MALWARE DETECTION USING ELECTROMAGNETIC SIDE-CHANNEL ANALYSIS

THESIS

Matthew A. Bergstedt, Captain, USAF

AFIT-ENG-MS-22-M-008

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-MS-22-M-008

MALWARE DETECTION USING ELECTROMAGNETIC SIDE-CHANNEL ANALYSIS

THESIS

Presented to the Faculty
Department of Electrical and Computer Engineering
Graduate School of Engineering and Management
Air Force Institute of Technology
Air University
Air Education and Training Command
in Partial Fulfillment of the Requirements for the
Degree of Master of Science in Cyber Operations

Matthew A. Bergstedt, B.S.E.E.
Captain, USAF

March 24, 2022

AFIT-ENG-MS-22-M-008

MALWARE DETECTION USING ELECTROMAGNETIC SIDE-CHANNEL ANALYSIS

THESIS

Matthew A. Bergstedt, B.S.E.E.
Captain, USAF

Committee Membership:

Lt Col James W. Dean, Ph.D.
Chair

Scott R. Graham, Ph.D.
Member

Michael A. Temple, Ph.D.
Member

AFIT-ENG-MS-22-M-008

Abstract

Many physical systems control or monitor important applications without the capacity to monitor for malware using on-device resources. Thus, it becomes valuable to explore malware detection methods for these systems utilizing external or off-device resources. This research investigates the viability of employing electromagnetic (EM) side-channel analysis (SCA) to determine whether a performed operation is normal or malicious. A Raspberry Pi 3 was set up as a simulated motor controller with code paths for a normal or malicious operation. While the normal path only calculated the motor speed before updating the motor, the malicious path added a line of code to modify the calculated speed. A script from a control terminal then sent a signal to the Pi to have it conduct either the normal or malicious operation while an EM probe was set up to collect emission traces of those operations. These traces were split into training and testing data sets, with the training set used to train a support vector classification (SVC) model. Afterwards, the model was run on the testing set and achieved 96% classification accuracy for classifying the trace as either normal or anomalous.

Table of Contents

Page

List of Figures

List of Tables

MALWARE DETECTION USING ELECTROMAGNETIC SIDE-CHANNEL ANALYSIS

I. Introduction

1.1 Background and Motivation

Across the defense and commercial sectors, control systems oversee and monitor critical infrastructure. As technology has improved, a move has been made to interconnect these systems, which has increased their vulnerability profiles. Despite this increased vulnerability, these systems lack on-device resources that could be used to monitor for potential malware. Thus, it is beneficial to explore options of providing malware detection using off-device resources.

More recently, malware attacks have started specifically targeting control systems. Examples of such attacks include Stuxnet [1], Havex [2], as well as the attacks on the Ukrainian power grid [3]. These demonstrate the need for malware detection options for control systems.

As a response to such attacks against control systems, the US Industrial Control Systems Cyber Emergency Response Team (ICS-CERT) launched an assessment program in 2009 to strengthen the security of control systems. As part of their operations, they began publishing an annual report detailing the state of security [4], with the first published in 2014. These show that the US is taking steps to address control systems security.

1.2 Problem Statement

The inability of control systems to monitor themselves for malware presents a weakness to their security. Thus, the questions that this research attempts to address are:

- Can an external device, in this case an electromagnetic (EM) probe, provide the capability of performing malware detection on a device.
- If malware can be detected, can the captured trace be decomposed to increase the accuracy.

1.3 Research Goals/Objectives

The goals for this research are to be able to detect malware running on a device utilizing an off-device resource such as an EM probe.

1.4 Hypothesis

This research hypothesizes that it is possible to detect anomalous activity using unintended EM emissions from a device. Additionally, by dividing the captured traces into intervals, it will be possible to more accurately detect anomalous activity.

1.5 Approach

To test the hypothesis, a Raspberry Pi will first be run in a bare metal environment to emulate a motor controller. Next, an EM probe connected to an oscilloscope will be used to capture emissions from the processing chip on the Pi. Then, a support vector classification (SVC) machine learning algorithm [5] is used to predict the class of a set of test data, which results in an accuracy metric for detecting malware.

1.6 Contributions

The research conducted contributes the demonstration of using direct unintended EM emissions to detect anomalous code behavior in a motor controller. This research also contributes the analysis of the captured data through time analysis, which better distinguishes between normal and malicious operations and determining where the malware occurs in the code. Lastly, this research contributes the documentation of a probability prediction for detecting anomalous behavior, which would likely prove to be more beneficial in an operational environment than direct classification.

1.7 Thesis Overview

The remainder of this document is as follows. Chapter II introduces technical background information related to the research and discusses other research relevant to this work. Chapter III covers the set up and design of the experiment, in addition to the statistical methods used for modeling the captured data. Chapter IV expounds upon the results of the experiments, laying out the captured data and the statistical results related to that data. Chapter V analyzes the results, draws conclusions from the research, and presents opportunities for future research to expand on this work.

II. Background and Literature Review

2.1 Overview

This chapter will briefly introduce topics relevant to the research conducted, as well as cover previously researched works related to this topic.

2.2 Bare Metal Programming

A typical programmable logic controller (PLC) that oversees critical infrastructure runs proprietary firmware. However, without access to such a PLC and the ability to modify its code, an alternate test device needs to be used. One such device capable of the desired functionality is a Raspberry Pi. The Pi normally runs with a Linux operating system, which contains a cooperative scheduler. Because the scheduler could impact the motor controller timing by preventing operations from running immediately when needed, the best method is to avoid it by running the Pi in a bare metal environment. With this environment, the Pi can avoid the scheduler, but will need to have each desired register to be used specified in the code. The Pi processor chip runs on an Advanced RISC Machines (ARM) architecture, but many Broadcom peripherals for the chip, such as timers, general-purpose input/output (GPIO), universal asynchronous receiver-transmitter (UART), and pulse-width modulation (PWM), are accessible through their memory addresses. Information about these peripherals and how to interact with them is accessible in the ARM peripherals documentation for the specific chip [6]. An additional web page provides a mapping of the peripheral registers to their memory addresses [7] allowing for mapping the registers in code.

2.3 Side Channel Analysis

Side-channel analysis (SCA) is the process of analyzing information about a device by observing a side channel off of that device. Some side channels are power leakage, electromagnetic (EM) emissions, and timing and they are events that are not considered as strongly when securing a device. More often, SCA is used as a means of performing an attack, where an attacker takes advantage of designers not considering power consumption or chip emissions in order to determine how a device is operating. Traditional exploitation of a device typically involves gaining control of the device through means such as hacking. A side channel attack attempts to use the side channel, such as power leakage, to gain information about the operation of the device. For instance, if a device draws different amounts of power each time it encrypts data, an attacker could potentially use known data to recover information about the encryption. Though SCA can be used for an attack, it can also be used for defensive purposes. While attackers may take steps to hide themselves from the network and host perspectives, they are significantly less likely take into consideration power consumption and chip emissions.

2.4 Support Vector Machines

Support vector machines (SVMs) are statistical learning methods for performing classification, regression analysis, and outlier detection. For this research, the classification capabilities were used. This allowed the data to be classed which can then used to train a model to recognize the features of each class. After training the model, the model can then be used to predict the class of test data. Alternatively, the model can be used to perform a probability prediction of the classes, where it will return the probability that a trace matches each of the trained classes rather than just returning the predicted class.

2.5 Related Work

Early research into SCA theorized the possibility for exploitation, observing that current flows through transistors used in semiconductor logic gates. By setting up a small resistor in the circuit, researchers could determine and observe the power drawn by the circuit, which could be used to demonstrate power SCA. Later researchers extended the idea that a device could exploited via power leakage, altering it to see if the same capability was possible through processor emissions. Kocher et al. [8] and Messerges et al. [9] showed how differential power analysis (DPA) could be used to recover parts of the Data Encryption Standard (DES) key used in smart cards. These early researchers demonstrated the success that such attack methods could have and paved the way for future SCA research. Following that success, Brier et al. [10] examined correlation power analysis (CPA) as an alternative side-channel attack method to DPA. The research addressed some problems with DPA that the CPA method seems to address.

Spawning from these works, Hanley et al. began looking at template attacks in order to recover the encryption keys [11]. Expanding on that, Lerman et al. looked at utilizing machine learning to help relax some assumptions needed for a template attack [12]. Along that same thread, Maghrebi et al. and Bartkewitz have looked into using Deep Learning models and other machine learning techniques to perform key recovery attacks or improve the success of those attacks [13] [14]. Others have researched methods of improving SCA attacks, with Socha et al. [15] focusing on AES implementations possible on a specific board, Xu and Heys [16] comparing static and dynamic side-channel attacks against a masked S-box circuit, Vosoughi and Köse [17] examining how distinguishers, described as classifiers that divide possible keys into correct and incorrect, could be combined to enhance the success of attacks, and Moos et al. [18] looking at various factors that could have an effect on information extraction with static power analysis.

While there are numerous research efforts around side-channel attacks, there are also efforts regarding how to protect against such attacks. Chevallier-Mames et al. [19] introduced simple ways of altering an algorithm in a manner that could help it be resistant to simple side-channel attacks. Fournaris et al. [20] examines several real-time embedded system device attacks and discusses some countermeasures to help prevent them.

Additionally, efforts are also being followed to see the effectiveness of SCA in detecting malware running on a system. Clark et al. [21] presents a power analysis method to monitor for malware in embedded medical devices, which would help ensure the security of crucial life-saving devices. This research focused on a need in the medical community due to many necessary devices not being able to support updates or antivirus software because of the manufacturer's policy. Clark and the others set up to measure system wide power consumption by measuring from the power outlet, which they deemed possible due to medical devices having specific applications per machine. For the research, the team used two devices, a pharmaceutical compounder and a supervisory control and data acquisition (SCADA) device. To perform the detection, the group used a supervised learning approach, using the mean, variance, skewness, kurtosis, root mean square, global minimum and maximum, and interquartile range as their features for training. The research successfully identified malware it was trained on with 94% accuracy for the compounder and 99% accuracy for the SCADA device, and identified malware it was not trained on with 88% accuracy for the compounder and 85% accuracy for the SCADA device.

Shende and Ambawade [22] also uses power analysis as a means of detecting hardware Trojans in a device. These two researchers fed numerous input vectors to a circuit with and without a hardware Trojan. They then used mean, variance, root mean square, median, and histogram as their features which they used with Principle Component Analysis to reduce the number of variables in the data. Linear discriminant analysis (LDA) was used to differentiate between the non-Trojan variables and the Trojan variables. After performing the reduction, LDA was able to completely differentiate between the circuit with the Trojan and without the Trojan, due to the circuit with the Trojan dissipating much more power than the other circuit.

Ding et al. [23] presents a Deep Learning model that observes power side-channels of Internet of things (IoT) devices to determine any malicious activity. The group focuses on Linux-based IoT devices, aiming to detect IoT malware attacks. They evaluate several common attacks to determine any common operations and commands used in infection and exploitation. To perform detection, they focus on the infection activities, and split the process into four phases:

- Detection of suspicious signals.
- Preprocessing of suspicious signals.
- Inferring activities from suspicious signals.
- Infection process modeling and correlation analysis.

The first and second phases identify signals and attempt to remove any noise before transforming the signal into a spectrogram for the later phases. The third and fourth phases apply a Seq2Seq architecture with Long Short-Term Memory networks to infer activity and then attempts to correlate the inferred activities with the infection process earlier identified. Across five IoT malware tested, they achieved around 90% accuracy on average.

Furthermore, more recently several efforts have been made in utilizing EM SCA as a means of detecting malware. Han et al. [24] focused their efforts on control flow integrity of PLC code execution. The group used frequency representations of EM signal sections to analyze their collected traces. With the representations, they train a model to learn control flow transitions. By comparing later collected signals with the learned transitions, the group attempts to alert on illegitimate control flows that do not match. Evaluating the model on Allen Bradley PLCs, they achieved 98.9% accurate detection of malicious code executions.

Sehatbakhsh et al. [25] propose a framework for detecting malware from EM emissions. For their framework, the group transforms signals into a sequence of spectral samples using short-time Fourier transform. They also devise a metric to determine distance between samples, which corresponds to how likely the samples are to have been produced by execution of the same code. After collecting and transforming signals, the framework is trained to separate the samples into an appropriate region of code. Then, during monitoring, it compares the new sample to the current region or a valid next region based on the training. It uses the distance measurement as a means of classifying a sample into an unknown category, representing anomalous behavior. Measuring across several devices and scenarios, the group achieved greater than 98% accuracy with their framework in all cases.

Chawla et al. [26] propose a framework utilizing a fast Fourier transform (FFT) and SVM and Random Forest based machine learning models. They apply the FFT to their EM traces to perform feature extraction. After performing a dimension reduction, the group trains a Random Forest model to learn non-linear complexities of the feature space and a SVM model to divide the feature space. They selected a Open-Q 820 Development Kit as the test device, running both malware detection and malware classification tests. The detection phase used benign and malware data sets and achieved 99% accuracy in detection. For the malware classification phase, the group selected eight malware families, achieving 88% accuracy in classifying between the different malware families.

Khan et al. [27] propose a framework to compare EM emanations with a reference dictionary using euclidean distance. After first capturing a trace, the group demodulates, low-pass filters, and samples the trace before passing it on for signal processing. They fill out the reference dictionary by collecting known safe EM signals and breaking down the signal into short-duration windows which serve as the dictionary entries. When monitoring, the signal is likewise broken down into a short-duration window, which is then compared to the entries in the dictionary using euclidean distance. The group then reconstructs the signal using the best matching dictionary entry, and the reconstructed signal is compared against the monitored signal for anomaly detection. If a moving average of the squared difference of the two signals surpasses a threshold, then it is determined that an anomalous behavior has been detected. Testing specific attacks on a few different systems, their implementation achieved greater than 98% accuracy in all cases.

In addition to the work examining embedded device processors, Ibrahim et al. [28] have proposed a framework for verifying the authenticity of a USB flash drive through its unintended magnetic emissions. In the research, the group was able to uniquely identify certain brands and models of USB flash drive based on the magnetic emissions during the boot procedure.

2.6 Summary

This chapter presents brief backgrounds on bare metal programming, side-channel analysis, and Support Vector Machines to aid the reader in understanding topics relevant to this research. Additionally, it discusses several related works relevant to the research which may provide further information regarding SCA. It covers the early implementation of side-channel attacks and how they were theorized, as well several attack efforts that were spawned from that early research. It also touches on some attempts at preventing such attacks from being able to take place. It further touches several efforts of using SCA from a defensive standpoint, similar to the goals of the research presented in this paper.

III. Methodology

3.1 Objective

The goal of this research is to determine whether electromagnetic (EM) side-channel analysis (SCA) can be used to successfully distinguish between normal operations and anomalous activity on a simulated motor controller. To accomplish this, a simple motor controller needed to first be developed to run normal and malicious operations. The research then aims to test the sensitivity of the support vector machine (SVM) model created based on the location of the EM probe in relation to the chip. Furthermore, the research examines the sensitivity of time alignment by splitting the collected traces into three intervals. Building on the goal of splitting the traces into thirds, the traces were set up as a sliding window to see if a specific section could be identified as being the best at distinguishing between the two operations. Finally, the collected operations were put together in a manner to attempt to create an operational environment in order to see if the process could successfully identify a single malicious operation in the midst of normal operations.

In order to accomplish these goals, the research was split into three parts. The first part covers the process of setting up the environment and getting the simulated motor controller running. The second part covers running the experiment to collect the EM trace data. The third part covers training a SVM model with the captured trace data and then using that model to perform analysis on a test group of data.

3.2 Assumptions

This research is conducted with three primary assumptions. First, the motor is run at a fixed revolutions per minute (RPM) that does not vary except when it is specifically changed. A varying RPM could affect the RPM calculations of future trials that is not accounted for in the code. Second, the motor load is constant. A non-constant load could alter the RPM between trials, which would affect the RPM calculations. Third, there is not any malware already present on the Raspberry Pi that actively interferes with the research in some manner. If malware were present, but it operated consistently between the designed normal and malicious operations, then it should not affect the capture process and the end results. However, if malware were present and operated differently for either of the designed operations, then that would produce some undesired effect that would be captured and affect the end results. Thus, it is assumed that the second situation is not the case.

3.3 Variables

Throughout the overall process the different parts need a way to identify the difference between a normal and a malicious operation, with the only difference between the two being whether or not the malicious code segment executes. Where needed, the value of 0 was assigned to normal operations, and the value of 1 was assigned to malicious operations or anomalous activity, depending on whether the malicious intent is already known. These values were then used in place of the operation type when communicating across platforms. Additionally, when conducting the statistical analysis, 0 and 1 were used as the classification for the two different operations when training and testing the algorithm.

3.4 Experimental Setup

The process for collecting traces consisted of three main platforms. First, a Raspberry Pi 3 simulated as a motor controller which ran either the normal or malicious operation. Next, an oscilloscope and EM probe captured a trace from the processing chip on the Pi. Lastly, a connected computer was set up to control which operation the Pi ran as well as collecting the captured trace from the oscilloscope and resetting it for the next capture. The platforms were functionally laid out and connected as shown in Figure 1.

The Raspberry Pi was connected to the control computer via USB to serial, connecting to universal asynchronous receiver-transmitter (UART) on the Pi. The Pi also output a general-purpose input/output (GPIO) signal to serve as a trigger, which was read by an oscilloscope probe. The oscilloscope was connected to the control computer via Ethernet cable. The oscilloscope also connected via a coax cable to an EM probe, which was positioned over the processing chip on the Pi.

3.4.1 Simulated Motor Controller

A Raspberry Pi 3 was set up to run with no operating system as a simulated motor controller. The basic control loop that the Pi simulates is shown in Figure 2. The C code for the main function can be found in Appendix B. With this setup, the Raspberry Pi simulates the logic controller, and the motor simulates the plant. The code base for the Pi begins with the uart1 program implemented by Zoldan Baldaszti

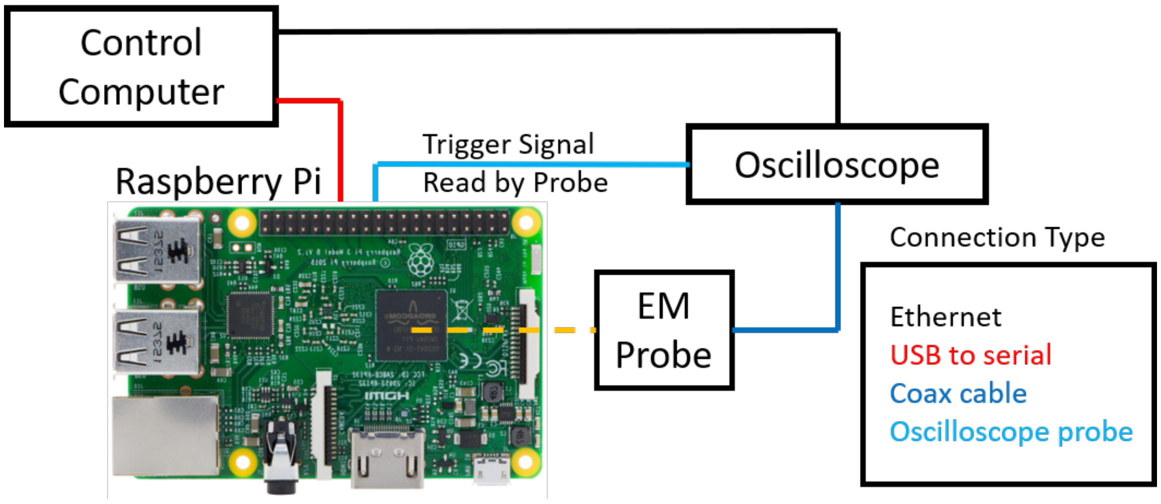


Figure 1: Functional layout of the experiment showing how each of the platforms were connected

and published on GitHub [29]. This provided the basic bare metal environment and communication between the Pi and the control computer. Additional functionality to alter the code for the purposes of this research was added. Several threads from the Raspberry Pi forums provided assistance in understanding how certain components were manipulated [30], [31], [32]. First, the Pi would set up a pulse-width modulation (PWM) signal which is output to a motor driver that drives a low-power (LP) 6V motor. The output from the connected motor encoder is then used as input to the Pi. Using rising edge detection on that input to generate a event that can be monitored, the Pi checks a system counter to determine the time between rising edges. This time between rising edges is used to calculate the period of the motor and then the RPM speed. Based on the calculated RPM, a check is made to see if the motor is spinning either too fast or too slow, passing the check if it does not meet either of those conditions. If the motor RPM fails the check, the PWM signal is updated to either speed up or slow down. The C code for the basic control loop can be found in Appendix C under the normOp function.

To add in a malicious path, code was added to subtract from the calculated RPM. The rest of the code path remains the same, however the subtracted value replaces the

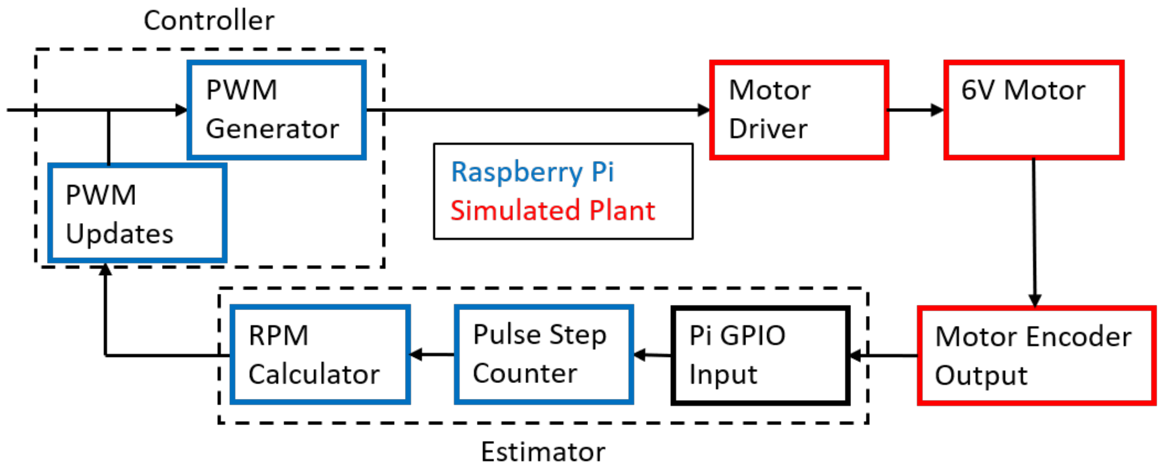


Figure 2: Diagram of base control loop of simulated motor controller

correctly calculated value for motor speed. The updated control loop highlighting the addition of a malicious effect is shown in Figure 3. As seen in the figure, the malicious path adds in its components after the normal RPM calculations but before the PWM updates. The C code for the malicious effect can be found in appendix C under the malOp function.

The program running on the Raspberry Pi begins by setting up the UART connection to the control computer, using Baldaszti's UART code. It then sets up and starts the motor. After the motor has had time to reach its designated starting speed, the program waits for an input from the control computer to determine which operation type should be run, which also uses Baldaszti's UART code. The program will then enter the code segment for the provided operation. Inside that operation, the Pi will set a GPIO output high, after it has detected rising edges from the motor, to signal the oscilloscope trigger for collection. Before exiting the operation, the Pi will set the GPIO output low, signalling the end of that capture. Thus, the motor controller code allows the EM trace collection to target the segment inside of the two operation types. The overall code flow on the Raspberry Pi is shown in Figure 4.

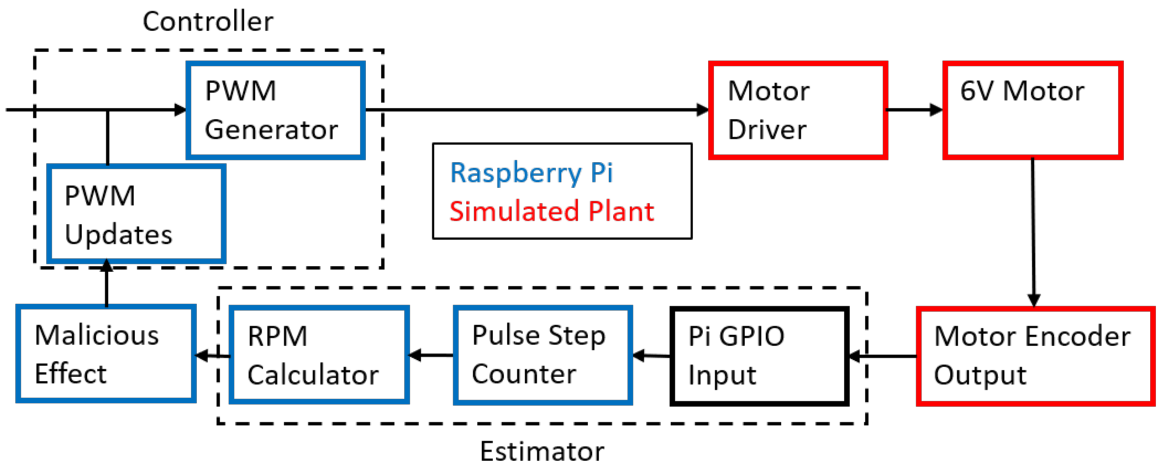


Figure 3: Diagram of control loop of simulated motor controller adding malicious code path

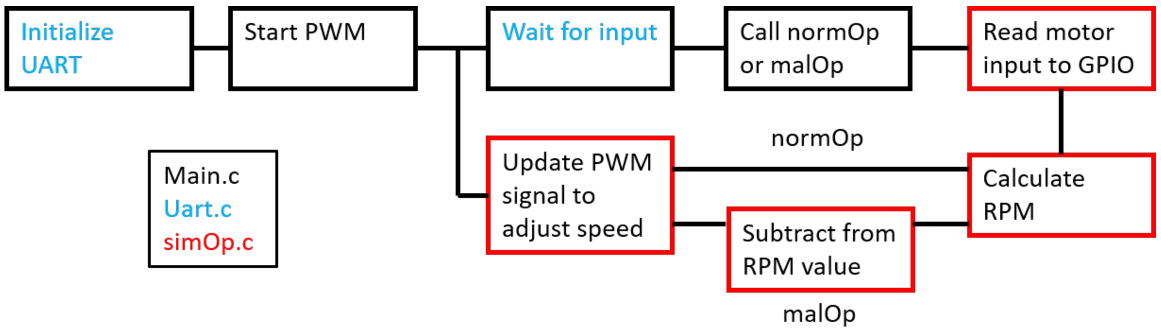


Figure 4: Diagram of the overall code flow on the Raspberry Pi

3.4.2 Oscilloscope

An oscilloscope was set up with an EM probe that was positioned above the processing chip on the Pi in order to capture traces when the Pi performs the designated operations. The various locations for the probe positioning are discussed in Section 3.6.1 below and shown in Figure 6. An oscilloscope probe is connected to a GPIO output from the Pi which serves as a trigger signal for the scope. When the trigger signal goes high, the scope will begin capturing and keeping the collected trace from the EM probe. The oscilloscope can then pass the trace to the control computer before a new trace is started, so that an individual trace can be obtained for every operation that is run.

3.4.3 Control Computer

A desktop computer was set up as the controller of the overall environment in order to tell the Pi which operation to run and also pull back captured traces from the oscilloscope. It was connected to the Raspberry Pi via a USB to serial connection, which allowed for bytes to be sent to and from the Pi. Using this connection, the computer would send the byte of the desired operation to be run to the Pi. The computer was connected to the oscilloscope via Ethernet. This connection allowed the computer to control the oscilloscope, which was used to reset the oscilloscope to single capture mode and to retrieve the individual traces.

3.5 Collection

With the environment set up, the control computer ran the experiment to collect traces. The computer did this through a Python script executed from the command line. The Python code can be found in Appendix D, with the code flow for the script shown in Figure 5. The script begins by setting up a connection to the oscilloscope, and then slightly later a connection to the Raspberry Pi. It then generates an array of 25,000 zeros and an array of 25,000 ones. After combining the two, it randomizes the order of them to provide variation in the operation being run. Next, it loops through the combined array. In the loop, it begins by rearming the oscilloscope for the trigger to successfully allow for the capture of relevant signal data. It then sends either a binary zero or one to the Pi over the serial connection to have the Pi run the appropriate operation. The code next collects the captured trace information as well as the trace for the trigger signal. Using the trigger signal, it tracks the minimums and maximums of the region of interest between the earliest time the trigger goes high until the latest time the trigger goes low. The loop ends by storing each retrieved trace into an array for either normal traces or malicious traces depending on the operation. After the loop, the code trims the the arrays down to just within the region of interest, resulting in the same number of samples across the arrays which is necessary for the machine learning. The collection experiment can then save the arrays of traces for further analysis.

From a simple examination of several traces, it appeared that every trace had an initial response to the trigger signal being set high. It also appeared that this trigger response lasted for about 100 samples, with the duration being consistent across traces. Thus, the first 100 samples were removed from the traces prior to actually

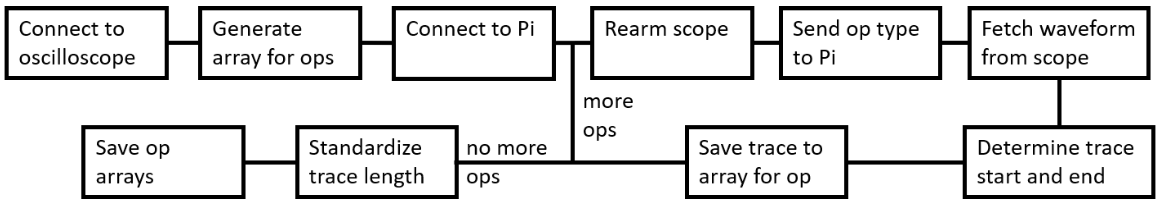


Figure 5: Diagram of the overall code flow for the Python collection script to generate and retrieve traces

running the machine learning parts of the analysis, since the trace during that time does not contain any capture related to the code execution of the motor controller.

As a simple means of testing for a difference between the normal and malicious operations, the average traces for each operation are plot against each other. If there is a difference between the operations, it would be expected that the average traces diverge at some point. In order to verify that any divergence is due to the identification of a difference in code, the collection experiment is repeated with no difference between the operations. The average traces for the operations when they are the same are then plotted against each other. It would be expected that the average traces for the operations when they are the same are either exactly identical or have very minimal differences.

3.6 Statistical Analysis

The statistical analysis is first followed to determine if it is possible to detect anomalous activity. Later statistical analysis was then divided into four additional main efforts:

- Varying probe location in order to test the sensitivity of the SVM model to the location.
- Splitting into intervals to test the sensitivity of the SVM model to time alignment.

- Further time dividing and sliding across the trace to identify specific sections that produce the most sensitive SVM model.

- Creating a scenario to mimic an operational environment in order to test the capability of probability prediction.

These efforts contained several steps that were common across the initial analysis and each effort. Since the trace data pulled from the oscilloscope included additional data not needed for analysis, the traces were reformatted to only include the amplitude data. As mentioned in Section 3.5, the first 100 samples of the traces were removed. Then, the traces were divided into five sets of 10,000 traces each, which were evenly split between normal and malicious operations. These sets were then further split into training and testing groups for use with a machine learning model to produce statistical results, with 5,000 traces put in the training group and 5,000 traces put in the test group. The support vector classification (SVC) implementation from the Scikit-learn library [5], which performs the classification functionality of a SVM, to first train a model on the training data, with the classes being either normal or anomalous, represented by a 0 and 1. After training the model, the model was used with the test data to generate predictions on the class of the each trace in the test group. Comparing the predicted class of each trace with the original class allows for a calculation of the number of true and false positives as well as true and false negatives. Using these results, the overall accuracy of the predictions can be determined, along with precision and recall of the predictions, for each set. If the initial analysis determines that anomalous activity can be detected, then the additional efforts can be investigated.

Equation (1) shows accuracy is defined as the number of correctly identified traces, true positives (TP) and true negatives (TN), over the total number of traces, TP and TN as well as false positives (FP) and false negatives (FN). The total number of traces will be the number of traces in the test group, which is 5,000.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{1}$$

Equation (2) shows precision is defined as the number of traces correctly identified as anomalous, TP, over the total number of traces identified as anomalous, TP and FP.

$$Precision = \frac{TP}{TP + FP} \tag{2}$$

Equation (3) shows recall is defined as the number of traces correctly identified as anomalous, TP, over the total number of traces that actually were anomalous, TP and FN.

$$Recall = \frac{TP}{TP + FN} \tag{3}$$

Precision and recall provide an alternate means of assessing prediction performance. Low precision indicates too many FP, which creates extra work for a defender. Low recall indicates too many FN, which means the most threatening malware could go unnoticed. In the event that both precision and recall are around the same value as accuracy, then accuracy successfully accounts for those situations.

3.6.1 Varying Probe Location

The code for this effort is found in Appendix E. The entire collection process was completed four times for various locations of the probe in relation to the processing chip on the Pi. The locations were empirically determined to test the sensitivity of the SVM model to the location of the probe. First, the probe was centered over the chip, which was performed as the initial phase of analysis. It is expected that this location would have the best results from statistical analysis due to being directly over the primary emission source, where it is expected that the most electrons are flowing to create the strongest EM field. Second, the probe was set up over one of the edges of the chip. This location is still partly over the primary emission source, so there should still be some ability to pick up the EM field and distinguish between the two operations, but it is expected that there is some drop from the accuracy of the centered location. Third, the probe was set up over a different part of the board that primarily performed an unused function. At this location, there is definitely an expectation that the accuracy would drop when compared to the centered location because there is no known electron flow directly beneath the probe, however it is still expected to have some ability to predict between the two operations due to emissions from the chip and other components on the board. Last, the probe was set up without the Pi underneath it in order to do a baseline test of capturing air. This test checks to confirm that there is no external means of a signal leaking into the trace, and it is expected to have an accuracy of around 50%. The first probe location was used initially as the means of testing the hypothesis, then the other locations were tested to account for other factors that may have impacted the results. Prior to running any experiments at each probe location, the shielding around the probe was reset so that the bottom of the shield was even with the bottom of the probe. Once each location has been evaluated, confidence intervals for the accuracy at each location are calculated. Then, a Student's two-sided t-test is conducted between the location with the highest accuracy and the other locations. The t-test produces a p-value. If the p-value produced is less than 0.05, then the null hypothesis that the two accuracy arrays are the same can be rejected. Thus, a p-value less than 0.05 indicates the accuracy of each location is statistically different. Figure 6 shows the probe locations relative to the Pi chip.

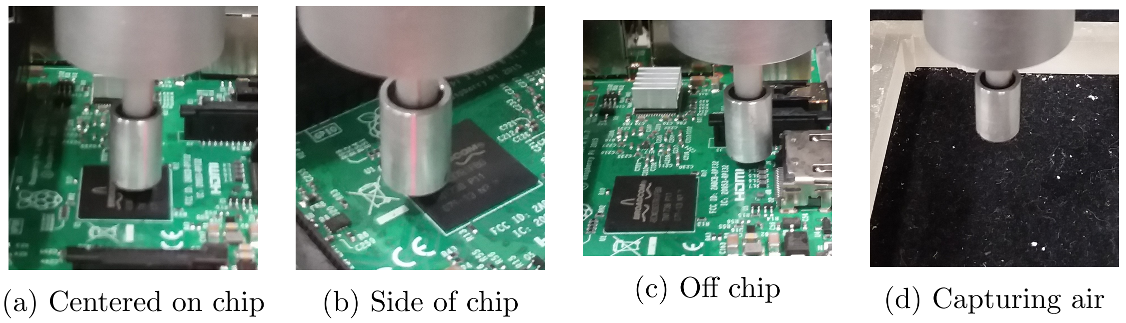

(a) Centered on chip (b) Side of chip (c) Off chip (d) Capturing air

Figure 6: Location of EM probe relative to the Raspberry Pi chip

3.6.2 Split into Thirds

After completing the analysis for the various probe locations, the location with the highest accuracy was selected for future efforts. The sets were then divided into three intervals, and the procedure for calculating accuracy was repeated for each of the thirds. To accomplish this, the training and testing section was split into three parts, covering the first, middle, and last thirds, with each third training a model and testing it on the appropriate data from that third. Due to requirements that the training and test data have the same number of samples in the trace, the originally trained model from the previous effort could not be used. It is expected that the last third will have the lowest accuracy since it contains the segments where the Pi reverts to a busy waiting after finishing the operation. The accuracy of the first and middle thirds is heavily dependent on where the divergence in the code appears in the trace. If the code divergence is captured in the middle third of the trace, then it is expected that the accuracy of the middle third would be the highest. If the code divergence is captured in the first third, then the first third would be expected to have the highest accuracy, although carry-through effects could still result in the middle third having a high accuracy as well. There are a few traces where the end of the operation, and thus the end of the relevant trace, falls near the end of the middle third, which would skew the accuracy of the middle third slightly lower than the accuracy of the first third for the same reason that the accuracy of the last third is expected to be lower. Once each third has been evaluated, confidence intervals for the accuracy at each location are calculated. Then, a t-test is conducted between the accuracy of each third to determine if each accuracy is statistically different from the others.

3.6.3 Sliding Window

After having split the trace into three intervals, it is possible to further examine the trace by creating a window that steps through the traces. For this method, a window size and step size need to be defined. The window size is the number of samples that will be included in each slice. The step size is the number of samples that the slice is increased by for each trial. Since the end of operations occurs in the last third of the capture traces, the sliding window is set up to run through the first two thirds of the traces. Additionally, the first 100 samples of the traces are included for this test in order to get the full picture. This test is first run with a window size of 100 samples and a step size of 50 samples, so the first trial would run the test on trace samples 0-100, the second trial would run the test on trace samples 50-150, and so forth. The test is then repeated with a window size of 50 samples and a step size of 25 samples, so the first trial would run the test on trace samples 0-50, the second trial would run the test on trace samples 25-75, and so forth. To accomplish this functionality, an additional loop is used for the training and testing section to go through the total number of steps needed to get though the section of the trace used. The accuracy of just the section that the sliding window includes is then calculated. While previous efforts predicted the class to determine the statistical results, this effort only calculates the resulting accuracy, since that is the only statistic of interest for this effort. This further narrows down the window to best identify the location of the traces that results in the highest accuracy. After determining the region for the highest accuracy, statistical analysis is conducted on that region as was done in the previous efforts. The statistical results for the region can then be compared to the previous statistical results.

3.6.4 Mimicking Operational Environment

After completing the previous efforts, it should be visible that there is some detectable difference between a normal and a malicious operation. However, these tests are performed outside of an operational environment and might not necessarily be indicative of how well it could perform in real world. Thus, a test was created to attempt to mimic an operational environment. The previous efforts fully predicted the class. This effort utilizes the probability prediction capability, which returns the probability that a test trace is in the normal or malicious class. Since this scenario is a two class problem, only the probability that a trace is normal is considered to simplify the analysis. To set up this test, a single malicious operation is put in the middle of several normal operations. This scenario is then compared against the previously trained model and the probability that each operation is normal is determined. Using those probabilities provides a glimpse at how a similar setup might be used in an operational environment, since it would be more realistic to use classification probability rather than direct classification.

3.7 Summary

This chapter discusses the methodology to collect traces from a Raspberry Pi 3. Additionally, it discusses the tests used to examine the captured traces in order to achieve an accuracy of predictions on the classification of those traces. These tests look at the overall accuracy for several EM probe locations, accuracy of a probe location when that data was split into thirds, accuracy of the data when further time divided into smaller slices, and usefulness of a probe collect in an operational environment.

IV. Results and Analysis

4.1 Overview

This chapter presents the results from the methodology efforts outlined in Chapter III. The average of both the normal and malicious traces is presented for the collection efforts. For the statistical analysis efforts, the results are presented for the four tests that were described: varying the electromagnetic (EM) probe location, splitting the captured traces into three intervals, creating a sliding window that steps through the captured traces, and creating a scenario to attempt to mimic an operational environment.

4.2 Collection

After having collected traces for all 50,000 operations run, the normal and malicious operations were averaged together and then plotted to provide an initial look at how two operations might differ. As seen in Figure 7, there do appear to be some identifiable differences between a normal and a malicious operation. These differences appear to begin, on average, at sample 500. Based on the code that was used for the operations, this time in the trace corresponds with where it would be expected for the differences to begin occurring.

For comparison, the averages of the traces when the code segments were the same can be seen in Figure 8. As seen in the plot, the average traces in this scenario are essentially identical, demonstrating that the differences detected in Figure 7 are due to a branch in the code paths.

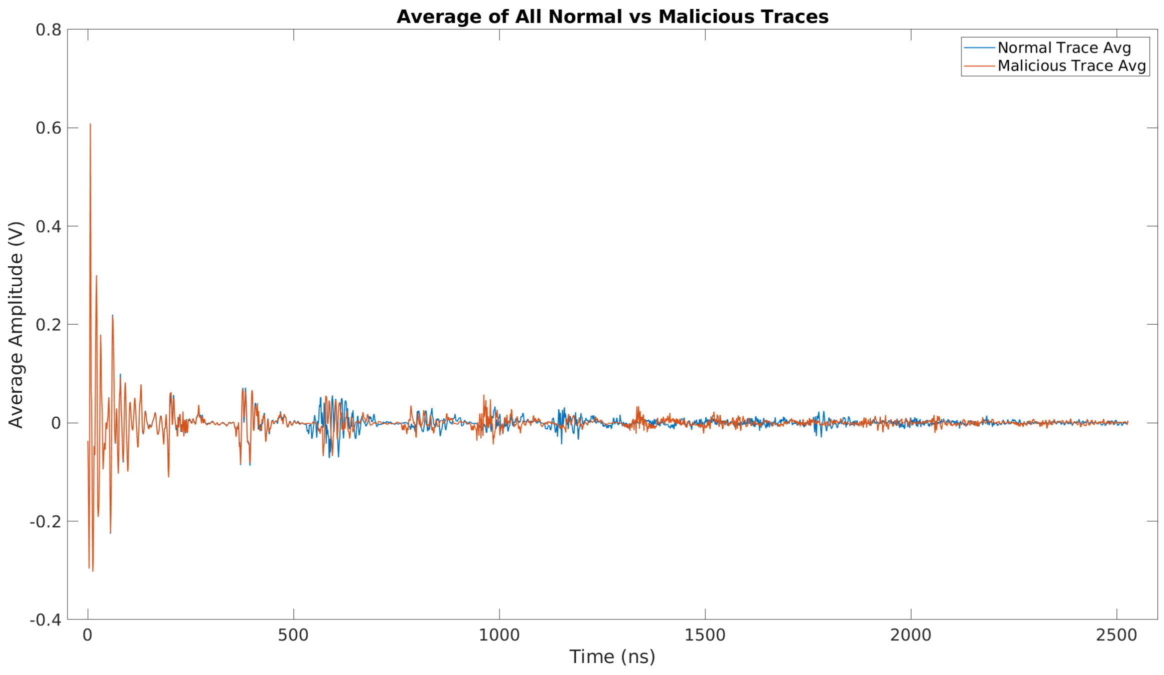


Figure 7: Plot of the average of all 25,000 normal and all 25,000 malicious traces

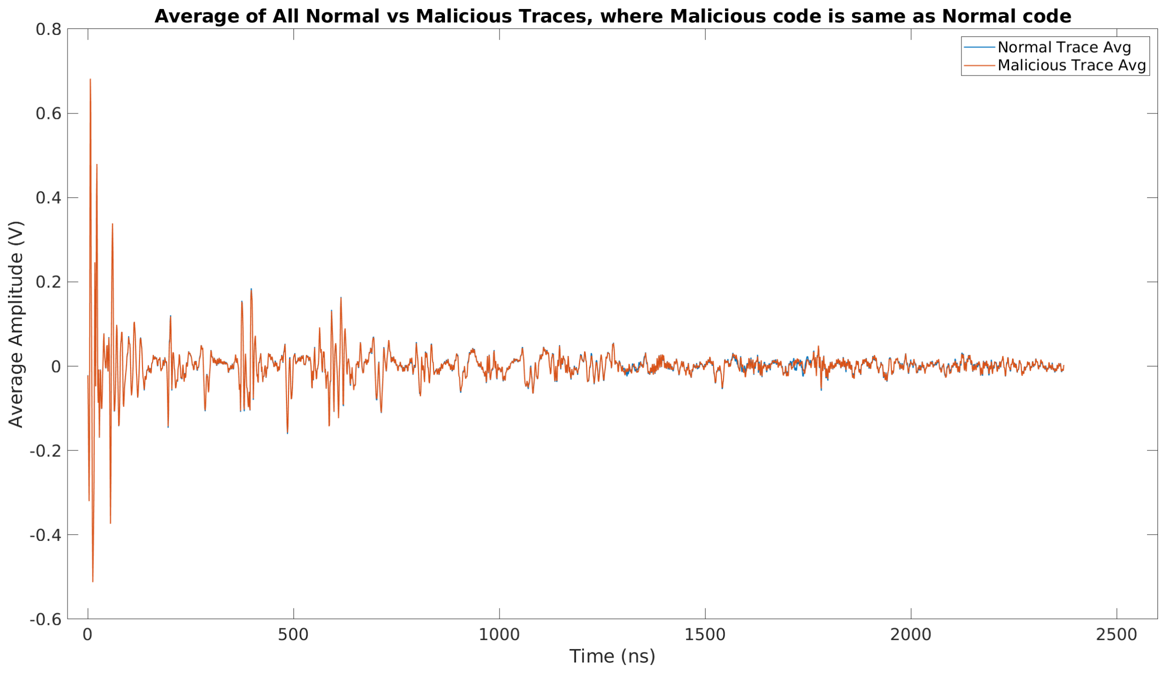


Figure 8: Plot of the average of all 25,000 normal and all 25,000 malicious traces when the code segments were the same

4.3 Statistical Analysis

With the traces collected, statistical analysis can now be applied to distinguish between the normal and malicious classes. In each of the statistical analysis efforts, the support vector classification (SVC) method was used to predict between classes, which could then be compared against the actual class to generate statistical results. In early testing, the linear discriminant analysis (LDA), quadratic discriminant analysis (QDA), and nu-support vector classification (NuSVC) classification methods from the Scikit-learn library were also tried, but the SVC method provided the best classification results. The statistical values for the initial phase of the experiment to determine whether anomalous activity can be detected are shown in Table 1. As seen in the table, the overall accuracy across all of the sets is 95.98%. Additionally, precision and recall also fall near that value, indicating that accuracy captures all relevant results. From this, we can determine that there is some detectable difference between the normal and malicious operations, allowing the additional efforts to move forward.

Table 1: Statistical Results for *Centered Probe* Location

Description	**Set 1**	**Set 2**	**Set 3**	**Set 4**	**Set 5**	**Mean of Sets**
Accuracy	95.54	96.54	95.94	95.88	95.98	95.98% ± 0.024
Precision	94.41	96.54	96.44	95.05	94.74	95.43% ± 0.066
Recall	96.74	96.50	95.39	96.67	97.24	96.51% ± 0.045

4.3.1 Varying Location

For the first location where the EM probe was centered over the Pi chip, the results from the initial phase. The statistical values for that experiment were shown in Table 1.

The statistical values for the experiment with the EM probe located over the side of the chip are shown in Table 2. As seen in the table, the overall accuracy across all of the sets is 82.33%, which indicates that there is still a detectable difference between the normal and malicious operations. Again, the precision and recall fall near the accuracy. Compared to the experiment with the probe centered over the chip, there is a drop-off in all of the statistical results. The results match the expectation that the centered location would have higher accuracy than the side location.

Table 2: Statistical Results for *Side Probe* Location

Description	**Set 1**	**Set 2**	**Set 3**	**Set 4**	**Set 5**	**Mean of Sets**
Accuracy	82.86	81.94	81.76	82.76	82.32	82.33% $\pm$ 0.032
Precision	82.19	82.61	80.25	82.18	82.64	81.98% $\pm$ 0.066
Recall	83.80	81.20	83.14	83.05	81.65	82.57% $\pm$ 0.073

The statistical values for the experiment with the EM probe off chip are shown in Table 3. As seen in the table, the overall accuracy across all of the sets for the off chip location is 81.00%, which still indicates there is a detectable difference between the operations. As before, precision and recall results are close to the accuracy, although the spread is larger for this experiment. Compared with the previous experiments, there is another drop-off in the statistical results, which matches the expectations for this location.

Table 3: Statistical Results for *Off Probe* Location

Description	**Set 1**	**Set 2**	**Set 3**	**Set 4**	**Set 5**	**Mean of Sets**
Accuracy	80.38	81.00	81.76	81.08	80.78	81.00% $\pm$ 0.034
Precision	77.04	81.47	82.62	81.74	79.30	80.43% $\pm$ 0.150
Recall	85.10	80.83	81.12	79.77	83.15	81.99% $\pm$ 0.142

The statistical values for the experiment with the EM probe just capturing air are shown in Table 4. As seen in the table, the overall accuracy across all of the sets is 50.74%. This indicates for this experiment that accuracy was as good as a coin flip to determine the difference between a normal and malicious operation. Since this experiment was not capturing any intended operations, this result is exactly what is expected.

Table 4: Statistical Results for *Air Probe* Location

Description	**Set 1**	**Set 2**	**Set 3**	**Set 4**	**Set 5**	**Mean of Sets**
Accuracy	50.94	51.22	49.86	50.90	50.80	$50.74\% \pm 0.035$
Precision	51.81	51.61	49.83	51.60	51.25	$51.22\% \pm 0.054$
Recall	44.63	50.24	46.22	45.37	48.89	$47.07\% \pm 0.160$

Each of the confidence intervals covering the accuracy for each of the four probe locations are plotted out in Figure 9. The confidence intervals fall within the boundaries of the markers in the image, with each interval plotted separately in Figure 10 to highlight each interval.

From the results and plot, it appears that the centered location provides the best distinction between normal and malicious operations, and should be the data

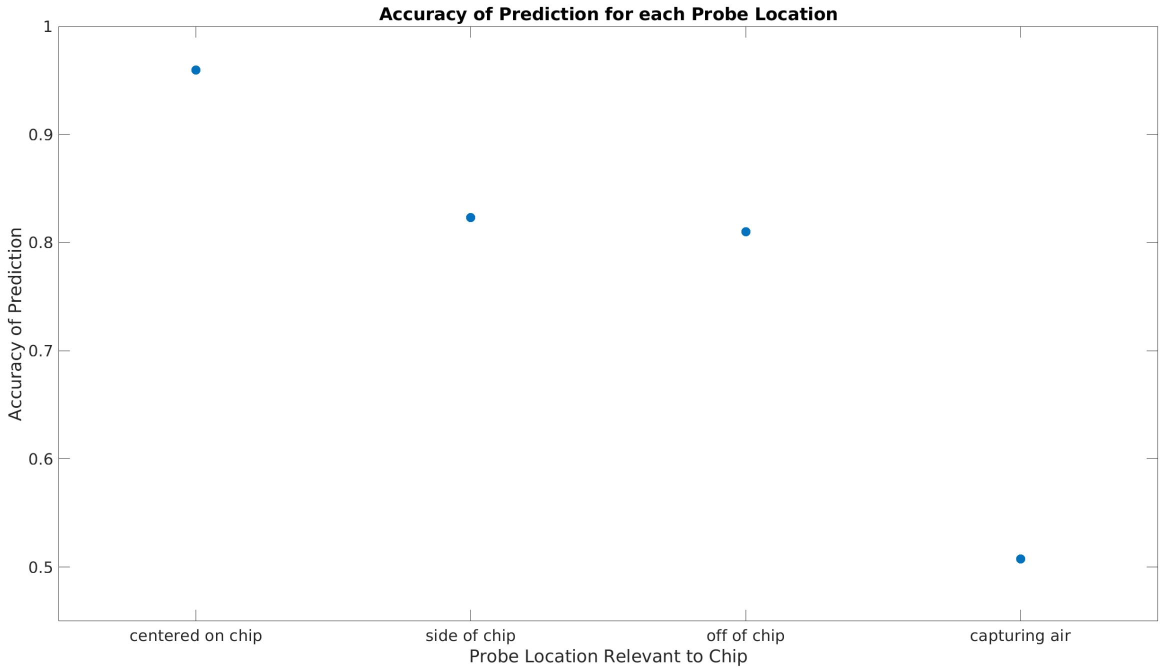
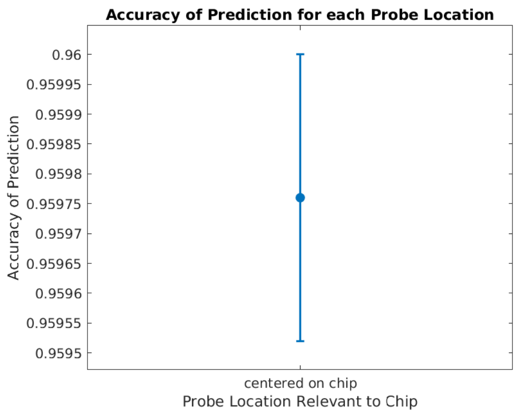
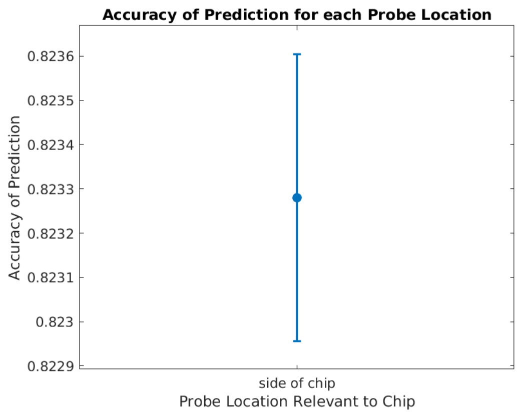


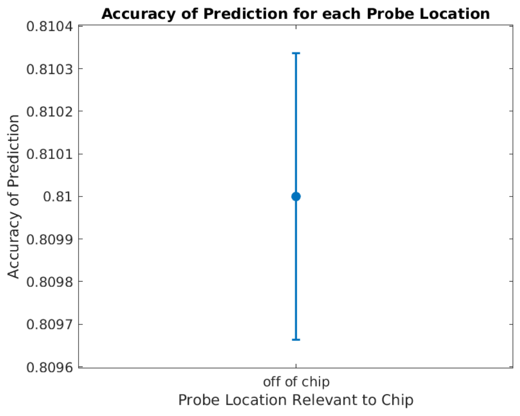
Figure 9: Means with confidence intervals for the accuracy of each of the four probe locations



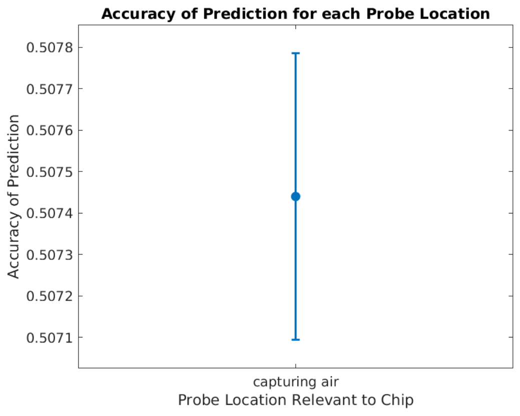
(a) Centered on chip



(b) Over side of chip



(c) Off of the chip



(d) Capturing air

Figure 10: Expanded means with confidence intervals for the accuracy of each of the four probe locations

set used for future experiments. In order to validate this, a t-test should first be conducted to verify that the accuracy results are statistically different. The t-test between the accuracy of the centered on chip location and the side of chip locations was $p = 2.6 \times 10^{-11}$, showing that these two accuracy results are statistically different. Conducting the t-test between the accuracy of the centered on chip and off of chip locations resulted in $p = 1.5 \times 10^{-11}$, also showing that the two accuracy results are statistically different. Thus, future experiments can continue using the centered on chip data set.

4.3.2 Splitting into Thirds

With the centered on chip data set, each of the traces are now split into three intervals, and the statistical values for each third are calculated. Due to the SVC method needing the same number of samples in the training group and test group, the previously trained models are not able to be used, so new models for each third are trained. The statistical values for the first third of the samples are shown in Table 5. As seen in the table, the overall accuracy of the first third is 97.23%. Of note, this is more accurate than the accuracy of all the samples in the trace, suggesting that performance of malware detection could potentially work better by just looking at the first third of the trace samples. This also indicates that the divergence of the code paths for the two operations occurs in the first third of the trace samples for this scenario.

Table 5: Statistical Results for *First Third* of Samples

Description	**Set 1**	**Set 2**	**Set 3**	**Set 4**	**Set 5**	**Mean of Sets**
Accuracy	96.40	97.42	97.38	97.80	97.14	97.23% $\pm$ 0.035
Precision	95.31	97.18	96.41	97.36	96.85	96.62% $\pm$ 0.055
Recall	97.71	97.69	98.45	98.29	97.50	97.93% $\pm$ 0.028

The statistical values for the middle third of the samples are shown in Table 6. As seen in the table, the overall accuracy of the middle third is 94.50%. Since it appears that the divergence of the code paths occurs in the first third, this accuracy matches the expectation that it is less than the accuracy of the first third. Also, since predictions are still fairly accurate, it indicates a carry-through effect after the code path divergence in the rest of the trace.

The statistical values for the last third of the samples are shown in Table 7. As seen in the table, the overall accuracy of the last third is 86.19%. This matches the expectation that the last third would be the least accurate. Although, there does still

Table 6: Statistical Results for *Middle Third* of Samples

Description	**Set 1**	**Set 2**	**Set 3**	**Set 4**	**Set 5**	**Mean of Sets**
Accuracy	93.76	94.74	94.88	94.78	94.32	$94.50\% \pm 0.031$
Precision	94.77	96.32	96.34	95.11	95.93	$95.70\% \pm 0.048$
Recall	92.82	93.06	93.35	94.47	92.66	$93.27\% \pm 0.048$

seems to be a detectable difference between normal and malicious operations, again indicating the carry-through effect.

Table 7: Statistical Results for *Last Third* of Samples

Description	**Set 1**	**Set 2**	**Set 3**	**Set 4**	**Set 5**	**Mean of Sets**
Accuracy	85.32	86.46	87.12	86.02	86.04	$86.19\% \pm 0.044$
Precision	86.30	87.46	88.82	86.11	88.70	$87.48\% \pm 0.086$
Recall	84.46	85.20	85.07	86.07	82.86	$84.73\% \pm 0.080$

Each of the confidence intervals covering the accuracy of the thirds are plotted out in Figure 11. Additionally, the confidence intervals for the overall accuracy of the data set is included for comparison. The confidence intervals fall within the boundaries of the markers in the image, with each interval plotted separately in Figure 12 to highlight the intervals.

From the results and plots, the different thirds appear to be different. To verify this, a t-test is conducted between each accuracy, like was done for the probe locations. The t-test between the accuracy of the first and middle thirds was $p = 2.2 \times 10^{-5}$. The t-test between the accuracy of the first and last thirds was $p = 2.0 \times 10^{-9}$. The t-test between the accuracy of the middle and last thirds was $p = 1.4 \times 10^{-8}$. Thus, the accuracy for each of the thirds are statistically different of the other thirds.

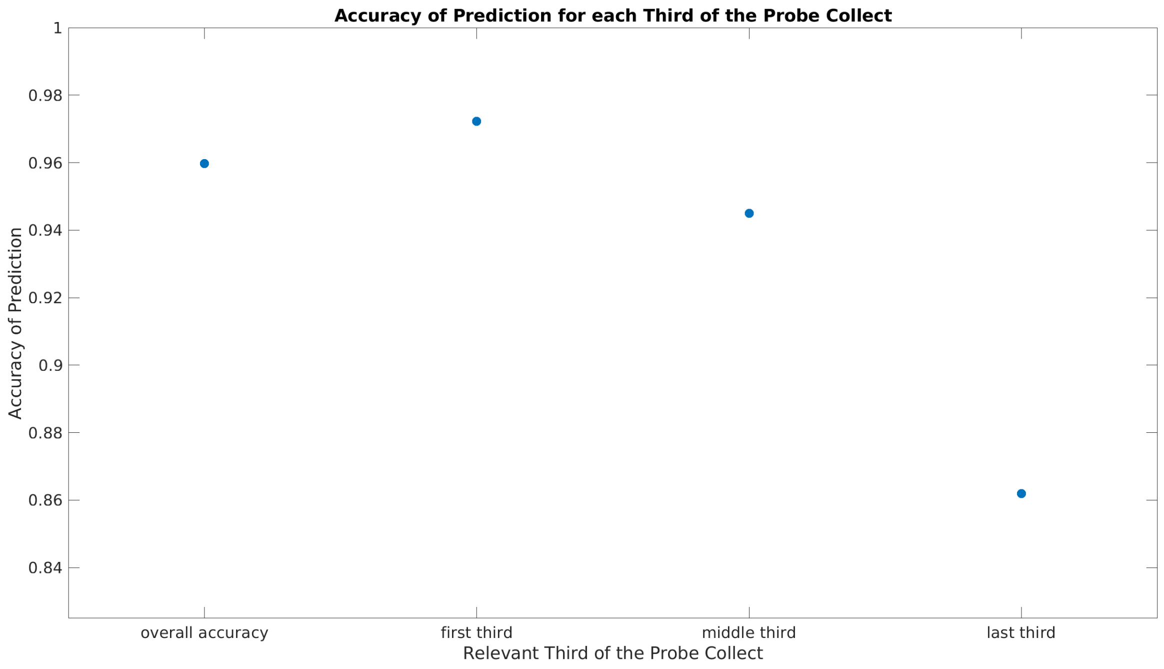
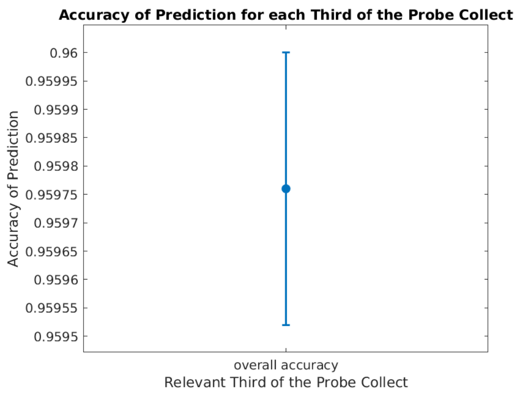


Figure 11: Means with confidence intervals for the accuracy of each of the thirds

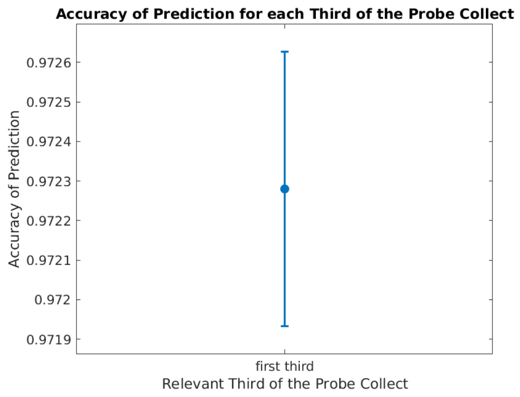
4.3.3 Sliding Window

Having examined the accuracy when the traces are split into three intervals, a window that steps across the traces can be created to attempt to further narrow the location where the code divergence occurs. While previous experiments cut out the first 100 samples of a trace, this experiment includes those beginning samples. The accuracy for a 100 sample window with a 50 sample step size are shown in Figure 13.

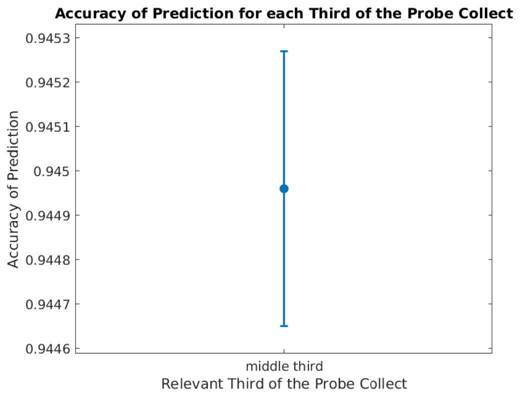
The accuracy for the window covering the first 100 samples are close to 50%, again justifying the decision to exclude those samples when running the previous experiments. Further examining the plot, it appears that the accuracy remains close to 50% for the first 400 samples, and the accuracy begins to rise after around 10 steps. Figure 14 shows the most accurate region from the previous figure. With the expanded plot, it is easier to see that the most accurate windows occur at samples 500-600, or after 10 steps, at samples 550-650, or after 11 steps, and at samples 600-700, or after 12 steps. This gives an overall range of samples 500-700 for the most



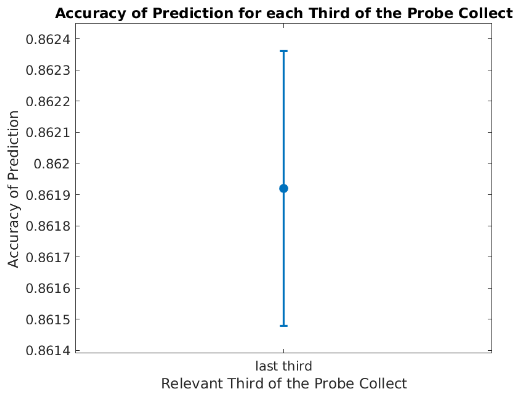
(a) Overall accuracy



(b) First third



(c) Middle third



(d) Last third

Figure 12: Expanded means with confidence intervals for the accuracy of each of the thirds

accurate region.

The window and sample size can be further reduced to 50 sample windows with a 25 sample step size to produce Figure 15. This plot with the smaller window shows a larger drop between the first higher accuracy region and the later high accuracy regions. It also shows that the areas between these higher accuracy regions trend closer towards 50%, which is not visible in the plot with the larger window size. It appears that the most accurate region begins after around 20 steps, which is shown in Figure 16. From the expanded plot, the region spans from samples 525-575 after 21 steps until samples 625-675 after 25 steps. This gives an overall range of samples 525-675 for the most accurate region.

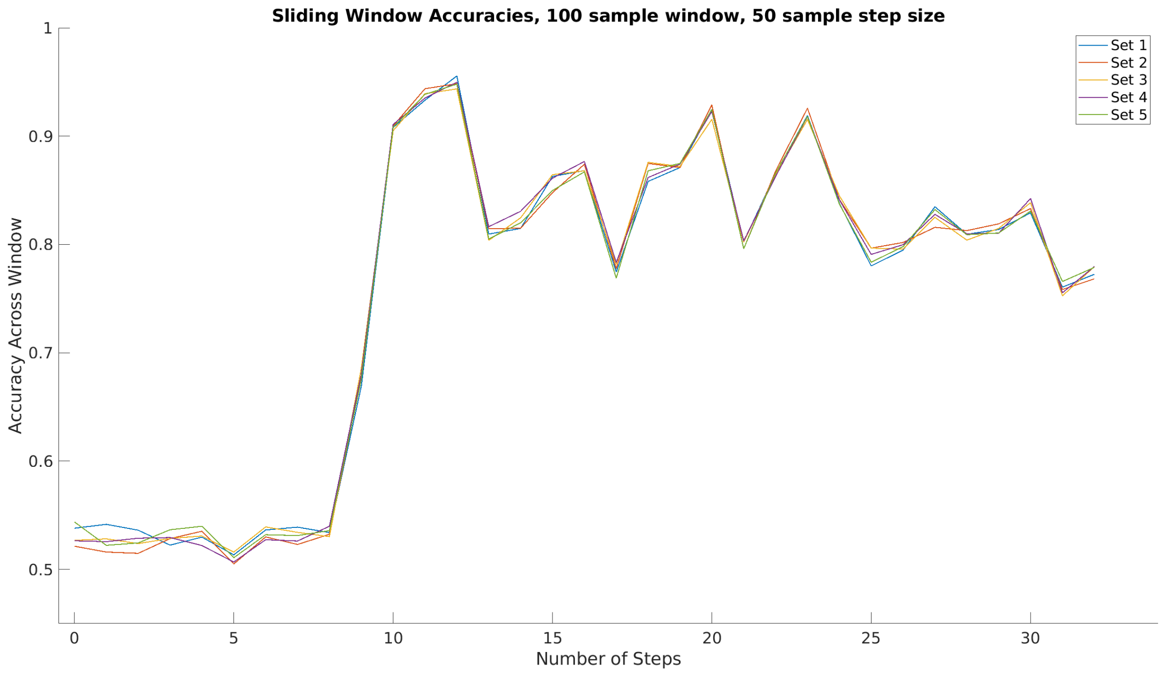


Figure 13: Accuracy of sliding window with a 100 sample window and a 50 sample step size

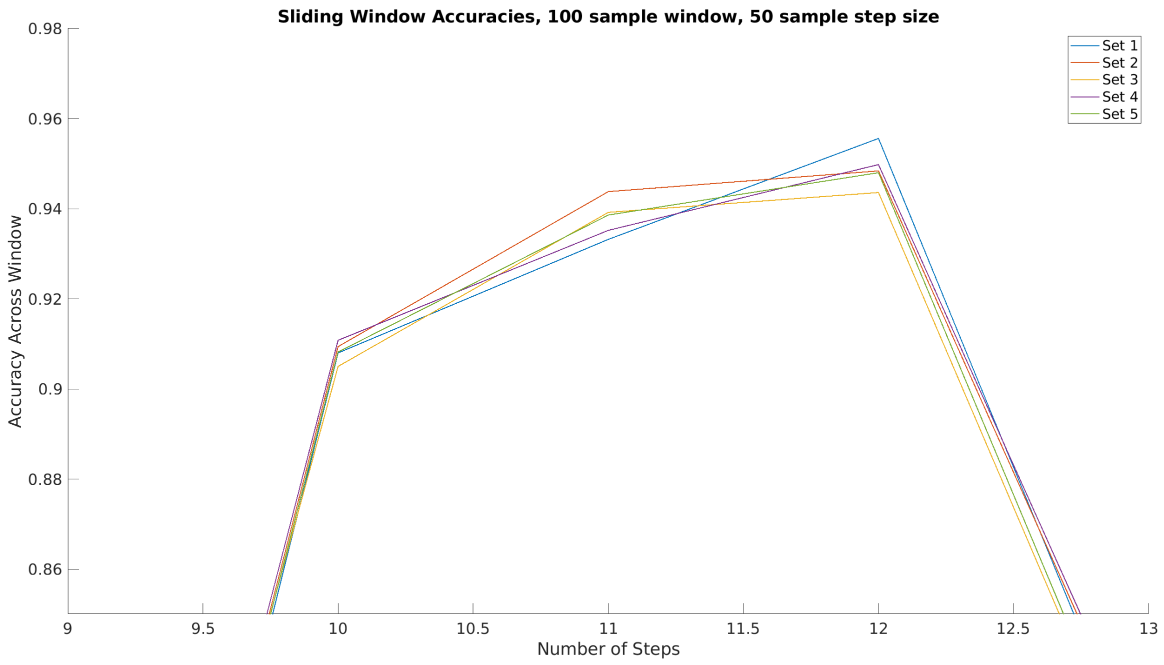


Figure 14: Most accurate windows from Figure 13 of 100 sample window and 50 sample step size

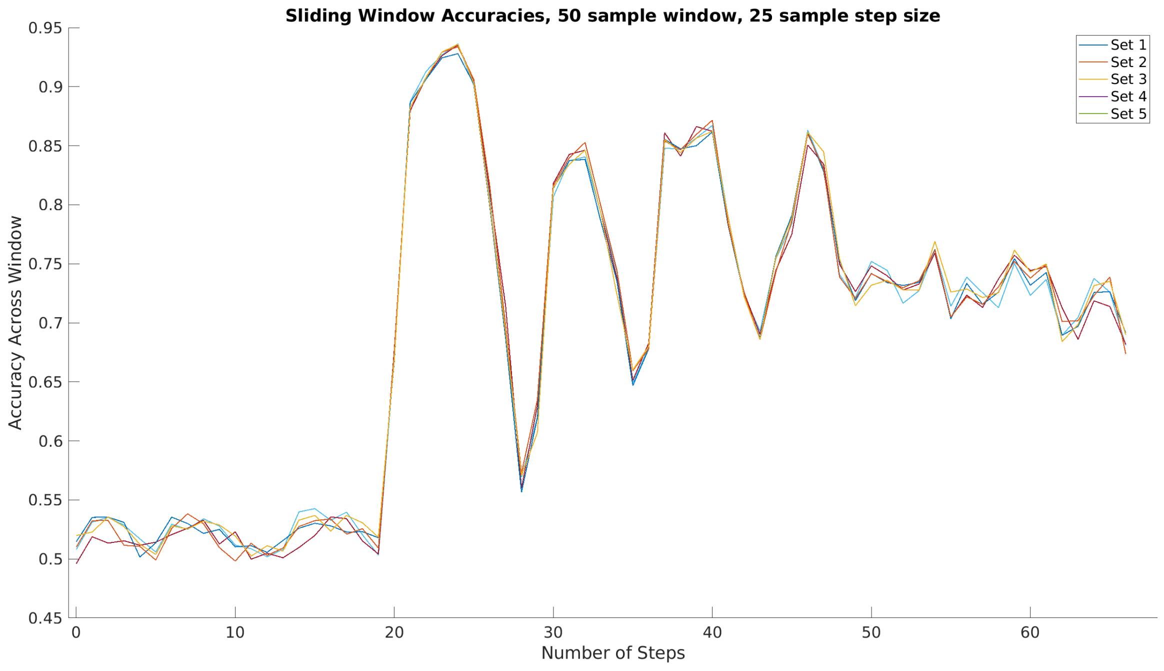


Figure 15: Accuracy of sliding window with 50 sample window and 25 sample step size

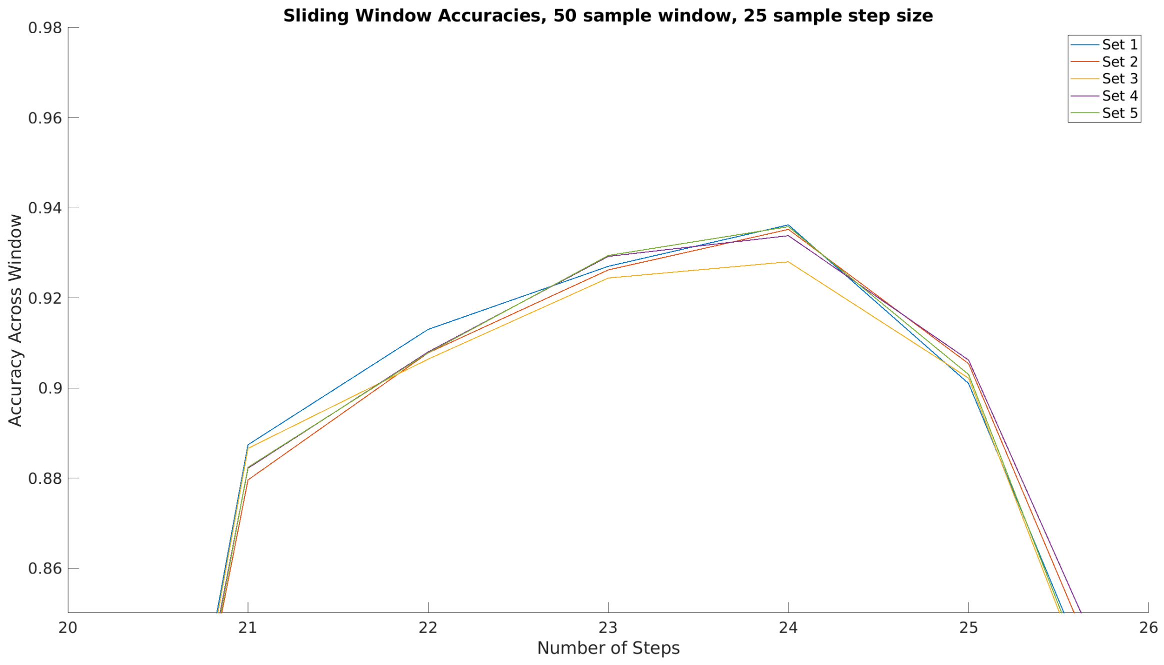


Figure 16: Most accurate windows from Figure 15 of 50 sample window and 25 sample step size

The determined range for the most accurate region from the sliding windows is now applied to the graph of the trace averages from Figure 7. Focusing on just the region around samples 500-700, shown in Figure 17, the traces show a noticeable difference from each other in this region. Beginning around sample 500-525, the two traces appear to fully separate from each other in a distinct manner that continues to around sample 700, where the difference between the traces significantly subdues, although it is still not as close of a match as before reaching this region.

From the sliding window and comparison back to the trace averages, it seems that the overall experiment could be repeated just around the sample 500-700 region and the statistical results should be the around the same. Thus, that was done, and the statistical results are shown in Table 8.

The 95.82% accuracy from just the most accurate sample region is only slightly off of the 95.98% accuracy from the overall accuracy. However this is less accurate than the 97.23% accuracy from the first third of samples, so the region could still be

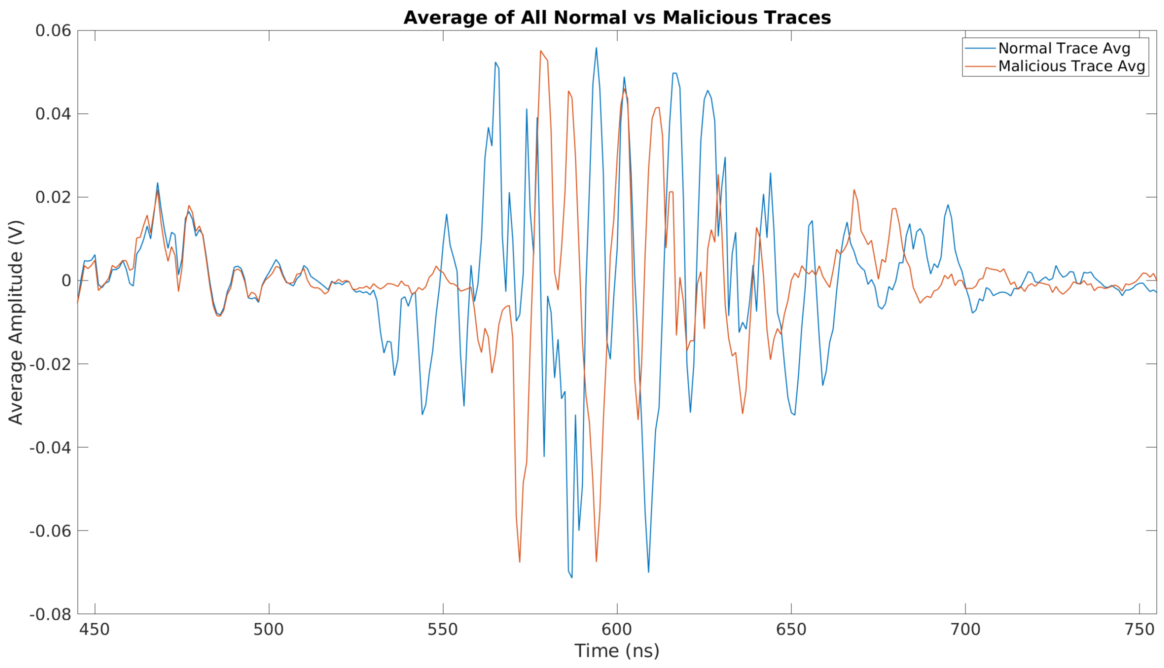


Figure 17: Trace averages around most accurate region from the sliding windows

Table 8: Statistical Results for Region of Samples 500-700

Description	**Set 1**	**Set 2**	**Set 3**	**Set 4**	**Set 5**	**Mean of Sets**
Accuracy	95.64	96.22	95.52	96.10	95.60	95.82% $\pm$ 0.021
Precision	95.75	96.03	95.41	96.16	94.50	95.57% $\pm$ 0.044
Recall	95.68	96.45	95.64	96.04	96.72	96.11% $\pm$ 0.032

tuned to improve accuracy results.

4.3.4 Mimicking Operational Environment

Although previous experiments show that normal and malicious operations can be distinguished from each other, they are not necessarily representative of an operational environment where there might only be a single malicious operation in the midst of countless normal operations. In order to attempt to emulate an operational environment, the probability predictions from the SVC method would be more beneficial than the normal prediction. An example of the predicted probabilities that the test traces in the first set represents normal is shown in Figure 18. As seen in the figure, most of the probabilities fall close to zero or one, which indicates that it would work as an emulated operational environment.

For the operational environment, 20 normal operations are run, then a single malicious operation, then 10 additional normal operations, which is repeated in five sets. The probability that each operation is normal is shown in Figure 19. Since the trials start at zero, the malicious operation corresponds with trial 20. From the figure, it appears that all of the malicious operations were correctly identified as malicious for all of the sets because none of the points at trial 20 are above 0.5.

Further focusing on just the trials that are identified as malicious as shown in Figure 20, where the probability is less than 0.5, false positives can also be examined to expand on the usability of this method. From the plot, there are five false positives across all of the sets, with one set having two false positives, and another set not

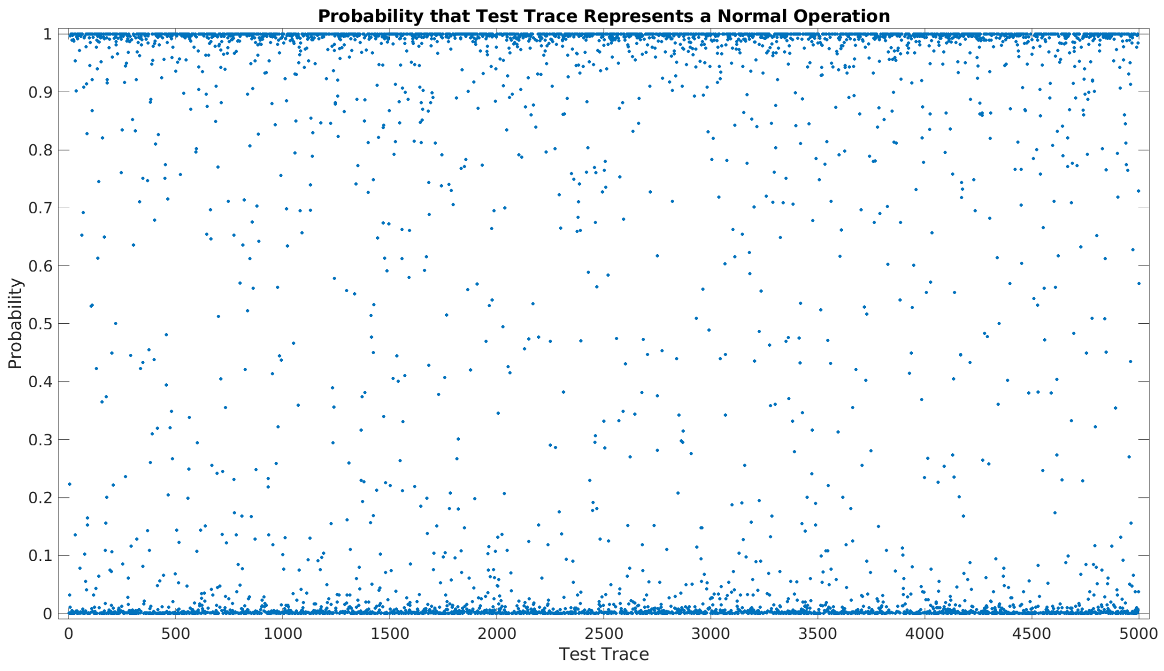


Figure 18: Probabilities that test data is a normal operation

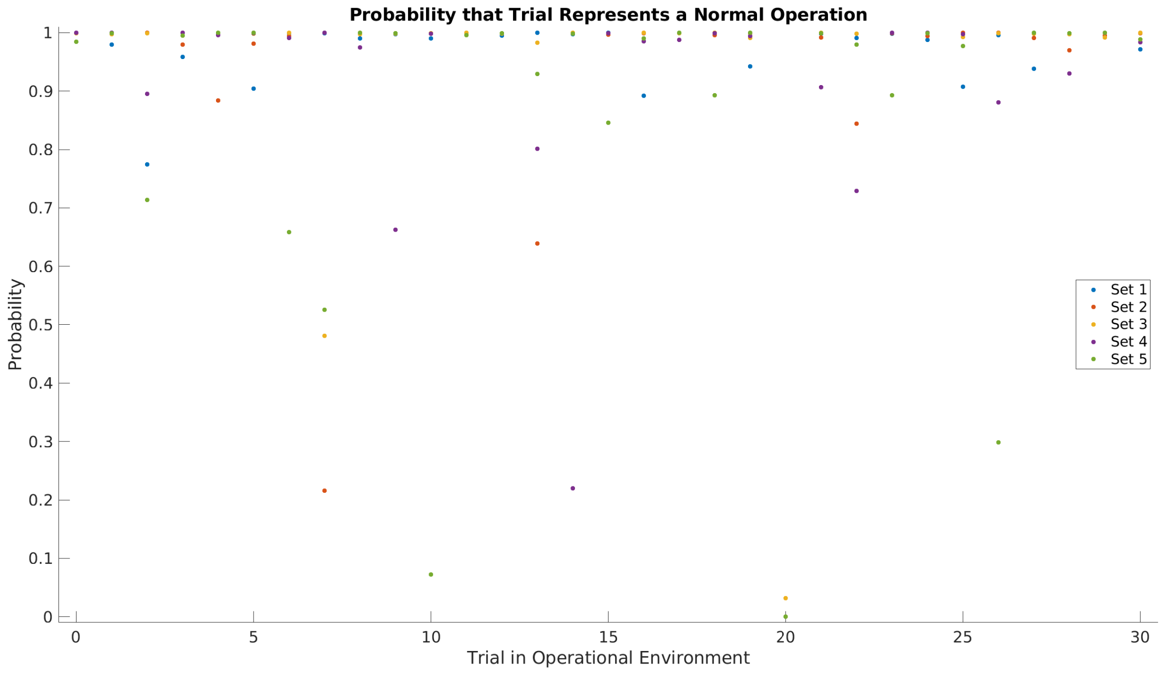


Figure 19: Probabilities that simulated operational event is a normal operation

having any false positives. Additionally, each of the false positives have a higher probability than the true positives, meaning there is less certainty that they are not normal. Thus in an operational environment, the probability levels could potentially be used as a factor in what type of check needs to occur in response to an event.

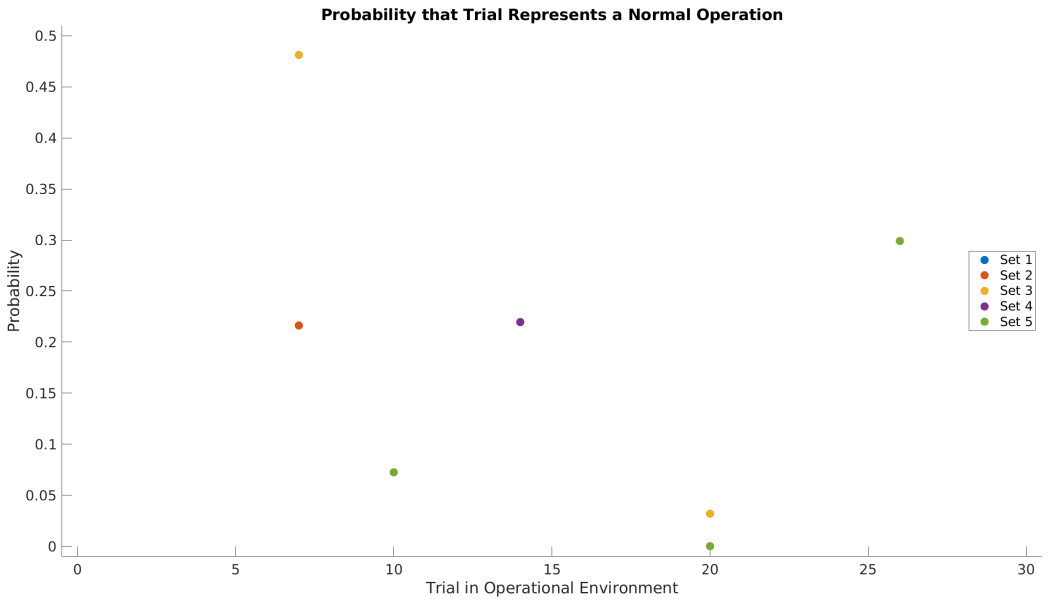


Figure 20: Probabilities less than 0.5 for simulated operational events

4.4 Summary

This chapter covers the results for the tests presented in Chapter III. For the varying EM probe locations, the location where the probe was centered over the chip performed the best, with a 95.98% classification prediction accuracy. When splitting the trace into three intervals, the first third achieved a best 97.23% classification prediction accuracy. By creating a window to step through the trace, the region for the best accuracy was found to be between samples 500-700 of the trace. When simulating an operational environment, all malicious operations were successfully predicted as malicious, although there were five false positives. However, this still shows that this type of malware detection scheme could work in an operational environment.

V. Conclusions

5.1 Overview

This chapter summarizes the results. Additionally, it discusses the significance of those results. Finally, the chapter will provide potential future areas of research that could be followed and investigated.

5.2 Research Conclusions

From testing the various probe locations relative to the Raspberry Pi chip, the location that was directly over the chip had the highest classification accuracy of 95.98%. Likewise, when splitting the traces into three intervals, the first third had the highest classification accuracy of 97.23%, while the middle third wasn't too far behind, with a classification accuracy of 94.50%. In conducting a sliding window test through the traces, the region contained between samples 500-700 was found to yield the highest classification accuracy, and by reducing the tests down to just this region, the classification accuracy was 95.82%. While this accuracy is comparable to the accuracy of the overall trace, it falls short of the accuracy of just the first third, showing that other trace information can help contribute to a higher accuracy. Lastly, utilizing the probability prediction results in a probability that an individual trace is either a normal operation or a malicious operation, which could be used and tuned in an operational environment to determine whether an operation needs to be looked into.

5.3 Significance

The results of the tests show that it is possible to detect malware using electromagnetic (EM) side-channel analysis (SCA). Furthermore, the tests show that splitting up the trace can produce a higher classification accuracy than using just the overall trace. The most accurate results would likely come from identifying where the split between normal and malicious might occur and utilizing the trace from that point on for classification. The location where the split might occur could be determined by understanding how an attacker could target the system and create an effect. Additionally, the tests using the probability prediction show that these methods could be viable in an operational environment as a means of malware detection.

5.4 Future Work

There are several paths that future research could follow. These possible directions are presented below.

- Increase the distance between the probe and the chip. The current setup has the probe close to the chip, which likely would not be possible in an operational environment. Thus, future research could look at extending the distance between the probe and the chip while still maintaining an accuracy above 90%.
- Expand the number of malicious operations being tested for. In this research, the only operations were normal and a single operation that was classified as malware. However, in the real world, there may be multiple different families of malware that could potentially target the same device. Thus, future research could examine adding another malware operation and look into classification results that distinguish between each operation.
- Focus on detecting a specific malware. While malware operation used in this research may have been inspired by a real world malware, it was not in fact a real malware. Thus, future research could look at setting up the experiment to use an actual malware sample for the malware operation.

- Expand on the use of the probability prediction. The probability prediction capability seems like it would be more useful operationally than direct classification, since it provides a score on anomalous activity. Thus, future research could focus specifically on the probability prediction for classification to fully explore it capabilities.

- Attempt to determine some sort of trace signature that could be pulled from the research. This research performed a sort of anomaly based malware detection, which requires being able to test a device that is free of malware in order to build the normal profile. However, in order to actually implement this setup, it would be beneficial to have a trace signature for malware. With that, the setup could be placed with a device that is already in use, and it would still be able to function without conducting a baseline to build the normal profile. In this case, malware could potentially be detected even if it were already on a device prior to the setup being installed.

Appendix A. Additional Results

While going through the initial testing, it seemed that the average of the traces provided a good indication of whether the two operations would be distinguishable as well as where in the trace those differences began. Comparing the averages from Figure 7, where the code segments are different, and Figure 8, where the code segments are the same, it can be seen how the averages show the operations separating in the traces. Thus, the code for the malicious operation was altered and the average traces recomputed in order to see how the new code might affect the averages.

First, the subtraction for the revolutions per minute (RPM) calculation was moved from being on its own line to be on the same line as where it was initially calculated. The average traces plot for this situation is shown in Figure 21. As can be seen in the plot, there is a split in the averages. Additionally, this split still occurs around sample 500 like previously, suggesting the compiler has created the machine code in basically the same manner.

Next, the subtraction for the RPM calculation was moved to be in the section that checks if the value is too fast or too slow. The average traces plot for this situation is shown in Figure 22. As can be seen in the plot, there is a split in the averages that also appears to occur around sample 500.

After these tests, the RPM subtraction location was reset, but the value being subtracted was changed to be much smaller to see if that would affect the outcome. The average traces plot for this situation is shown in Figure 23. As can be seen in the plot, the split between the averages is still present. This indicates that the presence of an additional machine code instruction is being picked up on.

Lastly, the RPM subtraction for the malicious operation was kept, but the code for it was also added into the normal operation. So the only difference between the two code segments was the size of the value being subtracted, which for the normal

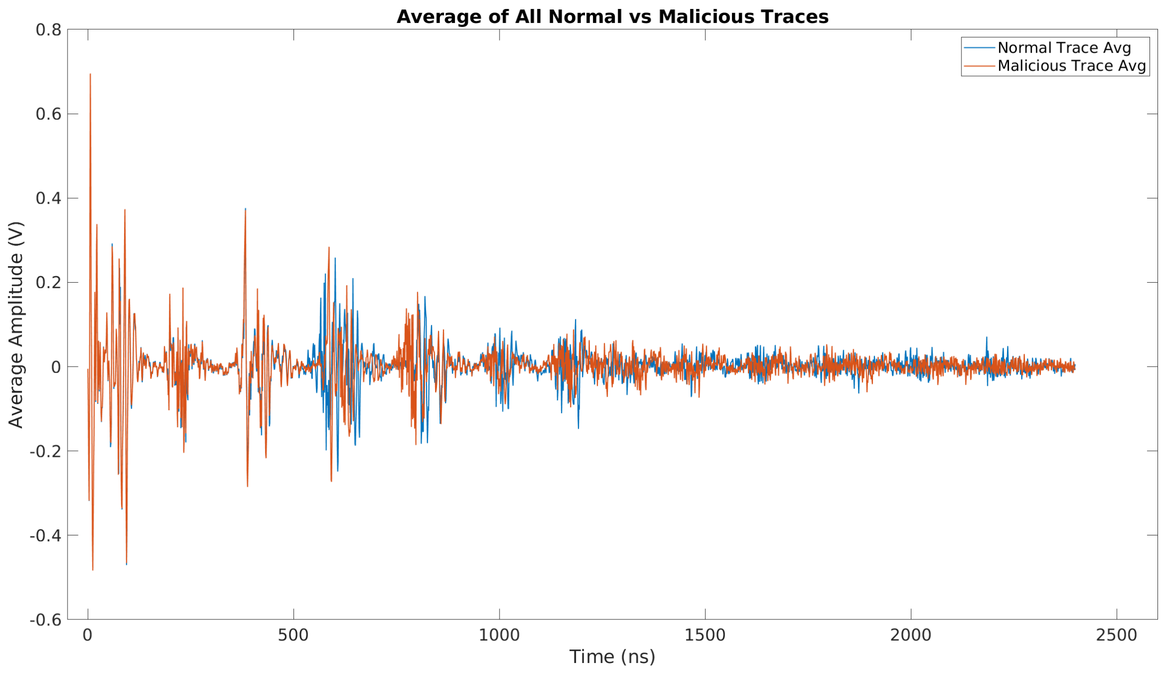


Figure 21: Plot of the average of all 25,000 normal and all 25,000 malicious traces when the RPM subtraction is done on the same line as the initial calculation

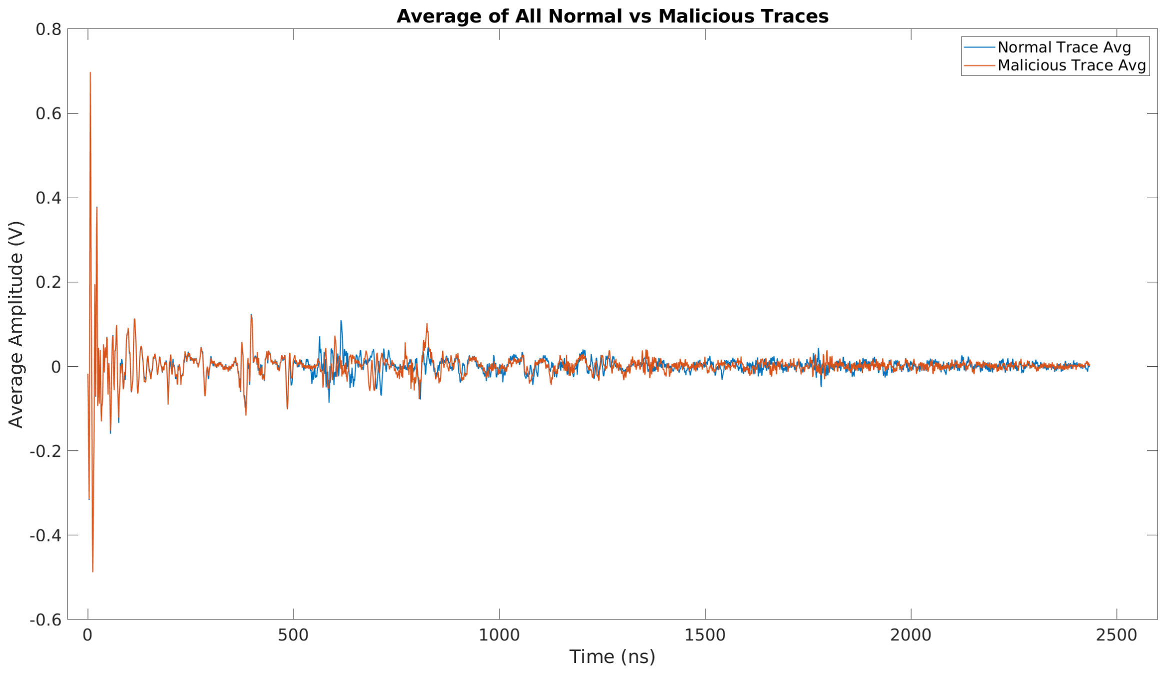


Figure 22: Plot of the average of all 25,000 normal and all 25,000 malicious traces when the RPM subtraction is done in the check to update the motor speed

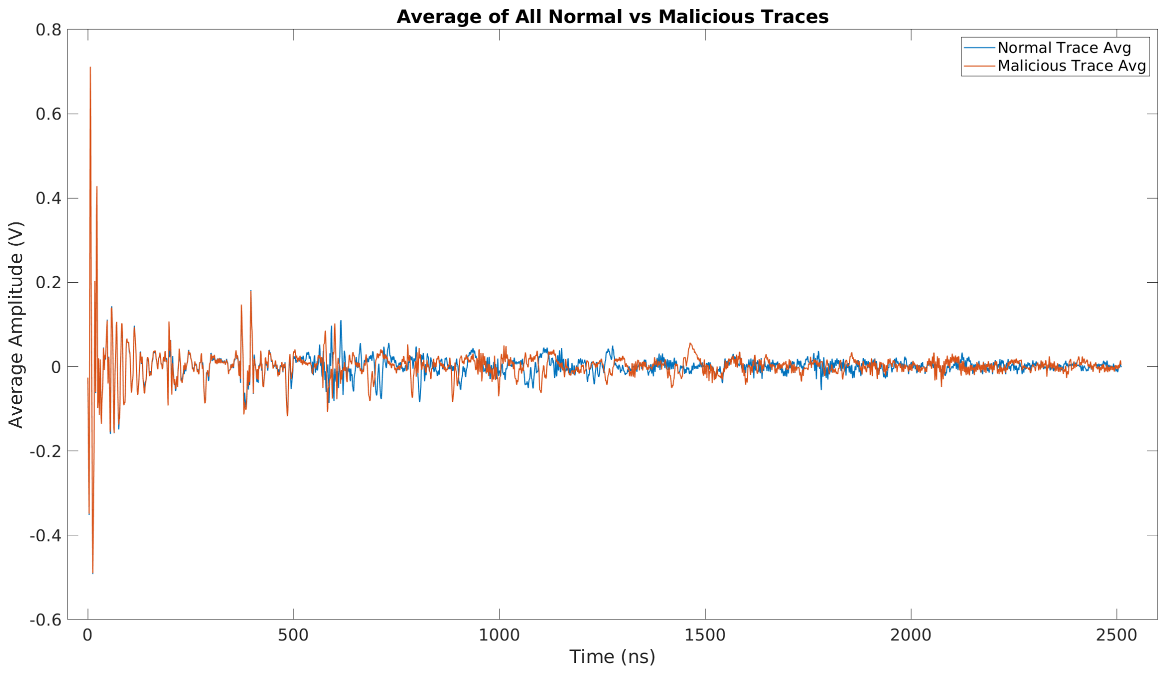


Figure 23: Plot of the average of all 25,000 normal and all 25,000 malicious traces when the RPM subtraction is of a smaller value

operation was 1 and for the malicious operation was 3. The average traces plot for this situation is shown in Figure 24. As can be seen in the plot, there is still a split between the two averages. However, there is much more overlap at later points than in the other plots.

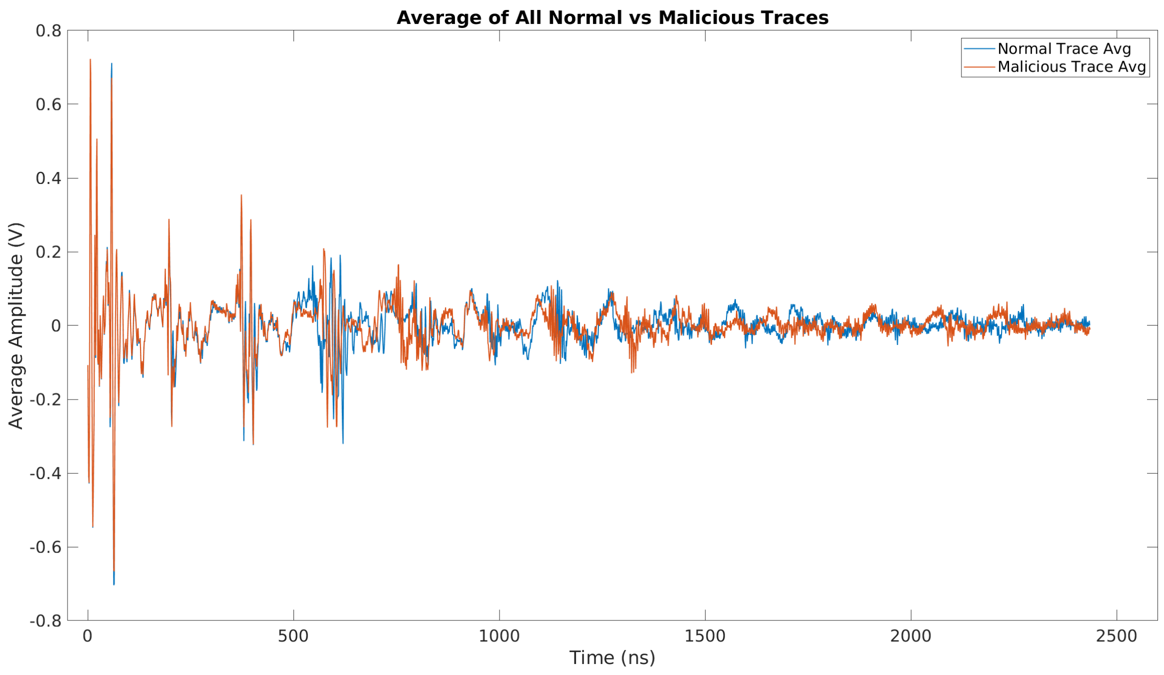


Figure 24: Plot of the average of all 25,000 normal and all 25,000 malicious traces when the RPM subtraction is done in both the normal and malicious operations

Appendix B. Main C Code

```c
#include "uart.h"
#include "simOp.h"
#include "gpio.h"

/* PWM registers */
#define PWM_CTL         ((volatile unsigned int*)(MMIO_BASE+0
   x0020C000))
#define PWM_RNG1        ((volatile unsigned int*)(MMIO_BASE+0
   x0020C010))
#define PWM_DAT1        ((volatile unsigned int*)(MMIO_BASE+0
   x0020C014))

#define CM_PWMCTL       ((volatile unsigned int*)(MMIO_BASE+0
   x001010A0))
#define CM_PWMDIV       ((volatile unsigned int*)(MMIO_BASE+0
   x001010A4))

void main()
{
  // initiallize character for operation
  char op;

  // initialize serial console
  uart_init();

  uart_puts("Beginning program\n");

  // start motor before running programs
  register unsigned int r;
  volatile unsigned int* p;
  volatile unsigned int* q;
  // set up clock for pwm
  *PWM_CTL = 0;
  p = CM_PWMCTL;
  q = CM_PWMDIV;
  *p |= (0x5A << 24)|(0 << 4);
  while(*p & (1 << 7)); // wait for clock to have stopped
  *q = (0x5A << 24)|(32 << 12)|(0 << 0); // 19.2MHz divided by 32 (val shifted 12) . 0 (val shifted 0)
  *p = (0x5A << 24)|(1 << 0)|(1 << 4); // the |(1<<4) needs to be on  this line to work
//  *p |= (1 << 4);
  while(!(*p & (1 << 7))); // wait for clock to have started
  // start generating pwm using gpio 12
  *PWM_RNG1 = 96; // further divide the 19.2MHz/(val shifted 12) by this to get frequency of pwm signal
  *PWM_DAT1 = 60; //58-67 gives 60-70% duty cycle, 25% - 24, 50% - 48, 75% - 72, 80% - 77
  r = *GPFSEL1; // taken from gpio pin int divide by 10
  r &= ~(7 << 6); // taken from (gpio % 10) * 3, clears those bits
  r |= (4 << 6); // sets mode to altfunc0 for pin (which is pwm0 on gpio 12)
  *GPFSEL1 = r;
  *PWM_CTL = (1 << 7)|(1 << 0);
  // set up gpio 16 for input and to look for rising edge
  r = *GPFSEL1; // taken from gpio pin int divide by 10
  r &= ~(7 << 18); // taken from (gpio % 10) * 3, clears those bits
  r |= (0 << 18); // sets mode to input for pin
  *GPFSEL1 = r;
  *GPREN0 |= (1 << 16); // sets gpio 16 to modify GPEDS on rising edge

```

```
// wait 20 cycles to try to give the motor time to reach the set
 speed
for(int k = 0; k < 20; k++)
{
  while(!(*GPEDS0 & (1 << 16)));
  *GPEDS0 |= (1 << 16);
}

// read input characters
while(1)
{
  // get input off uart
  op = uart_getc();
  // if 1, run malware operation, if 0, run normal operation
  if(op == '1') //not sure if I need to do character comparison
  {
    uart_puts("\n");
    uart_puts("Running malware operation\n");
    malOp();
    uart_puts("Malware op complete\n");
  }
  else if(op == '0') //might want to have as else if to account
 for potential error inputs that aren't 1 or 0
  {
    uart_puts("\n");
    uart_puts("Running normal operation\n");
    normOp();
    uart_puts("Normal op complete\n");
  }
  else if(op == 'q') //allow user choice of exiting
  {
    uart_puts("\n");
```

```
      uart_puts("Quitting program\n");
      break;
    }
    else if(op == 'r') //provide option to reset motor speed
    {
      uart_puts("\n");
      uart_puts("Resetting motor speed\n");
      *PWM_DAT1 = 60;
    }
    // in case want to account for possible input errors on device
   side
    else
    {
      uart_puts("Input error. Value not a 0 or 1, or q to stop.\n");
    }
    // reset the starting motor speed
    //*PWM_DAT1 = 60; // use if running multiple rounds in one test
  }
  // turn off motor
  *PWM_CTL &= ~((1 << 0)|(1 << 8));
  uart_puts("Program complete\n");
  while(1);
}
```

Appendix C. Simulated Operation Code

```
#include "uart.h" // for debugging
#include "gpio.h" // needed for the trigger

/* PWM registers */
#define PWM_CTL         ((volatile unsigned int*)(MMIO_BASE+0
   x0020C000))
#define PWM_STA         ((volatile unsigned int*)(MMIO_BASE+0
   x0020C004))
#define PWM_DMAC        ((volatile unsigned int*)(MMIO_BASE+0
   x0020C008))
#define PWM_RNG1        ((volatile unsigned int*)(MMIO_BASE+0
   x0020C010))
#define PWM_DAT1        ((volatile unsigned int*)(MMIO_BASE+0
   x0020C014))
#define PWM_FIF1        ((volatile unsigned int*)(MMIO_BASE+0
   x0020C018))
#define PWM_RNG2        ((volatile unsigned int*)(MMIO_BASE+0
   x0020C020))
#define PWM_DAT2        ((volatile unsigned int*)(MMIO_BASE+0
   x0020C024))

#define CM_PWMCTL       ((volatile unsigned int*)(MMIO_BASE+0
   x001010A0))
#define CM_PWMDIV       ((volatile unsigned int*)(MMIO_BASE+0
   x001010A4))
#define SYS_TIME_CLO    ((volatile unsigned int*)(MMIO_BASE+0
   x00003004))
volatile int rpm = 0;

void normOp()
```

```c
{
  volatile int timer_first = 0;
  volatile int timer_next = 0;
  volatile int period = 0;

  register unsigned int r;
  volatile unsigned int* p;

  // trigger for scope capture, using gpio 26
  r = *GPFSEL2; // 2 taken from gpio pin int divide by 10
  r &= ~(7 << 18); // 18 taken from (gpio % 10) * 3, clears those
   bits
  r |= (1 << 18); // sets mode to output for pin
  *GPFSEL2 = r;

  // wait for rising edge
  while(!(*GPEDS0 & (1 << 16)));
  // get system time
  timer_first = *SYS_TIME_CLO;
  *GPEDS0 |= (1 << 16);
  r=5; while(r--) { asm volatile("nop"); }

  // wait for next rising edge
  while(!(*GPEDS0 & (1 << 16)));
  // get next system time
  timer_next = *SYS_TIME_CLO;
  *GPEDS0 |= (1 << 16);

  // set trigger on gpio 26 high
  p = GPSET0;
  *p = (1 << 26);

```

```c
// calculate difference between two times, which should give
 period of 1 cycle
period = timer_next - timer_first;

// motor cycles 12 times for a full rotation
period = 12 * period;
// timer, and thus period, is in microseconds, so need to convert
 to minutes
// because of int math, do conversion when calculating rpm
// rpm is effectively 1 / period, account for conversion to
 minutes in numerator
rpm = 1000000 * 60 / period; // -1 for double

// check if the rpms are within a set value
if(rpm < 2500) // period of 2000 us
{
  // adjust rpm to be faster
  if(*PWM_DAT1 < 85) // avoid duty cycle >90%
  {
    *PWM_DAT1 += 2;
  }
}
else if(rpm >= 10000)
{
  // situation where rpm calculation definitely wrong, so don't
 change anything
}
else if(rpm > 5000) // period of 1000 us
{
  // adjust rpm to be slower
  if(*PWM_DAT1 > 25) // avoid duty cycle <25%
  {
      *PWM_DAT1 -= 2;
    }
  }

  // set trigger low and turn off pwm
  p = GPCLR0;
  *p = (1 << 26); // sets gpio 26 low
}

void malOp()
{
  volatile int timer_first = 0;
  volatile int timer_next = 0;
  volatile int period = 0;

  // trigger for scope capture, using gpio 26
  register unsigned int r;
  volatile unsigned int* p;
  r = *GPFSEL2; // 2 taken from gpio pin int divide by 10
  r &= ~(7 << 18); // 18 taken from (gpio % 10) * 3, clears those
   bits
  r |= (1 << 18); // sets mode to output for pin
  *GPFSEL2 = r;

  // wait for rising edge
  while(!(*GPEDS0 & (1 << 16)));
  // get system time
  timer_first = *SYS_TIME_CLO;
  *GPEDS0 |= (1 << 16);
  r=5; while(r--) { asm volatile("nop"); }

  // wait for next rising edge
```

```c
while(!(*GPEDS0 & (1 << 16)));
// get next system time
timer_next = *SYS_TIME_CL0;
*GPEDS0 |= (1 << 16);

p = GPSET0;
*p = (1 << 26);

// calculate difference between two times, which should give
 period of 1 cycle
period = timer_next - timer_first;

// motor cycles 12 times for a full rotation
period = 12 * period;
// timer, and thus period, is in microseconds, so need to convert
 to minutes
// because of int math, do conversion when calculating rpm
// rpm is effectively 1 / period, account for conversion to
 minutes in numerator
rpm = 1000000 * 60 / period; // -500 here for one line; -3 for
 double

// report that rpm is slower than calculated
rpm = rpm - 500;
//rpm = rpm - 2; // for smaller sub

// check if the rpms are within a set value
if(rpm < 2500) // period of 2000 us; sub 500 from rpm here for
 other check
{
  // adjust rpm to be faster
  if(*PWM_DAT1 < 85) // avoid duty cycle >90%
```

```
    {
      *PWM_DAT1 += 2;
    }
  }
  else if(rpm >= 10000)
  {
    // situation where rpm calculation definitely wrong, so don't
   change anything
  }
  else if(rpm > 5000) // period of 1000 us; sub 500 from rpm here
   for other check
  {
    // adjust rpm to be slower
    if(*PWM_DAT1 > 25) // avoid duty cycle <25%
    {
      *PWM_DAT1 -= 2;
    }
  }
  // set trigger low
  p = GPCLR0;
  *p = (1 << 26); // sets gpio 26 low
}
```

Appendix D. Experiment Collection Code

```
import ivi
import numpy as np
#import re
import pylab
import time
import serial
import scipy.io as sio

t = time.localtime()
current_time = time.strftime("%H:%M:%S", t)
print(current_time)

# create variable for number of each type of trial
num_trial = 25000
# set up scope to be able to communicate with it
scope = ivi.lecroy.lecroyWR104XIA("TCPIP0::192.168.1.11::inst0::
    INSTR")
# create array of random 1's and 0's to send to device
#opType = np.random.randint(2,size=(15000)) # increase to above
    10000
# create array of 1's and 0's, then randomize order to send to
    device
ops = np.zeros(num_trial, dtype=int)
ops = np.append(ops, np.ones(num_trial, dtype=int), axis=0)
opType = []
randperm = np.random.permutation(num_trial*2)
for i in randperm:
    opType.append(ops[i])
opType = np.array(opType)
# initialize arrays to hold trace data for the normal and malware
```

```
   operations
normOp = []
malOp = []
# initialize variables to hold the minimum rising and maximum
   falling edge indices of the trigger
rise_min = 0
fall_max = 0
# initialize serial connection
ser = serial.Serial('/dev/ttyUSB0', 115200) # verify device to use
   as port, based on what usb-to-serial cable connects as

for op in opType:
    # rearm the scope for a new trace capture before each
   transmission to the device
    scope.measurement.initiate()

    # initialize variables for rising and falling edges of each
   trial
    rise = 0
    fall = 0

    # send the operation type to the device to run either the normal
    or malicious operation
    # tests with variable weren't successful, so using if statements
    if op == 0:
      ser.write(b'0')
    elif op == 1:
      ser.write(b'1')
    else:
      print("Error with sending operation type")
      time.sleep(1)
      exit()

    # capture the trace data points from the scope
    waveform = scope.channels[0].measurement.fetch_waveform()
    # capture the trigger data points to narrow the window of
   interest
    trigger = scope.channels[2].measurement.fetch_waveform()
    # calculate the rough indices where the trigger rises and falls
    for i,volt in enumerate(trigger[1]):  # find min rise and max
   fall to give each event same number
        if volt > 1.5 and rise == 0:
            rise = i - 1
        elif volt < 1 and rise != 0 and fall == 0:
            fall = i + 1
    # determine if need to update minimum rising edge and maximum
   falling edge
    if rise < rise_min or rise_min == 0:
        rise_min = rise
    if fall > fall_max:
        fall_max = fall
    # based on the value that was sent, add the points that were
   just captured to the appropriate array
    if op == 0:
#         normOp.append(points)
        normOp.append(waveform)
    elif op == 1:
#         malOp.append(points)
        malOp.append(waveform)
    else:
        # if there was some issue with the value that was sent,
   print an error message and exit
        print("Error with sending operation type")
        time.sleep(1)
        exit()
# restrict the captures to just between the edges from the trigger
for i,op in enumerate(normOp):
    op = [op[j][rise_min:fall_max] for j in range(2)]
    normOp[i] = op
for i,op in enumerate(malOp):
    op = [op[j][rise_min:fall_max] for j in range(2)]
    malOp[i] = op
# convert the arrays to numpy arrays
normOp = np.array(normOp)
malOp = np.array(malOp)

# calculate and print out number of normal and malicious operations
numNorm = int(np.size(normOp)/np.size(normOp[0]))
numMal = int(np.size(malOp)/np.size(malOp[0]))
print(numNorm, "normal operations,", numMal, "malicious operations")

# save the numpy arrays for use elsewhere
#np.save('normOp.npy', normOp)
#np.save('malOp.npy', malOp)

# set up dictionary to export as mat file for use in matlab
#norm = np.delete(normOp, 0, 1)
#mal = np.delete(malOp, 0, 1)
#norm = np.reshape(norm, (num_trial, int(np.size(norm[0]))))
#mal = np.reshape(mal, (num_trial, int(np.size(mal[0]))))
#mdic = {}
#mdic['norm_same'] = norm
#mdic['mal_same'] = mal
#sio.savemat('same_sub.mat', mdic)

print("Experiment complete")
```

```python
t = time.localtime()
current_time = time.strftime("%H:%M:%S", t)
print(current_time)
```

Appendix E. Machine Learning Code

```python
import numpy as np
import scipy.stats as stat
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
    as LDA
from sklearn.discriminant_analysis import
   QuadraticDiscriminantAnalysis as QDA
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.svm import NuSVC
import sys
import scipy.io as sio

sample_size = 100  # num samples per division for the fingerprinting
exp_size = 0  # will hold the minimum between number of malicious
   and normal operations
num_tests = 5 # number of trials to split the two sets into
test_size = 0 # will hold the exp_size divided by the num_tests

sklearn_svc = SVC()

if len(sys.argv) == 3:
    # load the saved data from the input operations
    normArg = sys.argv[1]
    malArg = sys.argv[2]
    normOp = np.load(normArg)
    malOp = np.load(malArg)
else:
    # load the saved data from the pre-selected operations
    normOp = np.load('normOp.centered.npy') # centered, side, or off
   , or air
```

```python
    malOp = np.load('malOp.centered.npy') # centered, side, or off, or air

# determine the minimum between the two number of operations and resize each array to that min
exp_size = int(min(np.size(normOp)/np.size(normOp[0]),np.size(malOp)/np.size(malOp[0])))
normOp = normOp[:exp_size]
malOp = malOp[:exp_size]
test_size = int(exp_size / num_tests)

# extract just the voltage data from each trial of the operation for checking accuracy with just values
norm = np.delete(normOp, 0, 1)
mal = np.delete(malOp, 0, 1)
norm = np.reshape(norm, (exp_size, int(np.size(norm[0]))))
mal = np.reshape(mal, (exp_size, int(np.size(mal[0]))))
# remove first 100 samples since not really distinguishable
norm = np.array(norm[:,100:])
mal = np.array(mal[:,100:])

# split arrays into the number of tests
test = []
for i in range(num_tests):
    test.append(norm[i*test_size:(i+1)*test_size])
    test.append(mal[i*test_size:(i+1)*test_size])
test = np.array(test) # numpy array of numpy arrays, can be confusing to understand
# create X and y from the test array
X = test[0]
y = []
for i in range(test_size):
    y.append(0)
for i in range(1,num_tests*2):
    X = np.append(X, test[i], axis=0)
    if (i%2)==0:
        for i in range(int(np.size(test[i])/np.size(test[i][0]))):
            y.append(0)
    else:
        for i in range(int(np.size(test[i])/np.size(test[i][0]))):
            y.append(1)
y = np.array(y)

# create array of predictions for the number of tests
y_pred = []
y_actual = []
for i in range(num_tests):
    X_train, X_test, y_train, y_test = train_test_split(X[i*
   test_size*2:(i+1)*test_size*2,400:600], y[i*test_size*2:(i+1)*
   test_size*2], test_size=0.5)
    y_pred.append(sklearn_svc.fit(X_train, y_train).predict(X_test))
    y_actual.append(y_test)

# determine values for true and false positives and negatives
true_pos = []  # malicious predicted as malicious
true_neg = []  # normal predicted as normal
false_pos = []  # normal predicted as malicious
false_neg = []  # malicious predicted as normal
for j,pred in enumerate(y_pred):
    tpos = 0
    tneg = 0
    fpos = 0
    fneg = 0
    for i,cls in enumerate(pred):
        if y_actual[j][i] == cls:
            if cls:
                # case where true positive (malicious predicted as malicious)
                tpos = tpos + 1
            else:
                # case where true negative (normal predicted as normal)
                tneg = tneg + 1
        else:
            if cls:
                # case where false positive (normal predicted as malicious)
                fpos = fpos + 1
            else:
                # case where false negative (malicious predicted as normal)
                fneg = fneg + 1
    true_pos.append(tpos)
    true_neg.append(tneg)
    false_pos.append(fpos)
    false_neg.append(fneg)
# normal accuracy is true_neg / (true_neg + false_pos)
norm_acc = []
for tneg,fpos in zip(true_neg,false_pos):
    norm_acc.append(tneg / (tneg + fpos))
# malicious accuracy is true_pos / (true_pos + false_neg)
mal_acc = []
for i in range(num_tests):
    mal_acc.append(true_pos[i] / (true_pos[i] + false_neg[i]))
# precision is true_pos / (true_pos + false_pos)
precision = []
```

```
for tpos,fpos in zip(true_pos,false_pos):
    precision.append(tpos / (tpos + fpos))

# mean accuracy from the predictions output
calc_mean_acc = []
for i in range(num_tests):
    calc_mean_acc.append((true_pos[i] + true_neg[i]) / (np.size(
   y_pred[i])))
print("SVC mean accuracies:")
for acc in calc_mean_acc:
    print("%.2f%%" % (acc*100), end=" ")
print("\nAccuracy mean:", np.mean(calc_mean_acc))
print("Accuracy variance:", np.var(calc_mean_acc))

# print precision as well as normal and malicious accuracies
print("\nPrecision:")
for prec in precision:
    print("%.2f%%" % (prec*100), end=" ")
print("\nPrecision mean:", np.mean(precision))
print("Normal operation accuracies:")
for nacc in norm_acc:
    print("%.2f%%" % (nacc*100), end=" ")
print("\nNormal mean:", np.mean(norm_acc))
print("Malicious operation accuracies:")
for macc in mal_acc:
    print("%.2f%%" % (macc*100), end=" ")
print("\nMalicious mean:", np.mean(mal_acc))
# save statistical results as matlab dictionary
#mdic = {}
#mdic['airAcc'] = calc_mean_acc
#mdic['air_precision'] = precision
#mdic['air_norm_acc'] = norm_acc
```

```
#mdic['air_mal_acc'] = mal_acc
#sio.savemat('air.mat', mdic)
# save trace as numpy array
#calc_mean_acc = np.array(calc_mean_acc)
#np.save('mean_acc.npy',calc_mean_acc)
```

Bibliography

1. Nicolas Falliere, Liam O Murchu, and Eric Chien. W32.stuxnet dossier. Technical report, Symantec Security Response, 2011.

2. Julian Rrushi, Hassan Farhangi, Clay Howey, Kelly Carmichael, and Joey Dabell. A quantitative evaluation of the target selection of havex ics malware plugin. In *Industrial Control System Security (ICSS) Workshop*, 2015.

3. F-Secure Labs. Blackenergy & quedagh: The convergence of crimeware and apt attacks. Technical report, F-Secure Labs, 2014.

4. NCCIC and ICS-CERT. Ics-cert annual assessment report fy2016. Technical report, NCCIC, 2016.

5. Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

6. Broadcom Europe. *BCM2837 ARM Peripherals*. Broadcom Corporation, 406 Science Park Milton Road, Cambridge, UK, 2012.

7. Bcm2835 registers. `https://elinux.org/BCM2835_registers`, 2014. Accessed: Feb 2022.

8. Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 388–397. Springer Berlin Heidelberg, 1999.

9. Thomas S. Messerges, Ezzy A. Dabbish, and Robert H. Sloan. Investigations of power analysis attacks on smartcards. *Proceedings of the 1st Workshop on Smartcard Technology, Smartcard 1999*, 1999.

10. Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3156:16–29, 2004.

11. Neil Hanley, Michael Tunstall, and William P. Marnane. Unknown plaintext template attacks. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5932 LNCS:148–162, 2009.

12. Liran Lerman, Stephane Fernandes Medeiros, Nikita Veshchikov, Cédric Meuter, Gianluca Bontempi, and Olivier Markowitch. Semi-supervised template attack. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 184–199. Springer, 2013.

13. Houssem Maghrebi, Thibault Portigliatti, and Emmanuel Prouff. Breaking cryptographic implementations using deep learning techniques. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 10076 LNCS:3–26, 2016.

14. Timo Bartkewitz. Leakage prototype learning for profiled differential side-channel cryptanalysis. *IEEE Transactions on Computers*, 65:1761–1774, 2016.

15. Petr Socha, Jan Brejník, and Matěj Bartík. Attacking aes implementations using correlation power analysis on zybo zynq-7000 soc board. *2018 7th Mediterranean*

Conference on Embedded Computing, MECO 2018 - Including ECYPS 2018, Proceedings, pages 1–4, 2018.

16. Jiming Xu and Howard M. Heys. Template attacks of a masked s-box circuit: A comparison between static and dynamic power analyses. *2018 16th IEEE International New Circuits and Systems Conference, NEWCAS 2018*, pages 277–281, 2018.

17. M. Ali Vosoughi and Selçuk Köse. Combined distinguishers to enhance the accuracy and success of side channel analysis. *Proceedings - IEEE International Symposium on Circuits and Systems*, 2019-May, 2019.

18. Thorben Moos, Amir Moradi, and Bastian Richter. Static power side-channel analysis - an investigation of measurement factors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28:376–389, 2020.

19. Benoît Chevallier-Mames, Mathieu Ciet, and Marc Joye. Low-cost solutions for preventing simple side-channel analysis: Side-channel atomicity. *IEEE Transactions on Computers*, 53:760–768, 2004.

20. Apostolos P. Fournaris, Lidia Pocero Fraile, and Odysseas Koufopavlou. Exploiting hardware vulnerabilities to attack embedded system devices: A survey of potent microarchitectural attacks. *Electronics (Switzerland)*, 6, 2017.

21. Shane S Clark, Benjamin Ransford, Amir Rahmati, Shane Guineau, Jacob Sorber, Kevin Fu, and Wenyuan Xu. Wattsupdoc: Power side channels to nonintrusively discover untargeted malware on embedded medical devices. In *2013 {USENIX} Workshop on Health Information Technologies (HealthTech 13)*, 2013.

22. Roshni Shende and Dayanand D Ambawade. A side channel based power analysis technique for hardware trojan detection using statistical learning approach. In

2016 thirteenth international conference on wireless and optical communications networks (WOCN), pages 1–4. IEEE, 2016.

23. Fei Ding, Hongda Li, Feng Luo, Hongxin Hu, Long Cheng, Hai Xiao, and Rong Ge. Deeppower: Non-intrusive and deep learning-based detection of iot malware using power side channels. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security, ASIA CCS 2020*, pages 33–46. Association for Computing Machinery, Inc, 10 2020.

24. Yi Han, Sriharsha Etigowni, Hua Liu, Saman Zonouz, and Athina Petropulu. Watch me, but don't touch me! contactless control flow monitoring via electromagnetic emanations. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 1095–1108. Association for Computing Machinery, 10 2017.

25. Nader Sehatbakhsh, Alireza Nazari, Monjur Alam, Frank Werner, Yuanda Zhu, Alenka Zajic, and Milos Prvulovic. Remote: Robust external malware detection framework by using electromagnetic signals. *IEEE Transactions on Computers*, 69:312–326, 3 2020.

26. Nikhil Chawla, Harshit Kumar, and Saibal Mukhopadhyay. Machine learning in wavelet domain for electromagnetic emission based malware analysis. *IEEE Transactions on Information Forensics and Security*, 16:3426–3441, 2021.

27. Haider Adnan Khan, Nader Sehatbakhsh, Luong N. Nguyen, Robert L. Callan, Arie Yeredor, Milos Prvulovic, and Alenka Zajic. Idea: Intrusion detection through electromagnetic-signal analysis for critical embedded and cyber-physical systems. *IEEE Transactions on Dependable and Secure Computing*, 18:1150–1163, 5 2021.

28. Omar Adel Ibrahim, Savio Sciancalepore, Gabriele Oligeri, and Roberto Di Pietro. Magneto: Fingerprinting usb flash drives via unintentional magnetic emissions. *ACM Transactions on Embedded Computing Systems*, 20, 1 2021.

29. Zoltan Baldaszti. Raspberry pi 3 bare metal tutorial. `https://github.com/bztsrc/raspi3-tutorial`, 2021. Accessed: Feb 2022.

30. Raspberry pi forums: Beginner learning bare metal gpio. `https://forums.raspberrypi.com/viewtopic.php?t=202861`, 2018. Accessed: Feb 2022.

31. Raspberry pi forums: Rpi zero - bare metal - pwm. `https://forums.raspberrypi.com/viewtopic.php?t=279726`, 2020. Accessed: Feb 2022.

32. Raspberry pi forums: Working with pwm. `https://forums.raspberrypi.com/viewtopic.php?t=89122`, 2014. Accessed: Feb 2022.

Acronyms

ARM	Advanced RISC Machines. 4
CPA	correlation power analysis. 6
DES	Data Encryption Standard. 6
DPA	differential power analysis. 6
EM	electromagnetic. iv, vii, 2, 3, 5, 8, 9, 10, 12, 13, 14, 16, 17, 22, 23, 25, 27, 29, 30, 43, 44, 1
FFT	fast Fourier transform. 9
FN	false negatives. 20, 21
FP	false positives. 20, 21
GPIO	general-purpose input/output. 4, 14, 16, 17
ICS-CERT	Industrial Control Systems Cyber Emergency Response Team. 1
IoT	Internet of things. 8
LDA	linear discriminant analysis. 8, 29
LP	low-power. 15
NuSVC	nu-support vector classification. 29
PLC	programmable logic controller. 4, 8, 9
PWM	pulse-width modulation. 4, 15, 16

QDA quadratic discriminant analysis. 29

RPM revolutions per minute. 12, 13, 15, 16, 47

SCA side-channel analysis. iv, 4, 5, 6, 7, 8, 10, 11, 12, 44, 1

SCADA supervisory control and data acquisition. 7

SVC support vector classification. iv, 2, 20, 29, 33, 40, 1

SVM support vector machine. 5, 9, 12, 19, 20, 21

TN true negatives. 20, 21

TP true positives. 20, 21

UART universal asynchronous receiver-transmitter. 4, 14, 16

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704–0188*

1. REPORT DATE *(DD–MM–YYYY)* 24–03–2022	**2. REPORT TYPE** Master's Thesis	**3. DATES COVERED** *(From — To)* Jun 2020 — Mar 2022

4. TITLE AND SUBTITLE Malware Detection Using Electromagnetic Side-Channel Analysis	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Matthew A. Bergstedt	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 Hobson Way WPAFB OH 45433-7765	**8. PERFORMING ORGANIZATION REPORT NUMBER** AFIT-ENG-MS-22-M-008
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFRL/RYDA Building 620 WPAFB OH 45433-7765 COMM 937-656-9045 Email: pranav.patel.2@us.af.mil	**10. SPONSOR/MONITOR'S ACRONYM(S)** AFRL/RYDA
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Many physical systems control or monitor important applications without the capacity to monitor for malware using on-device resources. Thus, it becomes valuable to explore malware detection methods for these systems utilizing external or off-device resources. This research investigates the viability of employing EM SCA to determine whether a performed operation is normal or malicious. A Raspberry Pi 3 was set up as a simulated motor controller with code paths for a normal or malicious operation. While the normal path only calculated the motor speed before updating the motor, the malicious path added a line of code to modify the calculated speed. A script from a control terminal then sent a signal to the Pi to have it conduct either the normal or malicious operation while an EM probe was set up to collect emission traces of those operations. These traces were split into training and testing data sets, with the training set used to train a SVC model. Afterwards, the model was run on the testing set and achieved 96% classification accuracy for classifying the trace as either normal or anomalous.

15. SUBJECT TERMS

Side-channel analysis, Unintended radiated emissions, Anomaly detection

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Lt Col James W. Dean, AFIT/ENG
U	U	U	UU	89	**19b. TELEPHONE NUMBER** *(include area code)* (937) 255-3636, ext 4580; james.dean@afit.edu

Standard Form 298 (Rev. 8–98)
Prescribed by ANSI Std. Z39.18