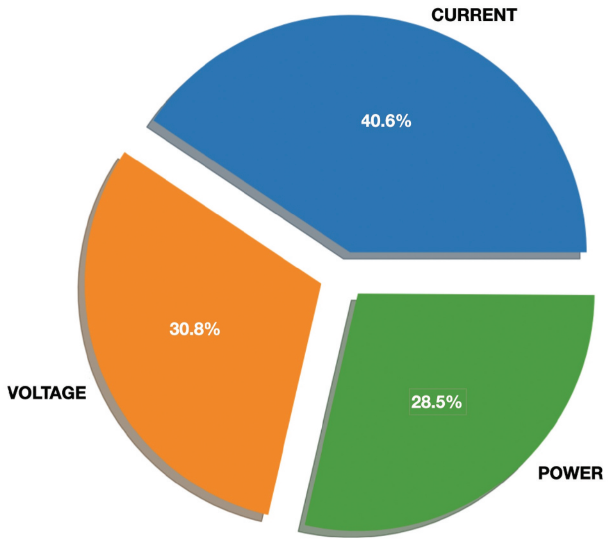

Figure 21. Virtual machine running feature importance by sensor type.

can indicate virtual machine usage. However, as previously mentioned, the additional CPU load did not change the MCC over time. This means that the model detects more complex patterns of resource fluctuations that specifically distinguish VM CPU usage from general process CPU usage.

Figure 21 shows an almost evenly balanced distribution of feature importance between features representing sensors that measure power, current and voltage.

The multiple detector comparison shown in Table 8 again yields the idea that the single detector model is the highest-performing. This is because there is a simple characterization of physical sensor changes of whether the virtual machine is running or not due to the fact that virtual machines require specific and repeatable allocation of resources. Since this characterization is simple, advanced models are not needed.

4.2.2. Simulated user process on virtual machine

Table 9 shows that the model trained to detect virtual processes that are running on a virtual machine has the highest DPS of 0.9682 in the random forest classifier. The fallout is low at 0.0113, and the RPD is 1, resulting in this

Table 8. Virtual machine running detector type comparison.

Detector	DPS	MCC	RPD	Fallout	TPD
Single	0.9839	0.6658	1.0	0.0065	0.425
Ens 6 Single	0.9724	0.5553	1.0	0.0064	1.825
Ens 11 Single	0.9766	0.6174	1.0	0.008	0.95
Ens 6 Dual	0.9544	0.6301	1.0	0.0183	1.25
Ens 11 Dual	0.9461	0.6743	1.0	0.0259	0.5

Table 9. Simulated user script running on virtual machine unknown additional load test results.

Algorithm	DPS	MCC	RPD	Fallout	TPD
Simple Random					
N Bayes	0.9145	0.5076	0.9667	0.025	0.5517
Ran For	0.8054	0.5422	0.8333	0.0101	1.72
KNN	0.75	0.5818	0.7667	0.0074	0.7826
Dec Tree	0.6844	0.5495	0.7	0.0072	1.0476
MLP	0.617	0.5489	0.6333	0.0056	1.7895
E Tree	0.5846	0.526	0.6	0.0055	1.7778
SVC	0.5825	0.5605	0.6	0.0039	2.7778
Log Reg	0.5759	0.5224	0.6	0.0122	1.8333
Advanced Random					
Ran For	0.9682	0.4535	1	0.0113	1.2
Log Reg	0.9383	0.3999	1	0.0089	5.35
N Bayes	0.9343	0.3972	1	0.0325	0.375
KNN	0.908	0.3017	0.95	0.006	3.8684
E Tree	0.8496	0.1873	0.925	0.0024	9.1892
SVC	0.8228	0.2718	0.925	0.0006	13.1081
Dec Tree	0.7711	0.1855	1	0.2429	1.775
MLP	0.6617	0.1056	0.8	0.0002	20.9688

high DPS. As seen previously for other models, the MCC is lower at 0.453 though still greater than 0, resulting in possible issues related to false negatives, but as mentioned earlier, for the detection of the process, this is much less important since we simply can average detections over multiple windows of the intervals and find the intervals where the process is running and starting. This DPS score of 0.9682 to detect the virtualized user behavior is less than the previous DPS of 0.989 and MCC of 0.6658 for detecting whether the virtual machine is running. This is unsurprising since the behavior of simply detecting a virtual machine is related to a predetermined and predictable resource allocation that the sensors can inform of, whereas detecting whether an actual user is running commands requires the model to more closely distinguish static virtual machine resource allocation from the resource allocations created by virtual machine interactions.

Another interesting behavior only seen for the detection of this process category is the finding that the simple random load performs worse. The top model for these test data is the naive Bayes with a DPS of only 0.9145 and an MCC of 0.5076. Even though the additional random CPU load is more complex, the model seems to use the complex additional load to distinguish between when the virtualized processes are running and when they are not running. We continue to use the results of the random additional CPU load since it is the more realistic testing scenario. Still, the possible reason for this improvement can be that when there is a more strenuous utilization of resources, every additional virtualized process has an even greater effect on the measurable sensor readings than when fewer resources are used.

Figure 22 shows that during the advanced random load testing, the random forest detection model maintained a high DPS and a low fallout rate without any major performance impact as the additional random load became larger.

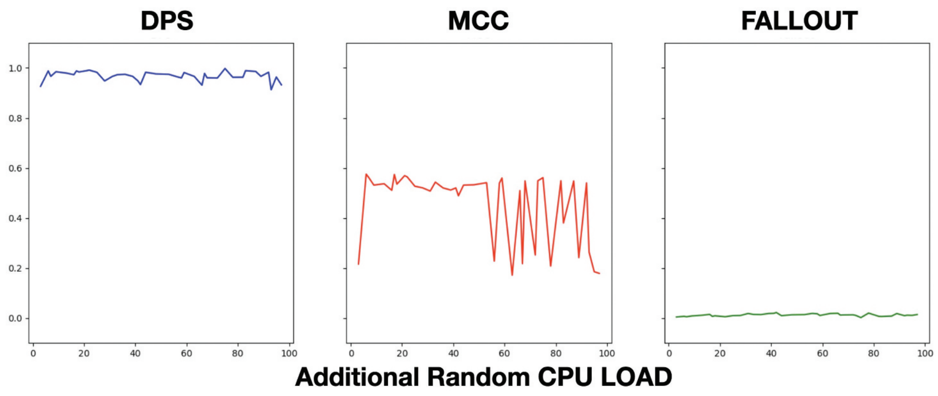

Figure 22. Simulated user script running on virtual machine binary classification metrics – single detector trained with random forest.

However, with around 55% additional random load, the MCC score became very erratic and appears to indicate that the prediction model is not a very good pure binary classification model.

Figure 22 shows the results consistent with the previous metrics of high DPS, low fallout and medium-level MCC. We do notice that the MCC starts off low and increases as the CPU load increases; then again, after a certain level of additional CPU load during testing, the MCC begins to fluctuate after the 55% additional random load.

In Figure 23, we visualize the testing of detection results and can visually see that there is a very high level of overlap between the predicted state and the actual state of the process running. As we mentioned previously, the fluctuations within the intervals themselves are related to the false-negative rates and are shown by the fluctuations resulting in dark-filled intervals and are indicated by the MCC of 0.4535. The overall detection of active process intervals starting and stopping without many false negatives seems to indicate that the detection model is a good process detection tool despite not having very strong binary prediction rates.

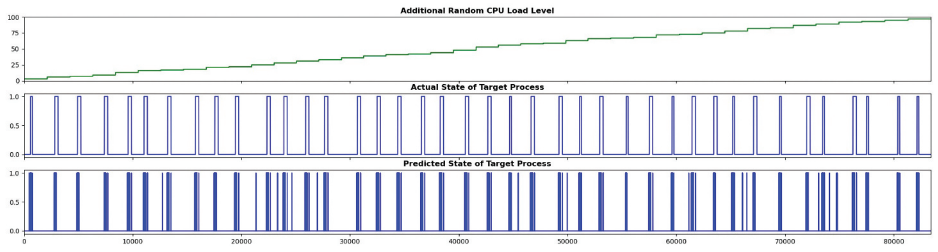

Figure 23. Simulated user script running on virtual machine prediction time series – single detector trained with random forest.

Figure 24 shows the distribution of feature importance based on the system component measured by the features. The most important physical system component to the detection of virtualized processes is the CPU, with a percentage of 53.8%. This model gives the CPU component more importance than all the other process detection model categories. This may be related to the fact that CPU utilizations for both virtualization tasks and, in particular, the processes running on virtual machines require fine-grained CPU information in order to consistently discriminate between whether the CPU usage is related to running a virtual process on the virtual machine or an additional process load. Since the DPS scores of the virtualization process detection tasks are high, the sensors seem to be measuring the stress on the physical system very precisely.

Figure 25 shows the feature importance breakdown by sensor type instead of system component. Here, we see that there is an even distribution of feature importance but also a slight increase in feature importance toward current sensors. This is most likely related to changes in the current sensor type, which changes more quickly during the start of the virtual machine and then increases further when a process runs on the virtual machine.

Table 10 reveals again that the single model detector has the best DPS. However, we notice that the TPD is significantly slower since it is around 1.2 seconds, whereas the more complex dual-ensemble models have a much lower TPD of 0.01 seconds with a reduction of DPS to 0.9148. This reveals that for complex process detector categories, such as

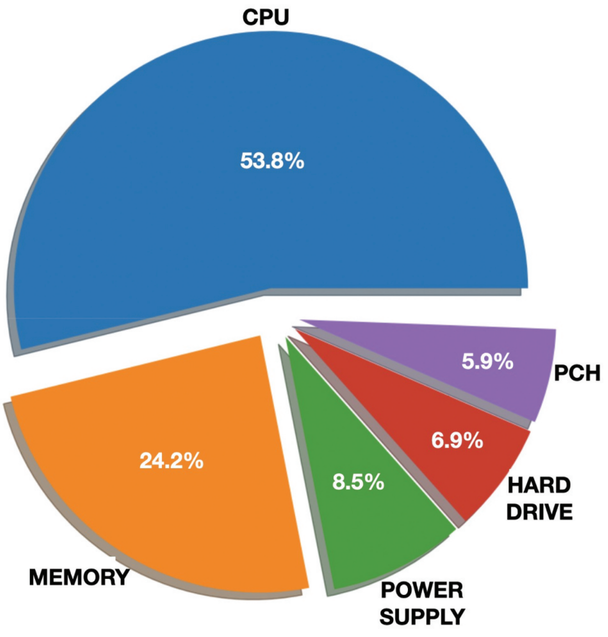

Figure 24. Simulated user script running on virtual machine feature importance by system component.

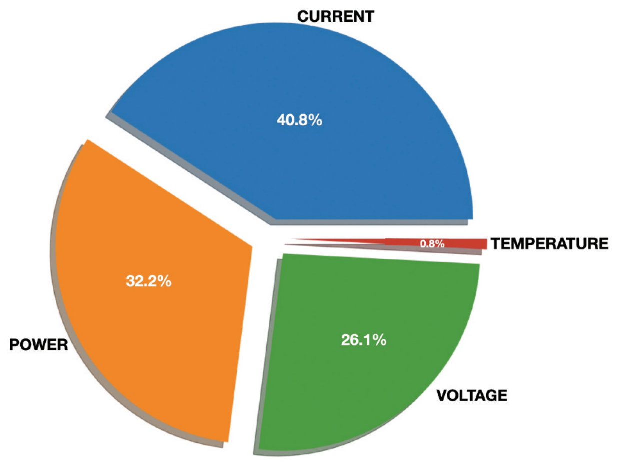

Figure 25. Simulated user script running on virtual machine feature importance by sensor type.

Table 10. Simulated user script running on virtual machine detector type comparison.

1Detector	DPS	MCC	RPD	Fallout	TPD
Single	0.9682	0.4535	1.0	0.0113	1.2
Ens 6 Single	0.9379	0.282	1.0	0.0289	0.775
Ens 11 Single	0.9395	0.3037	1.0	0.0306	0.2
Ens 6 Dual	0.9148	0.3928	1.0	0.0443	0.01
Ens 11 Dual	0.9101	0.4004	1.0	0.0467	0.01

Table 11. Selected detector types and their performance.

Process Type	Top Detector	DPS	MCC	RPD	Fallout	TPD
File I/O	Single	0.9968	0.9982	1.0	0.0005	0.275
CPU Intensive	Single	0.989	0.7878	1.0	0.0002	1.275
Network I/O	Single	0.5484	0.017	1.0	0.472	1.95
Virtual Machine Running	Single	0.9839	0.6658	1.0	0.0065	0.425
Simulated User Script	Single	0.9682	0.4535	1.0	0.0113	1.2

processes running on a virtual machine, accounting for factors such as CPU load can give more information that allows us to obtain better performance metrics.

5. Conclusion

Normally, attackers use side-channel information to obtain knowledge of the system, circumvent proper authorization and exploit security flaws. However, we show that side channel data can also be used by a defender or security administrator to find anomalous behaviors. We obtain sensor readings while different processes are running on a computer, use these as input into a machine learning classifier and use this to obtain a binary indication in which that process category is running. Our experiments describe the setup needed to train the classifiers, and we test the classifiers thoroughly with advanced loads that replicate the behaviors that we may see in real-world scenarios. When we run multiple process detection models at once, we can obtain side-channel information that gives us a fingerprint of the system through its regular behavior. This indicates the viability of using sensor information combined with machine learning models to develop Physical Sensor Side Channel (PSSC) data streams to detect arbitrary target processes. While prior studies have indicated the efficacy of process detection through complex behavioral analysis, they often come at the expense of speed [14] and generalization. Our novel detection model provides a rapid signal of a target process from sensors, facilitating timely safety precautions. A more in-depth behavioral analysis of processes can then serve as a subsequent verification step. In essence, our method complements traditional process detection techniques, aiming to bolster their effectiveness through real-time monitoring of sensors.

We can summarize the results of using PSSCs to detect the five classes, including the two VM classes, as follows: file I/O, CPU/ALU intensive, network I/O, virtualization instantiation and simulated user script. The results reveal that the detection performance differs among these classes. Classes file I/O and CPU/ALU intensive consistently demonstrate high detection rates across all background load conditions. In contrast, network I/O yields lower detection rates, while virtualization instantiation excels under low loads but sees diminishing accuracy as the load intensifies. The most challenging detection case arises with network I/O, which we believe stems from our NIC lacking an adequate set of physical sensors. Performance might improve with NICs that include more sensors or are positioned near other sensor-rich subsystems.

Our interpretation is that with this hardware setup, we will be able to detect certain behaviors involving file I/O and CPU-related tasks seen in ransomware but may not be able to detect certain network-level behaviors that could be similar to a network denial-of-service attack through hardware sensors alone unless an NIC with more sensors is added.

In our experiments, we visualize the DPS, MCC and fallout over time as CPU loads increase and obtain the RPD and TPD metrics across the entire test interval. We see that the effect of the load on the MCC being added does not impact file I/O, whereas all other process detectors rapidly degrade midway through the tests. However, when we visualize the binary vectors over time, we realize that DPS is more aligned with actual process detection than other classification metrics. This is shown when we see that the false negatives that are occurring are not as critical to the system as long as at least one true positive is detected in the interval where the process is detected. For this reason, DPS is the main metric we use to describe whether a certain process class can be used in a larger anomaly detection model. The MCC is significantly lower for classes network I/O, virtualization instantiation and simulated user script than for file I/O, though the DPS is just as high. The experiment reveals that when using these models to detect an anomaly, the classifiers only need to be trained to a certain minimal level of MCC that may not need to be near 1. Instead, the classifiers we train on must have a high DPS score. This makes our approach of using the general process detectors as an intermediate step of detecting anomalies an easier strategy when focusing on DPS.

Drawing insights from Tables 2, 4 and 6, it is evident that the single prediction model detector surpasses the ensemble detectors across all four target processes according to the DPS metric. Upon examining the data presented in Table 11, we can discern the models with the most favorable performance metrics, notably those demonstrating the highest DPS under the advanced random loads. The fallout rates are notably low, all under 0.01, except for Network I/O (0.5489) and Simulated User Script Virtual Machine (0.0113). When considering the Detector Performance Score (DPS), our chosen performance metric, it is evident that most categories score impressively, with DPS values surpassing 0.96, except for Network I/O, which achieved a DPS of 0.5444. The combination of low false-positive rates and high DPS scores makes the process classifiers reliable for determining the occurrence of these general process types using sensor data. Regarding model selection, it is notable that for most process classes, the single models exhibit the highest DPS and lowest fallout rates. The process classes that had the best results from the complex models, Network I/O and VM processes, still had low detection rates, showing that more complex models could not capture more information from sensors to make accurate classifications.

The usage of similar ensemble detection models in Taylor et al. [8] proved to have better results than single prediction models during the detection of more intricate ransomware attacks. However, from our resulting experiments, for the creation of simple binary prediction models for more general monitoring, the complexity tradeoff of the models does not seem to result in better discernment of process categorization. The interpretation of the fact that the simple model outperforms others is that the process detection task is not highly complex. In the future, using the single prediction models, we could examine the multiple binary vectors for multiple process detection classifiers at once and create more advanced anomaly detector models based on the binary prediction vectors or other intermediary information vectors. The anticipated complexity of the physical systems may result in the need for more complex machine learning models to deal with multiple process categories and multiple processes running at once. However, the next step of integrating the binary vector detection of these processes and integrating those into a higher-level customized malware and anomaly detector would necessitate a more complex model due to the variability in the way different attacks are performed. The next steps must also avoid using too much time since, as we see, TPD adds up quickly. Already, almost more than 1 second is needed to detect according to the TPD of most models in 11; depending on the method of attack, 1 second may cause significant damage in critical systems.

Along with the testing of process detectors, we also contribute importance values for sensor types and system components for detecting various target processes. In terms of what sensor type is most useful, the current power and voltage are all equally important for detecting all process categories. The importance values of hardware system components, including the CPU, PCH, memory, hard drive and power supply, can help system administrators focus on the certain sensor placement that the models will use to detect the specific target process. The knowledge can be used to understand whether a system has high monitoring capability for the detection of the target process based on the sensors available. For example, we show that CPU utilization is important to the four detection categories, all except for the CPU process category, which primarily uses memory. This indicates that hosts with sensors available for CPUs are very useful for process detection in general but also that CPU-intensive process category detection will need more sensors on the memory component to work effectively. The second most important system component varies; the file IO process requires an equal amount of importance from the PCH, memory, hard drive and power supply. Overall, across all process categories, the CPU is more important than memory and power supply. Hard drive system component sensors are only needed for file I/O-intensive processes and the processes that simulate processes on virtual machines.

Overall, there are some limitations we must address in hardware sensor placement, availability and speed of access. A lack of sensors can cause issues detecting certain types of anomalies. In our case, the lack of NIC sensors results in low network process detection. This could, in turn, impact our ability to detect DDOS attack scenarios that rely on network IO. Each general process detection model needs to be trained on an adequate number of sensors, and this varies from system to system, so the results of this work may not necessarily extend to systems where sensor data is purposely obfuscated. The other limitation is the time it takes for sensor data to be classified and the resulting damage done. For attacks that less than a few seconds, there may not be enough time for a higher level classifier to analyze the binary prediction vectors. This paper still highlights the potential of finding general process categories through the various sensors available on host machines and using the chosen general process detectors with high DPS scores. This would then allow us to use the models as part of automated responses or, as a next step, use the binary state vectors in a more complex model.

Disclosure statement

No potential conflict of interest was reported by the author(s).

ORCID

Aviraj Sinha http://orcid.org/0000-0002-5280-1585

References

[1] Lyu Y, Mishra P. A survey of side-channel attacks on caches and countermeasures. J Hardw Syst Secur. 2018;2(1):33–50. doi: 10.1007/s41635-017-0025-y

[2] Mangard S, Oswald E, Popp T. Power analysis attacks: revealing the secrets of smart cards, vol 31. Heidelberg, Germany: Springer Science & Business Media; 2008.

[3] Taylor MA, Larson EC, Thornton MA. General process detection through physical side channel characterization. In: 2022 IEEE International Systems Conference (SysCon) Montreal, QC, Canada. IEEE; 2022. p. 1–8.

[4] Zhang ZM, Guo Y-B. Survey of physical unclonable function. 2013;32(11):3115–3120. doi: 10.3724/SP.J.1087.2012.03115

[5] Kumar S, Mahapatra A, Swain A, et al. Microprocessor based physical unclonable function. In: IEEE International Symposium on Nanoelectronic and Information Systems (iNIS); 2017. doi: 10.1109/iNIS.2017.59

[6] Sinha A, Taylor M, Srirama N, et al. Industrial control system anomaly detection using convolutional neural network consensus. In: 2021 IEEE Conference on Control Technology and Applications (CCTA) San Diego, CA, USA. IEEE; 2021. p. 693–700.

[7] Jeffrey N, Tan Q, Villar JR. A review of anomaly detection strategies to detect threats to cyber-physical systems. Electronics. 2023;12(15):3283. doi: 10.3390/electronics12153283

[8] Taylor M, Larson E, Thornton M. Rapid ransomware detection through side channel exploitation. In: IEEE International Conference on Cyber Security and Resilience Rhodes, Greece; 2021.

[9] Bracken B. "What's next for ransomware in 2021?". 2020. Available from: https://threatpost.com/ransomware-getting-ahead-inevitable-attack/162655/

[10] Correa R. "How fast does ransomware encrypt files? Faster than you think". 2016 [cited 2016 Apr]. Available from: https://blog.barkly.com/how-fast-does-ransomware-encrypt-files note:This is an electronic document.

[11] U.S. Department of Homeland Security. Alert (TA16-091A) ransomware and recent variants. 2016. Available from: https://www.us-cert.gov/ncas/alerts/TA16-091A

[12] Suhag A, Daniel A. Study of statistical techniques and artificial intelligence methods in distributed denial of service (ddos) assault and defense. J Cyber Secur Technol. 2023;7(1):21–51. doi: 10.1080/23742917.2022.2135856

[13] Khonde S, Ulagamuthalvi V. Ensemble-based semi-supervised learning approach for a distributed intrusion detection system. J Cyber Secur Technol. 2019;3(3):163–188. doi: 10.1080/23742917.2019.1623475

[14] Rhode M, Burnap P, Jones K. "early-stage malware prediction using recurrent neural networks". Comput & Secur. 2018;77:578–594. doi: 10.1016/j.cose.2018.05.010

[15] Bresink M. "Hardware monitor: reference manual". 2017. Available from: https://www.bresink.com/osx/216202/Docs-en/index.html, note: This is an electronic document. Date of publication: [2017].

[16] Anaconda I. Individual edition. 2021. Available from: https://www.anaconda.com/products/individual

[17] Chollet F. Deep learning with Python. (NY): Manning Publications; 2017.

[18] Pedregosa F, Varoquaux G, Gramfort A, et al. Scikit-learn: machine learning in Python. J Mach Learn Res. 2011;12:2825–2830.

[19] Pedregosa F, Varoquaux G, Gramfort A, et al. “Scikit-learn user Guide”. Tech rep. Scikit-Learn Development Team; 2010.

[20] Norcott W, Capps D, Crawford I, et al. Iozone filesystem benchmark. 2021. Available from: https://www.iozone.org/

[21] Team FD. Ffmpeg: a complete, cross-platform solution to record, convert and stream audio and video. 2021. Available from: https://www.ffmpeg.org/

[22] Lyon G. Nmap security scanner. 2021.Available from: https://nmap.org/

[23] VMware, Inc. VMware fusion 12 pro. 2021.Available from: https://store-us.vmware.com/vmware-fusion-12-pro-5424173700.html

[24] Vikyd. Vikyd/go-cpu-load: generate CPU load on Windows/Linux/Mac. 2021. Available from: https://github.com/vikyd/go-cpu-load

[25] ColinIanKing. Colinianking/stress-ng: this is the stress-ng upstream project git repository. stress-ng will stress test a computer system in various selectable ways. It was designed to exercise various physical subsystems of a computer as well as the various operating system kernel interfaces. 2022. Available from: https://github.com/ColinIanKing/stress-ng

[26] Picek S, Heuser A, Jovic A, et al. Side-channel analysis and machine learning: a practical perspective. In: 2017 International Joint Conference on Neural Networks (IJCNN); 2017. p. 4095–4102. doi: 10.1109/ijcnn.2017.7966373

[27] Tharwat A. Classification assessment methods. ACI. 2020;17(1):168–192. doi: 10.1016/j.aci.2018.08.003

[28] Chicco D, Jurman G. The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation. BMC Genomics. 2020;21(1). doi: 10.1186/s12864-019-6413-7

[29] HNSM, Security HN. Increasing number of false positives causing risk of alert fatigue. 2020. Available from: https://www.helpnetsecurity.com/2020/03/24/alert-fatigue/.

[30] Perrier A. Feature importance in random forests. Available from: 2015. https://alexis perrier. com/datascience/2015/08/27/feature-importance-random-forests-gini-accuracy. html

Appendices

Appendix A. IO Process Run Commands

The iozone command to generate the IO-intensive process in Section 4.1.1 is shown here.

```
$ iozone −a
```

The flag -a runs all 13 tests back-to-back. This command requires around 8 minutes to complete on our Mac Mini system.

Appendix B. CPU-intensive Process Run

Commands

The CPU-intensive process used in Section 4.1.2 is generated using the command shown here.

```
$ ffmpeg −i             video.mov −c : v libx264 −crf 10 video.mp4
```

The -i flag marks the input video file name as an argument. The flag -c:v marks the encoder type as an argument. The -crf marks that constant rate factor used by the encoder. In this procedure, we utilized the Constant Rate Factor (CRF) scale to adjust video quality and file size, applying a value of 10 to achieve a very high-quality output. The CRF scale operates within a range from 0 to 51: a setting of 0 delivers lossless quality, indicating no loss of video data; 23 is established as the default value, offering a balance between quality and file size; and 51 marks the lowest quality level, significantly reducing file size. Opting for lower CRF values, such as 10, not only improves the video's output quality but also results in a larger file size and requires a higher computational effort. On our Mac Mini systems, executing this command with the specified CRF setting took approximately 6 minutes.

Appendix C. Network IO Run Commands

The network IO-intensive process used in Section 4.1.3 is generated using the nmap command shown here.

```
$ sudo nmap −Pn −T4 10.10.10.0/24
```

The -Pn flag indicates port scan test. The flag does not allow host discovery and searches through all hosts. The -T4 flag marks an argument between 0 and 5 to throttle the speed of the scan. In this case, values on the lower end indicate reduced performance speeds, whereas values on the higher end signify enhanced performance speeds. Specifically, for our experiment, the timing template is positioned just one step below the maximum speed, indicating that it will rapidly produce a significant volume of network traffic. The execution of this command on our Mac Mini systems took about 5 minutes to complete.

Appendix D. Virtualization Process Run

The vrun commands in this section launch, display the virtual machine, run a bash script and shut down the virtual machine in Section 4.2.

The command used to launch and display the virtual machine:

```
$ vmrun start            process.vmx gui
```

The path for the.vmx file extension is for the configuration file for the target virtual machine. The last command line argument -gui launches a visual of the virtual machine on a desktop monitor. The virtual machine was configured to log in automatically, bypassing the need for a password. It took approximately 2 minutes for the virtual machine to fully boot up. Following startup, we executed a command on the virtual machine that specified both the path to the Bash interpreter and the path to a script we had developed. This script was designed to simulate user activity on the virtual machine.

The command used to run the bash script:

```
$ vmrun –gu <USERNAME> –gp <PASSWORD> runScriptInGuest process
    .vmx –interactive"""/bin
    /bash usersim 5"
```

The -gu flag marks username input for accessing the virtual machine is the following command line argument. Additionally, the -gp flag allows for password input for logging in to the virtual machine with the username. The path to the.vmx file is for the configuration file for the target virtual machine. The option-interactive is required to run a script on the virtual machine without from the operating system. This argument is the command to be run on the guest virtual machine command line. The usersim Bash script leverages the Stress-ng stress test suite to initiate three different stressors on the guest virtual machine for a set duration, as detailed by [25]. It sets up a trio of stress tests: a CPU stressor focusing on matrix multiplication consuming 30% of CPU capacity, a mixed I/O stressor executing a variety of I/O tasks and a hard drive stressor engaged in constant reading and writing to the disk. These stressors aim to mimic actual user activity on the system. Each execution of the usersim script occurs over a 5-minute period following the virtual machine's complete boot process.

Once the usersim script has completed the 5-minute run, the virtual machine is shut down using the command:

```
$ vmrun stop           process.vmx soft
```

The path to the.vmx file serves as the configuration file for the designated virtual machine. The final command line argument, soft, indicates that the virtual machine is to be powered down via the system's shutdown script. The entire process, encompassing the startup of the virtual machine, execution of the usersim script and the subsequent shutdown, takes approximately 7 minutes to complete.