ENHANCED SECURITY UTILIZING SIDE CHANNEL DATA ANALYSIS

Approved by:

Dr. Mitchell Thornton - Committee Chairman

Dr. Eric Larson

Dr. Jennifer Dworak

Dr. Duncan MacFarlane

Dr. Ronald Rohrer

ENHANCED SECURITY UTILIZING SIDE CHANNEL DATA ANALYSIS

A Dissertation Presented to the Graduate Faculty of the

Lyle School of Engineering

Southern Methodist University

in

Partial Fulfillment of the Requirements

for the degree of

Doctor of Philosophy

with a

Major in Computer Engineering

by

Michael A. Taylor

(B.S., CSE, Southern Methodist University, 2017)
(M.S., CSE, Southern Methodist University, 2017)

December 18, 2021

ACKNOWLEDGMENTS

I am very grateful to everyone who has supported me through my education and helped me push through to the end. I would above all like to thank my advisor Dr. Mitchell Thornton who I have worked with through nearly all of my academic career. Without Dr. Thornton's mentorship and encouragement I would not have thought I was capable of reaching this point. I would also like to thank the other members of my committee Dr. Eric Larson, Dr. Jennifer Dworak, Dr. Duncan MacFarlane, and Dr. Ronald Rohrer for their insight and direction during my proposal which was instrumental in determining the focus of my research. Last but not least, I would like to thank my parents Jim and Julie Taylor for their support and encouragement through all the highs and lows of completing a PhD.

Taylor , Michael A. B.S., CSE, Southern Methodist University, 2017
M.S., CSE, Southern Methodist University, 2017

Enhanced Security Utilizing Side Channel Data Analysis

Advisor: Dr. Mitchell Thornton - Committee Chairman
Doctor of Philosophy degree conferred December 18, 2021
Dissertation completed November XX, 2021

The physical state of a system is effected by the activities and processes in which it is tasked with carrying out. In the past there have been many instances where such physical changes have been exploited by bad actors in order to gain insight into the operational state and even the data being held on a system. This method of side channel exploitation is very often effective due to the relative difficulty of obfuscating activity on a physical level. However, in order to take advantage of side channel data streams one must have a detailed working knowledge of how a target behavior, activity, or process effects the system on a physical level which may not always be available to a would be attacker. However, the owner of a system has unfettered access to their own system and is able to introduce a target, measure the effect it has on the physical state of the system through system side channels, and use that information to identify future instances of that same target on their system. System owners using the physical state of their own system in order to identify targeted behaviors, activities, and processes will have the benefit of faster detection with only a small amount of computational resources needed. In this research effort we show the viability of using physical sensor side channel data in order to enhance existing security methods by way of the rapid detection inherent in this technique.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ACC Accuracy

AES Advanced Encryption Standard

ALU Arithmetic Logic Unit

AUC Area Under Curve

AWS Amazon Web Services

CBC Cipher Block Chaining

CFB Cipher Feedback

CNN Convolutional Neural Network

CRF Constant Rate Factor

CSV Comma-Separated Values

DMA Direct Memory Access

DPS Detector Performance Score

ECB Electronic Codebook

Ens Ensemble

ICS Industrial Control System

KNN K Nearest Neighbor

LSTM Long Short-Term Memory

MCC Matthews Correlation Coefficient

MDI Mean Decrease Impurity

MLP Multilayer Perceptron

MTAR Mean Time to Attack Recognition

ML Machine Learning

NIC Network Interface Controller

NN Neural Network

OFB Output Feedback

OHM Open Hardware Monitor

PCA Principal Component Analysis

PCAP Packet Capture

PCH Platform Controller Hub

PMU Power Management Unit

PSSC Physical Sensor Side Channels

RAR Rate of Attack Recognition

ReLU Rectified Linear Units

RSA Rivest-Shamir-Adleman

RPD Rate of Process Detection

RTU Remote Terminal Unit

Saas Software as a Service

SMA Simple Moving Average

SoC System on a Chip

SVC Support Vector Classification

TPD Time to Process Detection

WMA Weighted Moving Average

WMI Windows Management Instrumentation

XOR Exclusive OR

Chapter 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 amount of functions and operations a system must perform at a given time while also having to account for such things as timing and concurrent data usage. Given some amount of 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 as well as sensitive information being used by the system. 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 very specific fingerprint begins to form in the physical system which is even easier to detect when the processes being performed are operating in a predictable and repeating manner.

This data that is able to be collected as a product of the regular physical operation of a system is known as side channel data. While side channel data is most widely known in the cybersecurity community as a method of attack and exploitation there is also the opportunity to use side channel data for enhanced system security.

1.1. Side Channel Data

Side channel data is any information that is gained from the regular operation of a system which allows for additional insight that is able to be exploited. Potential examples of side channel data include timing information, power consumption, monitoring cache accesses, and

even system acoustics. In an early example of side channel exploitation carefully measuring the time that was required to perform private key operations allowed the attackers to find fixed Diffie-Hellman exponents, factor RSA keys, and compromise various cryptosystems (Kocher, 1996). This exploit required designers to implement countermeasures in order for timing to no longer be used against the system during private key operations. However, the countermeasures require the system to physically perform a task based on some set of instructions and given enough time such side channels may again be able to be exploited. It is recommended that any channel carrying information from a secure area to the outside, no matter how innocuous the data appears, should be viewed as a potential risk and studied.

The power of side channel exploitation often comes from combining multiple different data channels that offer very little insight on their own but can be very insightful when viewed collectively. This concept is why the ever increasing complexity of computational systems has the potential to increase the ability of side channel exploitation.

1.2. Machine Learning

Many of the most effective side channel attacks have implemented complex statistical analysis in order to determine additional information about the state of a system. The increase in the complexity of computational systems offers many more channels to draw data from than were previously available. Machine learning algorithms utilize data analytical techniques in an attempt to create a predictive model from multi-dimensional data sets (Camacho et al., 2018). When considering the highly complex and interdependent nature of modern computational systems there is great potential in being able to create side channel models for an entire system rather than just specific subsystems or components. Experiments have shown that attackers are able to create highly complex machine learning models which leverage side channels in order to unveil secret AES encryption keys in systems which implement countermeasures (Kubota et al., 2019).

1.3. Thesis Statement

While more complex computational systems are able to be exploited in more advanced ways by attackers there is also the possibility of using the same approach for system security. Attackers use side channel data to determine information about a system which they otherwise would not have access to directly. While the owner of a system has direct access to their own system there may be times where the system is under attack but traditional methods of detection are either slow or altogether unable to detect the attack due to evasive action. The attack would likely be causing the physical state of the system to vary on even the smallest level from the expected state seen during normal operation. Given that a system was recorded during normal operation for a sufficient amount of time and in a sufficient amount of states it may be possible to train machine learning algorithms to detect anomalous system states which are due to malicious activity or faulty operation. Additionally, due to the machine learning models being trained based on physical side channel data there is a possibility for very rapid detection of malicious activity.

Chapter 2

Background

This research effort has focused on incorporating side channel data streams from physical sensors into machine learning prediction models for enhanced security of single physical systems. Experiments have been carried out in one instance using only the existing sensors in a physical system and in a second instance additional sensors and monitoring devices are introduced into the physical system solely for security. It is our hypothesis that the same basic side channel analysis principles implemented in a physical system can be adjusted and applied to remote and virtual environments for enhanced security.

2.1. Ransomware

Malware is a term that is used to refer to malicious software and is used to refer to all forms of software that can be used to compromise computer functions. This compromise causes harm to the victim computer and ultimately to the user or owner of the host computer. There are a large variety of types of malware including, viruses, worms, adware, bots, rootkits, spyware, Trojans, and the primary subject of this investigation, ransomware. Ransomware is a form of malware that holds a victim's computer system files hostage while demanding a ransom to release access to those files back to their legitimate owner.

A typical ransomware attack scenario involves infection of victim computer through penetration of an attack vector whereby the malware resulting from the attack contains a payload that, unbeknownst to the victim, engages in rendering important files as unusable, through their encryption with a key that is unknown to the victim. Upon completion of the initial silent encryption phase, the original unencrypted files are deleted and the victim is alerted

that their files are now inaccessible and will remain so until a ransom is paid. It is also often the case, that the attacker will demand ransom within some time period or otherwise the encryption key will be destroyed resulting in permanent loss of the victim's data. Figure 2.1 contains a high-level diagram of the chain of events characterizing a typical ransomware attack from the point of view of the adversary.

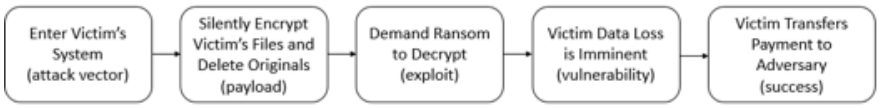


Figure 2.1. Typical Ransomware Attack

The largest ransomware attack in history, WannaCry, occurred in May 2017 with 230,000 computers in over 150 countries being infected within a few days. The spread of WannaCry was only halted by a web researcher in England who found a "kill-switch" which was engaged by registering a domain name found in the code (BBCNews, 2017). Just one month later, in June 2017, a ransomware attack known as Petya infected around 16,500 computers globally. One reported instance of the Petya attack requested 100 bitcoins, or about $250,000 dollars, in order to provide the key for decryption (Brandon R, 2017). The increase in ransomware attacks has also come with decreased infection times as attackers create new and better methods of file encryption. Testing performed by the cybersecurity company Barkley shows that many ransomware variants complete their encryption phase in under one minute. For instance, Petya finished encryption in 27 seconds and was still slower than Chimera which finished in only 18 seconds. Barkley found that 54% of attacks notified the victim of the ransom within one hour of infection (Correa R, 2016). More detailed information about well-known versions of ransomware, including how they infect computers can be found in (Abrams L, 2013), (O'Gorman G, McDonald G, 2012) and (Wyke J, Aijan A, December 2015).

Ransomware attacks on businesses, in 2016, were estimated to have occurred every two minutes. In 2017, the time between attacks fell to every 40 seconds—a tripling in the number of attacks. The variety and complexity of the attacks is also growing. For example, from 2016 to 2017, sixty-two new families of ransomware were documented with the number of new variants increasing eleven-fold (Kaspersky Labs, 2016). The following year several attacks occurred on an international scale by leveraging known NSA exploits and shared network resources of victims to propagate across entire networks rapidly. The largest of these attacks, WannaCry, infected more than 700,000 victims with 250,000 occuring in only four days (Kaspersky Labs, 2016). The spread of WannaCry was only stopped because a web researcher in England who found a "kill-switch" which was engaged by registering a domain name found in the code (BBCNews, 2017).

In 2019 a ransomware attack was estimated to occur once every 14 seconds in large part due to their success and the relative ease of launching an attack (Dobran, 2019). It is often pointed out that if you maintain rigorous backups of all critical data ransomware becomes more of an inconvenience than a true threat (Cybersecurity and Infrastructure Security Agency, 2020). The thought is that if your system is infiltrated and encrypted you can simply restore from a well maintained backup and continue working with the knowledge that none of your data actually left your system. However, in one instance of a ransomware attack it took just 45 minutes for consumer goods company Reckitt Benckiser to lose access to 17,500 assets in their network. The single company estimated their losses from the attack to be over $130 million dollars—not from data loss, but from the downtime of system restoration. During the time the company was under attack each second represented $48,000 worth of losses. It is estimated that the total ransom revenue collected by the attackers behind WannaCry was only $55,000 while the total economic loss is estimated to be as high as $4 billion (Monoghan, 2017).

Recently ransomware has been used to attack local and state governments by holding their critical systems hostage. Baltimore was attacked in May of 2019 with attackers demanding

a ransom of $76,000. The city government refused to pay resulting in millions of dollars being lost during their restoration efforts (Broadwater, 2019). The proximate threat posed by ransomware, then, comes from downtime rather than data loss or data security. Even if the systems can be restored from backups the downtime for restoration can be economically disastrous or even life threatening when critical services are unavailable (Ng, 2019). It has been suggested that ransomware may not always be about collecting money but instead a deliberate attack on a target's infrastructure and capabilities that can spread quickly. Ransomware attacks launched purely for destruction are effective due to how quickly they can take down entire networks of assets without the need to worry about being evasive (Fuller, 2017).

Combating attackers who weaponize ransomware requires detecting potential attacks quickly without resource intensive and time consuming analysis. In support of this, we use the physical state of a system, as it is measured through existing physical sensors, to generate prediction models that quickly flag physical system states consistent with a ransomware attack. Ideally this method would be effective alongside conventional methods of ransomware detection which are highly accurate but slower to complete analysis.

2.2. Physical Sensors

Most modern computer systems are comprised of sensors and associated processes that monitor the state of internal hardware components. These sensors continuously supply information that is communicated with other devices and subsystems for the intended purpose of ensuring that the system stays within specific operating specifications. If sensor data reveals that a system component is approaching a boundary of an operational specification, safety mechanisms are typically engaged to correct the internal environment so that system malfunctions can be prevented. For example, when the data from a temperature sensor that is monitoring a computer's central processing unit begins to increase in value, a signal is sent to the CPU cooling fan. This signal causes the fan to either become active or to increase the

fan speed to cool the CPU. Additionally, there are sensors that provide input to other subsystems such as internal power management units (PMU) to conserve power usage. Typically, computer system components are designed to be compact in size through the use of transistors with feature sizing in the nanometer scale. As a direct result, whenever computations become more complex, more stress is placed on a computer's hardware components. This increased stress occurs because a larger number of transistors are simultaneously switching in a circuit that correspondingly causes an increase in dynamic power consumption and results in more heat dissipation during heavy computational activity. Thus, monitoring the side channels of a system with embedded sensors that measure parameters such as temperature, power consumption, and battery voltage levels can give insight into the type of processing that is underway on a computer at a given time. Therefore, sensor data streams serve as side channels through which periodic observations can indicate when resource-intensive tasks, such as extensive file system I/O and encryption, are occurring. Because the silent phase of ransomware utilizes significant amounts of file system activity in combination with encryption, characteristic patterns present within a computer's sensor data may result in trends that are indicative of a ransomware attack.

A significant advantage of this approach, as compared to other side channel methods, is that the sensors and a means for querying them are natively provided. Thus there are fewer concerns in deploying and accessing sensors for the purpose of side channel exploitation. Furthermore, the trend has been that an increasingly diverse number of sensors are provided as integral components in modern computing devices. For example, a typical smart phone has many embedded sensors that could be used to support security applications including power monitors, accelerometers, ambient light sensors, antennas (including GPS receivers), fingerprint scanners, barometers, cameras, touchpad pressure sensors, and others. Even rack-mounted industrial servers contain a significant number of sensors that measure subsystem power consumption, temperature, and other environmental factors. All of these deployed sensors in modern computing devices provide a rich set of data sources that may be used to

provide internal side-channel information for the environment in which a computing device is operating. Sensors have been used in other security-related applications in the past. As an example, in (Alharbi A, Thornton M, Dec 2015), sensors present in mobile computing devices have been used to provide a user demographic classification capability for mobile devices with embedded touchscreens (Taylor et al., 2017).

2.3. Industrial Control Systems

Industrial control systems are used to control and carry out the the unique operations and automated processes of industrial environments. Industrial applications often have strict requirements for operation with little room for error. Critical infrastructure is almost always controlled by an industrial control systems which makes it a highly appealing target to attackers. Industrial control systems until fairly recently have been isolated systems with no need to be connected to any networks outside of the immediate area they are designed to operate in. The isolated nature of industrial control systems meant there was an inherent security to them which allowed for the use of very simple protocols. The Modbus RTU protocol has been used in industrial control systems since 1979 due to the relative easy of deployment and maintenance that is required. With Modbus RTU a programmable logic controller is connected to multiple devices using asynchronous serial data lines with simple unencrypted packets used for communication. In this type of system there is no means for identifying unanticipated devices, discovering new devices, or verifying the authenticity of a device on the bus (Gosine, 2020). Attackers could successfully monitor, disrupt, and modify communication on the bus if they were able to place an agent between two of the nodes in the system. With the reliance on the internet and the necessity for operators to access devices from a network, industrial control systems are having to be connected to networks. The reliance on the inherent security of isolation is no longer sufficient to keep these vital networks secure and operational (Jakaboczki and Adamko, 2015).

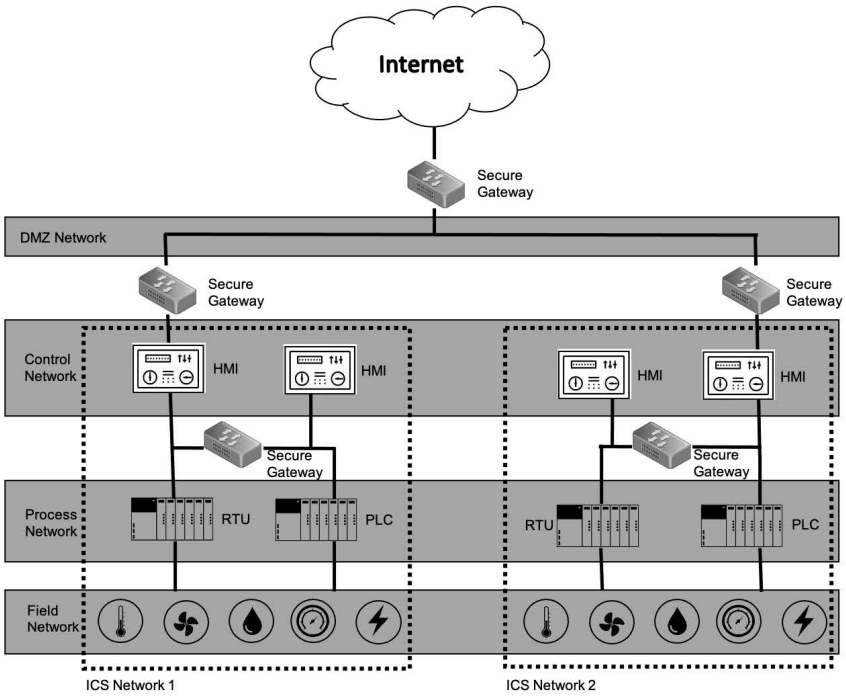


Figure 2.2. ICS Model

2.4. Industrial Network Protocols

Industrial network protocols have emerged that are able to connected industrial control systems together and communicate using concepts of computer networking. Modbus TCP is an industrial network protocol that is able to leverage the TCP/IP protocol stack by encapsulating Modbus RTU packets as the data in network packets. Using the Modbus TCP protocol, existing industrial control systems which have been operating using Modbus RTU protocol are able to be connected to more modern computer networks quickly and easily. However, the more advanced networking protocol allows for the underlying control system to be accessed much more easily and covertly. The same strict operational requirements must be maintained despite the more easily accessible and complex network. (Colbert and Kott, 2016)

Chapter 3

Industrial Control System Anomaly Detection

Manufacturing and industrial settings have become extremely complex. To address this increase in complexity, frameworks for automatic electronic control have been developed to keep large-scale processes running safely and without interruption. This collection of protocols that is essential for managing automation in manufacturing facilities, public infrastructure, and large-scale transportation networks is referred to as Industrial Control Systems.

An ICS is responsible for coordinating industrial operations so that they execute properly and on schedule. The protocols, connections, and devices that enable the communication between the active components in a factory setting are often collectively referred to as industrial ethernet. As these systems demonstrate state-like behavior to control industrial activity, monitoring this network is an effective way to perform reliability analysis to detect faulty equipment or processing errors. In this work, ICS network communication packets are analyzed by a long short-term memory (LSTM) based neural network (NN) in order to determine the overall health of the devices and components in an industrial environment. Machine learning techniques have been applied to industrial reliability analysis in the past (Alsina et al., 2018), but our technique differs in the sense that it is meant to require no additional overhead to initialize, just a connection to the industrial ethernet links that form the ICS network.

In this work, ICS network communication packets are analyzed by two convolutional neural networks (CNNs) in order to determine the overall health of the devices and components in an industrial environment. Machine learning techniques have been applied to industrial

reliability analysis in the past (Alsina et al., 2018), but our technique differs in that it is meant to require no additional overhead to initialize, just a connection to the industrial ethernet links that form the ICS network. The networks separately analyze the state of the systems using two different input streams: (1) the packet data sent along the network and (2) time series signals from an accelerometer and gyroscope. Each input corresponds to a different "view" of the system state. When the system is functioning properly, the state classified by each model should match or be reasonably similar. However, when faulty equipment or processing errors cause unexpected behavior in the system, the classification will diverge. Becasue the system diverges from normal behavior, this classification can also be described as anomaly detection. For example, [give example to be more clear].

Our goal is to use two streams of data to determine system anomalies. The command payloads must match with a specific sensor behavior. In the case of an ICS, there are different states the system can be in. Thus, by having two different models predict the state from either the payload or the sensor data, we can identify miscommunication errors between the network commands sent and the sensor behaviors. As a result, our model falls under the category of semi-supervised learning. This is due to the fact that, while the models use supervised training to identify state, the error between their detections is used to detect anomalies rather than the classifications themselves.

In this case, an error is defined as a difference between the predicted value of the two models. This technique contrasts with previous methods that would accumulate errors as standard deviations from the mean rather than model prediction errors. As a consequence, previous methods would require a normal distribution for both sensor and payload data, which is not always possible.

$$e_t = \begin{cases} 0 & y = \hat{y} \\ 1 & y \neq \hat{y} \end{cases}$$

Over time random errors can accumulate. However, over specific windows of time such as an attack, the percentage of errors per prediction should be high. Windows with high error rate will surpass a threshold and thus get labeled as an anomaly.

W is the window size and T being the number of times an error can occur over a window before the window is considered anomalous.

$$A_i = \sum_{t=i-W}^{i} e_i > T$$

3.1. Anomaly Detection Methodology

Our goal is to use two streams of data to determine system anomalies. The command payloads must match with a specific sensor behavior. In the case of an ICS, there are different states the system can be in. Thus, by having two different models predict the state from either the payload or the sensor data, we can identify miscommunication errors between the network commands sent and the sensor behaviors. As a result, our model falls under the category of semi-supervised learning. This is due to the fact that, while the models use supervised training to identify state, the error between their detections is used to detect anomalies rather than the classifications themselves.

In this case, an error is defined as a difference between the predicted value of the two models. This technique contrasts with previous methods that would accumulate errors as standard deviations from the mean rather than model prediction errors. As a consequence, previous methods would require a normal distribution for both sensor and payload data, which is not always possible.

$$e_t = \begin{cases} 0 & y = \hat{y} \\ 1 & y \neq \hat{y} \end{cases}$$

Over time random errors can accumulate. However, over specific windows of time such as an attack, the percentage of errors per prediction should be high. Windows with high error rate will surpass a threshold and thus get labeled as an anomaly.

W is the window size and T being the number of times an error can occur over a window before the window is considered anomalous.

$$A_i = \sum_{t=i-W}^{i} e_i > T$$

3.2. ICS Architecture

Our model architecture is described in the diagram. It uses a combination of convolutional, max pooling and dense layers. All activation functions are rectified linear units(ReLU) except for the final softmax activation for classification. The loss function used is categorical cross entropy for the 6 ICS states. An ADAM optimizer is at a learning rate of 1e-5 and is used to iteratively update the weights. The model is trained for 100 epochs. Proper training, validation, and testing splits are performed at the ratio of 70:20:10 to ensure the model can accurately detect ICS states from payloads and sensors. Data preprocessing, training results, and details on how models are used together are mentioned in the next sections.

3.3. Dataset Description

The dataset includes three axis gyrometer and three axis accelerometer input, as well as sensor data at a fixed sample rate of XXX. Network Data was preprocessed from PCAP files. There were 50 bytes between 0 to 255. These were converted to binary for machine learning input changing the input width from 50 to 424. Raw data is shown below for a single trial.

Though time deltas can be useful for anomaly detection, they do not supply enough information to help classify ICS states. However, since they can signify whether a state change is occurring, the data may become useful in the future applications.

3.3.1. Time Alignment

A key preprocessing step is aligning sensor data with corresponding packet payload time stamps and using them as input samples for machine learning. Though it takes some time for

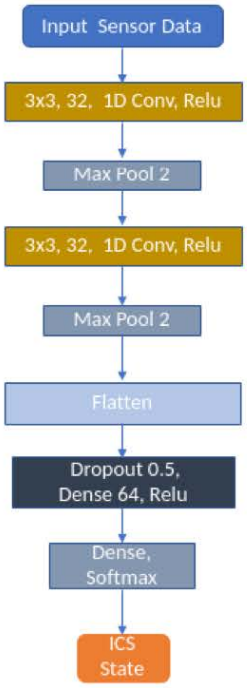


Figure 3.1. Diagram of Convolutional Architecture

the payload data to impact the ICS actuators, we can make this negligible by using multiple samples for machine learning input. While the various sensor inputs can be maintained at the same sample rate, the packet payload arrival time varies greatly depending on whether an ICS state transition is occurring and has a much lower sample rate. As a result, choosing a ratio of payload data to sensor data may need to be calculated. For our demonstration purposes, we used 100 samples of either payload and sensor information as input for one prediction.

3.3.2. Data Sampling and Preprocessing Steps

Our learning algorithm trains on chunks of data size 100 for both payload and sensor data streams. This setting lines up both data streams evenly across time and is also an

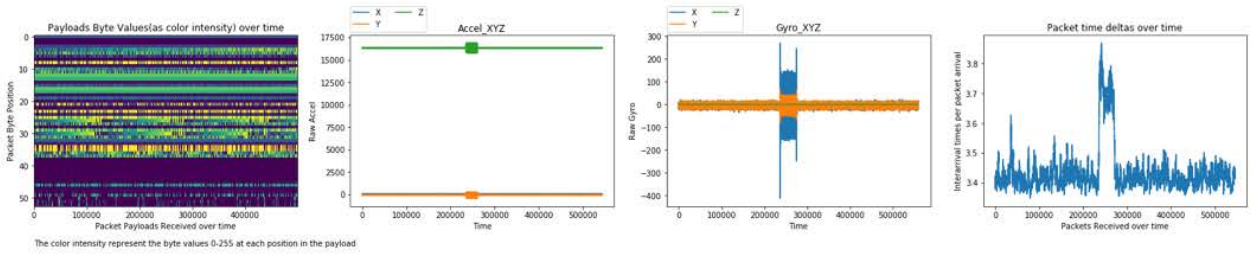


Figure 3.2. Visualizations of Dataset

important parameter for detection. A large chunk size may increase accuracy but may also make it slower to identify anomalies over small periods.

Accelerometer and gyrometer have 3 axis values. Raw sensor values are scaled by their minimum and maximum values to be between 0 and 1, due to their varying offsets.

Payload data widths were very large with 424 total bitstreams (converted from the original 50 bytes). For this reason, sample averaging was done to reduce 100 samples to 1 input. As shown in the training results next, averaging the packet bitstreams was able to improve classification by accounting for a constant amount of noise on the network.

3.3.3. Combining Models to Perform a Semi-Supervised Anomaly Detection

For the second part of our algorithm, the actual anomaly detection, we used the two previous models and analyzed when errors between the models would occur; as mentioned previously an error is defined as a disagreement between the two predictions. A sliding window of size 20 is used to calculate error percentage over time. In other words, every 20 predictions, produces an error rate. Plots are shown below.

This window size is another possible parameter for tuning. By using a smaller average window size, we can control false positive to false negative ratio, which can be optimized based on the costs of a misclassification.

By setting the error rate allowed to be around 18% per 20 predictions we could classify anomalies at a rate of 80% with a sample size of 25. There are 3 false negatives(undetected) and 2 false positives (detected too early). Results may be improved by better window sizes and other hyperparameters.

3.4. Experimental Results

Our results involve a combination of the accuracy of our supervised classifiers and their combine ability to distinguish anomalies. Alongside statistics such as accuracy, F1 score, AUC, and MCC are used to understand the skill of the classifier, its ability to classify better than random chance.

Another important statistic, is to look at latency of prediction. In fact, our method will have a sizeable latency due to determining anomalies only after the error rate of a window of time surpasses a certain point. For every prediction there are a 100 data points for sensor and packets, and for 20 predictions there is an anomaly flagged.

Table 3.1. Confusion Matrix for Anomaly Detection

		True Class		
		Anomaly	Baseline	Total
Result	Anomaly	19	7	26
	Baseline	4	10	14
	Total	23	17	40

3.4.1. Confusion Matrices

Here are the final statistics of the combined, unsupervised classifier: F1 score: 0.76 MCC: 0.42 ACC: 0.73 AUC:0.71

3.4.2. Latency of Classifier

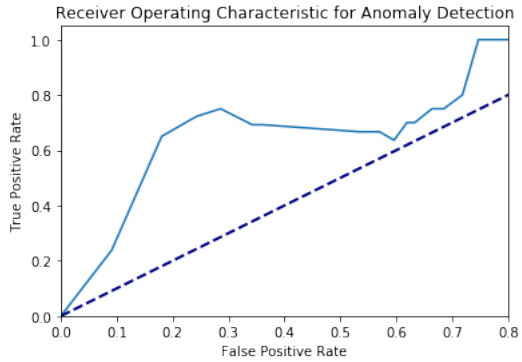


Figure 3.3. Receiver Operating Characteristics for Anomaly Detection

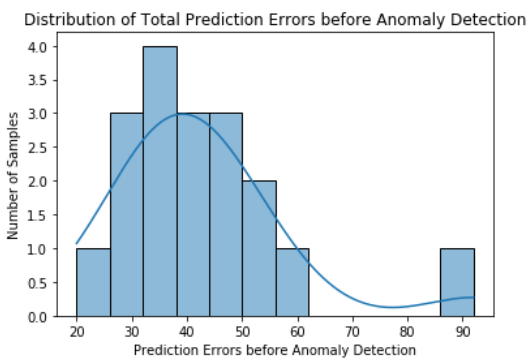
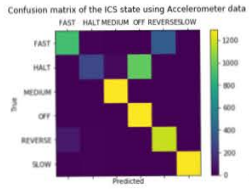


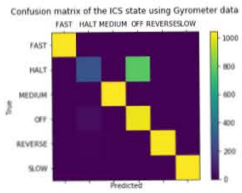
Figure 3.4. Distribution of Total Prediction Errors Before Anomaly Detection

The histogram above reveals latency in detection times(for true positive detections), before the anomaly is flagged. The median predictions before anomaly detection is 39.5. This means that about 3950 sensor and payload data were used in total to before the error was confirmed. This distribution has an outlier that took almost double the time to confirm the anomaly.

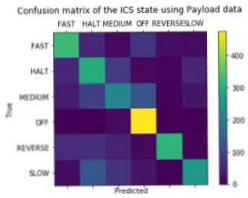
3.5. Summary and Contributions



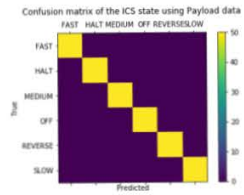
	precision	recall	f1-score	support
FAST	0.90	0.70	0.79	1300
HALT	0.84	0.22	0.34	1300
MEDIUM	0.97	0.99	0.98	1300
OFF	0.57	0.99	0.72	1300
REVERSE	0.76	0.91	0.83	1300
SLOW	1.00	0.99	0.99	1300
accuracy			0.80	7800
macro avg	0.84	0.80	0.78	7800
weighted avg	0.84	0.80	0.78	7800



	precision	recall	f1-score	support
FAST	1.00	0.99	0.99	1048
HALT	0.82	0.26	0.39	1048
MEDIUM	1.00	1.00	1.00	1048
OFF	0.57	0.97	0.71	1048
REVERSE	1.00	0.99	0.99	1048
SLOW	1.00	0.99	1.00	1048
accuracy			0.87	6288
macro avg	0.90	0.87	0.85	6288
weighted avg	0.90	0.87	0.85	6288



	precision	recall	f1-score	support
FAST	0.90	0.70	0.79	1300
HALT	0.84	0.22	0.34	1300
MEDIUM	0.97	0.99	0.98	1300
OFF	0.57	0.99	0.72	1300
REVERSE	0.76	0.91	0.83	1300
SLOW	1.00	0.99	0.99	1300
accuracy			0.80	7800
macro avg	0.84	0.80	0.78	7800
weighted avg	0.84	0.80	0.78	7800



	precision	recall	f1-score	support
FAST	1.00	1.00	1.00	50
HALT	1.00	1.00	1.00	50
MEDIUM	1.00	1.00	1.00	50
OFF	1.00	1.00	1.00	50
REVERSE	1.00	1.00	1.00	50
SLOW	1.00	1.00	1.00	50
accuracy			1.00	300
macro avg	1.00	1.00	1.00	300
weighted avg	1.00	1.00	1.00	300

Figure 3.5. Results of Classification for Individual Datastreams

The new classification technique reliably finds anomalies based upon the difference in prediction between two parallel neural networks, using two different, but complementary, views of the network to measure consensus: command packets and actuator sensors. This model can eventually be expanded to work with more complex and multistage ICS systems such as that of the SWaT dataset discussed in previous works. In the future, the area of multimodal machine learning may allow more complex interaction between payload and sensor data streams. This type of model would make it easier to incorporate data that does not directly identify ICS states such as interarrival times.

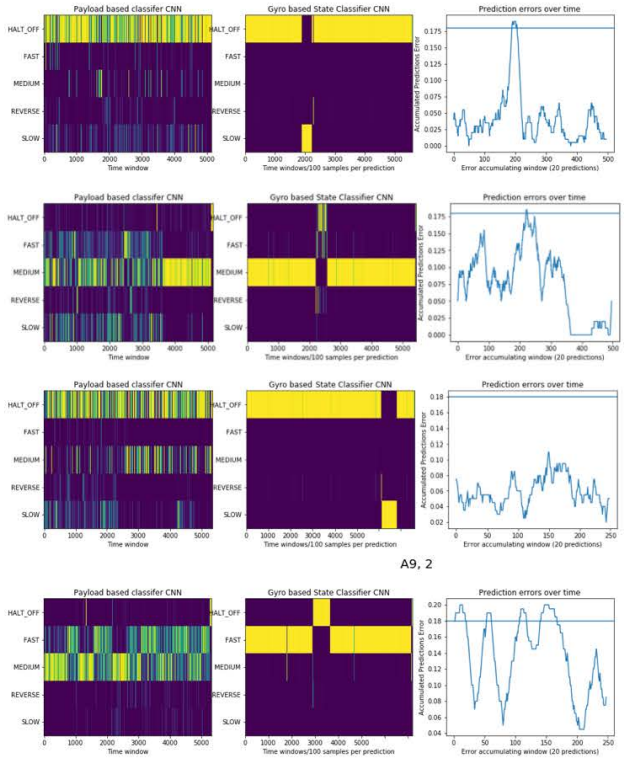


Figure 3.6. Anomaly Detection Results

Chapter 4

Rapid Ransomware Detection

Ransomware is launched in many different ways and with many different goals. Early ransomware simply encrypted the personal user files of the system under attack and would not provide the key for decryption until a ransom was paid to the attacker. In a later evolution the ransomware targeted and encrypted key system files and in most cases was able to lock a system from being used in only seconds. The attacker would place a message on the locked system which often claimed to be a legitimate entity and rather than a ransom the user was required to pay a "fine" in order to have their system unlocked. Eventually ransomware evolved into a variant which would quickly encrypt as much of a system's files as possible with little regard for evasive action. Additionally, this new ransomware would attempt to spread out to as many systems as possible across the networks connected to the system under attack. This was the point at which hundreds of thousands of systems were being compromised in a matter of only days. Eventually attackers found the most profitable way to use ransomware was against businesses and critical infrastructure. In the case of businesses the lost revenue from down time waiting for their systems to be restored was far greater than the ransom which was requested by the attackers. Critical infrastructure, especially hospitals, could not be unable to perform their functions for the time it would take for their systems to be restored and were often forced to pay the attackers ransom.

The many variations of ransomware are very different in their method of infiltration and replication. However, once ransomware infiltrates a system, regardless of the method used, files will be targeted, accessed, and encrypted in a relatively similar manner. Due to the core behavior of ransomware We use the physical state of a system, as it is measured

through existing physical sensors, to generate prediction models that quickly flag physical system states consistent with a ransomware attack. Ideally this method would be effective alongside conventional methods of ransomware detection which are highly accurate but slower to complete analysis.

Most modern computer systems are comprised of sensors and associated processes that monitor the state of internal hardware components. These sensors continuously supply information that is communicated with other devices and subsystems for the intended purpose of ensuring that the system stays within specific operating specifications. If sensor data reveals that a system component is approaching a boundary of an operational specification, safety mechanisms are typically engaged to correct the internal environment so that system malfunctions can be prevented. For example, when the data from a temperature sensor that is monitoring a computer's central processing unit begins to increase in value, a signal is sent to the CPU cooling fan. This signal causes the fan to either become active or to increase the fan speed to cool the CPU. Additionally, there are sensors that provide input to other subsystems such as internal power management units (PMU) to conserve power usage. Typically, computer system components are designed to be compact in size through the use of transistors with feature sizing in the nanometer scale. As a direct result, whenever computations become more complex, more stress is placed on a computer's hardware components. This increased stress occurs because a larger number of transistors are simultaneously switching in a circuit that correspondingly causes an increase in dynamic power consumption and results in more heat dissipation during heavy computational activity. Thus, monitoring the side channels of a system with embedded sensors that measure parameters such as temperature, power consumption, and battery voltage levels can give insight into the type of processing that is underway on a computer at a given time. Therefore, sensor data streams serve as side channels through which periodic observations can indicate when resource-intensive tasks, such as extensive file system I/O and encryption, are occurring. Because the silent phase of ransomware utilizes significant amounts of file system activity in combination with

encryption, characteristic patterns present within a computer's sensor data may result in trends that are indicative of a ransomware attack.

A significant advantage of this approach, as compared to other side channel methods, is that the sensors and a means for querying them are natively provided. Thus there are fewer concerns in deploying and accessing sensors for the purpose of side channel exploitation. Furthermore, the trend has been that an increasingly diverse number of sensors are provided as integral components in modern computing devices. For example, a typical smart phone has many embedded sensors that could be used to support security applications including power monitors, accelerometers, ambient light sensors, antennas (including GPS receivers), fingerprint scanners, barometers, cameras, touchpad pressure sensors, and others. Even rack-mounted industrial servers contain a significant number of sensors that measure subsystem power consumption, temperature, and other environmental factors. All of these deployed sensors in modern computing devices provide a rich set of data sources that may be used to provide internal side-channel information for the environment in which a computing device is operating. Sensors have been used in other security-related applications in the past. As an example, in (Alharbi A, Thornton M, Dec 2015), sensors present in mobile computing devices have been used to provide a user demographic classification capability for mobile devices with embedded touchscreens (Taylor et al., 2017).

Instead of monitoring file system attributes, the victim host system behavior is monitored by taking advantage of the increasingly large number of onboard sensors. In this sense, this new method uses a physical side channel approach where the victim's files are not directly monitored, rather the behavior of the victim machine is monitored and onboard sensor provided data is used as side channel information that can indicate when an encryption operation is occurring. This monitoring can be accomplished through a background process that is loaded at boot time and thus continuously monitors the system for suspicious behavior. Once this suspicious behavior is detected, the user can be alerted and the suspicious processes can be suspended. The central difference between this approach and other

previous approaches is that this approach uses secondary effects to detect the presence of malware rather than a direct effect, such as measuring increases in file entropy (Scaife N, Carter H, Traynor P, Butler K, June 2016).

Another recent approach for malware detection involves using embedded hardware performance counters that are present in most modern CPU architectures (Demme J, Maycock M, Schmitz J, Tang A, Waksman A, Sethumadhavan S, Stolfo S, June 2013) (Tang A, Sethumadhavan S, Stolfo S, Sept 2014). This approach uses machine learning to create detection models that monitor minor variations in malware execution characteristics. This new approach differs from the use of hardware performance counters in that it uses data being supplied from the suite of embedded sensors that are also present in modern computing platforms rather than performance counter data. Furthermore, this approach is designed to specifically detect ransomware since ransomware uses encryption to enable the victim's data files to be held hostage, and hence, allows them to be recoverable when a ransom is supplied in exchange for the decryption key. This approach uses data sources that are secondary to malware execution patterns and it does not rely upon the presence of performance counters. By targeting a specific class of malware, namely ransomware using encryption in the payload, it is possible to achieve high detection accuracy rates—and more importantly, the ransomware can be detected quickly to mitigate damage to data.

It is proposed that this new sensor-based detection methodology be used to complement more traditional signature-based approaches that are intended to prevent attack vector penetration. In contrast to prevention of attack vector penetration, the technique described here is designed to detect the presence of ransomware when penetration has been achieved. The side channel-based or sensor-based approach has an advantage in comparison to antivirus or IDS systems in that zero-day versions of ransomware can be detected since previously captured malware signatures are not required. Furthermore, it is not necessary to monitor individual files and calculate entropy or other metrics that must be continually re-computed and compared with one another as is the case in the solution provided in (Scaife N, Carter

H, Traynor P, Butler K, June 2016).

4.1. Training Prediction Models

4.1.1. Test Systems

To train and evaluate the methodology, two computer systems were chosen: (1) an older laptop (Hewlett Packard ENVY m4-1015dx) with fewer onboard sensors as compared to the size of a sensor suite found in more modern systems, and (2) a more modern system (MacBook Air 13-Inch Mid 2013) that had nearly three times the amount of sensors as compared to the older system. Access to the sensor output data was achieved through queries via the native operating systems and did not require the development of lower-level software. Because many third-party applications are developed that depend upon access to onboard sensor data, the means to access the sensors are generally available in most operating systems.

The Hewlett Packard m4-1015dx is the first system used in this experiment. When using Open Hardware Monitor the HP ENVY laptop returns 22 sensor values. The types of sensor values which are returned include clock, data, load, power, and temperature. While clock, data, and load are reported with the physical sensors they are considered "probes" which are relaying performance metrics provided by the CPU (Bresink M, 2017). This type of data was still used in the implementation of the new detection method but only as an aid in determining how best to apply the prediction models generated from the data obtained through physical measurements of the system. Thus the HP ENVY laptop only contains nine sensor values which are of direct use in creating the prediction models in this experiment.

It is important to note that of the nine sensors that are used in the evaluation of this system, eight of them directly measure aspects of the CPU thereby causing the predictive model to nearly entirely depend on CPU behavior. Therefore, the HP ENVY laptop can be seen as having a relatively "sensor-poor" environment for the evaluation of the sensor-based

ransomware detection method.

Table 4.1. HP ENVY m4-1015dx Reported Sensors

Type	**Count**
Clock	4
Data	3
Load	6
Power	3
Temp	6
	22

The MacBook Air 13-inch mid-2013 model is the second system used in this experiment. When using Hardware Monitor for Mac the MacBook Air laptop returns 70 sensor values. Two of the sensor values returned by the program measure battery capacity, but the values stop being reported when the system is connected to power and the battery is fully charged. Another sensor measures ambient light levels for adjusting the brightness of the screen. When the screen is turned off from a lack of user interaction this sensor also stops being reported. Although the sensors themselves are viable for use in the new prediction model they are unable to be utilized as the models require input vectors of a consistent size. Thus, the MacBook Air contains 67 sensor values which are of use in this experiment. Unlike the Hewlett Packard ENVY m4-1015dx, only 17 of the available 67 MacBook sensors directly measure CPU activity. Therefore, the availability of sensors that measure other system components and parameters allow for the development of predictive models that are more holistic to the system. The MacBook Air laptop thus provides a more inclusive sensor-suite with regard to monitoring the entire system and thus enables our methodology to have access to a richer set of side channel data.

Table 4.2. MacBook Air 13-Inch Mid 2013 Reported Sensors

Type	**Count**
Capacity	2
Current	16
Light	1
Power	18
RPMs	1
Temp	23
Voltage	9
	70

4.1.2. Simulating Ransomware Attacks

Physical sensor-based attack detection attempts to find a pattern in the physical state of a system that can identify the presence of an attack (many times referred to as a fingerprint). There are numerous variations of ransomware. Thus, training on the unique operation of a single variant would likely not be effective for detecting this general class of malware attacks. Ideally, the training data should be collected using a process that implements the most common and basic elements that are present in a variety of different ransomware attacks. The variation in ransomware attacks are usually due to the methods of infiltration, encryption, file system searching, file targeting, and the infiltration of additional attached systems.

It is assumed that any variation in the infiltration method does not affect the detection process. Thus, the proposed method is designed to detect ransomware that has recently infiltrated a system.

Additionally, the method an attacker uses for further propagating their attack to additional systems attached to the host is not a part of the attack that is considered in the

detection method. For these reasons, the focus of the proposed method is on the use of sensor data that shows characteristic patterns with respect to the type of encryption used, the process of iterating through the directories in a host's file system, and the targeting of files for encryption.

In this experiment a script was written that simulates the active encryption portion of a ransomware attack. The attack can be performed using many different choices of parameters based on criteria from all three of the previously noted areas of potential attack variation. The type of encryption is chosen from four different variations of AES encryption or a simple XOR encryption. The XOR encryption was included to simulate the behavior of more lightweight methods of encryption. The script accesses a host's file system based on the particular directories that it finds and searches through. To help avoid detection, the script uses intentional, random delays while file searching is occurring. The simulation script selects one or more starting points in the file system that are most likely to contain a host's personal and sensitive data. The script recursively traverses the directory and sub-directories of a starting point checking for files with the targeted file extensions. Targeted victim files are identified via the use of a list of file extensions that were historically targeted by multiple high-profile ransomware attacks. The script creates an encrypted version of the victim file, deletes the unencrypted version, and renames the encrypted file with the original unencrypted target file's name. Prior to operation the script is either set to run continuously or to wait a different random amount of time between 1 and 60 seconds each time after encrypting a target file

Randomly selecting parameters in the script creates a simulation of the active encryption phase of a ransomware attack variant. Although the methods used to infiltrate and propagate a ransomware attack vary widely, the methods of actually finding and encrypting as many of the host's personal files as possible is more constricted. Repeated testing with the script was accomplished and intended to offer attack simulations that differ enough in their approach such that the collective training data is a generalization of the active encryption phase of a

variety of different ransomware attacks.

4.1.3. Collecting Sensor Data

Physical sensors are present in systems in order to monitor and ensure operations within the safety specifications. For example, sensors measure the temperature of a CPU during operation in order to ensure it does not overheat and cause damage. The sensors are present for a very specific reason and generally offer little insight into the system beyond their intended use. The sensor readings in a system are usually readily available as they are implemented for safety. In implementation of our prototype detection system, sensor data is the input required to make predictions about the binary state of a system using machine learning algorithms. It is important that the methodology utilized in procuring sensor data is both quick and reliable. Many tools exist which allow users to simply monitor sensor data in a graphical user interface; however our prototype required the automated retrieval and parsing of sensor data at specific intervals in time. Both tools utilized in this experiment can be used to either write data to a file for future analysis or provide a feature vector of real-time sensor data to a prediction model. Writing data to a file allows for the direct comparison of machine learning algorithms in as far as how they would have predicted the state of the system given the same input data. Sensor data is collected for the test machine with a Microsoft Windows operating system using an open source program called Open Hardware Monitor (OHM). OHM is an especially powerful tool in the context of this experiment since it publishes all sensor data to Windows Management Instrumentation (WMI) for accessibility from the command line. Initially, OHM is capable of providing a comma separated list of the nomenclature for each sensor and the corresponding sensor category. OHM is continuously given a second command after a set time interval which returns a comma separated list of only sensor values that are in the same order as the initial list of sensor nomenclatures. Sensor data is collected for the test machine that has an Apple OSX operating system using a program called Hardware Monitor. Hardware Monitor allows sensor data to be accessed

from the command line that in turn allows automated scripts to work with real time sensor data (Bresink M, 2017). Prior to reading any sensor values, a command can be issued that returns a comma separated list of sensor names and categories that can then act as a header for future sensor data. In order to access the sensor data a command is issued to the command line that returns a string of comma separated sensor values in the same order as the header string.

4.1.4. Building Training Data Sets

The training data needs to represent periods of operation in a system both while under attack from ransomware and while not under attack. The ransomware was implemented as a daemon that ran in two hour blocks with each block having a predefined encryption method for all attacks. During the two-hour collection period, the system is either in a state of "normal operation" or "under attack". Normal operation is any time the system is not being attacked with the simulated ransomware script while "under attack" is any time that the simulated ransomware script is active. For each simulated ransomware attack, the method of attack is different by randomly selecting one or more starting positions in the file system for recursive searches. Additionally, the script randomly determines if the attack will use detection avoidance through randomizing the timing of directory access and file encryption. Before the two hour training block initiates, the time required to encrypt all target directories with the selected encryption method is measured. The script then begins in a state of "normal operation" for the amount of time that was previously measured before launching the encryption attack. During the time that the script is logging the sensor data, it is also adding system state labels so that a supervised machine learning model can be utilized. After the simulated attack is completed the logging stops and system is decrypted. There is a waiting period of two minutes for the system to return to a state of normal operation before logging continues and a new cycle is begun. Each of the five encryption methods is run in 12 different two-hour test blocks. Each encryption method is then repeated after additional

CPU load activity is initiated so that the training data can contain examples of sensor data with various amounts of background activity present in the runtime profiles. Test loads of 0%, 25%, 50%, 75%, and close to 100% system activity are each applied to the system in three separate two hour testing blocks for each encryption method. The complete training data set consists of 24 hours of regular training and 30 hours of simulated load training for each of the five encryption methods. The sensor readings and the CPU load of the system are polled every second during the training periods and labeled with a timestamp and either "normal operation" or "under attack." Data from each two-hour training block is collected in separate CSV files.

4.1.5. Training Prediction Models

The collected training data was used to to create several different prediction models available in the Python Scikit-learn library (Pedregosa F, Varoquaux G, Gramfort A, Michel V, 2010). In total, models were generated using 12 different machine learning algorithms that comprised 280 different combinations of parameter settings. Numerous combinations of methods were investigated including: data scaling method, feature selection method, prediction method, and moving average method (i.e., smoothing of the output predictions). The process of chaining methods together is often referred to as classification "pipelining" (Chang et al., 2007; Finkel et al., 2006). For test deployment, each model was stored in its own Python pickle file for use in various online tests.

The prediction method includes two options, binary classification and ordinal regression. During training each time interval is labeled with a binary state of "under attack" or "normal operation". Additionally, each time interval is labeled with a value between zero and seven which is determined based on both the presence and progression of attacks. When the system is in a binary state of "normal operation" and has been for at least 40 seconds the time interval is labeled with a zero. Once an attack does occur the binary label becomes "under attack" and the ordinal label becomes four which is near to the middle of the scale

but slightly favoring an attack state. Every ten seconds the system remains in a binary state of "under attack" the ordinal label is incremented until it is seven. The time interval label of seven indicates a system state in which an attack has been underway for a significant amount of time. Once the attack finishes and the binary system state become "normal operation" the ordinal label is changed to three which is near to the middle of the scale but slightly favoring a state of "normal operation". Every ten seconds the system remains in a binary state of "normal operation" the ordinal label is decremented until it is zero indicating the system state has had a significant amount of time to return to normal operation. The ordinal regression labels were used to train learning models which provided predictions about the attack state of the system and potential insight into the damage caused based on the time it has been active.

Binary classification models are trained with a dependent vector of binary values indicating true for "under attack" and false for "normal operation." Ordinal regression models are trained with a dependent vector consisting of values with seven being the highest and representing the highest likelihood of being "under attack" and zero being the lowest and representing the highest likelihood of being in "normal operation." Whenever the predicted value is greater than three, a prediction of "under attack" is made. Conversely, whenever the predicted value is less than or equal to three a prediction of "normal operation" is declared.

The data scaling methods investigated include four options: feature standardization, data normalization, feature min-max scaling, and no scaling. Feature standardization is the process of setting each feature of the data to have zero-mean and unit-variance.

$$X' = \frac{X - \frac{\sum X}{N}}{\sqrt{\frac{\sum (X - \bar{X})^2}{N}}} = \frac{X - Mean(X)}{Standard\ Deviation(X)}$$

Data normalization is accomplished such that every sample of a feature has the mean value of the feature subtracted from it after which it is divided by the standard deviation of the feature. This causes the mean of each feature to be zero with a standard deviation

of one. It is important to distinguish that feature standardization operates on individual feature columns of a data set (Pedregosa F, Varoquaux G, Gramfort A, Michel V, 2010).

Data normalization is the process of rescaling each data instance independently such that the L1 or L2 norm is equal to one.

$$L1 = \sum_{i=1}^{n} |y_i - f(x_i)| \qquad L2 = \sum_{i=1}^{n} (y_i - f(x_i))^2$$

Scaling the inputs to one, or unit norms, is a common operation when using classification and clustering machine learning algorithms.

Feature min-max scaling is a method used to standardize the range of the independent variables or features.

$$X' = \frac{X - Min(X)}{Max(X) - Min(X)}$$

Min-max scaling places all data on the same scale, usually zero to one, which in turn allows machine learning algorithms to weigh each feature equally. The standard deviations of the features tend to be smaller with min-max scaling which can suppress the effect of outliers. Scikit-learn includes data structures which fit and transform the training data and are also capable of transforming future test data as needed.

The dimensionality reduction method includes seven variations including the use of PCA, feature selection, and no dimensionality reduction. Several principal component analysis data reductions are performed, using the cumulative explained variance to guide the number of components chosen. Three variations of PCA dimensionality reduction are investigated based upon if the components maintain at least 70%, 80%, or 90% of the total variance. Feature selection is performed such that only the top 50%, 70%, or 90% of features are selected using F-tests from the Scikit-learn library to analyze variance in the training data (Pedregosa F, Varoquaux G, Gramfort A, Michel V, 2010).

Once the features are scaled and selected, they are used to train one of twelve different classification or clustering techniques. For clustering techniques, the methods are used as unsupervised classification methods where each cluster is assigned as a particular class and the total number of clusters equals the total number of classes. Parametric and non-parametric classification methods are employed, as well as supervised and unsupervised methods. The following algorithms are investigated:

- **Parametric Trees Methods**: Decision Tree (Breiman, 2017), Random Forest (Breiman, 2001), Extremely Randomized Trees (Geurts et al., 2006), One-versus-one Tree Ensemble, One-versus-rest Tree Ensemble
- **Parametric Methods**: Linear Regression, Logistic Regression, Support Vector Machine (Drucker et al., 1997), Naive Bayes, Two Layer Neural Network
- **Non-parametric Method**: K-Nearest Neighbors
- **Unsupervised Method**: K-Means

Each classifier is trained to classify every second of data from the sensors streams, resulting in a binary stream of predictions. This output stream is smoothed using various moving average methods. The moving average method consists of five options including simple moving average with window sizes of two and four, weighted moving average with window sizes of two and four, and no moving average. Moving average is a calculation to analyze data points by creating a series of averages of different subsets of the full data. One of the most common uses of moving averages is to smooth out short-term changes and emphasize long-term trends in time series data. The Simple Moving Average (SMA) is the simplest implementation of moving average that utilizes the unweighted mean of the previous n data points.

$$SMA_M = \frac{p_M + p_{M-1} + ... + p_{M-(n-1)}}{n}$$

Weighted moving average (WMA) gives different weights to data at different positions in the sample window. WMA allows more recent data to have more impact than previously seen data (Devcic J).

$$WMA_M = \frac{np_M + (n-1)p_{(M-1)} + ... + p_{(M-n+1)}}{n + (n-1) + ... + 2 + 1}$$

Moving averages are useful when dealing with real-time prediction models such as those used here that are based upon sensor data. If a prediction model has been making false predictions for an extended period of time it will require multiple true predictions to occur before the moving average becomes true. Utilizing a moving average may slow the response in reporting real attacks, or true positive predictions. However, the reduction in incorrect attack reports, or false positive predictions, is likely to have more of a positive impact on the prediction accuracy. Moving averages allow the tradeoff between responsiveness and accuracy to be easily adjusted by a user through increasing or decreasing the window size.

4.2. Testing Prediction Models

4.2.1. Building Test Data Sets

Test data was collected in one-hour test blocks in which a single ransomware attack would occur at a random time. The ransomware attack parameters were selected randomly for each attack. During training, the amount of time the system was recorded to be in the state of "normal operation" was similar to the amount of time the system was recorded in the state of "under attack" in order to create a balanced training set. However, during testing the use of a balanced set is not appropriate because most of the time a system would be in the "normal operation" state. Therefore, the testing was conducted in a manner such that there was a disproportionate amount of time the system was in the "normal operation" state. The purpose of this approach was to determine how many times the attacks would be correctly detected, the frequency of false positive predictions, and most importantly; how fast attacks

were predicted. Each of the five encryption methods underwent 24 of the one-hour test blocks. Afterward, each of the five encryption methods was tested with additional CPU loads of 0%, 25%. and 50%. Each of the encryption methods was tested in six different one-hour time blocks with each of the three additional CPU load levels. During each one-hour test block of the additional CPU load testing, there was a single ransomware attack that occurred at a random time. During the collection of data for testing the method, the total system CPU load is recorded each time the sensor data is polled. This CPU load data is not recorded during training and not used as a feature in the prediction models.

The initial phase of test data collection represents a system which is sitting unused and only running regular background processes. This phase of testing was designed to determine the performance of the proposed method in favorable conditions for detection. Ideally the models should have a low number of false positive attack predictions while being able to quickly determine when the system is under attack. The second phase of test data collection represents a system which has different levels of user activity in addition to the regular background processes from the initial phase of testing. This phase of testing was designed to determine if the prediction models are able to perform in a more realistic scenario in which the physical state of the machine is more dynamic.

4.2.2. Predicting System States

This experiment tests twelve different machine learning algorithms. The test data is collected and stored in CSV files with labels that indicate what the actual state of the system was each time the sensor data was probed. The models for each of the different algorithms are all used to make predictions with the same data set. Any of the models which use sliding windows for their final prediction (such as when a moving average is used) have the initial prediction vector iterated through in order to generate a final prediction vector. The predictions are generated using the Python Scikit-learn library by loading the previously constructed model from it's saved file. Models which use data preprocessing

(such as scaling or feature selection) also have the appropriate data structures loaded. The performance prediction is determined by comparing the actual system state vector from the test data to the final prediction vector generated from the prediction model.

4.2.3. Binary Classification Evaluation

In this experiment machine learning algorithms are each used to make a prediction about the binary ransomware attack status of a system. True positive predictions equate to periods of attack which are correctly identified as being under attack. False negative predictions equate to periods of attack which are incorrectly identified as not being under attack. True negative predictions equate to periods of no attack which are correctly identified as not being under attack. False positive predictions equate to periods of no attack which are incorrectly identified as being under attack. The final prediction vector of a model and the actual system state vector are compared to obtain the distribution of the four types of binary classification. The distribution of the four classifications was used to compute six metrics which offer more insight into prediction performance: sensitivity, precision, specificity, fallout, and accuracy. While each criterion is useful when viewed and evaluated together, none of them is a sufficient metric for performance when taken alone. In this experiment the test data contains a very disproportionate amount of one state to the other. As such, training a model which always predicted the high frequency state would result in a high accuracy score while actually being a poor predictor.

4.2.4. Matthews Correlation Coefficient (MCC)

The Matthews correlation coefficient (MCC) takes into account true and false positives and negatives and is generally regarded as a balanced measure which can be used even if the classes are of considerably different sizes such as the data in this experiment.

$$\frac{(TP * TN) - (FP * FN)}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

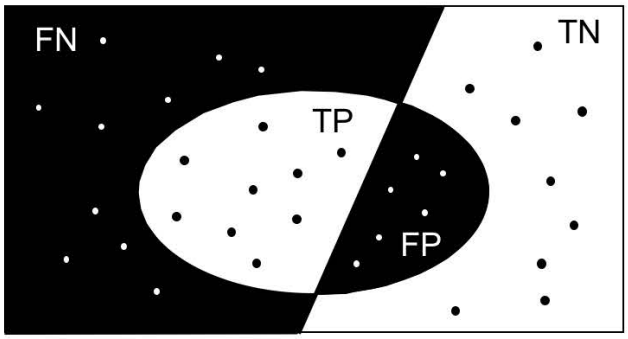


Figure 4.1. Binary Classification Evaluation

The MCC is a correlation coefficient between the observed and predicted binary classifications. Values range between -1 and +1. A coefficient of +1 represents a perfect predictor, 0 represents the same as random prediction, and -1 indicates total disagreement. The MCC is regarded as being one of the best measures for describing the relationship between the four possible binary classification outcomes in a single value. Thus, the MCC will be the most heavily weighed metric in determining the optimal machine learning algorithm for the problem presented in this experiment.

4.2.5. Rate of Attack Recognition (RAR)

When testing, an actual attack time series exists which defines the time periods during which there is an attack. Figure 4.2 displays an example time series plot with the actual attack time series located on the far left. For each time interval occurring during an attack in the actual attack time series, or every time interval after the rising edge and before the falling edge, is checked in the corresponding prediction time series for positive predictions. If a positive prediction exists then the attack instance is considered detected. Ideally there should exist at least one positive prediction during each attack instance which would result in a 1.0 or perfect rate of attack recognition. In the worst case scenario there would exist no positive predictions during any attack instance which would result in a zero. Unlike

traditional binary classification metrics, such as sensitivity, rate of detection is not concerned with the volume of correct positive predictions during an attack instance. Instead it considers the application of the binary classifier in which one positive prediction would perform the same as a high volume of positive predictions as the attack only has to be flagged once. It should be noted that the rate of attack recognition will always be one if all predictions are positive. Therefore, rate of attack recognition must be weighed in conjunction with a binary classifiers ability to make correct positive predictions such as precision.

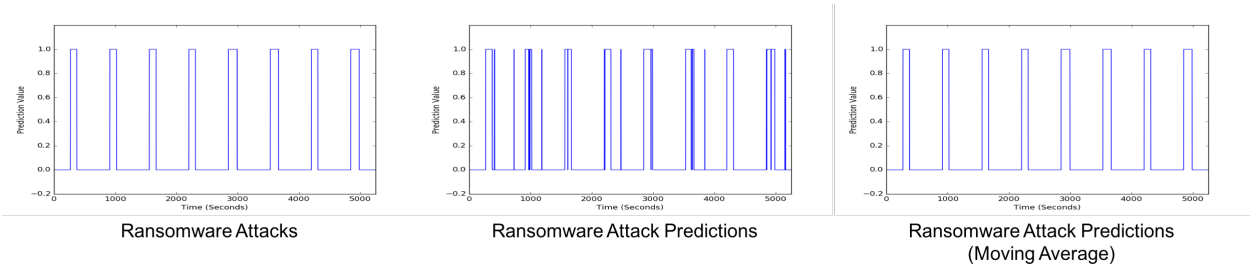


Figure 4.2. Example Time Series Plots

4.2.6. Mean Time to Attack Recognition (MTAR)

The rising edge of each attack instance in the actual attack time series represents the time interval at which the attack began. The first instance of a positive prediction at or after the rising edge and before the falling edge in the corresponding prediction time series represents the initial attack recognition. Ideally the rising edge itself would be a positive prediction, but in practice it is more likely that the sensors would need a small amount of time to reach the values at which positive prediction occurs. The number of time intervals until the first positive prediction is recorded for every attack instance which was successfully recognized. Afterwards all values are averaged to determine the mean time to attack recognition. It should be noted that attack instances that were not successfully recognized do not weigh negatively in this metric. Therefore, mean time to attack recognition

should be considered in conjunction with the rate of attack recognition to describe the effectiveness of detecting attack instances both quickly and consistently. Furthermore, the mean time to attack recognition will always be zero if all predictions are positive. Thus, mean time to attack recognition must also be weighed in conjunction with a binary classifiers ability to make correct positive predictions such as precision.

4.3. Experimental Results

4.3.1. Optimal Parameters for Algorithms

Stratified ten-fold cross validation is performed for each algorithm using the balanced training dataset in order to determine how well various test configurations are likely to perform. There exist a total of 280 different combinations of the test parameters that are all tested individually for each algorithm. The test parameters include the prediction method, data scaling method, dimensionality reduction method, and moving average method. For each combination the training data is separated into ten equal-sized subsets. The MCC is used rather than simple accuracy as it provides a more descriptive measure of the overall model performance where there is a class imbalance. Ten different MCC values are found by ten different tests in which each subset acts as the sole training data once and is part of the test data nine times. The ten MCC values are averaged to obtain the average MCC of the algorithm for a specific test parameter combination. The 280 average MCC values are ordered from greatest to least with the highest value belonging to the test parameter combination which is most likely to result in the highest performance during testing. The highest performing test parameter combination is used for the appropriate algorithm for the duration of testing instance.

The results shown in Table 4.3 rank the highest configuration MCC scores for each algorithm from top to bottom. The first test system, the HP ENVY m4-1015 dx, has its ranking displayed on the left and the second test system, the MacBook Air Mid 2013, has its

Table 4.3. Cross Validation Algorithm Rankings

HP ENVY m4-1015dx		**MacBook Air Mid 2013**	
Algorithm	**Max MCC**	**Algorithm**	**Max MCC**
Log Reg	0.9610	Rand Forest	0.9987
MLP	0.9600	Extra Tree	0.9985
One-V-One	0.9591	MLP	0.9984
One-V-Rest	0.9591	One-V-One	0.9983
SVC	0.9591	Log Reg	0.9979
Rand Forest	0.9570	One-V-Rest	0.9979
Extra Tree	0.9549	SVC	0.9979
Decision Tree	0.9528	KNN	0.9975
KNN	0.9311	Decision Tree	0.9970
Lin Reg	0.9285	Lin Reg	0.9965
N Bayes	0.9069	N Bayes	0.9917
K-Means	0.7074	K-Means	0.8327

ranking displayed on the right. In general, the first test system's optimal algorithm model configurations implemented data scaling with most using min-max, no feature selection, and a weighted moving average with a window size of four. All of the configurations implemented a moving average with 9 out of 12 being the largest window size tested. This is likely due to the relatively small number of sensors that are available for polling on the first test system. There are only 15 usable sensor readings for the first test system that most likely result in momentarily incorrect predictions with significant changes in the physical state of the system. However, once the system's physical state begins to level off the predictions become more accurate. The best performing models for the first system tended to be simpler or linear

models like logistic regression. These simple models tend to have less ability to formulate more complex patterns. Thus, this may indicate that it is advantageous to use simple relationships between the sensor streams for attack detection when the number of sensor streams is relatively few. The second test system, with 67 usable sensors, only has three algorithms in which the optimal configuration uses a moving average. The three algorithms that do use moving average use the smallest window size tested. With the much larger group of sensors to poll from the prediction models are likely relying on complex patterns rather than basic changes in the physical system state. These complex patterns are believed to represent the unique physical impact to the system that is a result of the combined actions of the ransomware process. This is also supported by the selection of more complex ensemble classifiers as the best performing models, such as Random Forests and Extremely Random Trees. These ensembles have the ability to detect more complex interactions among the input feature streams.

Comparing the scores seen in Figure 4.3 it can easily be determined that the second test system with the larger number of sensors scored higher for every algorithm. The one algorithm that scores significantly lower for both test systems is K-Means clustering which is the only unsupervised algorithm implemented in this experiment.

4.3.2. Prediction Evaluation on Test Data

The test data analysis is performed for each of the twelve machine learning algorithms. There exists five different encryption modes with each having separate training and testing data. Every combination of encryption modes of every size from one to five has a prediction model trained, and the same combination of testing data is used to assess how well the model performs when making predictions for data it has been trained with.

The analysis is split into five parts with different combination selection sizes for the models. The test procedure can be outlined as follows:

1. Select combinations of one encryption mode from five total encryption modes, $\binom{5}{1}$,

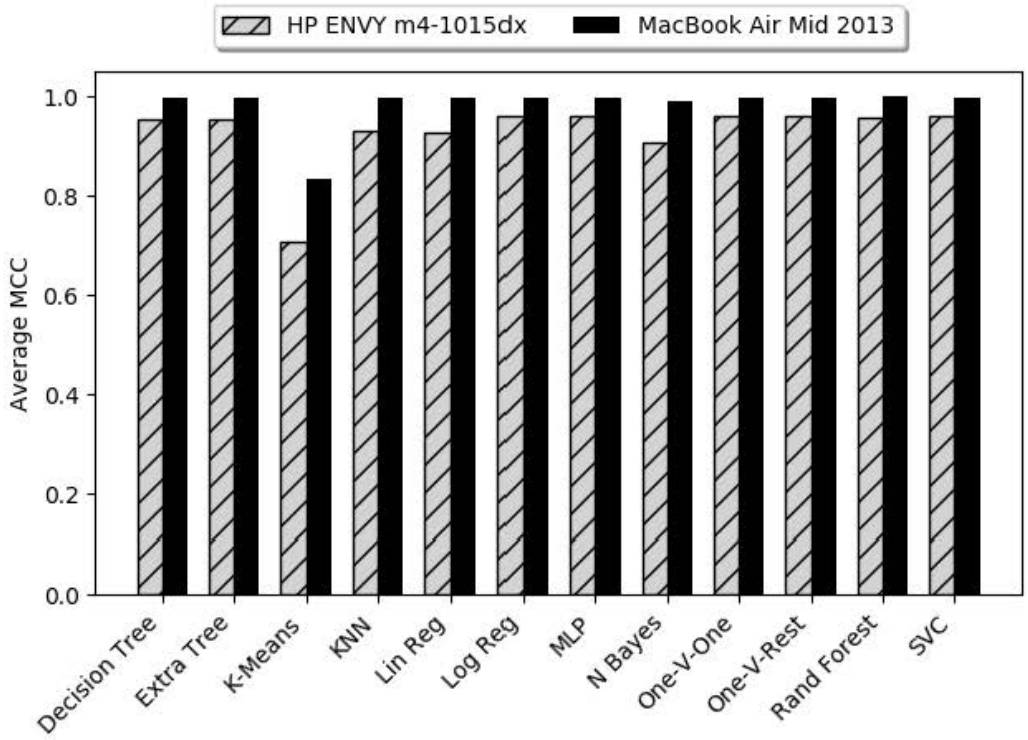


Figure 4.3. Stratified 10-Fold Cross Validation Results

resulting in five models. Test each model with the corresponding encryption mode test data.

2. Select combinations of two encryption modes from five total encryption modes, $\binom{5}{2}$, resulting in ten models. Test each model with the corresponding encryption mode test data.

3. Select combinations of three encryption modes from five total encryption modes, $\binom{5}{3}$, resulting in ten models. Test each model with corresponding encryption mode test data.

4. Select combinations of four encryption modes from five total encryption modes, $\binom{5}{4}$, resulting in five models. Test each model with corresponding encryption mode test

Table 4.4. Accuracy Analysis MCC Results

HP ENVY m4-1015dx		**MacBook Air Mid 2013**	
Algorithm	**Avg MCC**	**Algorithm**	**Avg MCC**
Log Reg	0.4928	Extra Tree	0.9981
Extra Tree	0.4882	KNN	0.9980
One-V-One	0.4882	Log Reg	0.9980
One-V-Rest	0.4882	Rand Forest	0.9980
SVC	0.4882	MLP	0.9970
MLP	0.4717	Lin Reg	0.9964
Rand Forest	0.4707	One-V-Rest	0.9916
Lin Reg	0.4414	SVC	0.9914
KNN	0.4267	N Bayes	0.9790
N Bayes	0.4137	Decision Tree	0.9687
Decision Tree	0.3922	One-V-One	0.9528
K-Means	0.1355	K-Means	0.7743

data.

5. Select combinations of five encryption modes from five total encryption modes, $\binom{5}{5}$, resulting in one model. Test the model with all encryption mode test data.

There are 31 total models and tests for each algorithm during the evaluation analysis. For each algorithm, the scores are computed from the averages of the 31 different tests carried out on the 31 different models.

Table 4.4 shows the prediction models from the first test system all scoring below 0.5 while all of the prediction models from the second test system, except K-means clustering, score above 0.95. The prediction models used on the first test system, with the exception

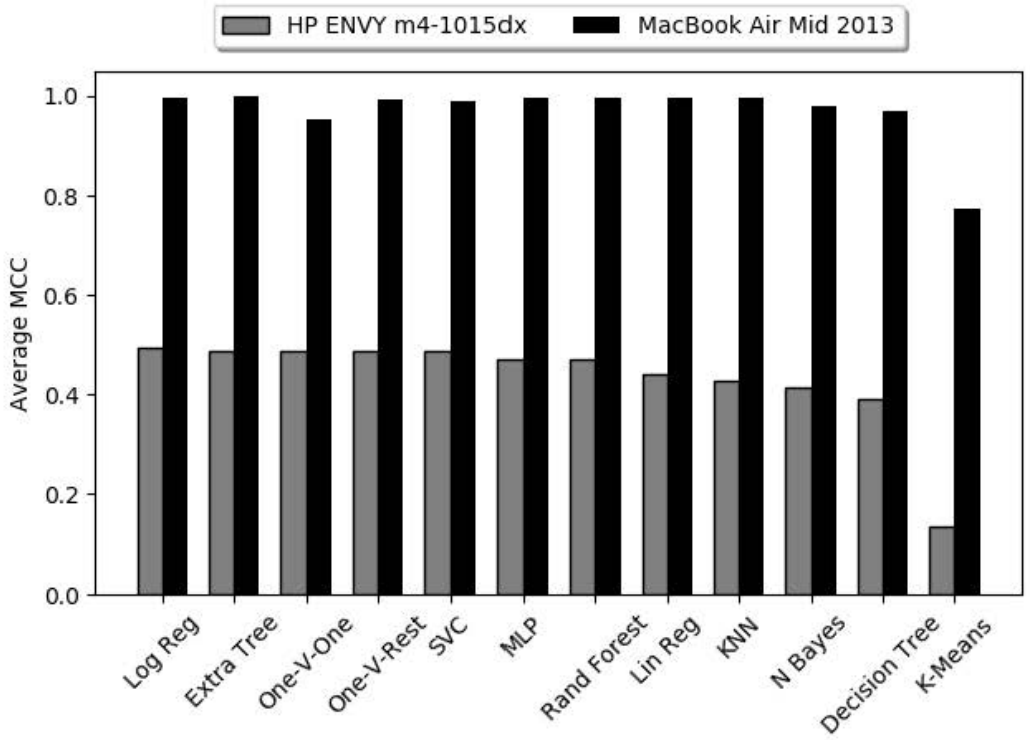


Figure 4.4. Accuracy Analysis MCC Results

of K-means clustering, were all able to detect every ransomware attack while maintaining low false positive rates. The accuracy scores for these prediction models, which measures the rate of correct predictions, were all above 0.9. The sensitivity scores, which measure the rate of correctly predicted polling times which occurred during attacks, was never above 0.3 for any of the prediction models.

Table 4.5 is a ranking of the mean time for attack recognition, or MTAR, for the prediction models. The majority of the prediction models for the first test system took over one minute to make a prediction of "under attack." The second test system, which has a large collection of sensors, is able use its prediction models to make the first "under attack" prediction in less than one second for all algorithms except two. The ten prediction models that had a mean time for attack recognition of less than one second were able to detect all of the ransomware

Table 4.5. Accuracy Analysis MTAR Results

HP ENVY m4-1015dx		MacBook Air Mid 2013	
Algorithm	**Avg MTAR**	**Algorithm**	**Avg MTAR**
Decision Tree	48.10	MLP	0.3891
K-Means	52.01	One-V-Rest	0.4044
Extra Tree	53.03	SVC	0.4050
Rand Forest	59.60	Decision Tree	0.4113
Log Reg	61.99	Rand Forest	0.4539
N Bayes	65.08	Extra Tree	0.4794
KNN	66.41	Log Reg	0.4924
MLP	69.47	Lin Reg	0.5600
One-V-One	69.81	One-V-One	0.5700
One-V-Rest	69.81	KNN	0.5812
SVC	69.81	N Bayes	1.511
Lin Reg	70.97	K-Means	54.31

attacks during the accuracy testing with an average MCC score of 0.989.

The results of this portion of the testing are not necessarily meant to reflect the actual performance this method would achieve when deployed on a live system. The test systems are running regular background processes and the ransomware attacks are performed using the encryption modes that the current prediction model was directly trained to detect. The extremely high performance of the second test system's prediction models along with the inconsistent performance of the first test systems prediction models suggests the prediction models using many sensor values are exploiting physical interactions in known attacks which are complex and unique enough that they are almost instantly recognized and nearly always

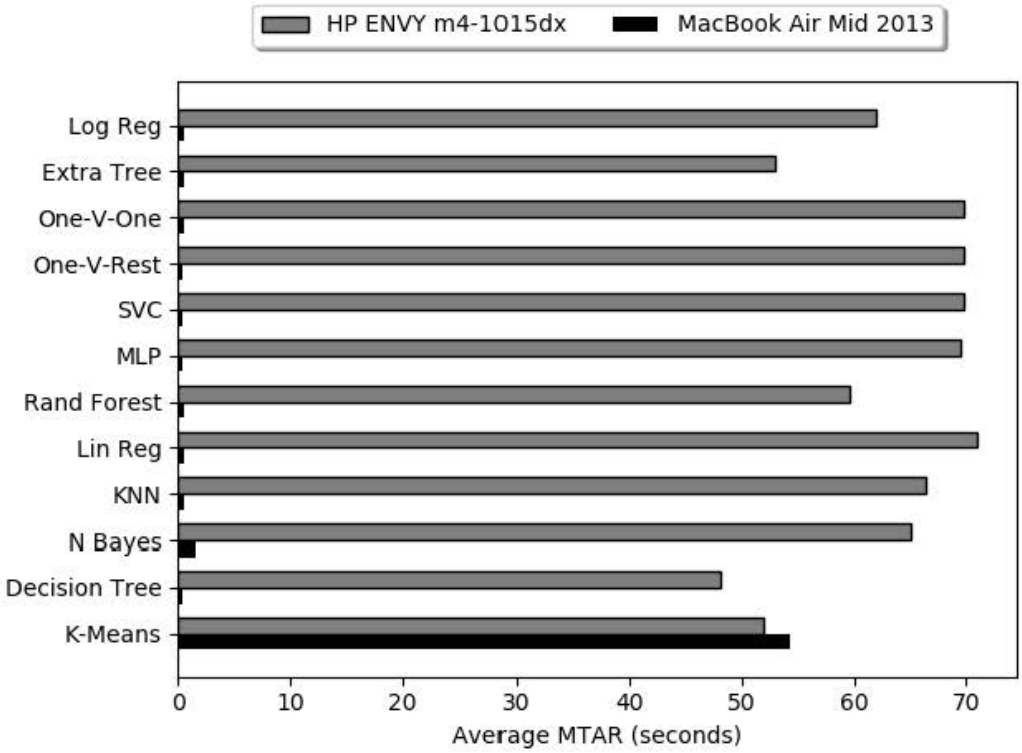


Figure 4.5. Accuracy Analysis MTAR Results

classified correctly on an idle system.

4.3.3. Robustness of Prediction Ability

The robustness analysis is performed for each of the twelve machine learning algorithms. The same 31 models generated during the previous analysis are all tested with attacks using each of the five encryption modes. Robustness, in the context of this analysis, is used to refer to the ability of an algorithm to perform well given encryption mode it has been trained to predict as well as encryption modes it has not been directly trained to predict. This analysis will convey whether models that have not been explicitly trained to detect certain encryption modes can still detect them with relative success. There are 31 total models with each model requiring five tests for each encryption mode. Therefore, 155 tests are required

Table 4.6. Robustness Analysis MCC Results

HP ENVY m4-1015dx		**MacBook Air Mid 2013**	
Algorithm	**Avg MCC**	**Algorithm**	**Avg MCC**
Log Reg	0.4952	Extra Tree	0.9980
One-V-One	0.4885	Log Reg	0.9980
One-V-Rest	0.4885	Rand Forest	0.9979
SVC	0.4885	KNN	0.9978
Extra Tree	0.4864	Lin Reg	0.9960
MLP	0.4747	One-V-Rest	0.9958
Rand Forest	0.4707	SVC	0.9958
KNN	0.4526	N Bayes	0.9839
Lin Reg	0.4433	Decision Tree	0.9559
N Bayes	0.4125	MLP	0.9378
Decision Tree	0.4013	One-V-One	0.9040
K-Means	0.1363	K-Means	0.7681

for each algorithm during the robustness analysis. The scores for each algorithm are found based on the averages of the 155 tests carried out on the 31 models. The algorithm with the highest MCC average in both the previous analysis and robustness analysis is selected and used in the remainder of the tests.

4.3.4. Effect of Training Time on Performance

The effect of training time on the prediction performance of the model was tested with the highest performing algorithm for each test system. Based on the outcome of the accuracy and robustness testing Logistic Regression was selected for further evaluation with the HP

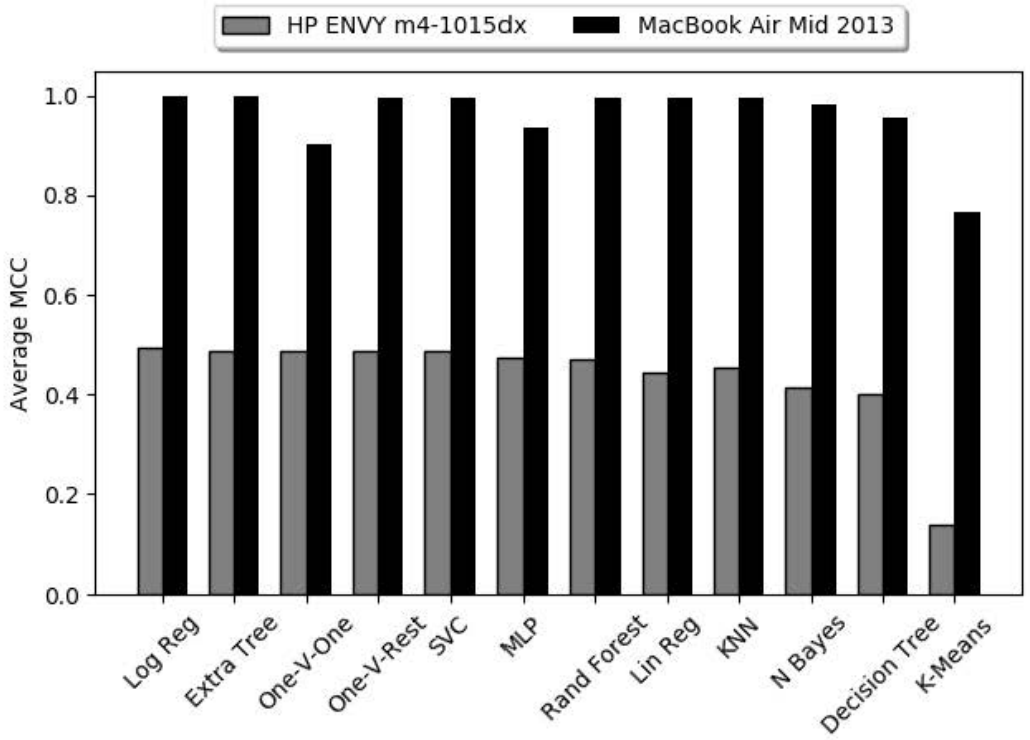


Figure 4.6. Robustness Analysis MCC Results

ENVY m4-1015dx as it had the highest average MCC score during both tests. Likewise, the Extra Tree model was selected for further evaluation with the MacBook Air Mid 2013 as it also had the highest average MCC score during both tests. Twelve prediction models where created for each system from training data collected over 24 hours. The first model was trained with only the first two hours of the data and subsequent models were trained with an additional two hours of data until all 24 hours worth of training data were utilized. It can be seen in table 4.8 that the performance of the HP ENVY m4-1015dx system was highest after 4 hours of training with additional training causing the performance to decrease until leveling off around 18 hours. The MacBook Air Mid 2013 system showed very little change in performance after 4 hours with the MCC score only increasing to a maximum of 99.85% and the mean time to attack recognition only dropping by 0.1166 seconds at its peak performance

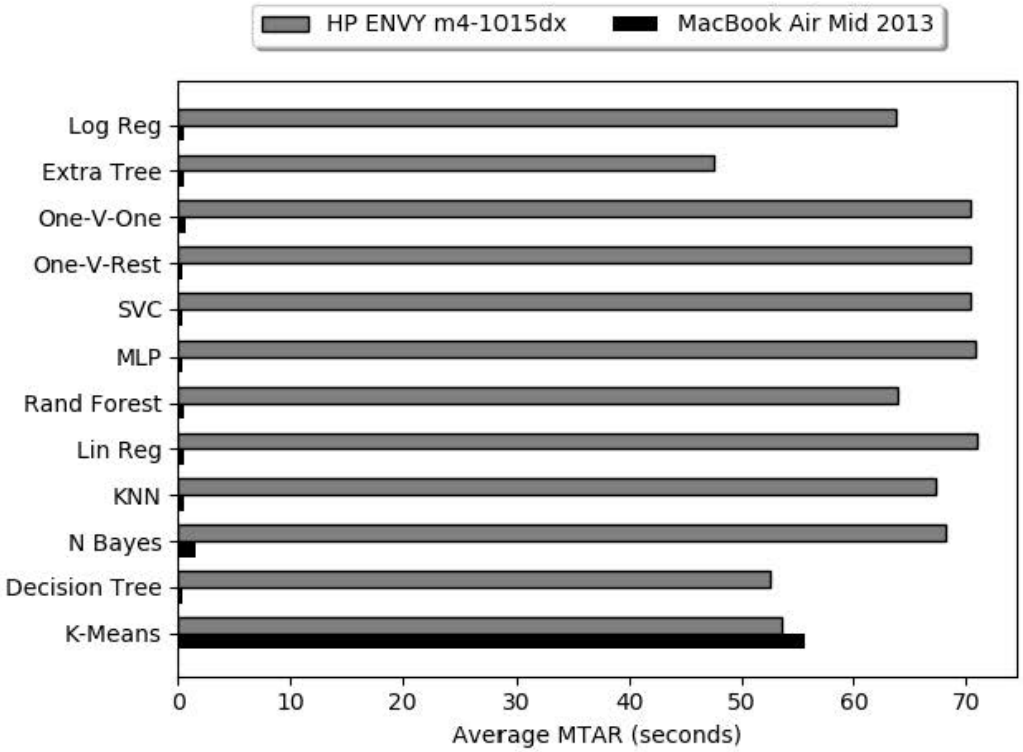


Figure 4.7. Robustness Analysis MTAR Results

Table 4.7. Robustness Analysis MTAR Results

HP ENVY m4-1015dx		**MacBook Air Mid 2013**	
Algorithm	**Avg MTAR**	**Algorithm**	**Avg MTAR**
Extra Tree	47.51	Decision Tree	0.4010
Decision Tree	52.63	SVC	0.4078
K-Means	53.57	One-V-Rest	0.4070
Log Reg	63.76	MLP	0.4207
Rand Forest	63.97	Rand Forest	0.4530
KNN	67.29	Extra Tree	0.4667
N Bayes	68.17	Log Reg	0.4733
One-V-One	70.3618	Lin Reg	0.5613
One-V-Rest	70.3618	KNN	0.5739
SVC	70.3618	One-V-One	0.6223
MLP	70.83	N Bayes	1.520
Lin Reg	70.98	K-Means	55.73

after 20 hours of training. While the results of this experiment do indicate that there is small increases in performance with additional training time the models with the least amount of training time are still able to perform at a comparable level. This conclusion seems especially true in the MacBook Air test system which seems to indicate that systems with a large array of sensors are able to be effectively trained quickly. Considering the dynamic and modular nature of modern computing devices this property of the new prediction technique is very useful as hardware modification would likely decrease the performance of a previously trained model and greatly benefit from a newly trained model.

Table 4.8. Training Time Analysis Results (Hours)

	HP ENVY m4-1015dx		MacBook Air Mid 2013	
	Avg MCC	**Avg MTAR**	**Avg MCC**	**Avg MTAR**
2	0.5364	39.44	0.9972	0.6667
4	0.5785	14.23	0.9982	0.5083
6	0.5398	35.48	0.9984	0.4833
8	0.4911	69.52	0.9984	0.4583
10	0.5247	45.67	0.9984	0.5083
12	0.5122	57.73	0.9984	0.4500
14	0.4906	69.65	0.9985	0.4083
16	0.4993	66.75	0.9984	0.4833
18	0.4851	70.28	0.9983	0.4333
20	0.4854	70.35	0.9985	0.3917
22	0.4849	70.34	0.9984	0.4417
24	0.4898	70.07	0.9984	0.4833

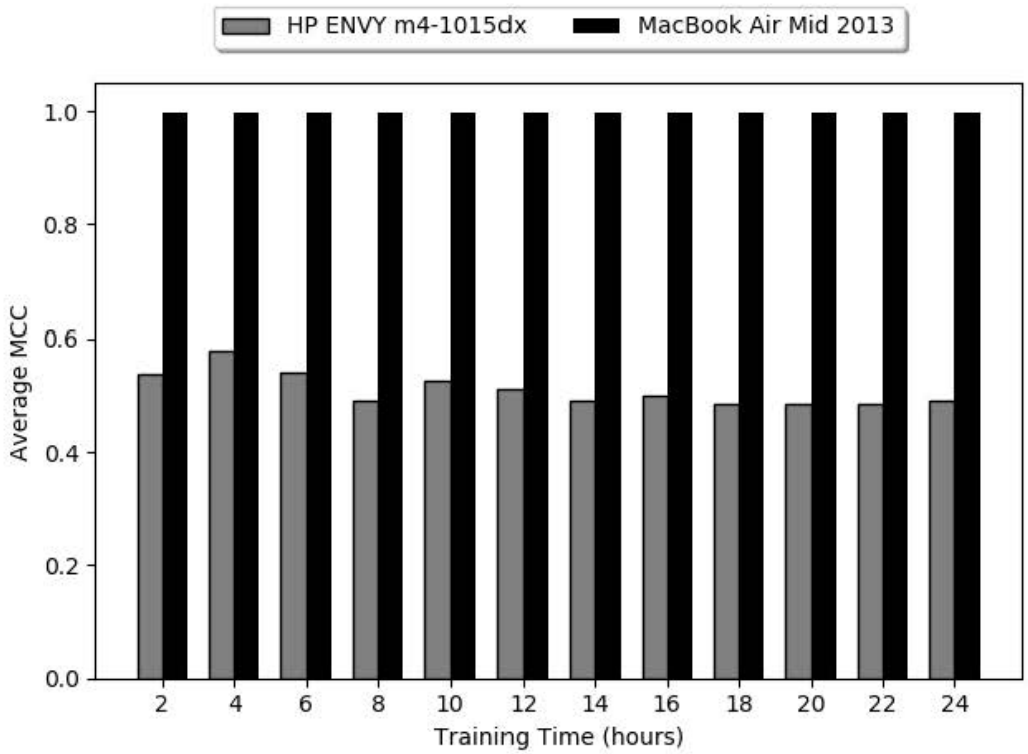


Figure 4.8. Training Time Analysis MCC Results

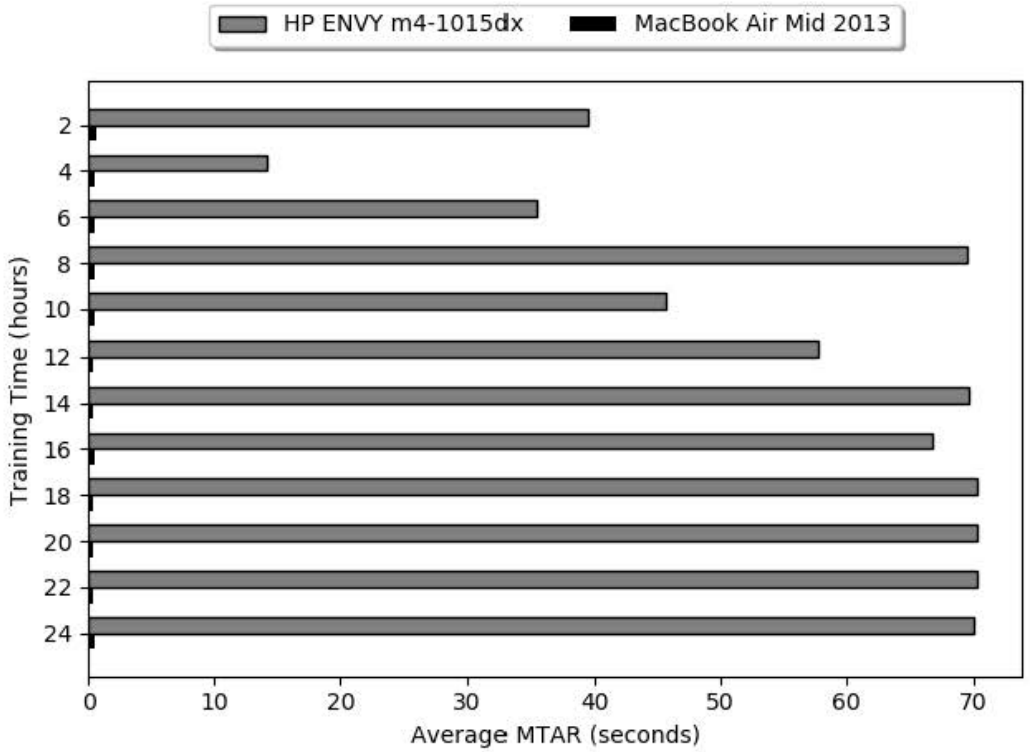


Figure 4.9. Training Time Analysis MTAR Results

4.3.5. Effect of User System Load on Performance

The previous testing demonstrated the viability and potential of the new ransomware detection method under favorable system conditions. Testing with additional system loads represent a more complex and realistic situation which in turn requires a more complex prediction model. Figure 4.10 shows the more complex model used for detecting ransomware when unknown additional system loads are present. This method uses a collection of five prediction models which implement the the same machine learning algorithms selected for each of the two systems during the training time testing which are trained with 0%, 25%, 50%, 75%, and 100% additional system loads present. Each of the five models provides a confidence score for a state of "normal operation" and a state of "under attack" from the same input vector of sensor data. In order to determine which of the five models to utilize the ransomware attack process is run with only regular background processes and monitored to determine a value to use as the additional CPU load the processes is likely to introduce on the system. During testing the current system CPU load is used to determine which of the models to use for a "normal operation" confidence score and which of the five models to use for an "under attack" confidence score. Given that no ransomware is attacking the system the behavior of the system would be best represented by the model trained with an additional CPU load closest to the current CPU load resulting in a higher "normal operation" confidence score. However, given that there is currently ransomware attacking the system the current system CPU load minus the CPU load which the ransomware process is likely to place on the system would be used to select the prediction model to supply an "under attack" prediction. The two confidence scores are compared and the higher score is used as the final system state prediction for the ensemble model. The ensemble model is first tested with no additional CPU load present on the system, similar to the previously conducted tests. However, unlike the previous tests which always used the prediction model generated with no additional system load this more complex model has to select a model which may not always be the right one. In table 4.9 it can be seen that the performance of the model

has decreased due to the requirement of determining which models to use, but in the case of the MacBook Air test system there was still MCC score above 90%. When comparing table 4.9 to tables 4.10 and 4.11 it can be seen that the ensemble prediction model was more effective with rising additional unknown CPU loads. All MCC scores for the MacBook Air test system were all above 90% with additional CPU loads of 25% and 50%. Even with the additional requirement of detecting ransomware attacks with unknown additional system loads the new ransomware detection method was able to detect all attack instances with most mean time to attack recognition results just over 7 seconds on the system with a large array of sensors.

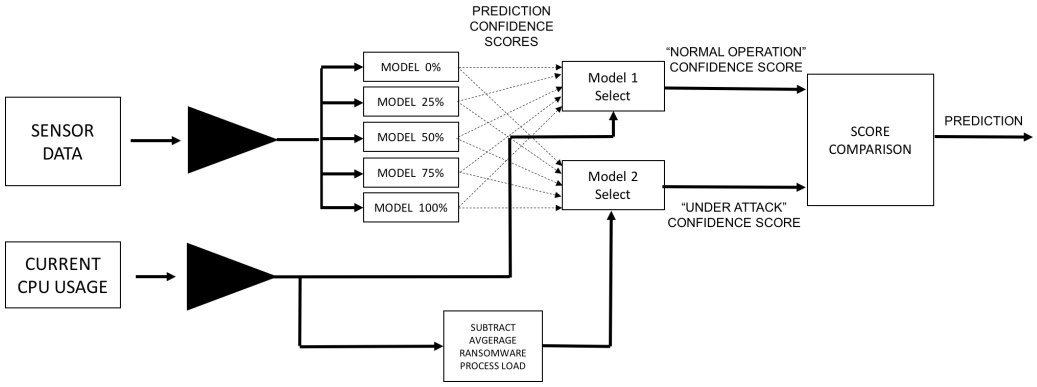


Figure 4.10. Simulated Load Analysis Ensemble Predictive Model

4.4. Summary and Contributions

In this experiment we show that ransomware attacks can be effectively detected in a rapid manner before significant amounts of data are encrypted during an attack. Once the machine learning algorithms have been trained and a predictive model has been generated, predictions about the state of a system are calculated quickly and with a low computational overhead.

Perhaps the most important aspect of this experiment is the speed in which an attack

Table 4.9. Effect of User System Load on Performance (0%)

	HP ENVY m4-1015dx		MacBook Air Mid 2013	
	Avg MCC	**Avg MTAR**	**Avg MCC**	**Avg MTAR**
ECB	0.4724	31.334	0.9410	8.6667
CBC	0.3057	53.334	0.9561	3.1667
CFB	0.3313	120.84	0.7238	6.6667
OFB	0.3921	50.167	0.8430	1.3334
XOR	0.5695	52.0	0.6985	5.8334
ALL	0.5076	57.467	0.6098	9.0

Table 4.10. Effect of User System Load on Performance (25%)

	HP ENVY m4-1015dx		MacBook Air Mid 2013	
	Avg MCC	**Avg MTAR**	**Avg MCC**	**Avg MTAR**
ECB	0.2449	92.0	0.9368	13.0
CBC	0.3168	111.5	0.9267	6.0
CFB	0.4219	142.83	0.9134	10.834
OFB	0.3707	91.0	0.9303	8.0
XOR	0.2344	36.334	0.9237	10.167
ALL	0.4755	55.167	0.9132	11.867

Table 4.11. Effect of User System Load on Performance (50%)

	HP ENVY m4-1015dx		MacBook Air Mid 2013	
	Avg MCC	**Avg MTAR**	**Avg MCC**	**Avg MTAR**
ECB	0.3293	58.834	0.9693	8.8334
CBC	0.4789	63.667	0.9590	7.0
CFB	0.5928	123.0	0.9164	7.75
OFB	0.3576	77.334	0.9408	7.1667
XOR	0.3619	67.834	0.9745	7.5
ALL	0.5148	78.667	0.9518	7.5

may be detected. Detection speed is a very important security concern in attacks on critical infrastructure as the paramount concern is reducing any downtime. Rapid detection could mean a significant reduction in downtime from an attack as less of the system would be corrupted. In experimental testing, the highest performing system had an average time to attack recognition which was as little as 1.3 seconds and never exceeded 13 seconds in even the lowest performing models. However, it has been found that even the best predictive models generate some false positive predictions. For this reason it is believed that the most effective method for implementing this technique would occur when more detailed analysis follows the indication of a positive prediction. When the quick acting predictive model indicates a positive prediction some degree of preventative measures would be taken to slow or stop the ransomware process from inflicting further damage. These preventative measure could include stopping the system for more in depth analysis in the most sensitive environments, briefly enforcing stricter firewall rules to defend against further attacks on network entities, or even simply notifying the user to potential risk.

Chapter 5

Rapid General Process Detection

The ability to detect and characterize processes running within the working set of a computer system via side channels enables a variety of applications including such as malware detection and white- and black-list validation. The use of side channels for general process detection likewise allows for such processes to be identified and characterized in a manner that is difficult to spoof since the physical characteristics of the process are used rather than signatures or other means.

Previous work in this area targeted the detection of a specific process such as ransomware (Taylor et al., 2021). In this past work certain physical sensors present in modern System-on-Chips (SoC) that comprise CPU cores were used to gather real-time data that was provided as input to a machine learning classifier to determine if an instance of ransomware was present. This past approach involved training the classifier with instances of processes that exhibited ransomware behavior. Specifically, many ransomware processes can be generally characterized as first identifying victim files for encryption followed by a phase where the identified files underwent encryption. From a general process point of view, one can view these activities as first comprising a phase of file I/O activity followed by a phase of CPU-intensive operations when the encryption algorithms are run.

The objective of this work is to perform general process detection at a finer-grained level such that sensor data can indicate and discriminate between processes that are heavily biased toward file I/O activity or CPU/ALU-intensive activity. If a suitable set of general processes could be detected, then the sequence described by the ransomware process would involve first detecting file I/O intensive activity followed by the detection of CPU/ALU-intensive activity. Thus, given the sequence behavior of any type of application, whether it

was an instance of malware or not, the ability to detect the sequence and type of processes running on a CPU-based system could represent a set of basic building blocks that allow for arbitrary types of applications to be detected via physical side channels by characterizing an application of interest as a sequence of activities that are dominated by a particular type of process. For example, when a process is identified as first consisting of file I/O activity followed by bursts of heavy CPU/ALU activity then a process is detected that is searching for target files followed by processing them through heavy computation such as is the case for many instances of ransomware. Such a capability greatly generalizes the approach taken for ransomware detection in (Taylor et al., 2021) and furthermore does not require customized training of process classifiers for each different instance of applications to be detected. Rather, the system administrator can specify a particular sequence of processes to characterize a process of interest, and when such a sequence is detected, an alert can be issued indicating a high probabllity that a particular process sequence has been detected. Another benefit of this research is the evaluation of the effectiveness of using physical sensor side channels (PSSC) or different types of general processes.

The efficacy of the use of PSSC-based detection and characterization is largely dependent upon the presence, or lack of presence, of appropriate sensors in the host architecture. For example, if a particular host architecture does not have sensors present inside or near the network interface circuitry (NIC), then the detection of a process that contains a lot of communication with the network is more difficult to achieve with PSSC.

Another contribution of this work is the incorporation of models to account for sensor output due to other processes concurrently running on a system. From a signal processing point of view, a particular sensor can be viewed as a data collector whose output is due to the composite set of processes that are instantiated at any instance. While it is the case that operating systems generally use temporal sequencing of multiple processes to provide the illusion of concurrent execution, the time slices are very small and modern architectures have significant levels of concurrency due to clever architectural features. For example, a file

I/O intensive process may be accessing memory through a DMA while the CPU/ALU is concurrently engaged in heavy number-crunching activities. Furthermore, these two activities may have originated from different processes altogether. For this reason, the sensor, when viewed as a data collector, is providing composite information regarding the environment within the system. We have devised our classifier models to account for the presence of such background loads and we have found that such background load models should necessarily vary depending upon the type of process being detected.

We identified four classes of general processes and report on the ability of a PSSC detection and characterization approach for each process category. These are: i) file I/O, ii) CPU/ALU intensive, iii) network I/O, and iv) virtualization instantiation. Our results indicate varying levels of detection ability for these classes of processes. Classes i) and ii) are shown to have high accuracy over all background load conditions whereas class iii) can be reasonably detected under low load conditions and class iv) has very good detection capabilities for low load conditions but accuracy drops off as load conditions increase. The worst performing process detection case was that for network I/O. We attribute this poorest performing network I/O class to the fact that our NIC did not comprise a suitable set of physical sensors. NIC's that contain more sensors or for cases where the NIC is physically located adjacent to other sensor-rich subsystems would perform better. This indicates that alternative side channels for detecting network I/O may be desirable in our target architecture such as the use of network traffic patterns (Kadloor et al., 2010; X.H. Wu, 2015) or the use of programmed event monitors (Li and Gaudiot, 2021; Oshana et al., 2021).

This experiment demonstrates the viability of detecting arbitrary target processes by measuring the physical state of a system through existing system sensors. Previous work has shown that process detection often requires complex behavioral analysis which has shown to be very accurate, but often at the cost of speed (Rhode et al., 2018). The new detection model presented in the experiment is meant to act as an initial rapid indication of a target process which would allow for safety measures to be taken until such time as more tradi-

tional behavior analysis can be performed as verification. In this way the detection method presented in this paper is meant to act in tandem with existing methods of process detection in an effort to augment their performance.

5.1. Detection Models

Detection Models utilize machine learning (ML) algorithms in order to generate a binary prediction as to whether a target process is or is not currently running on a system. In the context of this study a Detection Model is an obfuscation of either a single machine learning model or a collection of machine learning models which when provided a vector of a system's current physical sensor readings provides a binary output representing whether the target process is currently running on a system. Once a Detection Model has been trained and implemented the resources required to perform the underlying machine learning predictions is minimal as there is no complex behavioral analysis necessary. Additionally, as all of the data that is used for process detection is considered side-channel data there is minimal risk to privacy and user data leakage.

5.2. Training and Building Detection Models

5.2.1. Experimental Environment and Setup

This experiment utilized an Apple Mac Mini MGEM2LL/A with a 1.4 Ghz Intel Core i5 processor, 4 GB of LPDDR3 RAM, and a 500 GB HDD. Using a software application name Hardware Monitor we were able to access the current sensor readings for 50 different system sensors through the command line (Bresink M, 2017). The systems sensors were comprised of 16 temperature sensors, 6 voltage sensors, 12 current sensors, 15 power sensors, and 1 sensor for the exhaust fan RPMs. The testing scripts and automated data collection was performed using the Continuum Anaconda Python 3 package and associated libraries (ana). This includes the Scikit-learn library which is utilized in transforming data, creating

prediction models, and performing target process predictions (Pedregosa et al., 2011). In order to create additional CPU loads for training the GO programming language was installed in order to run the Go script "go-cpu-load" (Vikyd). This script maintains a desired CPU load by creating a continuous loop and adjusting a delay period after each loop in order to adjust CPU usage. In order to create additional CPU loads for testing we used the Stress-ng project package which is capable of creating a number of different system stressors with a high level of control (ColinIanKing). We were able to create the CPU load using over 70 different methods included in Stress-ng. All of which were different than the method used during training with "go-cpu-load".

This experiment was carried out with three different target processes which represent different types of processes likely to be run by users. The first target process involves heavy file I/O and utilizes the Iozone filesystem benchmarking tool (ioz). The second target process involves heavy CPU resource usage and utilizes the FFmpeg multimedia framework (ffm). The third target process involves heavy network I/O and utilizes the Nmap network discovery and security auditing tool (nma).

5.2.2. Experimental Process

In order to perform this experiment a python script is executed which performs a series of training data collection followed by a series of test data collection for each test processes. Prior to beginning the data collection for a new process the system is cycled in order to free all resources. After every individual training and test cycle a comma separated data file is stored on the test system until all testing ends and the data can be collected. Once the training and test data are collected the training data is used to create all of the necessary detector models for the various combinations of machine learning algorithms, test processes, and detection model type. Once all of the models are created the test data collected for each test processes is used to generate the detection predictions for each time interval. The same test data is used for all models created for each test process in order to directly compare their

performance with a controlled data set. Once all of the detection prediction vectors have been generated performance metrics are computed in order to analyze and draw conclusions about the new detection method.

5.2.3. Collecting Training Data

In the previous ransomware detection experiment it was shown that training a model for 24 hours resulted in an almost identical result to a model that had been trained for only 2 hours (Taylor et al., 2021). Due to this knowledge training data for each model was collected for two hours. Ensemble detector models are comprised of up to 11 ML models for each test process. The ML models required training with additional system loads from 0% to 100% with one model being trained at each interval of 10%. The additional load is achieved by creating a process on all Hyper-threads which run an empty loop at a dynamic rate that maintains the desired overall CPU load. Maintaining the desired additional load level relative to each Hyper-thread ensures the additional load is balanced over the system's total CPU resources. 22 hours of training is required for each test process in order to accommodate ensemble detector models which utilize the maximum number of ML models.

During each two hour training cycle three threads are created. The first thread controls the operation of the process being tested, the second thread adds a desired additional CPU load onto the system, and the third thread runs a data logger. The test process control thread starts by running the test process for an entire cycle and recording the amount of time that it requires to complete. Afterwards, the test process control thread waits 2 minutes for the system sensors to re-stabilize and then it sets a Boolean variable that indicates that the data logging thread should begin recording data. The test process control thread then begins a loop which first waits for the amount of time previously recorded for the test process to complete, then the test process is run, and finally the test process control thread waits again for the amount of time previously recorded for the test process to complete. The test process control thread then waits 2 minutes for the system sensors to re-stabilize and if the

total time has met or exceeded the desired total training time the loop stops. This data collection routine results in two hours of data which has a 2 to 1 ratio of the test process not running to the test process running. This ratio is not a realistic representation of what is likely to be experience in a real world environment. However, for the purposes of training the detection models this ratio is more beneficial.

5.2.4. Training Machine Learning Models

Machine Learning models are trained with a known additional system CPU load level present while the test process is both running and not running. This allows for the selection of a ML model in a detection model which is more likely to have been trained only under conditions similar to the system's current physical state based on the total CPU utilization at each point in time a determination is being made as to whether the target process is currently running. Ensemble detectors are comprised of these more targeted individual ML models. Additionally, a ML model is trained for each test process which includes all of the training data collected at various additional CPU load levels. This single model is implemented alone rather than as part of an ensemble detector.

5.2.5. Building Ensemble Detectors

Ensemble detectors are built by including a collection of ML models which have each been trained with a single known additional system CPU load present. The maximum size of the detection models collection is 11 as ML models have been trained at known additional CPU load level intervals of 10% from 0% to 100%. The ensemble detector utilizes a ML model selection function which receives the current total system CPU utilization at each instance a target process is determined to be either running or not running. The ML model selection function can be implemented to select two ML models or a single ML model for each detection decision. When two ML models are selected one ML model provides a probability value representing whether the target process is not currently running and the

second ML model provides a probability value representing whether the target process is currently running. The ML models are selected by assuming the load created by the target process is or is not included in the current total system CPU utilization respectively. In the event that the target process is not currently running then the total system CPU utilization would be considered the additional load level present during training conditions most similar to the current state. Conversely, in the event that the target process is currently running then the total system CPU utilization must be reduced by the load created by the target process in order to be considered the additional load level present during training conditions most similar to the current state. When the ML model selection function is implemented to select only a single ML model the added load of the target process is not considered for the condition where the target process is currently running and instead the probability of both conditions is determined by the ML model trained with the closest additional system CPU load as the current total system CPU utilization.

5.3. Testing Process Detectors

5.3.1. Collecting Test Data

Test data is collected using the same method as training data. However, the test process is not initially timed and there is no loop for running the test process until the desired time has elapsed. Instead, the test process control thread waits a random amount of time between 10 and 40 minutes, runs a single instance of the test process, and then waits for the remainder of the desired test time. This process results in data that has a very short amount of time where the test process is actively running and it is occuring at a random time.

5.3.1.1. Zero Additional Load

Test data is collected while no additional CPU load preset on the system beyond what is present due to regular background processes. Each test process is run a single time over the

course of one hour at a randomly determined time. This is repeated 24 times to complete the data set for each test process.

5.3.1.2. Simple Random Additional Load

Test data is collected with a randomly selected additional CPU load preset on the system beyond what is present due to regular background processes. Each test process is run a single time over the course of one hour at a randomly determined time. Prior to performing each test run an additional CPU load level is randomly selected from a specific range of values and is applied to the system using the same method implemented in training. Four values are selected from a range of 10 values before moving to the next range of 10 values starting with 1 through 10 and ending with 91 through 100. This results in a data set of 40 tests with equal representation of random additional load levels at intervals between where training data was collected with known additional CPU loads. This data set allows for the possibility of randomly selected additional load levels to be the same as the known load levels used during training.

5.3.1.3. Advanced Random Additional Load

Test data is collected with a randomly selected additional CPU load preset on the system beyond what is present due to regular background processes. Each test process is run a single time over the course of one hour at a randomly determined time. Prior to performing each test run an additional CPU load level is randomly selected from a specific range of values and is applied to the system using a different method from what was implemented during training. The randomly selected load level is further randomly divided into unbalanced load levels to maintain on each Hyper-Thread that cumulatively apply the overall desired additional system CPU load level. The method of maintaining the load level on each Hyper-Thread is randomly selected from 70 different CPU stress methods which does not include the empty loop method utilized in training. Four values are selected from a range of 9 values

before moving to the next range of 9 values, skipping those used during training, starting with 1 through 9 and ending with 91 through 99. This results in a data set of 40 tests with equal representation of random additional load levels at intervals between where training data was collected with known additional CPU loads. This data set does not allow for the possibility of randomly selected additional load levels that are the same as the known load levels used during training.

5.3.2. Target Process Detection Methodology

5.3.2.1. Process Detection With Unknown Additional Loads

Section 5.3.1 details test data collection with three different types of loads present on the system which are all unknown to the detector. The data set collected from each type of unknown additional load implementation is used for a separate test of process detector's performance ability.

Zero additional load test data is designed to serve as an ideal environment for process detection as relatively simplistic physical sensor activity is likely to be sufficient for accurate target process detection. The test on this data set is used for showing the viability of the process detectors as a simple proof of concept. Analysis of detection ability with this data set allows for easily identifying process detectors which are unsuitable for more rigorous testing. The results of this testing is unlikely to be highlighted.

Simple random additional load test data is much more realistic and rigorous for testing the process detectors than the zero additional load data set test. However, the implementation of the same method of applying the additional balanced CPU load as well as allowing for the possibility of selecting random load levels which are the same as those used during training incorporate some favorable aspects for process detection. The test on this data set is used for showing the ability of the process detectors when the system is likely in an unseen state from what was used for training, but the system exhibits similar patterns of behavior. Analysis of

detection ability with this data set allows for identifying process detectors which are capable of performing in the most basic realistic scenarios.

Advanced random additional load test data is much more realistic and rigorous for testing the process detectors than the zero additional load and simple random additional load data sets. The implementation of multiple unseen methods of applying the additional unbalanced CPU load as well as not allowing for the possibility of selecting random load levels which are the same as those used during training truly tests the detection predictors in a scenario where obvious behavioral similarities are not present. The test on this data set is used for showing the ability of the process detectors when the system is in an unseen state from what was used for training with comparable system states achieved through completely different methods. Analysis of detection ability with this data set allows for identifying process detectors which are capable of performing in unfavorable and unseen realistic scenarios.

5.3.2.2. Detection Model Type Comparison

Section 5.2 details how ML models are trained and section 5.2.5 details how a collection of ML models can be used to build an ensemble detector which attempts to select ML models which are optimal for the system's current physical state. Five different methods of process detection utilizing the trained ML models were evaluated in this experiment. Each of the five methods was tested on all 3 unknown additional load for each test processes.

The most simple detection method is utilizing a single ML model which has been trained with all of the training data collected at the various different known additional load levels. The simplicity of this method is illustrated in figure 5.1. This detection method is labeled as "Single" in the experimental results.

Ensemble detection methods are more advanced and select from a collection of up to 11 different ML models based on the current system state. Single selection ensemble detectors select the ML model which was trained with an additional CPU load that is closest to the current overall CPU load of the system. In this experiment, single selection ensemble

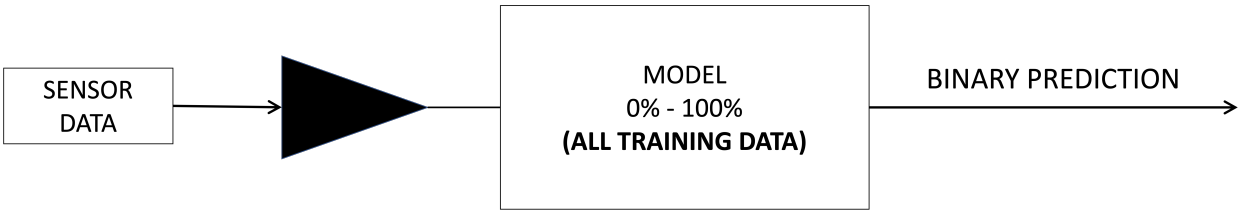


Figure 5.1. Single Detection Model (Single)

detectors with 6 ML models and 11 ML models are evaluated. These detectors are shown in figure 5.2 and figure 5.3 and will be labeled as "Ens 6 S" and "Ens 11 S" respectively in the experimental results.

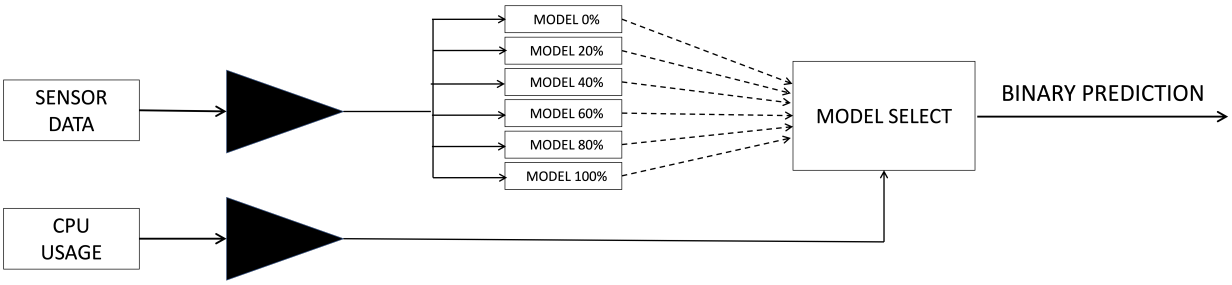


Figure 5.2. Ensemble Detector - Single Model Selection - 6 Models (Ens 6 S)

Dual selection ensemble detectors select the ML model which was trained with an additional CPU load that is closest to the current overall CPU load of the system. However, this model is used only to provide the probability that the target process is not currently running. A second ML model is selected for providing the probability that the target process is currently running which was trained with an additional CPU load that is closest to the current overall CPU load of the system after the average load created by the target process is subtracted. This is because the ML model load level is only the additional load added not including the load added by the target process. Therefore, the ML model which most likely was trained under similar conditions when the target process is running is actually a lower

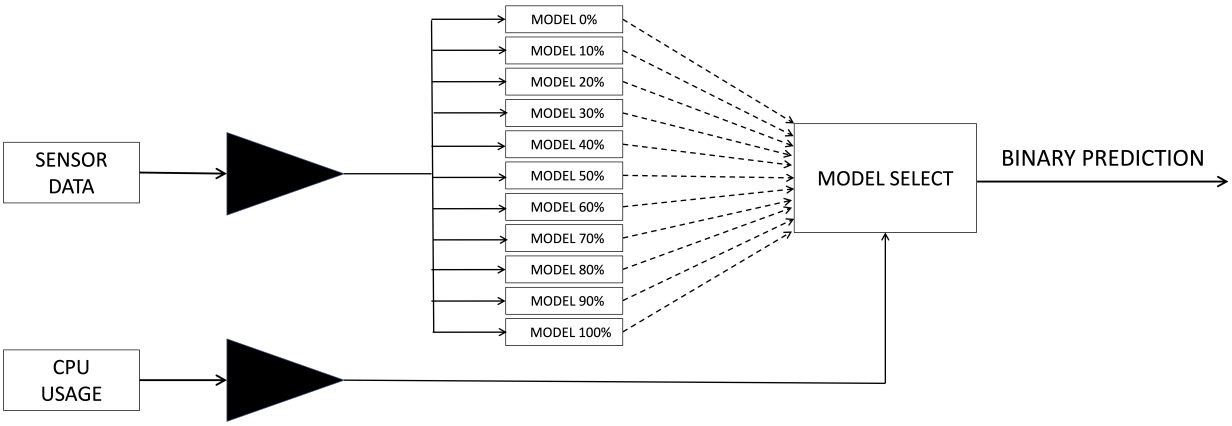


Figure 5.3. Ensemble Detector - Single Model Selection - 11 Models (Ens 11 S)

load level. In this experiment, dual selection ensemble detectors with 6 ML models and 11 ML models are evaluated. These detectors are shown in figure 5.4 and figure 5.5 and will be labeled as "Ens 6 D" and "Ens 11 D" respectively in the experimental results.

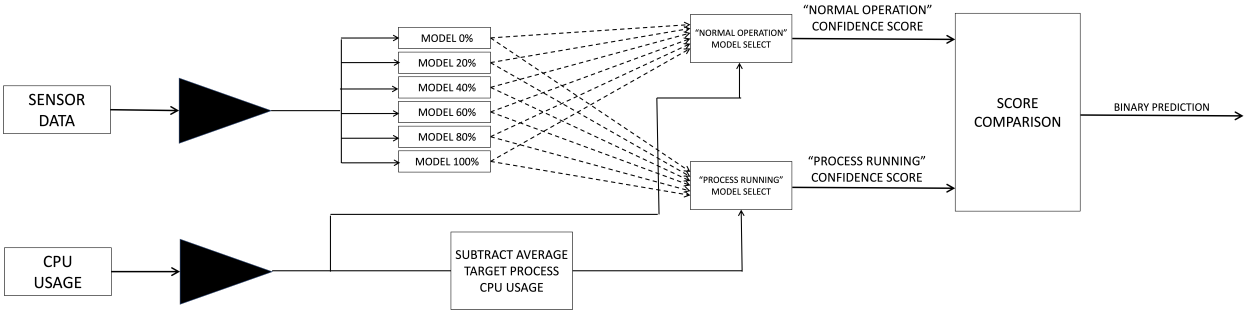


Figure 5.4. Ensemble Detector - Dual Model Selection - 6 Models (Ens 6 D)

The detection performance results of the five different process detection methods can be directly compared to draw conclusions about the effectiveness of more advanced techniques and whether the additional complexity introduced justifies any potential performance increase as was seen in (Taylor et al., 2021).

5.3.3. Performance Evaluation Methodology

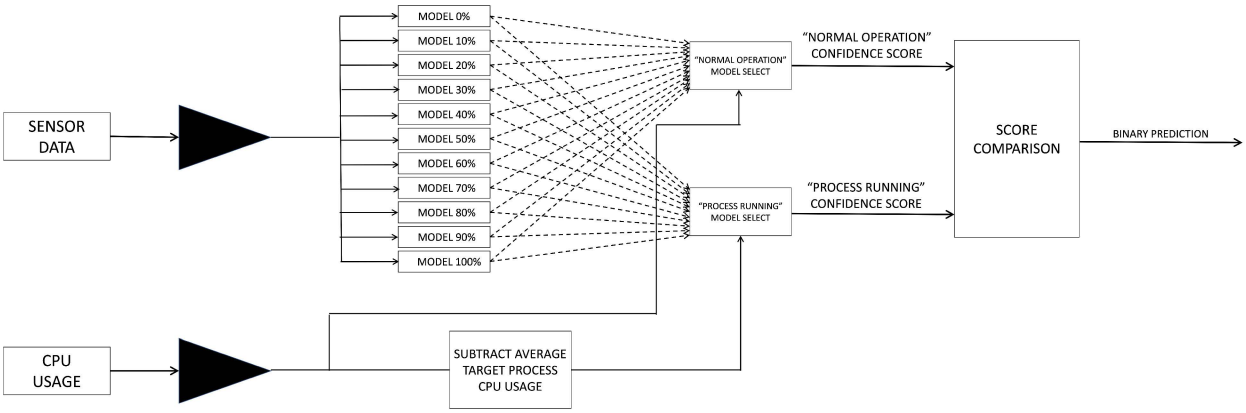


Figure 5.5. Ensemble Detector - Dual Model Selection - 11 Models (Ens 11 D)

5.3.3.1. Binary Classification Evaluation

In this experiment machine learning algorithms are each used to make a prediction about the binary status of a process on a system. The final prediction vector of a model and the actual system state vector are compared to obtain the distribution of the four types of binary classification. The distribution of the four classifications was used to compute five metrics which offer more insight into prediction performance: sensitivity, precision, specificity, fallout, and accuracy (Tharwat, 2020).

5.3.3.2. Matthews Correlation Coefficient (MCC)

The Matthews correlation coefficient (MCC) takes into account true and false positives and negatives and is generally regarded as a balanced measure which can be used even if the classes are of considerably different sizes such as the data in this experiment (Chicco and Jurman, 2020).

The MCC is a correlation coefficient between the observed and predicted binary classifications. Values range between -1 and +1. A coefficient of +1 represents a perfect predictor,

0 represents the same as random prediction, and -1 indicates total disagreement.

5.3.3.3. Rate of Process Detection (RPD)

When testing, an actual process presence time series exists which defines the time periods during which the target process is actually running on the system. Target process runs start when a labeled system state of "normal operation" transitions to a labeled state of "process running". Conversely, a target process run ends when a labeled system state of "process running" transitions to a labeled state of "normal operation". If a positive prediction exists during the period of time representing a target process run then the target process is considered detected. Ideally there should exist at least one positive prediction during each target process run which would result in a 1.0 or perfect rate of process detection.

5.3.3.4. Time to Process Detection (TPD)

The initial labeled system state of "process running" for each target process run in the actual process presence time series represents the time interval at which the target process run began. The first instance of a positive prediction in the corresponding prediction time series at or after this initial "process running" state and before the next "normal operation" system state represents the initial target process detection. Ideally the initial "process running" state itself would be a positive prediction, but in practice it is more likely that the sensors would need a small amount of time to reach the values at which positive prediction occurs. This metric counts the number of time intervals until the first positive prediction is recorded for every target process run which was successfully recognized. Afterwards all values are averaged to determine the time to process detection.

5.3.3.5. Detector Performance Score (DPS)

The new detection model is designed to rapidly detect a target process while maintaining a low fallout. In other words we are most concerned with three aspects of the detector's

performance. The first aspect is to maintain a low TPD which would mean the detector is able to quickly identify when the target process begins running on a system. Based on previous work we determined that in order for our model to be effective it must have a TPD of no more than 60 seconds (Rhode et al., 2018). The second aspect of performance most important for our new detection model is detection models maintaining a low fallout during operation. This would mean that the detection model is not creating so many false positive detection alerts that it becomes a detriment to operation. In a 2020 report by Neustar Security Solutions it was found that on average 26% of security alerts experienced by organizations were deemed false positives (24 et al., 2020). Therefore, we determined that in order for our model to be effective it must have a fallout of no more than 0.26. The final aspect of performance most important for our new detection model is that it should miss minimal test instances of a target process. We determined that if the detection model misses an instance of the target process during testing it should be heavily penalized during assessment of performance. Taking into account the three most important aspects of performance we used a metric called Detection Performance Score (DPS) in order to help in our assessment of performance alongside the other more traditional binary classification metrics.

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

5.4. Experimental Results

The results for each of the four test processes were compared using the four different tests outlined in 5.3.2. The results for the simple random additional load test, and the advanced random additional load test are shown for each algorithm. However, only the single model detector results are included. This provides detailed insight into the performance of the process detectors in their simplest form of implementation. The results for the detection model comparison, performed for each process, shows the highest performing machine

learning algorithm for each detection method. The results for the zero additional load are not shown in the data as they were only used to determine if additional testing should be performed with a specific target process.

5.4.1. File I/O Process

With this process we were aiming to create a significant amount of file I/O on the system in order to determine if the new method is capable of detecting a specific pattern of file I/O with a random amount of CPU usage occurring at the same time. We implemented the filesystem benchmarking tool IOzone to act as our file I/O process. IOzone generates and measures a variety of file operations in order to measure a system's file I/O performance. There are a total of 13 test types which includes 6 write tests and 7 read tests. The write tests are defined as Write, Re-write, Random Write, Record Re-Write, Fwrite, and Frewrite. The read tests are defined as Read, Re-read, Random Read, Backward Read, Stride Read, Fread, and Freread. The actual process we ran was as follows

```
$ iozone -a
```

The option "-a" means that iozone should run all 13 tests back-to-back. This command required roughly 8 minutes to complete on the Mac Mini systems.

5.4.1.1. Test Results

The results of the unknown additional load tests can be seen in table 5.1. During the advanced random load test the detector which was trained using the logistic regression machine learning model had the best performance. The detector correctly caught all instances of the target process during testing in an average of 0.275 seconds.

Examining the series of plots in figure 5.6 it can be seen that the logistic regression model maintains almost perfect DPS and MCC scores throughout all test instances with varying additional random loads. The plot on the right also shows that the logistic regression model

maintains a fallout rate very close to zero throughout all of the test instances as well.

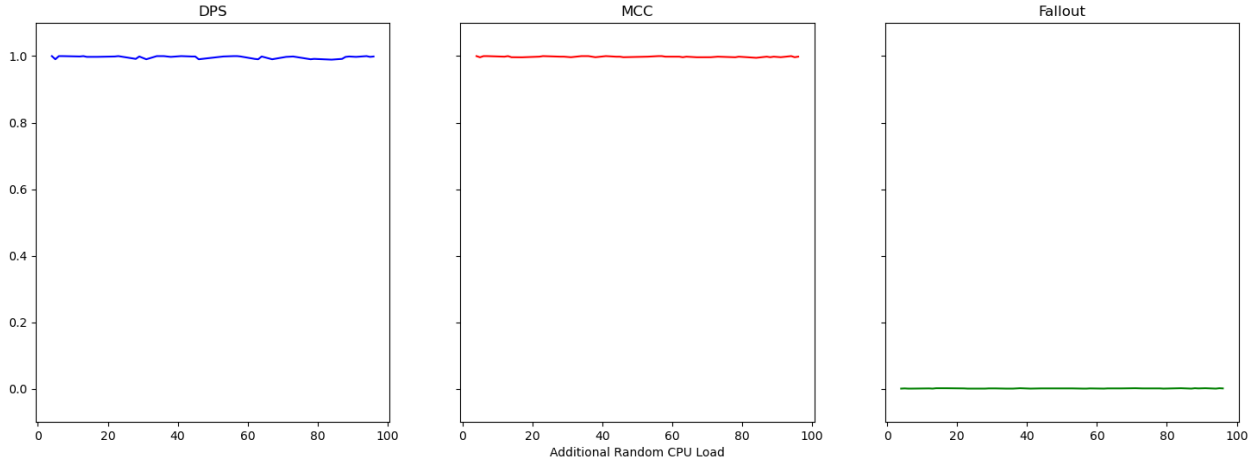


Figure 5.6. File I/O Process Binary Classification Metrics - Single Detector Trained With Logistic Regression

Figure 5.6 shows three time series plots which are all aligned on the same x-axis which represent prediction time intervals. The top timeseries plot shows the total random additional load present on the system at a given time. The middle timeseries plot shows the actual state of the system during testing with a rising edge representing the start of the target process and the falling edge representing the end of the target process. The bottom timeseries plot shows the prediction made by the detector for each time interval. Ideally the middle and the bottom plots would be identical and the more similar the two plots are the better the detector performed. In figure 5.7 the two plots are very similar with the exception of a limited number of false positives which occurred over the entire 40 hour testing block. These results show that the new method of process detection was highly successful when the process was mostly performing file I/O tasks.

Table 5.2 shows the features with the highest feature importance scores based on the built in Scikit-Learn "feature_importance_" attribute for random forest models. The feature importance scores are determined using gini importance or mean decrease in impurity (MDI).

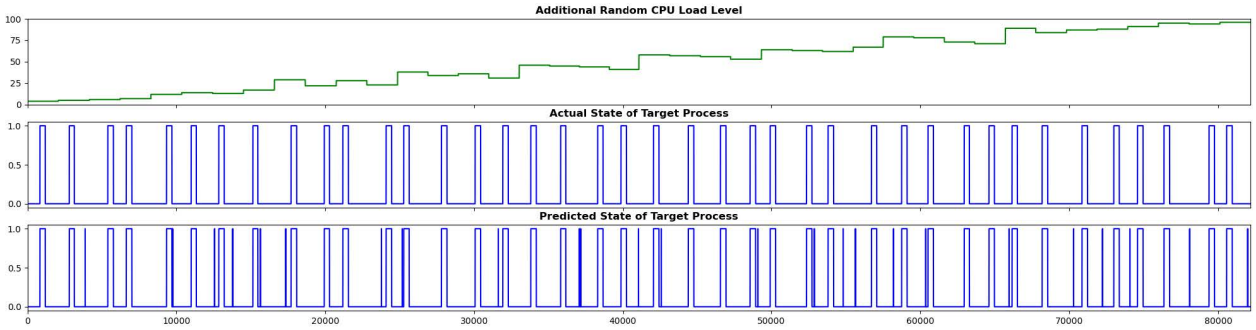


Figure 5.7. File I/O Process Prediction Time Series - Single Detector Trained With Logistic Regression

MDI calculates each feature importance as the sum over the number of splits across all tress that include the feature proportionally to the number of samples it splits (Pedregosa et al., 2011) (per, 2015). We can see that most of the top scoring features are sensors that measure the CPU and the hard drive.

Figure 5.8 shows the percentage of the total feature importance values based on the system components they measure. The CPU accounts for almost half of all feature importance with the remainder being nearly evenly distributed between the power supply, the hard drive, the memory, and the platform controller hub (PCH). It is interesting to note that despite there being unknown and randomized loads placed on the CPU during testing the file I/O process is still mostly able to be detected by monitoring the physical state of the CPU.

Figure 5.9 shows the percentage of the total feature importance values based on the type measurements the features perform. Current and power sensors account for just under 90% of the total feature importance values. This likely means that the detection of the file I/O process is achieved through complex patterns of rapid changes in the operational needs of the CPU, hard drive, memory, and PCH. When considering that file I/O generally requires the CPU to read and write from the hard drive using the motherboard and PCH to move and store the intermediate working data in memory the feature importance scores are right in

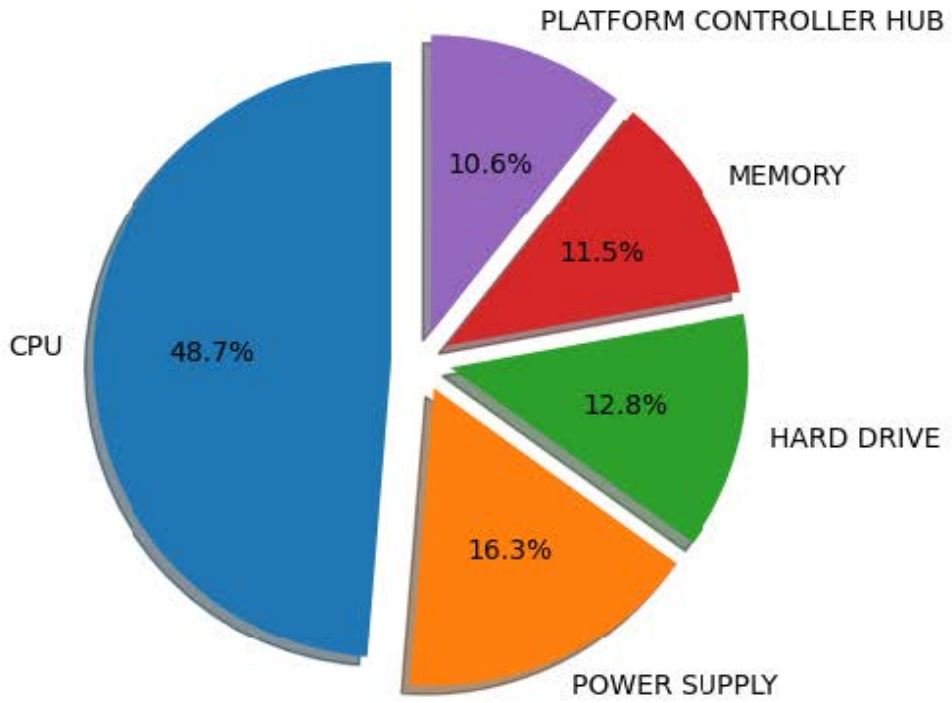


Figure 5.8. File I/O Process Feature Importance by System Component

line with what we should expect to see. While this is a more simplistic process to understand and make sense of in terms of feature importance it is important to note that we trained and implemented the detector successfully without ever having to rely on our understanding of how the process works.

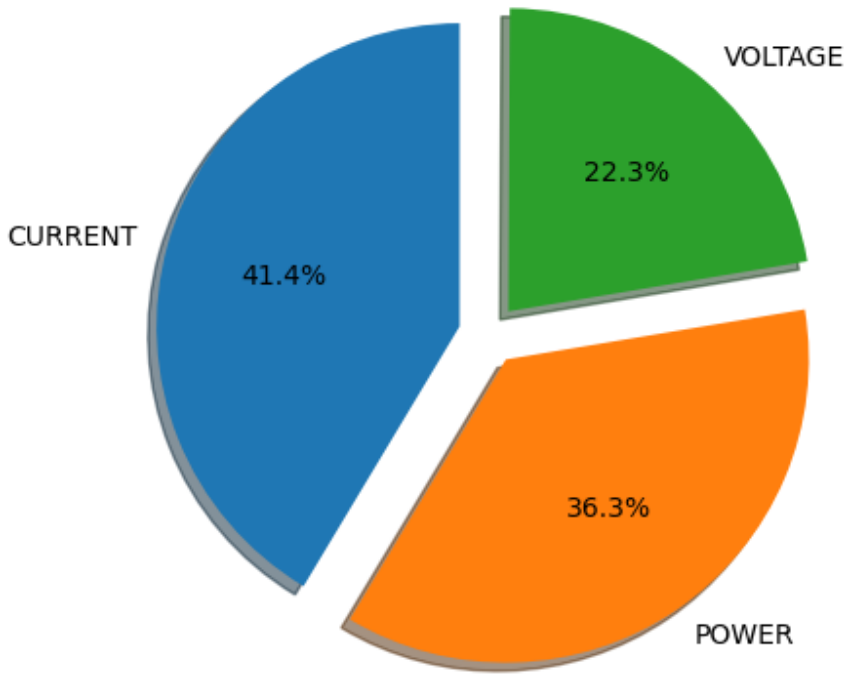


Figure 5.9. File I/O Process Feature Importance by Sensor Type

When comparing the highest performing machine learning algorithm from all of the detector models in table 5.3 it can be seen that the all of the detection model types displayed very high performance. All instances of the target process were caught by all detection model types in an average of less than half a second.

This test process demonstrates a situation where the target process utilizes many different types of system resources during operation, including the system component with random loads introduced during operation. This test demonstrated the viability of utilizing the new detection models for target processes which fit this category.

5.4.2. CPU Intensive Process

With this process we were aiming to create a significant amount of CPU usage on the system to determine if the new method can detect a CPU intensive process with an additional random amount of CPU usage occurring at the same time. The actual process we ran utilized the multimedia tool ffmpeg to convert a video in mov format to a video in mp4 format. This requires ffmpeg to reencode the entire video which requires a large amount of computational resources to perform. The video being converted is exactly 100 MB, 3 minutes in length, and high definition resolution. The actual process we ran was as follows

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

The "-i" option means that the next argument is the path to the input video file. The option "-c:v" mean that the next argument should be used as the encoder. The "-crf" option means the next argument is the constant rate factor that should be used by the encoder. The range of the CRF scale is 0-51, where 0 is lossless, 23 is the default, and 51 is the worst quality possible. Low CRF values result in higher quality output videos, but also increase the total size of the video file. We used a CRF of 10 which is very high quality output and thus a higher amount of computation. This command required roughly 6 minutes to complete on the Mac Mini systems.

5.4.2.1. Test Results

The results for the CPU intensive unknown additional load testing can be seen in table 5.4. When looking at the results of the random additional load test it can be seen that the

detector which was trained using the random forest machine learning algorithm performed the best with all target process instances being detected in an average of 1.275 seconds.

Table 5.10 shows that the fallout rate remains almost at zero through all 40 hours of testing. However, there is a noticeable decline in DPS and MCC as the random additional CPU load increases. The DPS begins to drop slightly once the additional CPU load level reaches 75%, but at no time does DPS ever go below 0.8. However, the MCC score drops dramatically starting at around an additional CPU load level of 50%. Once the additional load level reaches around 90% the MCC score drops to almost 0.5.

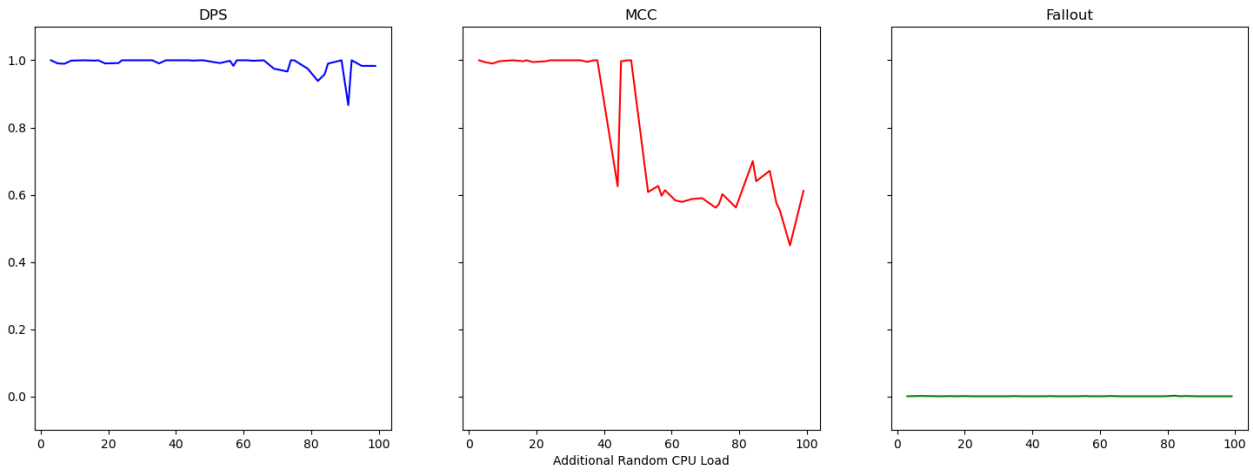


Figure 5.10. CPU Intensive Process Binary Classification Metrics - Single Detector Trained With Random Forest

The MCC scores of the detectors dropped to levels which, on their own, appeared to suggest poor performance. However, the DPS of the same models remained very high. This can be explained by examining the timeseries plots in figure 5.11. The prediction timeseries appears to be almost identical to the actual system state timeseries with the exception of the target process run instances appearing to be filled solid at high additional CPU load levels.

Figure 5.17 shows a much smaller section of the same time series plot as figure 5.11 and shows that during the time the target process is running the detector demonstrates poor

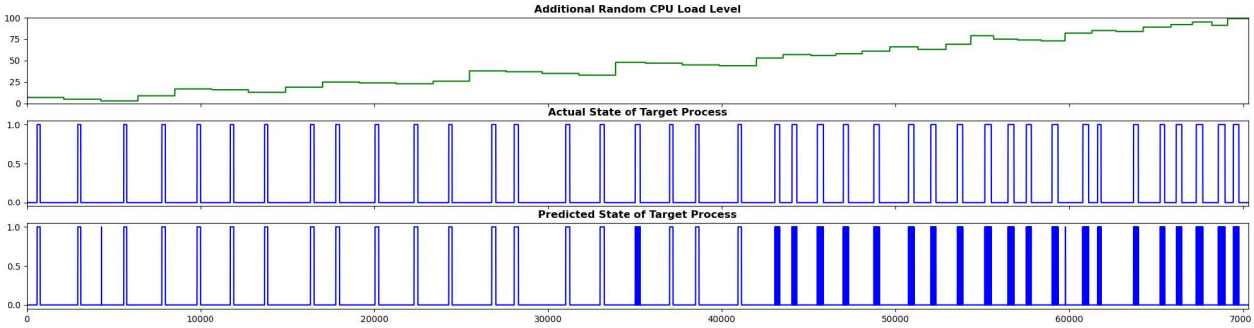


Figure 5.11. CPU Intensive Process Prediction Time Series - Single Detector Trained With Random Forest

performance at sustaining the positive prediction and instead rapidly alternates between positive and negative predictions. In terms of a binary classification model this is a very undesirable performance which is reflected in the MCC scores. However, in the context of rapid process detection we are more concerned with the initial positive prediction as that would act as a trigger for more in depth behavioral analysis. Groupings of positive predictions would be accounted for at this more advanced stage of analysis. This also means that fallout must be kept to a minimum in order to avoid excessive instances of in depth behavior analysis which are falsely triggered by the detection model. The lack of false positives in the timeseries plots once again supports the high DPS as this detector, while not a high performing binary classifier, is a high performing rapid process detector.

Table 5.5 shows the MDI feature importance scores from the random forest trained single model detector. The most notable observation from the table is that features representing senors measuring the CPU are not nearly as important as they were for the file I/O process. Instead, most of the top feature names appear to indicate that the system memory and the power supply are the most important system components when it comes to the CPU intensive process.

Table 5.13 supports this observation by showing that 48.6% of feature importance is

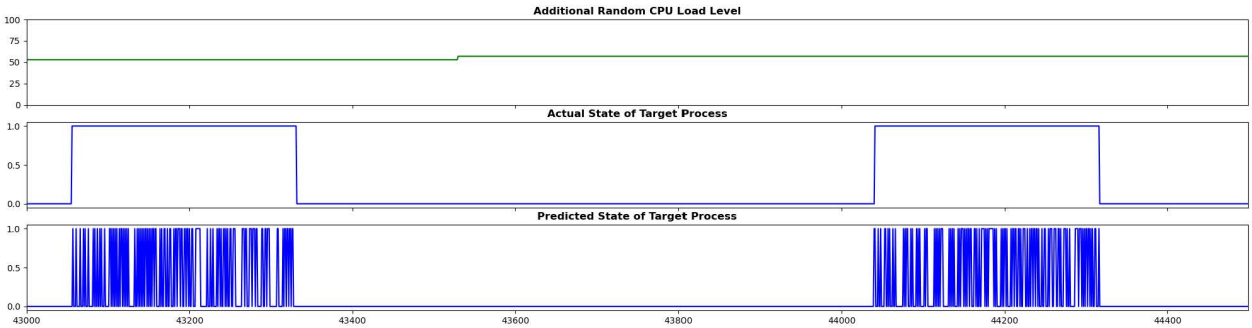


Figure 5.12. CPU Intensive Process Prediction Time Series Zoomed

assigned to features which represent sensors measuring the system memory and 26.6% of feature importance is assigned to features which represent sensors measuring the power supply. Although 24.5% of feature importance is assigned to features representing sensors measuring the CPU it could be initially surprising as one might expect a CPU intensive process to heavily rely on the CPU for detection. However, as we can see in the feature importance scores the features which represent sensors measuring the CPU are likely diminished in their capacity to help in detection due to the additional random CPU load placed on the system. This forced the detection model to identify patterns and trends in alternative system components in combination with the measurements from the CPU in order to detect the CPU intensive process.

Based on the feature importance assigned to the different sensor types seen in figure 5.14 it is likely that the detector is using very small and rapid fluctuations in power in multiple system components, especially power supplied to the system memory, in order to identify the CPU intensive process. The process is a video processor and heavily utilizes system memory and CPU processing during operation which would also causes spikes in the power required for operation.

When comparing the random forest single model detector to the four other detection model types the 5 models all performed very well with the single detector model performing

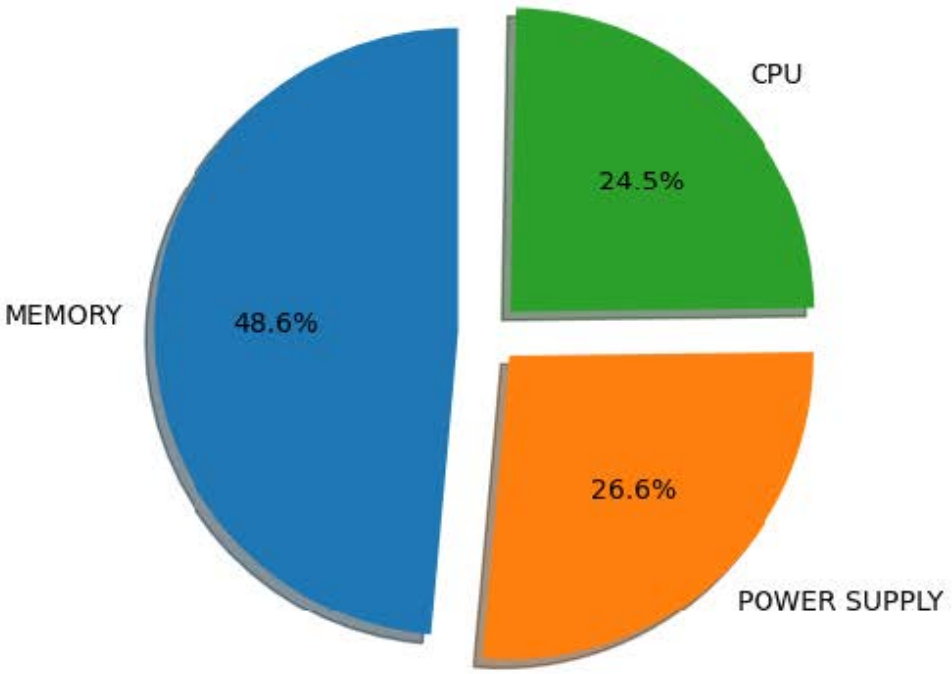


Figure 5.13. CPU Intensive Process Feature Importance by System Component

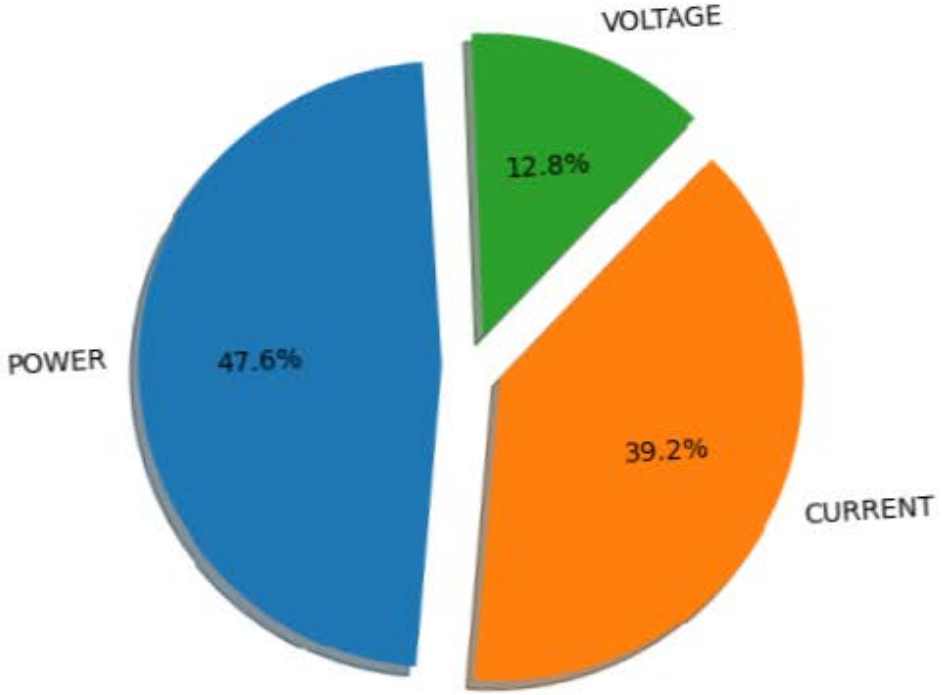


Figure 5.14. CPU Intensive Process Feature Importance by Sensor Type

slightly better than the other four as seen in table 5.6.

This test process demonstrated a situation where the target process heavily utilizes a system resource which is also heavily utilized by other processes on the system. This test demonstrated the viability of utilizing the new detection models for target processes which fit this category.

5.4.3. Network I/O Process

With this process we were aiming to create a significant amount of network traffic on the system to determine if the new method can detect a process which generates heavy network traffic with an additional random amount of CPU usage occurring at the same time. The actual process we ran utilized the network discovery and testing tool NMAP to carry out a full port scan of every IP address in the subdomain of the test systems very quickly. The actual process we ran four consecutive times each test cycle and is as follows

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

The "-Pn" option indicates that the desired test is a port scan which does not perform host discovery and instead treats all hosts as if they were up. The "-T4" option indicates a desired timing template with a value ranging from 0 to 5. Lower values represent slower performance while higher values represent faster performance. In this instance the timing template is only one position from being the fastest and thus will generate a large amount of network traffic very quickly. This command required roughly 5 minutes to complete on the Mac Mini systems.

5.4.3.1. Test Results

The results for the network I/O process unknown additional load tests can be seen in table 5.7. The data shows that the performance is significantly lower than either of the previously tested processes. The highest performing machine learning algorithm for the

advanced random load test was k-nearest neighbor (KNN). Using the KNN algorithm all instances of the target process were detected and in an average time of only 1.95 seconds. However, the fallout for this detector was 0.472 which makes implementation unlikely as it would result in far too many false positives for most applications. This same trend can be seen with the other machine learning algorithms tested.

Figure 5.15 shows that the DPS for the single model detector trained with KNN does perform well as a rapid process detector during a small number of tests. However, the majority of test instances result in a poor DPS of 0.5 or lower. As previously noted, the single model detector trained with KNN caught all target process instances during testing in an average time of only 1.95 seconds, but the fallout rate throughout testing averaged 0.472 with some test instances having a fallout rate over 0.9. The erratic and high levels of fallout during testing can be seen on the far right plot. Due to the extremely high fallout rate the MCC scores during testing stayed around 0 which means that strictly as a binary classification model the single model detector trained with KNN is about as effective as random selection.

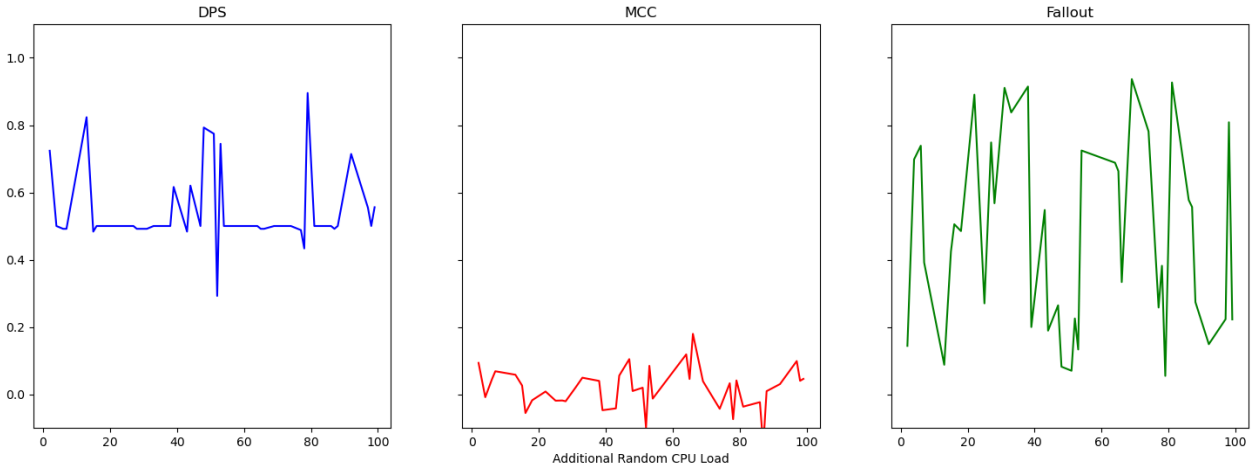


Figure 5.15. Network I/O Process Binary Classification Metrics - Single Detector Trained With KNN

The timeseries plots in figure 5.16 would indicate that the detector is making 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.

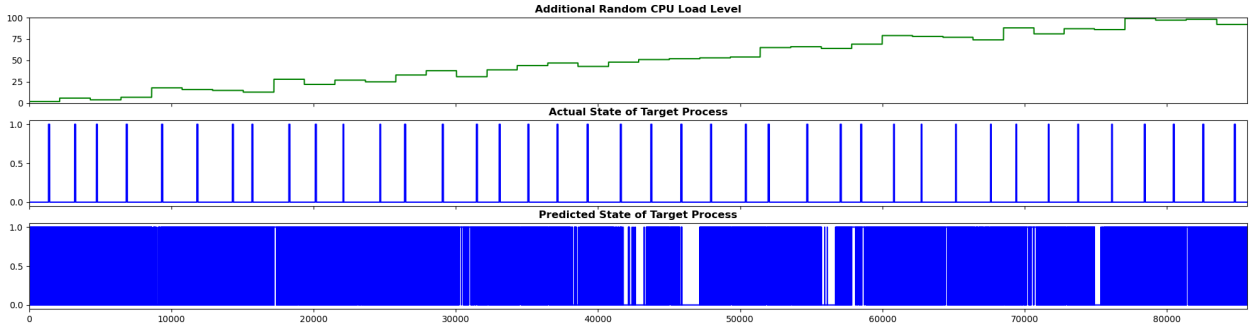


Figure 5.16. Network I/O Process Prediction Time Series - Single Detector Trained With KNN

While the main takeaway from figure 5.16 is correct is should be noted that due to the large number of prediction time intervals in the data the plot can appear worse than it actually is. Instances with false positives will require at least one pixel to visual represent, but that one pixel actually appears to cover many time intervals. Figure 5.17 shows a small section of the timeseries plot in figure 5.16. only 2.5% of the time intervals in the network I/O process test data set are displayed in a plot with the same number of pixels for visualization as figure 5.16. The detector is clearly not making true predictions at almost every time interval, but it does appear to be functionally no better than random classification.

Reviewing the feature importance scores in table 5.8 show that there are no features representing sensors which directly measure the physical state of the network interface. Instead, there is a fairly even mix of features which represent sensors that measure system components which are highly unlikely to undergo a significant enough physical change to be of any use in process detection.

The distribution of feature importance based on the system components being measured

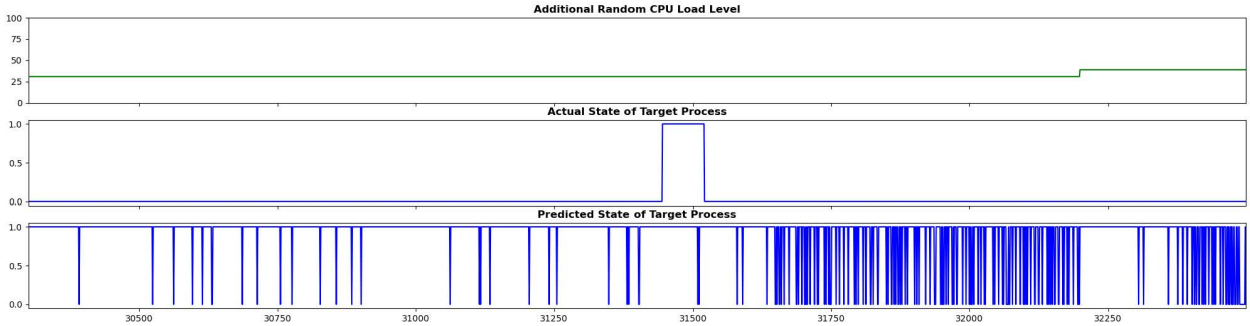


Figure 5.17. Network I/O Process Prediction Time Series Zoomed- Single Detector Trained With KNN

is shown in figure 5.18. It can be seen that the system component with the highest amount of feature importance is system memory which is not heavily utilized in the network I/O test process. Furthermore, the hard drive and the system air temperature also comprise a large amount of the feature importance. Clearly there would be very little physical change in the hard drive or the temperature of the air inside the system as a result of sending a large number of network probes over a short amount of time.

Perhaps the most simplistic way to make the determination that the rapid detector models are ineffective with this process on this system can be seen in figure 5.19. The amount of feature importance assigned to features representing temperature sensors is 44.9%. This indicates that the detection ability of the models in this system are relying heavily on temperature changes which are far slower and far less accurate than current, power, and voltage sensors. This is likely because there is no features which directly measure the network interface and the model must instead use sensors which measure the entire system such as ambient air temperature.

Table 5.9 shows the highest performing algorithms used in models for each detector type. The DPS for all five detector types in about the same with some having a higher fallout rate and a faster detection rate and others having a lower fallout rate and a slower detection rate.

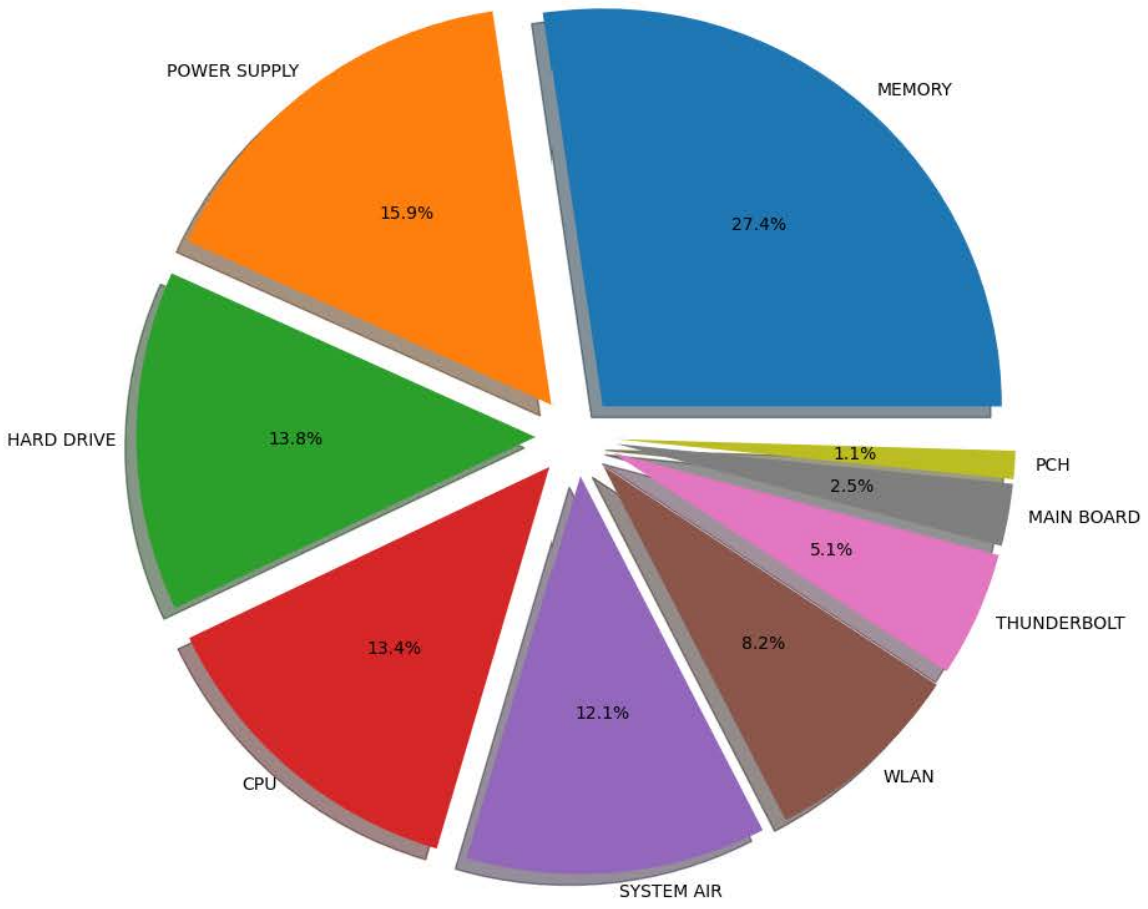


Figure 5.18. Network I/O Process Feature Importance by System Component

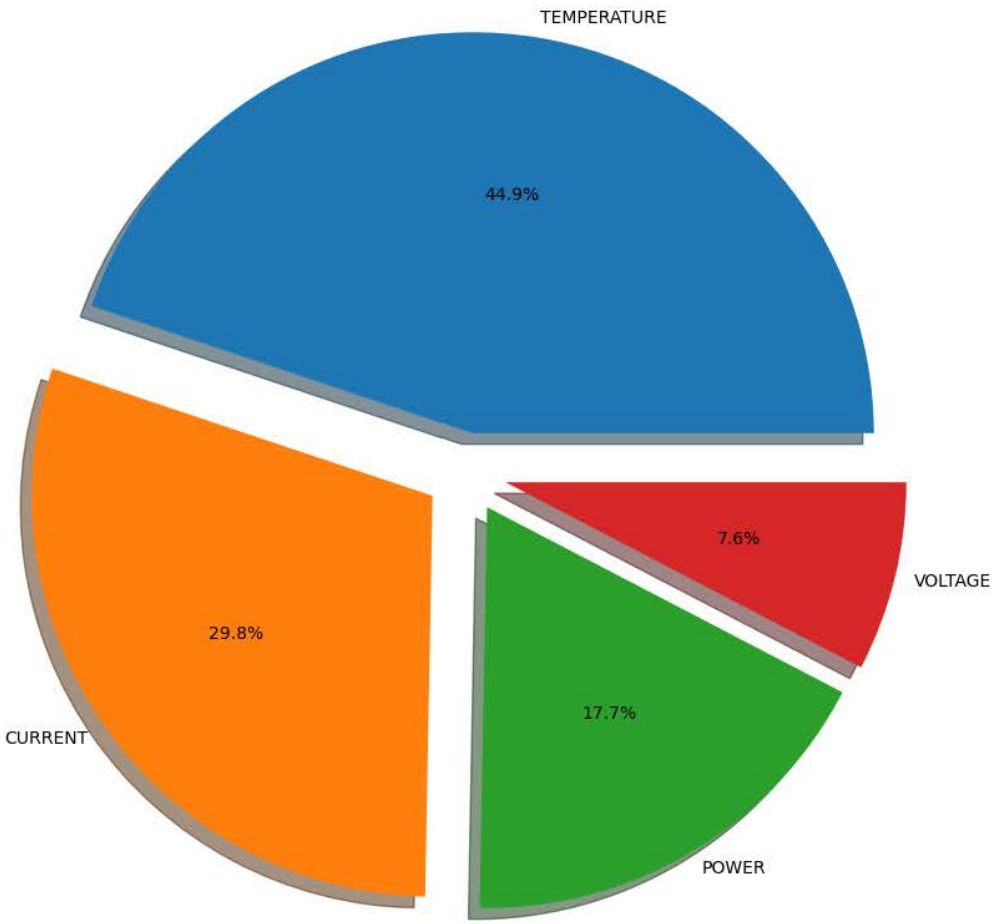


Figure 5.19. Network I/O Process Feature Importance by Sensor Type

In both instances the models do not perform well enough to be realistically used as a rapid process detector.

This test process demonstrates a situation where the target process predominantly utilizes a system resource which does not have a means of direct measurement through the system's physical sensors. This test demonstrated the limited viability of utilizing the new detection models for target processes which fit this category.

5.5. Summary and Contributions

This experiment demonstrated that the new ransomware detection model presented in the experiment in chapter 4 is a viable detector for arbitrary processes. The new detection model exhibited a high level of performance when trained to detect processes which predominately utilized resources which were specifically monitored by a subset of the system's sensors. We believe that as system complexity inevitably increases we are also likely to see an increase in physical system sensors in order to monitor and regulate operation. The need to monitor system components with a higher level of resolution will almost certainly increase the performance of the new detection model and allow for additional use cases.

We can also see from tables 5.3, 5.6 and 5.9 that the single prediction model detector outperformed the ensemble detectors for all 3 target processes. Previous experiments with the new detection model in chapter 4found the ensemble detector outperformed the single prediction model detector when the target process was more complex and utilized a wider variety of system resources (Taylor et al., 2021). It is reasonable to infer that there is a tradeoff point at which the complexity of the process causes the more complex detector models to overtake simpler detector models in performance. We plan to further experiment with the new detection model and investigate criteria for making a determination about which type of detection model best suits a target process.

Table 5.1. File I/O Process Unknown Additional Load Test Results

	Algorithm	**DPS**	**MCC**	**RPD**	**Fallout**	**TPD**
Zero	SVC	0.9952	0.9981	1.0	0.0003	0.5
	KNN	0.9949	0.9963	1.0	0.0005	0.5
	Random Forest	0.9942	0.9979	1.0	0.0003	0.625
	MLP	0.9942	0.9941	1.0	0.001	0.4583
	Logistic Regression	0.994	0.9952	1.0	0.0013	0.4167
	Extra Tree	0.9931	0.9938	1.0	0.0011	0.5833
	Naive Bayes	0.99	0.9916	1.0	0.0009	1.0
	Decision Tree	0.9869	0.989	1.0	0.003	0.875
Simple Random	SVC	0.996	0.9955	1.0	0.0013	0.175
	Random Forest	0.9951	0.9943	1.0	0.0017	0.2
	Extra Tree	0.9944	0.9919	1.0	0.0019	0.25
	Logistic Regression	0.9926	0.9899	1.0	0.0032	0.15
	Naive Bayes	0.9897	0.9848	1.0	0.0042	0.275
	KNN	0.9865	0.9809	1.0	0.0064	0.15
	MLP	0.9857	0.9777	1.0	0.0069	0.125
	Decision Tree	0.9001	0.8785	1.0	0.051	0.225
Advanced Random	Logistic Regression	0.9968	0.9982	1.0	0.0005	0.275
	Random Forest	0.9964	0.9984	1.0	0.0004	0.35
	KNN	0.9956	0.9972	1.0	0.0008	0.35
	Extra Tree	0.9952	0.995	1.0	0.001	0.35
	SVC	0.9949	0.9948	1.0	0.0014	0.275
	Naive Bayes	0.9948	0.9946	1.0	0.0011	0.375
	Decision Tree	0.9921	0.9963	1.0	0.0007	0.775
	MLP	0.538	0.1692	1.0	0.8151	0.0667

Table 5.2. File I/O Process Top Feature Importance Scores

Feature	**Importance Score**
CPU SUPPLY 1 POWER	0.16692266735584263
CPU SUPPLY 1 VOLTAGE	0.15222755254576856
CPU SUPPLY 1 CURRENT	0.1403713089595106
HARD DRIVE POWER	0.1063958593775839
PCH 1.05V LINE CURRENT	0.10570759090994092
1.8V S3 LINE CURRENT	0.06837460029350595
1.8V S3 LINE POWER	0.044147384145957785
HARD DRIVE CURRENT	0.0418151518970547
5V LINE VOLTAGE	0.03387261408253847
CPU A VOLTAGE	0.02743776022050085
DIMM 1.5V S3 LINE CURRENT	0.026168232938537397
DIMM 1.5V S3 LINE POWER	0.022893870276736995
SSD 3.3V S0 LINE CURRENT	0.021663835991360663
DDR3 MEMORY 1.35V LINE POWER	0.012148017781998794
TOTAL SYSTEM SUPPLY POWER	0.00983400662100652
DDR3 MEMORY 1.35V LINE VOLTAGE	0.009308429505969626
DC INPUT CURRENT	0.0071807934059072666
DDR3 MEMORY 1.35V LINE CURRENT	0.0025460589388404233
PLATFORM CONTROLLER HUB CHIP TEMPERATU	0.0006553833125740882

Table 5.3. File I/O Process Detector Type Comparison

Detector	**Top Algorithm**	**DPS**	**MCC**	**RPD**	**Fallout**	**TPD**
Single	Logistic Regression	0.9968	0.9982	1.0	0.0005	0.275
Ensemble 6 Single	Logistic Regression	0.9967	0.998	1.0	0.0005	0.275
Ensemble 11 Single	Extra Tree	0.9944	0.9937	1.0	0.0014	0.35
Ensemble 6 Dual	Logistic Regression	0.9967	0.998	1.0	0.0005	0.275
Ensemble 11 Dual	Extra Tree	0.9944	0.9937	1.0	0.0014	0.35

Table 5.4. CPU Intensive Process Unknown Additional Load Test Results

	Algorithm	**DPS**	**MCC**	**RPD**	**Fallout**	**TPD**
Zero	SVC	0.9987	0.9989	1.0	0.0001	0.125
	KNN	0.9987	0.9988	1.0	0.0001	0.125
	Logistic Regression	0.9987	0.9987	1.0	0.0002	0.125
	Decision Tree	0.9985	0.9978	1.0	0.0003	0.125
	Random Forest	0.9984	0.9981	1.0	0.0003	0.125
	MLP	0.9977	0.9932	1.0	0.0007	0.125
	Extra Tree	0.9976	0.993	1.0	0.0007	0.125
	Naive Bayes	0.9951	0.9912	1.0	0.0006	0.4583
Simple Random	Decision Tree	0.9988	0.9987	1.0	0.0002	0.1
	KNN	0.9987	0.999	1.0	0.0001	0.125
	Logistic Regression	0.9986	0.9991	1.0	0.0002	0.125
	SVC	0.9983	0.9989	1.0	0.0002	0.15
	Random Forest	0.9983	0.9983	1.0	0.0004	0.1
	Extra Tree	0.9974	0.9951	1.0	0.0007	0.15
	MLP	0.9972	0.9944	1.0	0.001	0.1
	Naive Bayes	0.9267	0.8958	1.0	0.0844	0.25
Advanced Random	Random Forest	0.989	0.7878	1.0	0.0002	1.275
	Decision Tree	0.9875	0.7613	1.0	0.0004	1.4
	Extra Tree	0.9861	0.5901	1.0	0.0005	1.55
	KNN	0.9776	0.6876	1.0	0.0008	2.5
	Naive Bayes	0.7896	0.6577	1.0	0.3329	0.6
	MLP	0.5484	0.4717	0.55	0.0003	0.2273
	Logistic Regression	0.4988	0.4756	0.5	0.0001	0.25
	SVC	0.4738	0.4738	0.475	0.0001	0.2632

Table 5.5. CPU Intensive Process Top Feature Importance Scores

Feature	**Importance Score**
DIMM 1.5V S3 LINE CURRENT	0.13084167347811235
1.8V S3 LINE CURRENT	0.12690068235864524
DDR3 MEMORY 1.35V LINE VOLTAGE	0.11755596775437337
DDR3 MEMORY 1.35V LINE POWER	0.09879153804845998
DIMM 1.5V S3 LINE POWER	0.06939175162219618
DDR3 MEMORY 1.35V LINE CURRENT	0.06919840988098663
CPU HIGH SIDE POWER	0.06389207490906618
1.8V S3 LINE POWER	0.06113221533693672
CPU A POWER	0.055223508428859154
CPU PACKAGE CORE POWER	0.04625019647991447
DC INPUT POWER	0.04304485357736412
CPU A CURRENT	0.030399367834582476
CPU PACKAGE TOTAL POWER	0.01859136828060757
TOTAL SYSTEM SUPPLY POWER	0.01768053858166743
DC INPUT CURRENT	0.01738988585543115
CPU HIGH SIDE CURRENT	0.01638827564124705
CPU A VOLTAGE	0.010221401493927125
CPU PACKAGE GPU POWER	0.0022629476485113344
CPU CORE 1 TEMPERATURE	0.0018502909317993422
PLATFORM CONTROLLER HUB CHIP TEMPERATU	0.0012879949166906335
SSD 3.3V S0 LINE CURRENT	0.0007286477722697663
MLB AMBIENT TEMPERATURE TEMPERATURE	0.0007059067646638107
WLAN CARD TEMPERATURE.1	0.000143849698768454
MAIN LOGIC BOARD TEMPERATURE	0.00011531944275823139

Table 5.6. CPU Intensive Process Detector Type Comparison

Detector	**Top Algorithm**	**DPS**	**MCC**	**RPD**	**Fallout**	**TPD**
Single	Random Forest	0.989	0.7878	1.0	0.0002	1.275
Ensemble 6 Single	Naive Bayes	0.9781	0.7271	1.0	0.002	2.15
Ensemble 11 Single	Naive Bayes	0.9836	0.7835	1.0	0.004	1.05
Ensemble 6 Dual	Decision Tree	0.988	0.7524	1.0	0.0002	1.4
Ensemble 11 Dual	Extra Tree	0.9854	0.5758	1.0	0.0004	1.65

Table 5.7. Network I/O Process Unknown Additional Load Test Results

	Algorithm	**DPS**	**MCC**	**RPD**	**Fallout**	**TPD**
Zero	SVC	0.9843	0.7535	1.0	0.0057	0.5833
	MLP	0.9802	0.6261	1.0	0.0069	0.7917
	KNN	0.9052	0.1949	1.0	0.0413	1.8333
	Logistic Regression	0.9005	0.1199	1.0	0.0059	10.5833
	Decision Tree	0.8339	0.3383	1.0	0.088	0.5417
	Extra Tree	0.7173	0.1485	0.8333	0.0148	12.9
	Naive Bayes	0.0	-0.002	0.0	0.0009	nan
	Random Forest	0.0	-0.0014	0.0	0.0006	nan
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
	Logistic Regression	0.7568	0.2759	0.875	0.0492	4.0571
	Extra Tree	0.7296	0.1318	0.975	0.1662	3.3333
	Decision Tree	0.6966	0.1754	1.0	0.1725	1.8
	Naive Bayes	0.4965	0.0479	0.775	0.4243	2.8065
	Random Forest	0.4318	0.1693	0.5	0.0327	2.9
Advanced Random	KNN	0.5484	0.017	1.0	0.472	1.95
	Decision Tree	0.5114	0.0114	1.0	0.4957	3.7
	Logistic Regression	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
	Extra Tree	0.3904	-0.0756	0.775	0.5175	5.129
	Naive Bayes	0.3625	-0.0337	0.725	0.7077	0.0
	Random Forest	0.3354	0.0135	0.575	0.3946	2.7826

Table 5.8. Network I/O Process Top Feature Importance Scores

Feature	**Importance Score**
SSD 3.3V S0 LINE CURRENT	0.1382299805992649
DIMM 1.5V S3 LINE CURRENT	0.06578903979426579
MEMORY SLOTS TEMPERATURE	0.06281660320171678
AMBIENT AIR POSITION 3 TEMPERATURE	0.05920182751612403
1.8V S3 LINE CURRENT	0.05719195405054438
THUNDERBOLT TEMPERATURE	0.05118092347087696
1.8V S3 LINE POWER	0.05068674597877819
WLAN CARD TEMPERATURE	0.04086890755016818
WLAN CARD TEMPERATURE.1	0.04062318834921384
DIMM 1.5V S3 LINE POWER	0.0381753411378859
MEMORY BANK A POS 1 TEMPERATURE	0.036491722938217785
CPU A VOLTAGE	0.03580001958339358
AMBIENT AIR TEMPERATURE	0.030837969990001813
AMBIENT AIR POSITION 2 TEMPERATURE	0.02984843808351909
CPU A PROXIMITY TEMPERATURE.1	0.027565928006298066
DDR3 MEMORY 1.35V LINE VOLTAGE	0.027442260008739096
MAIN LOGIC BOARD TEMPERATURE	0.025309164013207368
CPU A PROXIMITY TEMPERATURE	0.02514906413286043
DDR3 MEMORY 1.35V LINE CURRENT	0.021693161396651662
DDR3 MEMORY 1.35V LINE POWER	0.0211093104123621
DC INPUT POWER	0.018000880771544697
CPU A POWER	0.01223523877093388
PLATFORM CONTROLLER HUB CHIP TEMPERATU	0.01140814018034716
1.8V S3 LINE VOLTAGE	0.011208493058368352

Table 5.9. Network I/O Process Detector Type Comparison

Detector	**Top Algorithm**	**DPS**	**MCC**	**RPD**	**Fallout**	**TPD**
Single	KNN	0.5484	0.017	1.0	0.472	1.95
Ensemble 6 Single	Random Forest	0.506	-0.0119	0.975	0.8284	0.2308
Ensemble 11 Single	Logistic Regression	0.5444	0.0076	0.95	0.5489	4.9211
Ensemble 6 Dual	Random Forest	0.5062	-0.0116	0.975	0.8375	0.2051
Ensemble 11 Dual	Random Forest	0.5401	-0.0053	1.0	0.8598	0.3

Chapter 6

Virtualization Detection

Virtualization allows businesses to utilize technological infrastructure without the need to purchase, install, or maintain anything themselves. Cloud computing services such as Amazon Web Services (AWS) and Microsoft Azure will process 94% of all enterprise workloads in 2021. Software as a Service (SaaS) processes are projected to account for 75% of the total workload (Sumina and Ivey, 2021). When using cloud services, security is a concern for many businesses as the cloud service provider must be trusted to stay up to date and vigilant against attacks. It was found in a survey that 75% of businesses consider security a top concern when moving to the cloud (Freeze, 2021). Virtualization obfuscates and isolates virtual entities from each other and the host system. However, although the virtual entities and the host are all isolated and obfuscated from each other they share the same physical hardware.

When considering the work done in chapters 4 and 5 it has been shown that the physical state of a system can be measured with system sensors. Despite the obfuscation and isolation inherent in virtualization the operation of the virtual system still effects the physical system in a way which makes detection of virtual activity possible. Hosts would then have the ability to train detection models using the system's physical sensors to detect specific activity of interest being carried out on the virtual entities which it is hosting.

Recently cyber attackers have been discovered to be launching ransomware attacks through virtual machines in order to evade existing system security methods. Perhaps the most relevant aspect of such attacks is instances where an attacker on a virtual machine will create a ransomware payload in a shared folder which effectively allows evasion of security monitors

and easy system penetration. Accounting for bad actors in a cloud environment means host systems need to be able to identify potentially malicious activity from its guests systems while ensuring that the guest's isolation and privacy are not compromised in the process. Using physical sensor data in order to detect target activity would allow a host to accomplish this task.

6.1. Training and Testing Detection Models

This experiment utilizes the same test environment as described in section 5.2.1. Rather than having multiple different process data sets this experiment utilized only a single data set with multiple binary flags indicating when virtual activity occur ed. The data was collected in the same manner outlined in section 5.2.3. During the data collection process a virtual machine was launched, ran a simulated user script, and then shutdown at the time instances where target processes were run in chapter 5. Additionally, this experiment built and tested the same machine learning algorithms and detection model types outlined in the experiment in chapter 5.

6.2. Implementation of Target Activity for Detection

The target activity implementation aimed to launch a virtual machine using a type 2 hypervisor with a simulated user load running on the virtual machine and an additional random amount of CPU usage occurring at the same time on the host system. We utilized VMware Fusion 10 as the type 2 hypervisor running on the Mac Minis outlined in section 5.2.1. VMware Fusion includes a command line tool called "vmrun" which allows a virtual machine to be controlled and interacted with on the host system through the command line (vmw). We implemented commands to start and display, run a Bash script, and shutdown and close a specific virtual machine. The virtual machine used in the testing ran Xubuntu which is an Ubuntu variant that uses the XFCE desktop environment which is lighter weight and requires fewer resources. The virtual machine was allocated one of the Mac Mini's

processing cores, 2 GB of memory out of a total of 4 GB, and a pre-allocated virtual hard drive of 100GB. The command used to launch and display the virtual machine is as follows

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

The vmx file path is for the file that acts as the configuration for the target virtual machine. The last command line argument "gui" means the virtual machine should be shown on the screen once launched. The virtual machine was set to automatically log on without a password. Fully starting up the virtual machine required roughly 2 minutes. We then passed a command to the virtual machine which included a path to the Bash interpreter and a path to a script that we created to add a simulated user load to the virtual machine. The command used to run the bash script is as follows

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

The "-gu" option means the username for accessing the virtual machine is the following command line argument. Additionally, the "-gp" option means the following command line argument is the password for logging in to the virtual machine with the username. The vmx file path is for the file that acts the configuration for the target virtual machine. The option "-interactive" is required to run a script on the virtual machine to avoid warnings from the operating system. The last argument is the command to be run on the guest virtual machine command line. The Bash script usersim utilized the stress test suite Stress-ng to create three stressors that run for the desired amount of time on the guest virtual machine (ColinIanKing). The script creates one CPU stressor, one mixed I/O stresser, and one hard drive stressor. The CPU stressor performs matrix multiplication and is set to utilize 30% of the available CPU resources. The mixed I/O stressor performers randomly selected types of I/O operations, and the hard drive stressor continuously reads and writes from the hard drive. The stressors were run to simulate the load of a user on the virtual machine. The usersim script is run for 5 minutes each time after the virtual machine has fully started.

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

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

The vmx file path is for the file that acts as the configuration for the target virtual machine. The last command line argument "soft" means the virtual machine should be shut down using the system's shutdown script. Starting the virtual machine, running the usersim script, and shutting down the virtual machine requires roughly 7 minutes.

Once the start command is run for the virtual machine a flag is set in the data to indicate that the virtual machine is operational. This flag stays set until the stop command is run. During this period of time the command to run the user simulation script on the virtual machine is executed which causes a second flag to be set indicating that the simulated user script is operational. This flag stays set until the simulated user script completes on the virtual machine. This creates two possible events to detect from the data. The first event to detect is when the virtual machine is launched. Afterwards, several minutes go by until the second event to detect, the simulated user script, is run on the virtual machine. The detection model trained to detect the virtual machine launch and operation would ideally make positive predictions from the time the virtual machine starts to the time it stops. The potential complication could come from the increased usage of allocated resources to the virtual machine once the simulated user script begins running. The detection model trained to detect the simulated user script running on the virtual machine would ideally make positive predictions from the time the simulated user script began running until the time it completed. It is possible that the detection model can simply detect the instances when the virtual machine is running which includes all instances of when the simulated user script is running. However, if this occurs the false positive predictions prior to every instance of the simulated user script would have a noticeable negative effect on the detector performance metrics.

6.3. Experimental Results

6.3.1. Virtual Machine Running on a Host

6.3.1.1. Test Results

The results for detection of a virtual machine running on a host can be seen in table 6.1. Both the simple random load test and the advanced random load test resulted in a DPS higher than 0.9 for all machine learning algorithms except for one. The highest performing machine learning algorithm was KNN as it had the highest DPS for the advanced random load test and the second highest DPS for the simple random load test. The detection model which used KNN detected all instances of the target process during the advanced random load test in an average time of 0.425 seconds while maintaining an average fallout of only 0.0065.

Figure 6.1 shows the DPS, MCC, and fallout rate for all additional random load levels used during testing. The DPS plot on the left shows that the new detection model has a very high level of performance with only a very minor drop at higher additional random load levels. This can also be seen in the plot on the right which shows the fallout rate which remains very close to zero with only a minor increase at the higher additional random load levels. The middle graph shows that the MCC scores were only around 0.6 which indicates that while this model performs well as a rapid detector it is only a moderately effective binary classification model.

The reason for the lower MCC scores can easily be seen in figure 6.2 which shows that while the virtual machine being launched was detected with success there also exists a noticeable amount of false positives. However, as outlined in 5.4.3.1, the timeseries graphs can appear to be worse than they actually are. This is due to there being 40 hours of data with a prediction occuring every second. This can cause false positives to appear much worse on the timeseries as one pixel is actually the size of many prediction instances.

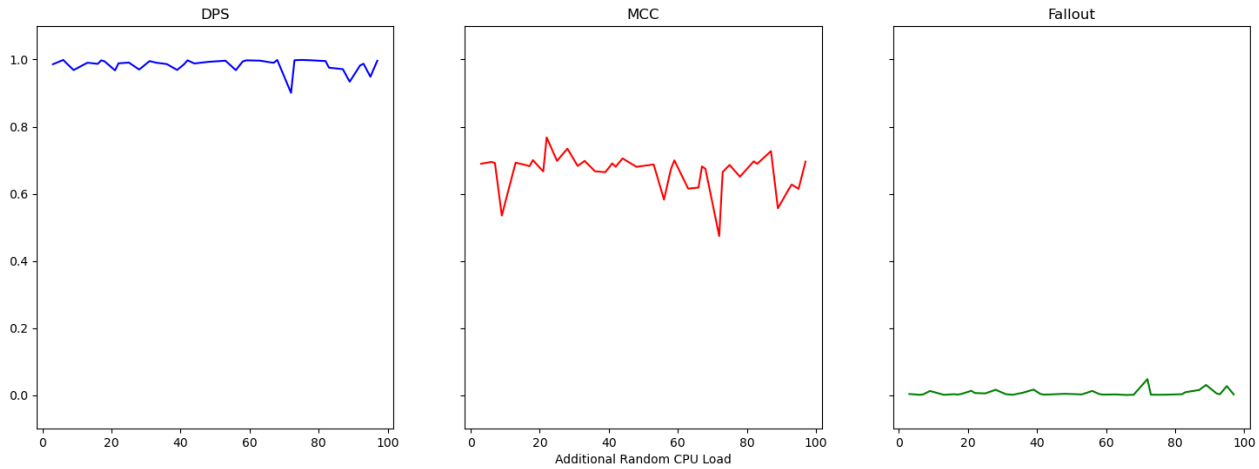


Figure 6.1. Virtual Machine Running Binary Classification Metrics - Single Detector Trained With KNN

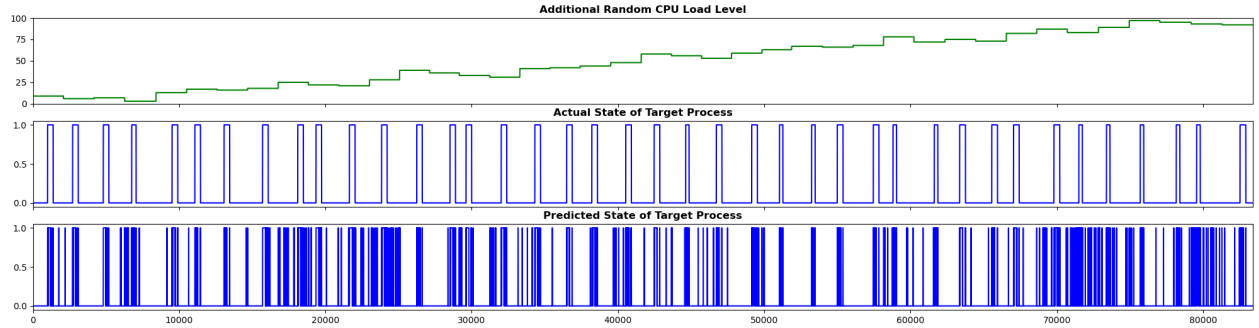


Figure 6.2. Virtual Machine Running Prediction Time Series - Single Detector Trained With KNN

Table 6.2 shows the mean decrease in impurity (MDI) feature importance scores from the single model detector trained with the random forest algorithm. The top 3 features are all sensors that measure the CPU. This is worth noting since the CPU is also having a random additional load placed on it during testing but the physical changes in the CPU due to the virtual machine being launched would seem to be distinct enough that it can be observed despite the "noise" of the additional load.

Figure 6.3 shows that 51.7% of feature importance is assigned to features which represent sensors that measure the CPU. It is also interesting to note that features representing sensors that measure the system memory were assigned 32.0% of feature importance. It would seem from the feature importance scores that the detection model is able to identify when the virtual machine is launched and running by the power fluctuations caused by the CPU and memory resources allocated to the virtual machine.

Figure 6.4 shows an almost evenly balanced distribution of feature importance between features representing sensors that measure power, current, and voltage.

Table 6.3 shows the top performing algorithm for each of the 5 detector model types. It would seem that as the detectors become more complex their performance diminishes. Considering what was found during the feature analysis this would seem to suggest that when a virtual machine is launched and is running on a system there exists a relatively simple but distinguishable pattern of physical changes to the system. Virtual machines are allocated specific amounts of system resources which are often left unchanged by a user which could explain the distinguishable signature on the system.

The new detection model was shown to be not only effective in detecting the launch and operation of a virtual machine, but was also able to perform detection without the need for the more complex ensemble detection models. This seems to suggest that virtual operations which have a regularly allocated amount of the system resources for operation are good use cases for physical sensor based detection.

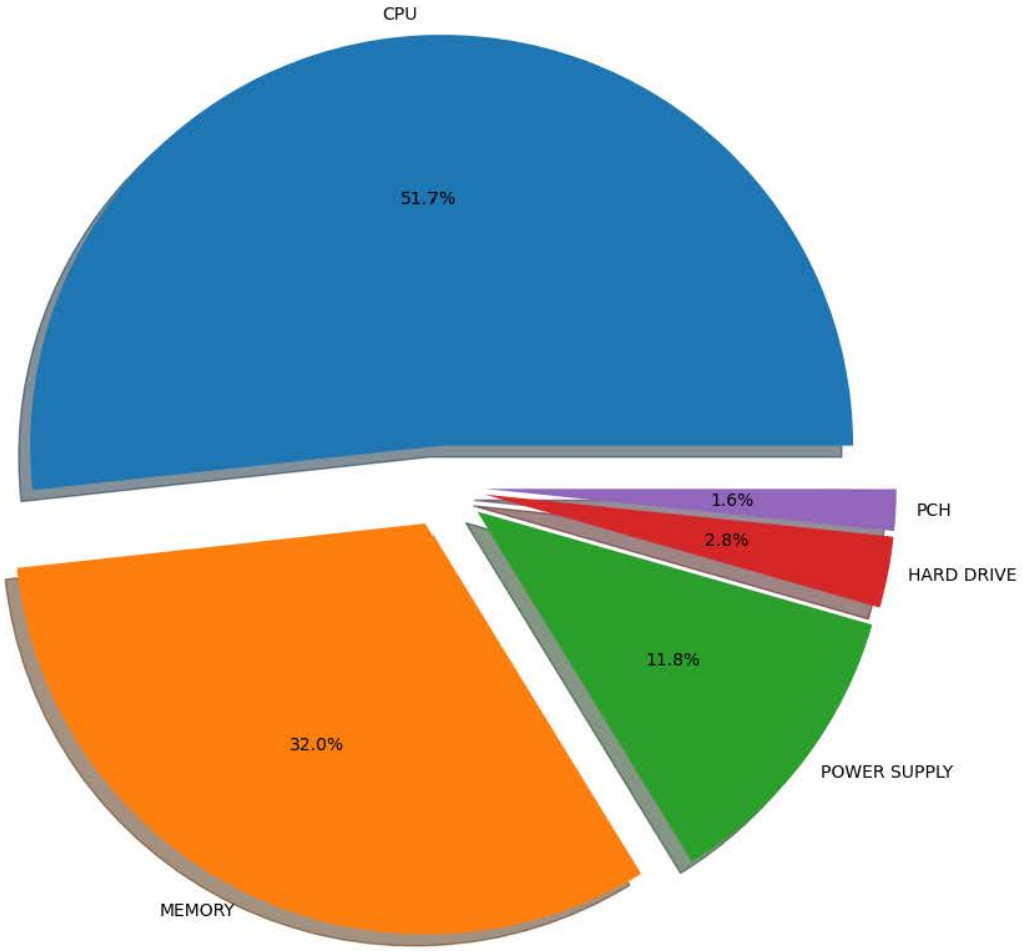


Figure 6.3. Virtual Machine Running Feature Importance by System Component

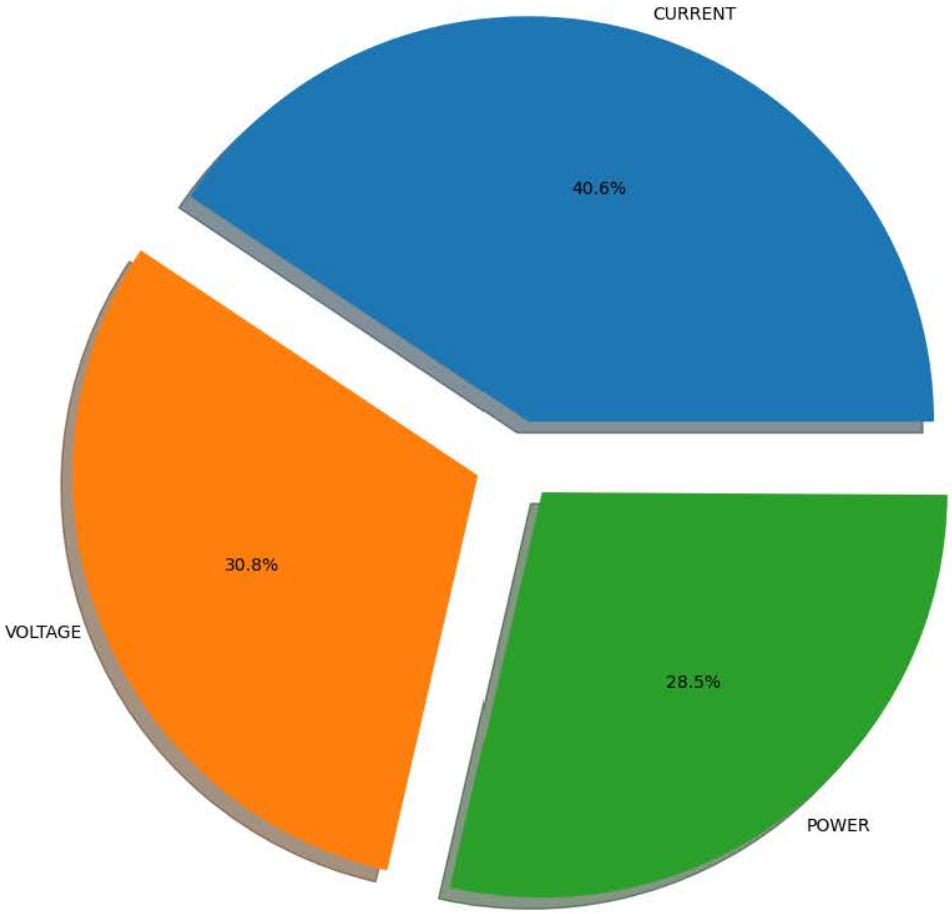


Figure 6.4. Virtual Machine Running Feature Importance by Sensor Type

6.3.2. Simulated User Script Running on Virtual Machine

6.3.2.1. Test Results

The results for detection of a simulated user script running on a virtual machine can be seen in table 6.4. The highest performing detection model when there was an unknown additional random CPU load present was trained with the random forest machine learning algorithm. The detection model trained with random forest had a DPS of 0.9682 for the advanced random load test and a DPS of 0.8054 for the simple random load test. What is interesting to note in this experiment was that the detection models increased in performance as the testing environment became more complex. In fact during the zero load and simple random load tests no detection model was able to detect all of the instances where the simulated user script was running on the virtual machine. However, during the advanced random load test 4 detection models were able to detect all of the simulated user script runs and of those 3 had a DPS above 0.9. Based on these results it would seem that as the system came under increased stress and resources were being utilized in very different and random ways the effect on the physical state of the system from the simulated user script running on the virtual machine became more pronounced and distinguishable.

Figure 6.5 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. However, 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.

Examining the timeseries plot of all tests conducted with the random forest detection model during the advanced random load test offers insight into the low MCC scores. As was the case with previous experiments the detection model appears to have difficulty in maintaining positive prediction during the duration of an instance in time where the simulated user load is running on the virtual machine. In terms of binary classification metrics

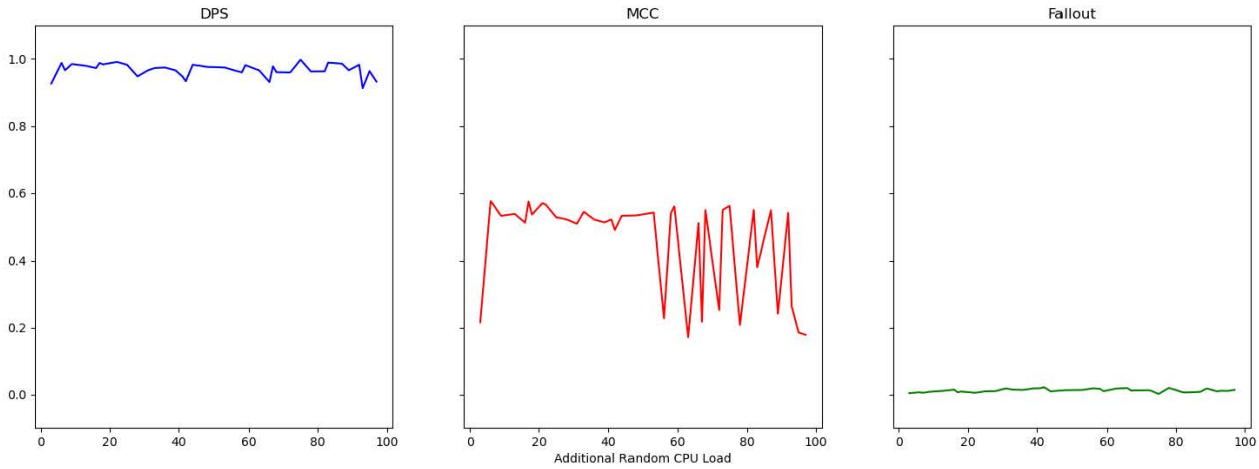


Figure 6.5. Simulated User Script Running on Virtual Machine Binary Classification Metrics - Single Detector Trained With Random Forest

the detection model maintains a low fallout rate and a level of specificity which are good indications it is a good rapid detection model. However, the detection model had an average miss rate of 0.6805 and an average sensitivity of only 0.3195. The high miss rate and low sensitivity is due to the detection model going back and forth between positive and negative predictions during instances of the simulated user script running on the virtual machine. In the context of a rapid detection model this is not a significant concern but in terms of a binary classification model this is an indication that it is not a strong predictor. Furthermore, when considering that the MCC metric accounts for disproportionate positive and negative system states during testing and balances the prediction ability for both evenly this causes the MCC to be drastically lower despite the detection model performing at a high level in the context of rapid detection.

Table 6.5 shows the top features for the random forest trained detection model as determined through MDI. It is notable that the top features are a mixture of CPU sensors, memory sensors, and hard drive sensors since the simulated user script on the virtual machine is running one CPU stressor, one I/O stressor, and one hard drive stressor.

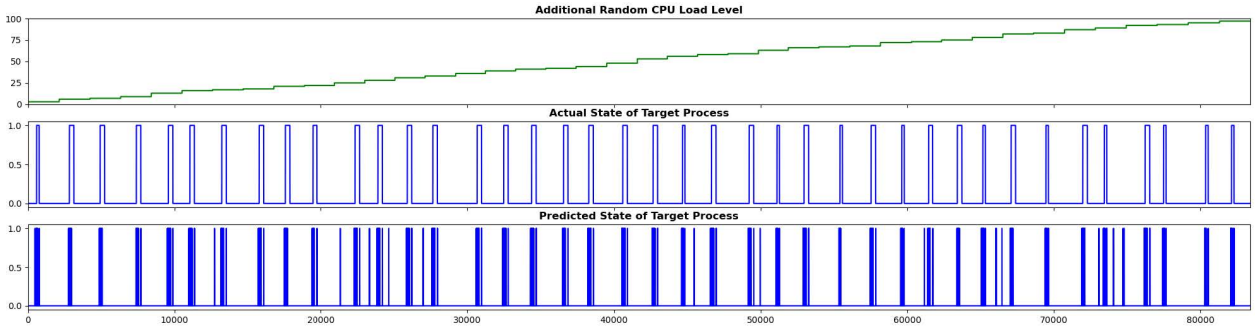


Figure 6.6. Simulated User Script Running on Virtual Machine Prediction Time Series - Single Detector Trained With Random Forest

Figure 6.7 shows the distribution of feature importance based on the system component measured by the features. It is worth noting that despite the CPU having random unknown loads it is still by far the most important system component for the detection model. This appears to indicate that when the physical system is under a large amount of stress and resources are not as readily available the virtual machine operations have a more pronounced physical effect on the system and in turn are more easily detected. This was also seen in the results from section 6.3.1.1.

Also similar to the results in section 6.3.1.1 is the distribution of feature importance based on the type of measurement performed by each sensor. Figure 6.8 shows that there is an almost evenly balanced distribution of feature importance between features representing sensors that measure power, current, and voltage. Although, in this experiment current measurements have a more pronounced importance at the expense of voltage measurements. It is likely this is due to the model being trained to detect the simulated user script running on the virtual machine which involves operations that cause changes in current that make it further distinguishable from the virtual machine itself simply running.

Table 6.6 shows that the single model detector had the highest DPS out of the five model type. However, it was also was the slowest where the more complex model types

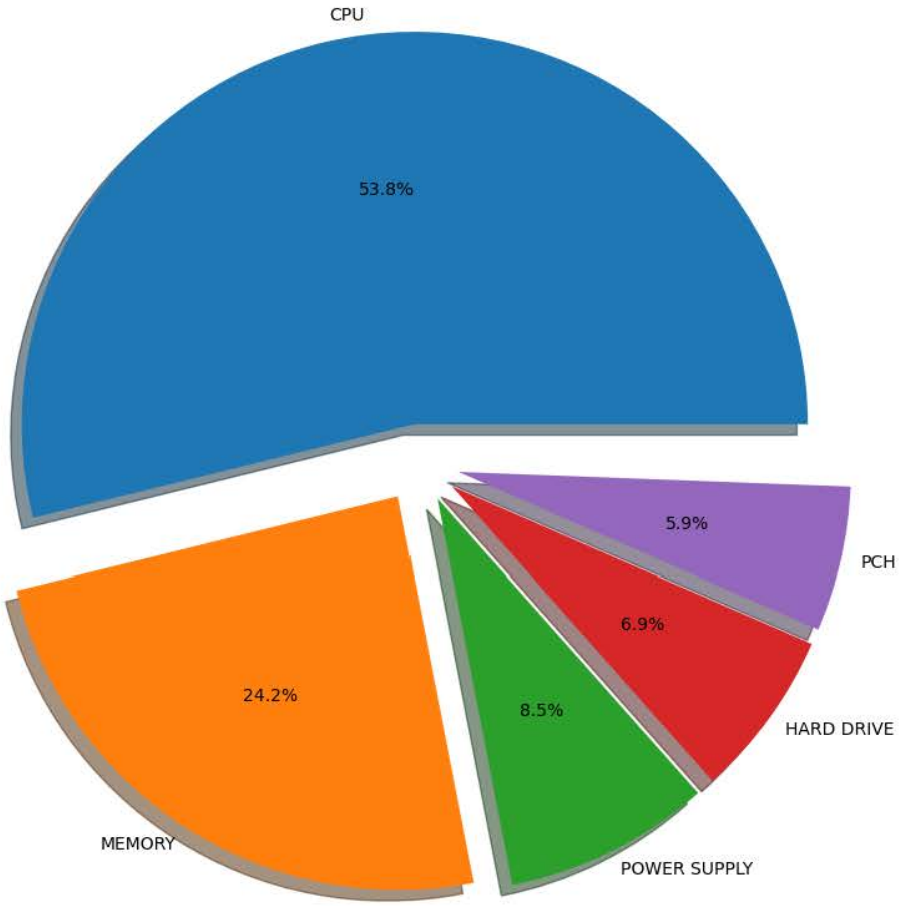


Figure 6.7. Simulated User Script Running on Virtual Machine Feature Importance by System Component

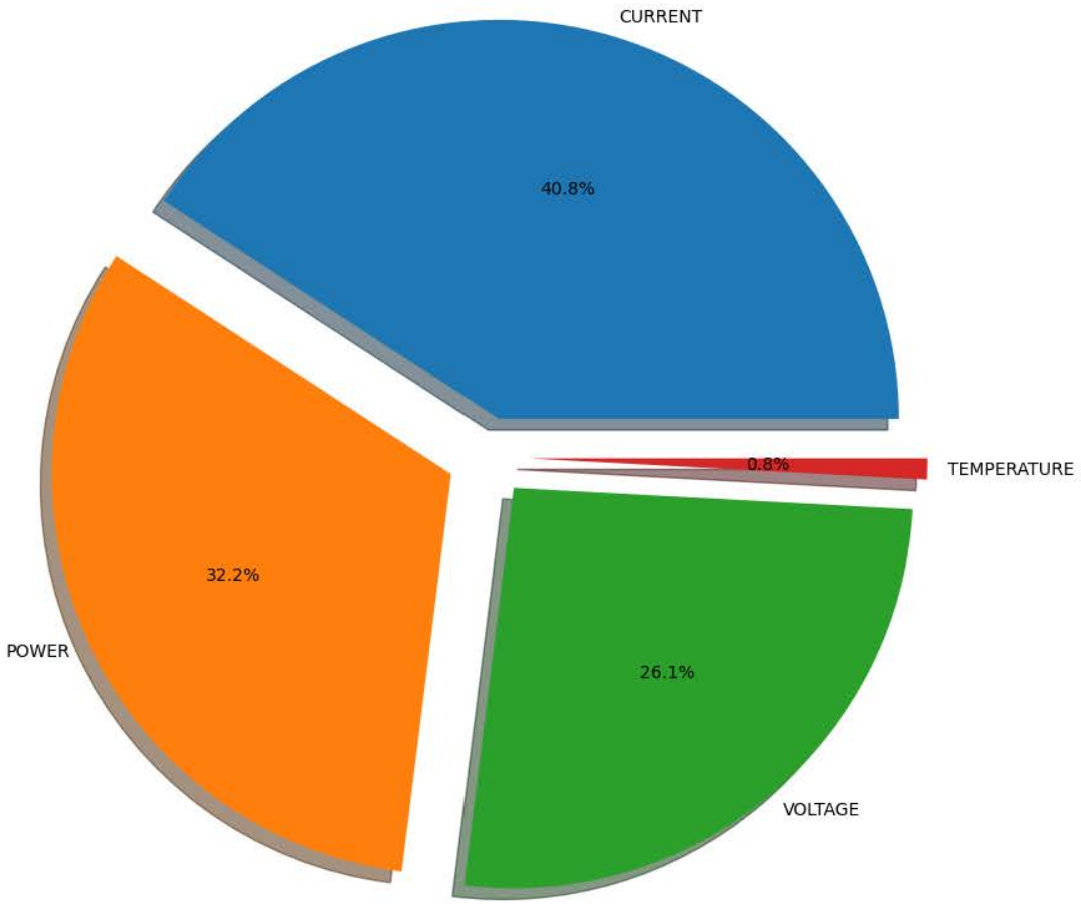


Figure 6.8. Simulated User Script Running on Virtual Machine Feature Importance by Sensor Type

were very fast with only a slightly high fallout rate. This is likely an indication that the complexity of the behavior being targeted by the detector is reaching a point where the more complex ensemble detection models are close to overtaking the more simplistic single model in performance.

6.4. Summary and Contributions

Virtual entities are obfuscated and isolated from each other as well as the host system causing traditional security methods which rely on behavioral analysis to be very ineffective (Palmer, 2021). However, it often overlooked that the physical state of the host system will change based on specific activity carried out by a hypervisor. In this experiment we were able to use this idea to expand the new detection model from chapters 4 and 5 to include activity carried out by a type two hypervisor from a host system. We were able to detect when a virtual machine was launched and run on a host. Additionally, we were able to detect when a specific process was run on the same virtual machine. It can be seen in figure 6.7 and 6.8 that the features used to detect these test cases were very similar as the actual test data set is the same. However, the features used for detecting the process run on the virtual machine, rather than simply the virtual machine running, do show importance scores which indicate the actual process run on the virtual machine causes a slightly more specific change to the physical system. While this same change being accounted for in the detection models used for detecting the virtual machine simply running it seems that the feature importance scores are altered due to the period prior to the process running when the virtual machine is launching. This experiment served as a proof of concept for virtualization detection and shows that further experiments and analysis in this area is warranted.

Table 6.1. Virtual Machine Running With Unknown Additional Load Test Results

	Algorithm	**DPS**	**MCC**	**RPD**	**Fallout**	**TPD**
Zero	Decision Tree	0.9898	0.9433	1.0	0.0039	0.3333
	KNN	0.9896	0.9661	1.0	0.0045	0.2083
	SVC	0.9889	0.9658	1.0	0.0048	0.2083
	Logistic Regression	0.9888	0.956	1.0	0.004	0.4167
	Extra Tree	0.9872	0.9528	1.0	0.0057	0.2083
	Random Forest	0.9856	0.8877	1.0	0.0039	0.8333
	MLP	0.9853	0.9462	1.0	0.0064	0.2917
	Naive Bayes	0.9829	0.5045	1.0	0.0022	1.5417
Simple Random	Logistic Regression	0.9896	0.9402	1.0	0.004	0.325
	KNN	0.9865	0.9375	1.0	0.006	0.225
	SVC	0.9863	0.9365	1.0	0.0061	0.225
	Random Forest	0.9753	0.884	1.0	0.0106	0.525
	Extra Tree	0.9721	0.8756	1.0	0.0136	0.2
	MLP	0.9609	0.8591	1.0	0.0194	0.225
	Decision Tree	0.9481	0.8228	1.0	0.0259	0.25
	Naive Bayes	0.8566	0.6393	1.0	0.1339	0.8
Advanced Random	KNN	0.9839	0.6658	1.0	0.0065	0.425
	SVC	0.9836	0.6735	1.0	0.0071	0.325
	Random Forest	0.9822	0.6697	1.0	0.0081	0.275
	MLP	0.9753	0.5257	1.0	0.0073	1.275
	Extra Tree	0.9566	0.5926	1.0	0.021	0.375
	Logistic Regression	0.9386	0.6506	1.0	0.037	0.425
	Naive Bayes	0.9029	0.5785	1.0	0.0772	0.3
	Decision Tree	0.7266	0.2298	1.0	0.2829	0.2

Table 6.2. Virtual Machine Running Top Feature Importance Scores

Feature	**Importance Score**
CPU SUPPLY 1 CURRENT	0.16209144360458197
CPU SUPPLY 1 VOLTAGE	0.13858467696758017
CPU SUPPLY 1 POWER	0.1373765892816713
1.8V S3 LINE CURRENT	0.09161934004015827
DDR3 MEMORY 1.35V LINE VOLTAGE	0.08926097645867695
CPU A VOLTAGE	0.06401910693608606
DIMM 1.5V S3 LINE POWER	0.05738645184415804
DIMM 1.5V S3 LINE CURRENT	0.05444884551848147
DDR3 MEMORY 1.35V LINE CURRENT	0.046280768340159505
DDR3 MEMORY 1.35V LINE POWER	0.042101770415796755
HARD DRIVE CURRENT	0.030777365863822632
HARD DRIVE POWER	0.02765657137570172
PCH 1.05V LINE CURRENT	0.0163088628394155
5V LINE VOLTAGE	0.015925282215279175
1.8V S3 LINE POWER	0.008401920710889925
CPU HIGH SIDE POWER	0.003642497614649063
CPU A CURRENT	0.0035535214212078785
CPU PACKAGE TOTAL POWER	0.002843679739648582
CPU A POWER	0.0023610812127552922
CPU PACKAGE GPU POWER	0.00174899038706063
TOTAL SYSTEM SUPPLY POWER	0.0012430376419228925

Table 6.3. Virtual Machine Running Detector Type Comparison

Detector	**Top Algorithm**	**DPS**	**MCC**	**RPD**	**Fallout**	**TPD**
Single	KNN	0.9839	0.6658	1.0	0.0065	0.425
Ensemble 6 Single	Logistic Regression	0.9724	0.5553	1.0	0.0064	1.825
Ensemble 11 Single	Logistic Regression	0.9766	0.6174	1.0	0.008	0.95
Ensemble 6 Dual	Logistic Regression	0.9544	0.6301	1.0	0.0183	1.25
Ensemble 11 Dual	Logistic Regression	0.9461	0.6743	1.0	0.0259	0.5

Table 6.4. Simulated User Script Running on Virtual Machine Unknown Additional Load Test Results

	Algorithm	**DPS**	**MCC**	**RPD**	**Fallout**	**TPD**
Zero	KNN	0.8195	0.8035	0.8421	0.0023	2.6875
	Decision Tree	0.8179	0.7612	0.8421	0.0047	2.375
	Naive Bayes	0.816	0.3127	0.8421	0.0024	3.1875
	MLP	0.8141	0.775	0.8421	0.003	3.25
	SVC	0.812	0.7981	0.8421	0.001	4.0625
	Extra Tree	0.8085	0.7804	0.8421	0.0017	4.375
	Logistic Regression	0.8063	0.7387	0.8421	0.0036	4.25
	Random Forest	0.7953	0.7138	0.8421	0.0012	6.375
Simple Random	Naive Bayes	0.9145	0.5076	0.9667	0.025	0.5517
	Random Forest	0.8054	0.5422	0.8333	0.0101	1.72
	KNN	0.75	0.5818	0.7667	0.0074	0.7826
	Decision Tree	0.6844	0.5495	0.7	0.0072	1.0476
	MLP	0.617	0.5489	0.6333	0.0056	1.7895
	Extra Tree	0.5846	0.526	0.6	0.0055	1.7778
	SVC	0.5825	0.5605	0.6	0.0039	2.7778
	Logistic Regression	0.5759	0.5224	0.6	0.0122	1.8333
Advanced Random	Random Forest	0.9682	0.4535	1.0	0.0113	1.2
	Logistic Regression	0.9383	0.3999	1.0	0.0089	5.35
	Naive Bayes	0.9343	0.3972	1.0	0.0325	0.375
	KNN	0.908	0.3017	0.95	0.006	3.8684
	Extra Tree	0.8496	0.1873	0.925	0.0024	9.1892
	SVC	0.8228	0.2718	0.925	0.0006	13.1081
	Decision Tree	0.7711	0.1855	1.0	0.2429	1.775
	MLP	0.6617	0.1056	0.8	0.0002	20.9688

Table 6.5. Simulated User Script Running on Virtual Machine Top Feature Importance Scores

Feature	**Importance Score**
CPU SUPPLY 1 CURRENT	0.18598349523946825
CPU SUPPLY 1 POWER	0.16244837692082506
CPU SUPPLY 1 VOLTAGE	0.13312233269140145
DDR3 MEMORY 1.35V LINE VOLTAGE	0.07181538692907137
HARD DRIVE CURRENT	0.07160332692209632
HARD DRIVE POWER	0.06865739571118795
PCH 1.05V LINE CURRENT	0.05935060514588598
1.8V S3 LINE CURRENT	0.05004190779956479
CPU A VOLTAGE	0.04139858571755868
DIMM 1.5V S3 LINE POWER	0.04065020079118585
DDR3 MEMORY 1.35V LINE POWER	0.022345117730539682
DIMM 1.5V S3 LINE CURRENT	0.020561776426695162
DDR3 MEMORY 1.35V LINE CURRENT	0.014770014323036547
5V LINE VOLTAGE	0.014232748297133857
1.8V S3 LINE POWER	0.012745097779953602

Table 6.6. Simulated User Script Running on Virtual Machine Detector Type Comparison

Detector	**Top Algorithm**	**DPS**	**MCC**	**RPD**	**Fallout**	**TPD**
Single	Random Forest	0.9682	0.4535	1.0	0.0113	1.2
Ensemble 6 Single	Naive Bayes	0.9379	0.282	1.0	0.0289	0.775
Ensemble 11 Single	Naive Bayes	0.9395	0.3037	1.0	0.0306	0.2
Ensemble 6 Dual	Naive Bayes	0.9148	0.3928	1.0	0.0443	0.0
Ensemble 11 Dual	Naive Bayes	0.9101	0.4004	1.0	0.0467	0.0

Chapter 7

Conclusion

7.1. Summary

System side channel data has commonly been used to attack systems in which an attacker has knowledge of how a target behavior or activity physically effects a system. Instead of attacking a system with side channel data we show that it is possible to defend a system by training machine learning algorithms to detect intricate patterns in the physical behavior of a system which correlate to a target behavior or activity.

This research effort has investigated and carried out experiments to determine the viability of utilizing side channel data streams from physical devices along with machine learning models for enhancing the security of a single physical system. Our results have shown that when sufficient side channel data is collected and meaningful labels are applied the new security approach is able to identify potential problems and irregularities very quickly. It is our belief that incorporating our new method with existing methods could help reduce the time it currently takes for detecting such events by acting as an early warning for more in depth security analysis.

This new method works by collecting a large amount of readily available side channel data and finding extremely complex relationships which can be used to draw conclusions about the system. It is our hypothesis that by implementing the same core principles in remote and virtual environments a user may be able to identify potential security threats by collecting and analyzing readily available side channels data streams in order to identify extremely complex patterns and relationships which are able to be used to draw conclusions about the environment and potential security threats.

7.2. Future Work

This research effort was able to show a proof of concept for physical sensor side channel security. However, there is still a large amount of research that can be conducted in order to refine the detection methods and identify new use cases. The addition of additional sensors into a system, as seen in chapter 3, was limited to an ICS in the scope of this study. However, we believe it may be possible to implement this technique in additional systems such as rack mounted servers. This would allow for rapid monitoring and detection of anomalies at a scale desired by an administrator. We also believe that the new detection model investigated in chapters 4, 5, and 6 requires additional investigation into how the complexity of the behavior being targeted effects the detection model type selection for optimal performance. For instance, the simulated ransomware process in chapter 4 was complex enough that a more advanced ensemble detection model was superior in performance to that of a single detection model. However, for the simple processes in chapter 5 and even the hypervisor activity in chapter 6 the single detection model was superior in performance to the more complex ensemble detection models.

Due to the relatively small amount of research that has been conducted in this area many different aspects of performance and implementation of physical sensor side channel security warrant further investigation. This research effort's largest contribution is that we were able to show the viability of physical sensor side channel security for several different use cases which are are high value to the cybersecurity community.

REFERENCES

Individual edition. URL https://www.anaconda.com/products/individual.

ffmpeg: A complete, cross-platform solution to record, convert and stream audio and video. URL https://www.ffmpeg.org/.

iozone filesystem benchmark. URL https://www.iozone.org/.

nmap security scanner. URL https://nmap.org/.

Vmware fusion 12 pro. URL https://store-us.vmware.com/vmware-fusion-12-pro-5424173700.html.

Feature importance in random forests, Aug 2015. URL https://alexisperrier.com/datascience/2015/08/27/feature-importance-random-forests-gini-accuracy.html.

Help Net Security March 24, Help Net Security, and March 24. Increasing number of false positives causing risk of alert fatigue, Mar 2020. URL https://www.helpnetsecurity.com/2020/03/24/alert-fatigue/.

Abrams L. "Cryptolocker Ransomware Information Guide and Faq". http://www.bleepingcomputer.com/virus-removal/cryptolocker-ransomware-information, 2013. Note: This is an electronic document. Date of publication: [October 14, 2013];.

Alharbi A, Thornton M. "Demographic Group Classification of Smart Device Users". *IEEE Int. Conf. on Machine Learning and Applications*, pages 481–486, Dec 2015.

Emanuel F Alsina, Manuel Chica, Krzysztof Trawiński, and Alberto Regattieri. On the use of machine learning methods to predict component reliability from data-driven industrial case studies. *The International Journal of Advanced Manufacturing Technology*, 94(5-8): 2419–2433, 2018.

BBCNews. "Cyber-attack: Europol Says it was Unprecedented in Scale". http://www.bbc.com/news/world-europe-39907965, 2017. Note: This is an electronic document. Date of publication: [May 13, 2017];.

Brandon R. "Petya Ransomware Authors Demand $250,000 in First Public Statement Since the Sttack". https://www.theverge.com/2017/7/5/15922216/petya-notpetya-ransomware-authors-bitcoin-demand-decrypt, 2017. Note: This is an electronic document. Date of publication: [July 5, 2017];.

Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

Leo Breiman. *Classification and regression trees*. Routledge, 2017.

Bresink M. "Hardware Monitor: Reference Manual". https://www.bresink.com/osx/216202/Docs-en/index.html, 2017. Note: This is an electronic document. Date of publication: [2017];.

Luke Broadwater. Baltimore transfers $6 million to pay for ransomware attack; city considers insurance against attacks, 2019. URL https://www.baltimoresun.com/politics/bs-md-ci-ransomware-expenses-20190828-njgznd7dsfaxbbaglnvn.bkgjhe-story.html.

Diogo M. Camacho, Katherine M. Collins, Rani K. Powers, James C. Costello, and James J. Collins. Next-generation machine learning for biological networks. *Cell*, 173(7):1581–1592, 2018. doi: 10.1016/j.cell.2018.05.015.

M Chang, Quang Do, and Dan Roth. Multilingual dependency parsing: A pipeline approach. *AMSTERDAM STUDIES IN THE THEORY AND HISTORY OF LINGUISTIC SCIENCE SERIES 4*, 292:55, 2007.

Davide Chicco and Giuseppe Jurman. The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation. *BMC Genomics*, 21 (1), 2020. doi: 10.1186/s12864-019-6413-7.

Edward J. M. Colbert and Alexander Kott. Springer International Publishing, 2016.

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. URL https://github.com/ColinIanKing/stress-ng.

Correa R. "How Fast Does Ransomware Encrypt Files? Faster Than You Think". https://blog.barkly.com/how-fast-does-ransomware-encrypt-files, 2016. Note: This is an electronic document. Date of publication: [April 2016];.

Cybersecurity and Infrastructure Security Agency. "Ransomware Activity Targeting the Healthcare and Public Health Sector". https://us-cert.cisa.gov/ncas/alerts/aa20-302a, Oct 2020.

Demme J, Maycock M, Schmitz J, Tang A, Waksman A, Sethumadhavan S, Stolfo S. "On the Feasibility of Online Malware Detection with Performance Counters". *40th Annual Int. Symposium on Comp. Arch*, June 2013.

Devcic J. "Weighted Moving Averages: The Basics". http://www.investopedia.com/articles/technical/060401.asp. Note: This is an electronic document. Date of publication: [2009];.

Bojana Dobran. 27 Terrifying Ransomware Statistics & Facts You Need To Read, 2019. URL https://phoenixnap.com/blog/ransomware-statistics-facts.

Harris Drucker, Christopher JC Burges, Linda Kaufman, Alex J Smola, and Vladimir Vapnik. Support vector regression machines. In *Advances in neural information processing systems*, pages 155–161, 1997.

Jenny Rose Finkel, Christopher D Manning, and Andrew Y Ng. Solving the problem of cascading errors: Approximate bayesian inference for linguistic annotation pipelines. In *Proceedings of the 2006 Conference on Empirical Methods in Natural Language Processing*, pages 618–626. Association for Computational Linguistics, 2006.

Di Freeze. The world will store 200 zettabytes of data by 2025, Mar 2021. URL https://cybersecurityventures.com/the-world-will-store-200-zettabytes-of-data-by-2025/.

Jason Fuller. Cyber-security emergency: Weaponized ransomware attack spreading like wildfire, 2017. URL https://www.cnet.com/news/ransomware-devastated-cities-in-2019-officials-hope-to-top-a-repeat-in-2020.

Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Machine Learning*, 63(1):3–42, Apr 2006. ISSN 1573-0565. doi: 10.1007/s10994-006-6226-1. URL https://doi.org/10.1007/s10994-006-6226-1.

Anil Gosine. Industrial control systems: Cyber policies and strategies. *Journal - American Water Works Association*, 112(6):48–54, 2020. doi: 10.1002/awwa.1518.

Gabor Jakaboczki and Eva Adamko. Vulnerabilities of modbus rtu protocol - a case study. *ANNALS OF THE ORADEA UNIVERSITY. Fascicle of Management and Technological Engineering.*, XXIV (XIV), 2015/1(1), 2015. doi: 10.15660/auofmte.2015-1.3111.

S. Kadloor, X. Gong, N. Kiyavash, T. Tezcan, and N. Borisov. Low-cost side channel remote traffic analysis attack in packet networks. *2010 IEEE International Conference on Communications*, 2010. doi: 10.1109/icc.2010.5501972.

Kaspersky Labs. Story of the year The ransomware revolution, 2016. URL https://media.kasperskycontenthub.com/wp-content/uploads/sites/43/2018/03/07182404/KSB2016_Story_of_the_Year_ENG.pdf.

Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. *Advances in Cryptology — CRYPTO '96 Lecture Notes in Computer Science*, page 104–113, 1996. doi: 10.1007/3-540-68697-5_9.

Takaya Kubota, Kota Yoshida, Mitsuru Shiozaki, and Takeshi Fujino. Deep learning side-channel attack against hardware implementations of aes. *2019 22nd Euromicro Conference on Digital System Design (DSD)*, 2019. doi: 10.1109/dsd.2019.00046.

C. Li and J.-L. Gaudiot. Detecting spectre attacks using hardware performance counters. early pre-print:55, 2021.

Angela Monoghan. Massive-cyber-attack could cost nurofen and durex maker £100m, 2017. URL https://media.kasperskycontenthub.com/wp-content/uploads/sites/43/2018/03/07182404/KSB2016_Story_of_the_Year_ENG.pdf.

Alfred Ng. Ransomware froze more cities in 2019. next year is a toss-up, 2019. URL https://www.cnet.com/news/ransomware-devastated-cities-in-2019-officials-hope-to-top-a-re.peat-in-2020.

O'Gorman G, McDonald G. "Ransomware: A Growing Menace". Technical report, Symantec Corporation, 2012.

R. Oshana, M. A. Thornton, E. C. Larson, and X. Roumegue. Real-time edge prrocessing detection of malicious attacks using machine learning and processor core events. In *Proceedings of 2021 IEEE Systems Conference*, pages 1–8. IEEE, 2021.

Danny Palmer. Ransomware: Now gangs are using virtual machines to disguise their attacks, Jun 2021. URL https://www.zdnet.com/article/ransomware-cyber-criminals-are-using-virtual-machines-to-hide-attacks-from-being-dete

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python . *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Pedregosa F, Varoquaux G, Gramfort A, Michel V. "Scikit-Learn User Guide". Technical report, 2010.

Matilda Rhode, Pete Burnap, and Kevin Jones. "early-stage malware prediction using recurrent neural networks". *Computers & Security*, 77:578–594, 2018. doi: 10.1016/j.cose.2018.05.010.

Scaife N, Carter H, Traynor P, Butler K. "Cryptolock (and Drop It): Stopping Ransomware Attacks on User Data". *IEEE Int. Conf. on Dist. Computing Systems*, June 2016.

Vladimir Sumina and Elisabeth Ivey. 26 cloud computing statistics, facts & trends for 2021, Jul 2021. URL https://www.cloudwards.net/cloud-computing-statistics/.

Tang A, Sethumadhavan S, Stolfo S. "Unsupervised Anomaly-based Malware Detection using Hardware Features". *Int. Symposium on Research in Attacks, Intrusions, and Defenses*, Sept 2014.

M.A. Taylor, K.N. Smith, and M.A. Thornton. Sensor-based Ransomware Detection. In *Future Technologies Conference*, pages 794–801, 2017.

M.A. Taylor, E.C. Larson, and M.A. Thornton. Rapid Ransomware Detection Through Side Channel Exploitation. In *IEEE International Conference on Cyber Security and Resilience*, 2021.

Alaa Tharwat. Classification assessment methods. *Applied Computing and Informatics*, 17 (1):168–192, 2020. doi: 10.1016/j.aci.2018.08.003.

Vikyd. vikyd/go-cpu-load: Generate cpu load on windows/linux/mac. URL https://github.com/vikyd/go-cpu-load.

Wyke J, Aijan A. "The Current State of Ransomware". Technical report, SophosLabs, December 2015.

D.F. Gu X.H. Wu, H.T. Ma. An automatic network protocol reverse engineering method for vulnerability discovery. *Network Security and Communication Engineering*, page 65–70, 2015. doi: 10.1201/b18660-14.