2018-06

DETECTING RANSOMWARE THROUGH POWER ANALYSIS

Melton, Jacob D.

Monterey, CA; Naval Postgraduate School

https://hdl.handle.net/10945/59721

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

DETECTING RANSOMWARE THROUGH POWER ANALYSIS

by

Jacob D. Melton

June 2018

Thesis Advisor:	John D. Roth
Co-Advisor:	Roberto Cristi

Approved for public release. Distribution is unlimited.

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

1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** June 2018	**3. REPORT TYPE AND DATES COVERED** Master's thesis

4. TITLE AND SUBTITLE DETECTING RANSOMWARE THROUGH POWER ANALYSIS	**5. FUNDING NUMBERS**
6. AUTHOR(S) Jacob D. Melton	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.	**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

Cyber criminals are increasingly using malicious programs to take control of and exploit individuals', businesses', and governments' data. A large portion of malware is a type called ransomware, which finds a way to restrict the infected user's access to data until a payment is obtained. Current detection solutions include programs that analyze file system changes and registry events, employ honeypot techniques, and identify anomalies in network patterns. This research presents an algorithm developed to detect ransomware by analyzing a computer's power consumption. Specifically, the algorithm identifies features of the computer's power consumption that are indicative of encryption operations. We can successfully identify encryption of files with sizes of 500MB and greater with a high degree of success. By applying our encryption detection algorithm to the Cryptographic Ransomware, we are able to successfully identify the execution of WannaCry Ransomware samples.

14. SUBJECT TERMS solid-state drive, power analysis, ransomware, encryption	**15. NUMBER OF PAGES** 103
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

DETECTING RANSOMWARE THROUGH POWER ANALYSIS

Jacob D. Melton
Ensign, United States Navy
BS, United States Naval Academy, 2017

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
June 2018

Approved by: John D. Roth
Advisor

Roberto Cristi
Co-Advisor

Clark Robertson
Chair, Department of Electrical and Computer Engineering

ABSTRACT

Cyber criminals are increasingly using malicious programs to take control of and exploit individuals’, businesses’, and governments’ data. A large portion of malware is a type called ransomware, which finds a way to restrict the infected user’s access to data until a payment is obtained. Current detection solutions include programs that analyze file system changes and registry events, employ honeypot techniques, and identify anomalies in network patterns. This research presents an algorithm developed to detect ransomware by analyzing a computer’s power consumption. Specifically, the algorithm identifies features of the computer’s power consumption that are indicative of encryption operations. We can successfully identify encryption of files with sizes of 500MB and greater with a high degree of success. By applying our encryption detection algorithm to the Cryptographic Ransomware, we are able to successfully identify the execution of WannaCry Ransomware samples.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

AES	Advanced Encryption Standard
ALU	Arithmetic Logic Unit
ATX	Advanced Technology eXtended
CPU	central-processing unit
DES	Data Encryption Standard
FPGA	field-programmable gate array
HDD	hard disk drive
OS	operating system
RAM	random access memory
RSA	Rivest, Shamir, and Adelman
SATA	Serial Advanced Technology Attachment
SSD	solid-state drive
UBA	User Behavior Analytics
VM	virtual machine

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgments

I would like to thank my advisor, Professor John Roth, for his tremendous help with analyzing data and putting together this thesis. I would also like to thank Donna Miller for her help setting up and performing experiments in her lab. It was a privilege to conduct research with both of you.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

In the modern era of the global Internet, information systems are increasingly falling victim to malicious programs. Ransomware is one extortion method that prevents the target from accessing their data until a payment is extracted. In the last few years, ransomware has been a problem worldwide, with one attack, Wannacry, affecting more than 200,000 victims across the globe [1]. The two main types of ransomware are locker ransomware and crypto ransomware [2]. Locker ransomware prevents the user from accessing or logging on to the device and, as a result, denies access to the user's data [3]. Crypto ransomware encrypts data throughout the targeted system, causing the files to become useless without the encryption key. The user is then prompted to pay for the encryption key. Crypto ransomware accounts for over 90 percent of the ransomware attacks executed in 2016 [4]. Since 2013, the threat from crypto ransomware has only grown in volume and sophistication. The ability to detect and prevent ransomware attacks is beneficial to individuals, businesses and governments by reducing costs and increasing information availability.

1.1 Objective

The objective of this research is to develop a method for detecting ransomware that can be applied with great effect to the current state of ransomware activity. Current proposed methods for detection include signature detection, honeypot techniques, and machine learning methods [5]–[7]. Each of these approaches has drawbacks, including the inability to detect new types of ransomware and a long lag time between detection and distribution of a defense. In order to target these failures, we propose a method to detect a common trait across the scope of all ransomwares. Since crypto ransomware is the majority of ransomware, in this research we propose a method to detect this type of malware. Specifically, the method targets the detection of encryption operations used in crypto ransomware. As for the method to detect encryption, past research in [8] showed that detection of encryption operations in field-programmable gate arrays (FPGAs) was possible through power analysis techniques. In the experiment, encryption of plain text was performed while the power of the FPGA was recorded. The execution of encryption operations was shown to have a unique and consistent effect on the device's power consumption. In addition, other research in [9] and [10] use power analysis to detect read and write operations occurring on a computer. Specifically, in this research we propose a method for detection of a computer's execution of encryption operations. Power consumption of a computer's solid-state drive (SSD) and central-processing unit (CPU) are monitored while ransomware activity is preformed. More specifically, this involves recognizing the procedure taken to encrypt a file through identifying representative features within the computer's power consumption. Since the process for encrypting a file causes interaction with the computer's SSD and CPU, the energy used by these components yields a classifiable pattern.

The ultimate goal of this research project is to create an algorithm to recognize crypto ransomware activity using power consumption data. Since crypto ransomware uses encryption to render a user's files unusable, this algorithm specifically looks for features that indicate encryption operations while limiting the identification of false positives. The ability to recognize the occurrence of encryption allows for intervention in the computer's process to limit the data rendered inaccessible by ransomware. For example, when encryption is detected, the process can be halted while the user is asked if the encryption operations are authorized. Having a high degree of accuracy is crucial when predicting the presence of encryption operations. Alerting the user to false positives can be desensitizing and decrease the value of a true positive alert. A successful algorithm enables the majority of a victim's data to remain accessible while detecting the presence of malware on their device.

1.2 Thesis Contributions

To detect crypto ransomware on computers, we provide a unique solution using the device's power consumption. The proposed method detects the use of encryption operations that are innate characteristics of crypto ransomware.The contributions of this thesis are as follows.

1. We propose a method for using the power consumption of a computer's SSD and CPU to detect the execution of crypto ransomware. First, an automated power gathering system is used to collect the power traces of the SSD and CPU. Second, an algorithm extracts features from the power traces and predicts the occurrence of encryption operations inherent to crypto ransomware.
2. In this thesis, we analyze the performance of the proposed crypto ransomware detec-

tion algorithm in terms of its ability to detect encryption operations. Furthermore, the algorithm is shown to detect a sample of Wannacry ransomware.

1.3 Organization

This thesis is organized into five chapters. A discussion of previous work with ransomware and its history is provided in Chapter 2. Additionally, background on the power analysis techniques used in this research is provided in Chapter 2 as well. The method for data collection as well as the algorithm used to extract features from power traces and identify encryption operations is explained in Chapter 3. The results of several experiments to test the proposed method of encryption detection is presented in Chapter 4. A final overview of the results and proposed methods for further work are described in Chapter 5.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 2: Background

In this chapter, we explore previous work investigating the detection of ransomware. Topics covered include ransomware history, ransomware characteristics, current ransomware detection methods, and power analysis techniques. In Ransomware History, we introduce the reader to the evolution of ransomware to its modern-day state. In Ransomware Characteristics, we cover the common characteristics seen with this malware's behavior. In Current Ransomware Detection Methods, we focus on current methods for detecting and eliminating the execution of ransomware on a victim's computers.

2.1 Ransomware History

The idea of using a malicious program to hold a target's data for ransom has been around for decades. The first credited crypto ransomware attack occurred in 1989 and targeted health-care data through the use of a hidden program distributed on floppy disks and became recognized as the AIDS Trojan [11]. Fortunately, this attack was relatively unsuccessful because of several limiting factors. At the time, processing payments to regain access to the data was more complex than today. Additionally, encryption algorithms allowing the encryption keys to be retrieved and allowing for recovery of data were not as advanced. Furthermore, the Internet was not as widely used, so propagation and infection of malware was more difficult than in current circumstances. Finally, in many ransomware code samples, the encryption key could be viewed in plain text and not properly hidden to prevent easy decryption. Today these barriers have all been overcome, enabling ransomware to have a large and devastating impact.

The second major wave of crypto ransomware occurred in 2005 with the appearance of the Trojan Gpcoder [4]. These malicious programs contained several flaws including using weak custom encryption or having the encryption key visible within the code of the malware. These failures were quickly overcome with newer versions using Rivest, Shamir, and Adelman (RSA) encryption for stronger encryption and generating random encryption keys for each victim. A few years later, in 2008, ransomware began using crypto currency as a form of payment, giving malware an easier platform to extract payments from victims.

Crypto currency is a digital asset that has a monetary value. Due to the digital nature of crypto currencies, transactions occur globally, with great speed, and are secure and anonymous [12]. As a result, revenue from ransomware can be generated worldwide almost instantly.

The evolution of ransomware continued with the creation of locker ransomware. Instead of encrypting the victim's data, locker ransomware restricts access to the device until a payment is extracted. Some examples include programs that displayed threats of law enforcement prosecution for downloading copyrighted materials or faking expired software licenses [13].

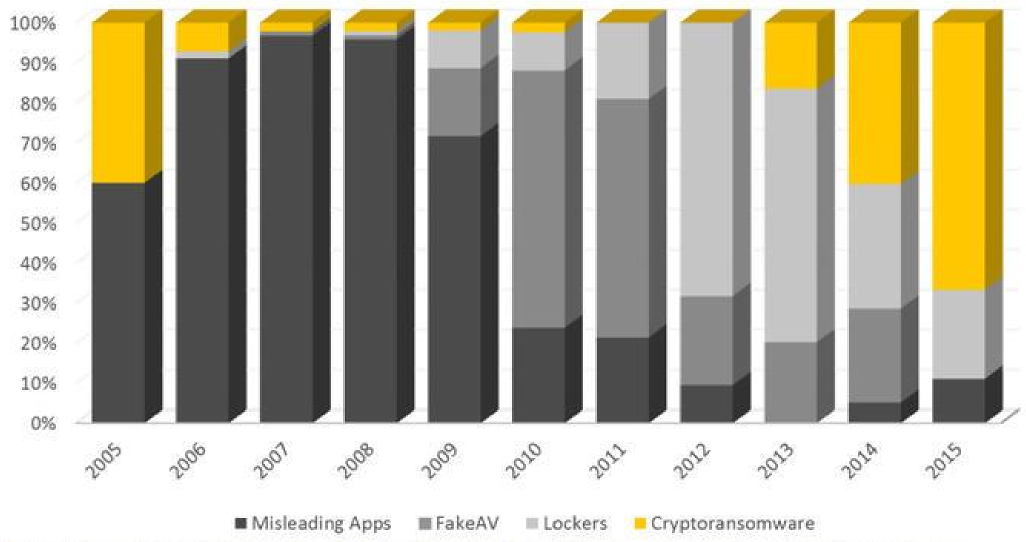

Figure 2.1. Percentage of Malware Identified between 2005 and 2015. Source: [4].

In 2013, there was a shift from the social-engineering aspects of locker ransomware back to the crypto-ransomware attacks. In Figure 2.1, we see the popularity of crypto ransomware explode starting in 2013. This is a result of cyber criminals learning that through the use of strong encryption algorithms and proper encryption key management it is almost impossible to recover the compromised data without paying the ransom fee. For instance, one modern ransomware called CryptoWall has earned over 300 million U.S. dollars in revenue [11].

2.2 Ransomware Characteristics

Cyber criminals create ransomware with the goal of gaining access to a victim's system, restricting access to key data files, and then extorting the user to regain control of their files. With this procedure in mind, a generic ransomware attack includes the following stages: deployment, installation, destruction, and extortion [14].

Deployment

In this phase, the malware seeks to gain access to the target device. Typical attack vectors include the following.

- Drive-by downloads: Malicious code is placed on websites to exploit a visitor's browser in an attempt to download malware onto the targeted machines [15].
- Phishing attacks: Phishing uses social-engineering methods to attract victims to spoofed websites and links. Often trustworthy websites are impersonated to get victims to give away personal information or download malicious software [16].
- Exploit kits: These software tools were created to automatically take advantage of vulnerabilities in a victim's machine. These kits can be used to create code that targets applications such as Abode Flash Player and Java Runtime Environment. Because of their automated nature, exploit kits have lowered the technical ability required to create and use malicious code [17].

Installation

After the malware has successfully been downloaded to the victim's computer, the ransomware attempts to gain control over the system's functions. In many cases with ransomware, the infected machine attempts to establish a connection to a domain over the Internet to communicate information of the victim's machine as well as encryption keys generated to be used on the device [18]. Generally at this stage, the malware also scans the local connections for other vulnerable systems [19].

Destruction

In this phase, the malware encrypts files on the victim's system. Encryption methods range from using asymmetric algorithms, such as RSA, to symmetric algorithms, such as Advanced Encryption Standard (AES) and Data Encryption Standard (DES). Asymmetric algorithms use two keys to encrypt data were symmetric algorithms only use one key. There are even families of ransomware that use self-designed encryption systems [20].

Extortion
Once the victim's files have been encrypted, the malware displays information explaining that the device has been compromised. Payment, typically in the form of a crypto currency because of its users' anonymity, is requested for the service to decrypt the victim's data.

2.3 Current Ransomware Detection Methods

Each year the number of new malware specimens increases, and with this growth, new security methods are invented to protect networks and devices [21]. From Figure 2.2, we see that cyber attacks against federal agencies grow yearly. Current methods can be divided into three categories: signature, behavioral, and heuristic [22]. Signature-based detection is the most commonly used today. In the rest of this section, we investigate proposed defenses against ransomware.

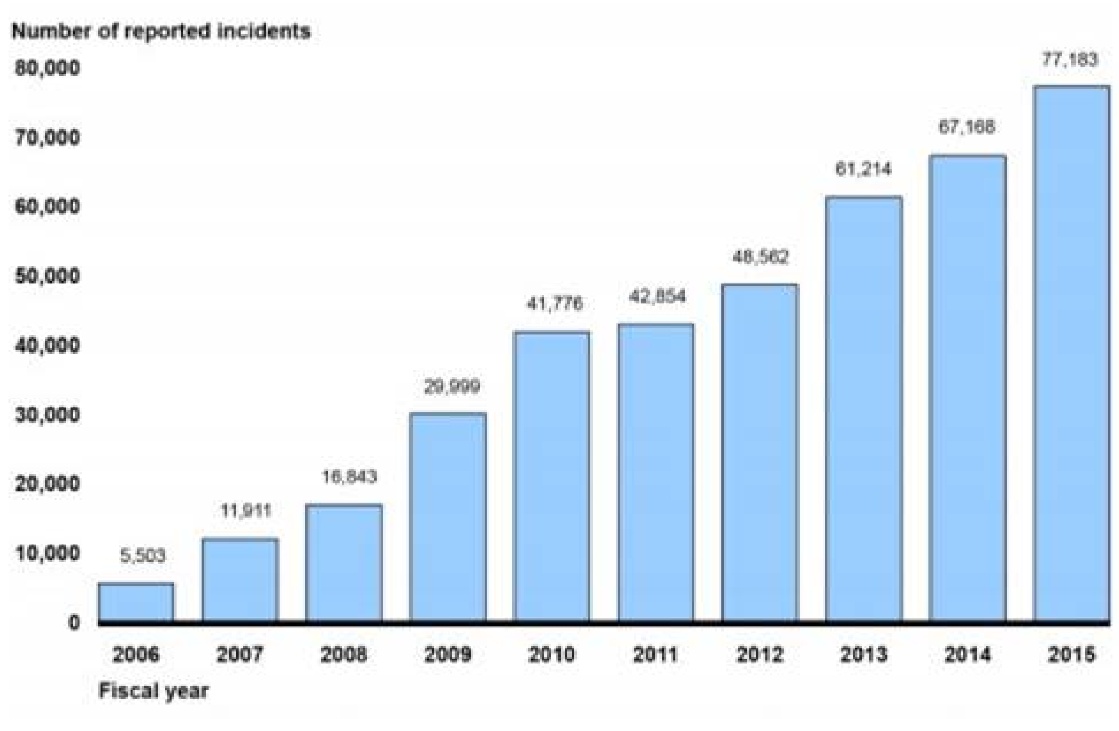

Figure 2.2. Incidents Reported for Cyber Attacks on Federal Agencies between 2006 and 2015. Source: [23].

Signature Detection
Pattern matching is the backbone of signature-based detection [5]. Using a unique feature or a developed fingerprint is common in traditional anti-virus products. The process involves capturing a sample of the malware, developing its signature, then distributing that pattern to anti-virus software. Anti-virus software then scans files on the device looking for the extracted signature, and if found, the file is red flagged and quarantined. The main issue with this method is that zero-day attacks or brand-new malware cannot be detected. The delay until the malware's signature updates can be created to protect systems from infection can leave the door open for many victims to be infected. As a consequence, more specific detection techniques have been suggested.

Honeypot
The use of a honeypot in a computer system refers to monitoring for unwarranted use of a resource by a program. For example, User Behavior Analytics (UBA) can be applied to identify abnormal behavior of ransomware. Specifically, identifying when a user modifies a large number of files in a relatively short amount of time can indicate the execution of ransomware [24]. Another approach uses the placement of files and folders throughout the device and constantly monitors them for modifications. If these tripwire files are accessed and modified with encryption, then ransomware is likely present [6]; however, this method can only raise the alarm that ransomware is present if the specified files are modified. Unfortunately, ransomware can be programmed to only encrypt certain files and directories, and if the tripwire files are not among these, the honeypot technique fails to detect the ransomware. Even in the case of the tripwire files being modified, there is a likelyhood that much of the data on the device has already been encrypted and is inaccessible by the user.

Hashing
A hash function is typically a mathematical algorithm that takes input data and determines an output value as a direct result of the input [25]. In some cases, these functions have a specified output length. For example, the text file of a book could be put through a hash function and only a 20 character output is provided. Each output is directly determined by the data throughout the input. If the same book was modified slightly, the 20 character output is different. The advantage of these functions is that they can be used to determine when data has been modified based on a change in the hashing functions output. In [26], hashing is

used to monitor files and identify when they have been modified. Additionally, through the use of similarity-preserving hash functions [27], an altered file can be analyzed and inferred if encryption has occurred. Since encryption results in plain text that is unrelated to the original document, these hash functions can indicate whether a file was just altered slightly or entirely encrypted. Using similarity-preserving hash functions to monitor recently modified files can give an indication to the presence of ransomware; however, using hashing to detect when files are modified incurs a large overhead in CPUs time for continuously calculating the hashes of files. Additionally, for large directories, it takes a significant amount of time to calculate the hashes of the stored data. As a consequence, the delay for recognizing a modified file is large and gives the ransomware a nontrivial amount of time to encrypt files. This can lead to a substantial amount of data already encrypted before the ransomware is detected and stopped.

Shannon's Entropy

In the context data, Shannon's entropy is used to describe the amount of randomness, or disorder, that is contained in a given sample of data [28]. Specifically, this technique presents a value to represent the amount of information and randomness in a given data sample. For example, if a string contains ten characters, the entropy may suggest that the same amount of information can be expressed in eight characters. The reason files may have low entropy values is repetition of the same data, and as a result that data can be expressed with less data. An example is the string "thethethe" which represented in fewer characters, perhaps with "3xthe," representing three times that string. A low value of entropy suggests that the information in that file can be expressed using a smaller number of bits. A large value of entropy suggests that the raw data cannot be expressed with a smaller number of bits.

The research from [28] shows that it is possible to identify encrypted files from the file's entropy. In [29], a process for monitoring the entropy of the data buffer is proposed as a way to identify encryption operations from ransomware. Specifically, this research looks for increases in entropy in the data buffer because encryption methods increase the randomness and disorder or a file. This method for detecting encryption depends on the entropy of a file increasing for detection, and as a result will fail to detect encryption of a file already having a larger amount of entropy. Files that are stored or compressed by the user have larger entropy values before the execution of ransomware and, therefore, do not yield a detection with this method.

Machine Learning

Past research has used machine learning to detect malware executables. The work in [7] tackled the issue as a text-classification problem. With support-vector machines, naïve Bayes, and decision trees, the researchers demonstrated they could detect malicious executables. In another study, machine learning algorithms were applied to detect cryptographic algorithms from program binaries. A large number of encryption programs were generated using known encryption libraries and compiled into binary files. Using decision tree models, we see that an algorithm was presented to detect ransomware in [30]. These methods have large overhead requirements in terms of computational power, which causes a latency between infection of a device with ransomware and its detection. In addition, these methods had trouble with malware that uses compressed or encrypted payloads. Malware using compression or encryption to hold its executable code remained undetected using this proposed method.

2.3.1 Limitation of Current Detection Methods

Current ransomware methods have several limitations. Using signature detection and machine learning requires a sample of the malware that is at least similar to the ransomware infecting your device to yield detection. If this is not the case, then there is a large latency between when these methods stop ransomware attacks and the introduction of the malware. Additionally, other methods such as hashing and entropy techniques have large overheads and, in some cases, can result in a large delay before the encryption operations are detected on a device. Finally, honeypot techniques rely on tripwire files which can be coded to be avoided, and the method also has a large delay before encryption detection. These limitations mainly result in an inability to recognize the presence of ransomware in a timely manner in order for the victim to avoid being denied access to their data. The method proposed in this research is able to consistently detect ransomware at the beginning stages of infection.

2.4 SSD Architecture

The demand for SSDs has increased in recent years due to their faster storage operations and smaller power characteristics. The benefits of SSDs has allowed them to be in almost every new computer sold. As a result, these devices' behavior is important to understand with the current threat from ransomware. Compared to traditional hard disk drive (HDD)s, which use spinning magnetic disks to store data, SSDs use NAND-based flash memory [9]. This type of memory has groups of memory cells that are arranged in blocks and managed by a controller as shown in Figure 2.3. Due to the nature of the design, read, write, and delete operations are limited to the size of the entire block [31]. Additionally, each of these cells is made up of a floating-gate transistor. These transistors contain an electrically secluded region where a charge or absence of a charge can be stored representing a logical 1 or 0, respectively. In order to modify the isolated region, electron tunneling is used to write a logical bit [32]. It is important to note that the high voltage differential required to write a bit is not needed when reading data from the electrically secluded region.

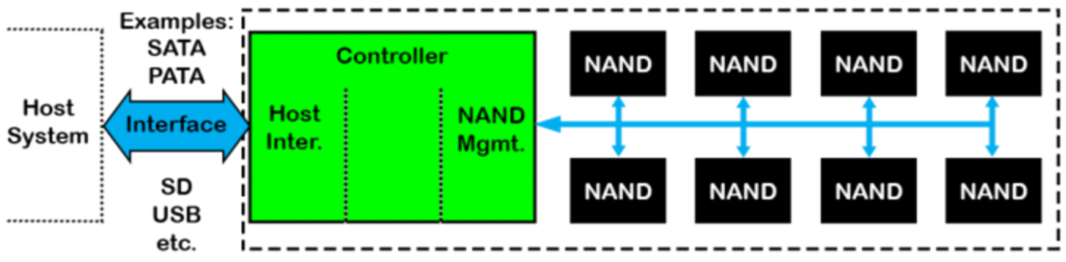

Figure 2.3. Basic SSD Architecture. Source: [33].

2.5 Power-Analysis Techniques

The power consumption of devices can often be used to gain insight into the operations being preformed by the electronic system. Some of the most powerful examples are side-channel attacks against cryptographic implementations. More specifically, physical measurements of a device's power consumption while performing encryption can lead to the discovery of the key used in algorithms ranging from AES to RSA [34]. The implication of these attacks have been profound, implying that even technically sound security algorithms may have unintentional back doors.

In other research, power analysis has been used to show that devices such as SSDs do not behave as manufacturers advertise [9], [35]. The authors of [35] found that SSDs do not erase all their data when a delete command is issued. In addition, when a TRIM command is enabled, which is supposed to ensure data deletion as opposed to just file pointers, some manufacture's drives do not perform the operation [9]. Another paper found that through measuring the current in the power line of an SSD, operations such as file read and writes can be predicted [10].

The research presented in this thesis uses similar power analysis and gather techniques as [9], [10]. In addition to past research, the power consumption of the CPU is be analyzed in conjunction with the SSD. The algorithm for detecting encryption also draws from the techniques presented in [10] for classifying read and write operations. These techniques are used to identify features in the power traces of the SSD and CPU. A unique algorithm was developed to compare these features and extract information that can identify the execution of encryption algorithms.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 3: Ransomware-Detection Algorithm Design

The objective of this chapter is to describe the process behind the proposed ransomware detecting algorithm. Since crypto ransomware is the most prevalent type of ransomware in today's market [4], our goal is to develop a method to detect this type of ransomware. With encryption being crucial to this type of malware's operation, the objective is to recognize encryption operations being executed. Through the analysis of a computer's CPU and SSD power consumption, features can be extracted to detect the occurrence of encryption. By recognizing the presence of encryption, we can stop the malware before the majority of the victim's data is encrypted. Additionally, if the ransomware's activity is stopped prematurely, the key is often still in memory and can be used to decrypt the initial modified data [36].

3.1 Determination of Method and Data Sources

Encryption is the backbone of modern day digital security, providing the ability for secure communications and data storage. The security of cipher text is determined by the strength of the algorithm. Basically, the security of the cipher text is determined by the length of time it takes to convert the cipher text back to plain text by trial and error. Modern encryption algorithms such as implementations of AES can take over hundreds of years to crack by brute force [37]; however, there are other ways to attack encryption such as through side channel attacks.

A side-channel attack is a technique that extracts information on cryptographic devices by gathering physical information about the device while cryptographic operations are occurring [38]. For example, in [8] DES encryption was executed on an FPGA while the power consumption of the device was recorded. By analyzing the power consumption of the FPGA, the authors of [8] were able to detect the period when encryption was occurring and even recover the encryption key used. For detection of encryption on a computer, power analysis can potentially yield similar results.

To determine the parts of a computer where power consumption will yield information about ransomware, an understanding of the basic components of a computer is necessary. Modern computers share similar characteristics to the well known Von Neumann architecture shown in Figure 3.1 [39]. This architecture uses memory called primary memory to store data and programs, and in modern computers, this is referred to as random access memory (RAM). The Arithmetic Logic Unit (ALU) carries out calculations with data such as add and multiply. The control unit manages moving data into and from memory and executing program instructions. In a modern computer, a CPU contains the ALU and control unit. For interaction with secondary memory, the data must be directed through the input and output terminals displayed in Figure 3.1. In a computer, the operating system (OS) and user's files are stored in secondary memory, typically on an SSD. When an application or file is being used, that data is loaded from secondary memory into primary memory referred to as RAM. From RAM, the CPU runs the application or modifies the file and eventually updates the data back to secondary memory.

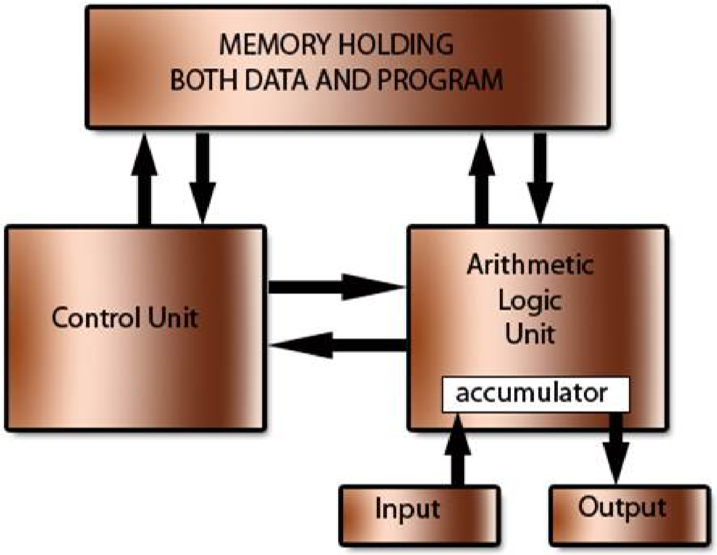

Figure 3.1. Von Neumann Architecture. Source: [40].

3.2 Signal Processing

In order to reach the goal of identifying when encryption is occurring, some sort of machine learning or classification algorithm must be used. These algorithms tend to have certain steps, as shown in Figure 3.2, that enables new data to be analyzed and classified based on extracted features from training data.

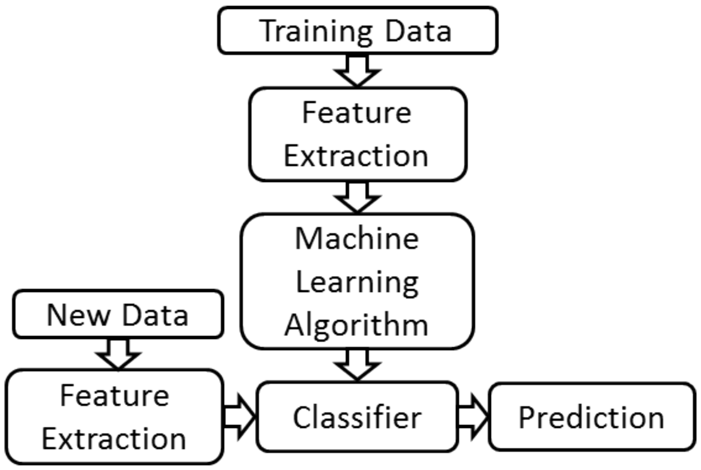

Figure 3.2. Flow Chart for Classification Algorithm

When beginning with a classification problem to separate data into groups, training data or data from known sources is used to instruct the algorithm on how to classify samples; however, the machine-learning algorithm is only as good as the identifying inputs extracted from the training data. In order for best results, these features should be impacted as a direct consequence of differences in the known groups in the training data. Once the features are extracted from the training data, they are used to form a model by the machine-learning algorithm that can be used to classify new data. When new data is analyzed, features are extracted and compared against the model in a classifier to predict the new data's characteristics.

3.3 Algorithm for Identifying Encryption

In the following section, we describe the algorithm's process for taking the inputs of SSD and CPU power consumption and determining if and when an encryption operation has occurred or is occurring. The flowchart in Figure 3.3 describes at a high level the stages within the presented algorithm.

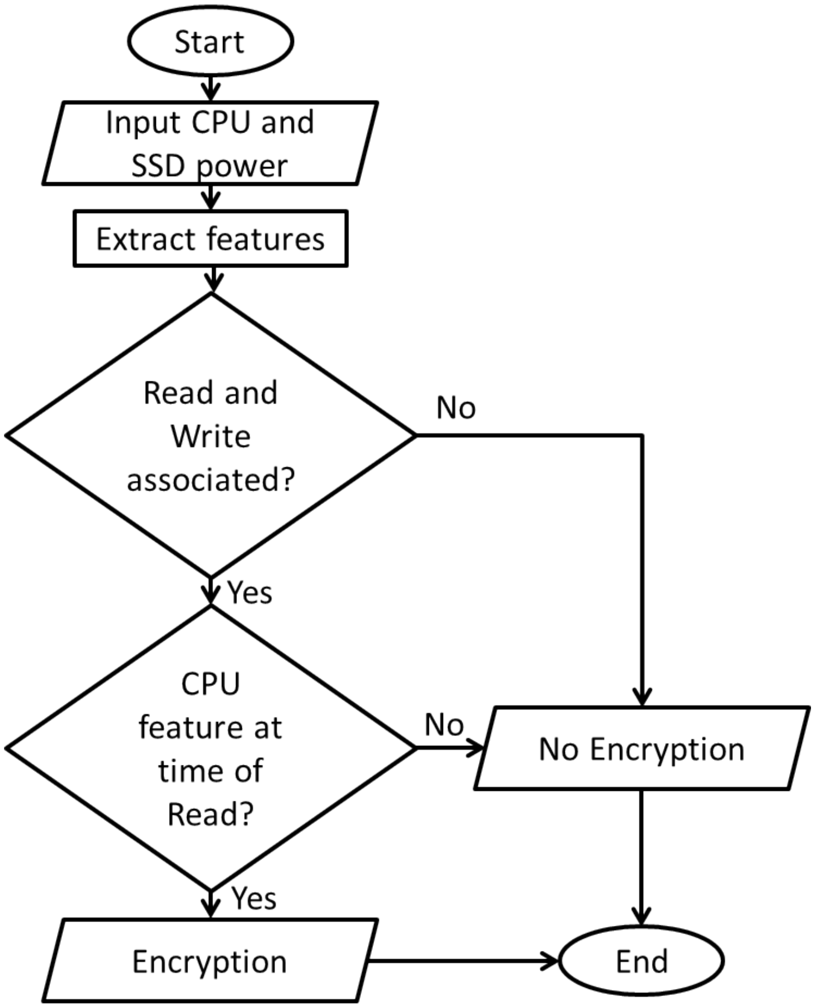

Figure 3.3. Flowchart of for Identifying Encryption

3.3.1 Determination of Features

When looking for features that signify the presence of encryption in the power consumption of an SSD and CPU, it is necessary to understand the innate process of encryption and how it is preformed on a computer system. For a file from secondary memory to be encrypted, the following three actions shown in Figure 3.4 have to occur:

1. The file must be loaded from main memory into RAM in the form of a file-read operation.
2. The encryption algorithm performs the process of translating the plain text into cipher

text and updating the values in RAM.

3. Encrypted data is written back to main memory in the form of a file-write operation and in the process overwrites the old file.

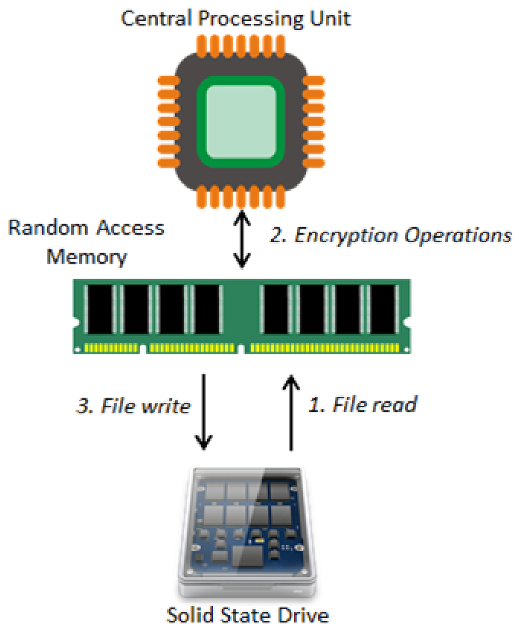

Figure 3.4. Encryption Operation Steps

These three steps are what the algorithm recognizes as features, and encryption can be identified through associating these features within certain parameters. More specifically, the file read and write operations are features ascertained using the SSD's power consumption, and the encryption algorithm's activity is extracted using the CPU's power consumption. In order to extract features from the power traces, a median filter is implemented to reduce noise in the signals.

The median filter was computed as

$$y[i] = median\{x[i] \in \omega\}, \tag{3.1}$$

where ω represents the neighborhood centered around the location $[i]$ in the signal. In general, a median filter takes a block of data centered at a location, determines the median value, and replaces the centered value with the median value. This type of filtering is typically used to reduce noise while still preserving the edges in the signals [41]. In the implementation of this filter in the algorithm, a *w* value of 10,000 was used. This equates to a window size of a fifth of a second in the time domain in our implementation. Once the power consumption traces are filtered, the algorithm goes on to extract features representative of encryption operations.

In [10], the authors use different threshold values of the SSD's power consumption to signify operations either being a file read, file write, or an idle operation; the amplitude of the power consumption signal is the most consistent feature across several SSD manufacturers. In addition, the researchers in [10] show that across several manufacturers, in combination with either a Windows or Linux OS, a write operation has a higher amplitude than a read operation. This observation is consistent with the discussion in Chapter 2, where the voltage required to write to a floating-gate transistor is significantly larger than what is required to perform a read operation. Furthermore, through experimentation, we see in (a) in Figure 3.5 the power consumption of a 1.0-GB read operation and in (b) in Figure 3.5 a 1.0-GB write operation. From Figure 3.5, it is apparent that for the same size file, a write operation consumes more power.

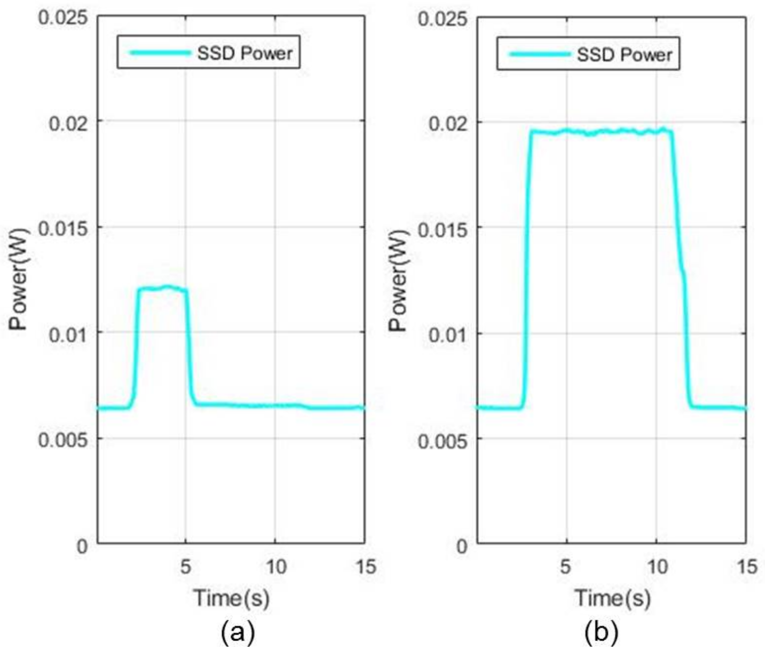

Figure 3.5. Power Plot of 1.0 GB File Read in (a) and 1.0 GB File Write in (b)

The presented ransomware detection algorithm applies a method of using thresholds to determine file read or file write features. Additionally, threshold limits are used to determine CPU features from the processor power draw. In (a) from Figure 3.6, we see a power trace of a file-write operation followed by a file-read operation. The displayed thresholds represent the values that the power trace must exceed to be classified as a read or write feature. Once the algorithm analyzes the power trace of the SSD against the thresholds, probable features are identified as a potential read feature if the power consumption exceeds only the lowest threshold and a potential write feature if the trace surpasses the second threshold. The thresholds were determined by visual inspection, and the values of the read and write threshold are 0.007 W and 0.0158 W, respectively. For the area to be deemed a true feature, the integrated value of the power signal exceeding the threshold is compared to a minimum size to ensure the location is significant enough to be labeled a feature. In the implementation of the algorithm, the minimum feature size is determined by looking at the power consumption of the SSD while in an idle state and visually identifying a minimum threshold size of 0.0028 W-s. The minimum threshold size allows only features of true operations to be considered in the algorithm. In Figure 3.6, the read and write features are shown in (b) to be extracted from the power trace from (a).

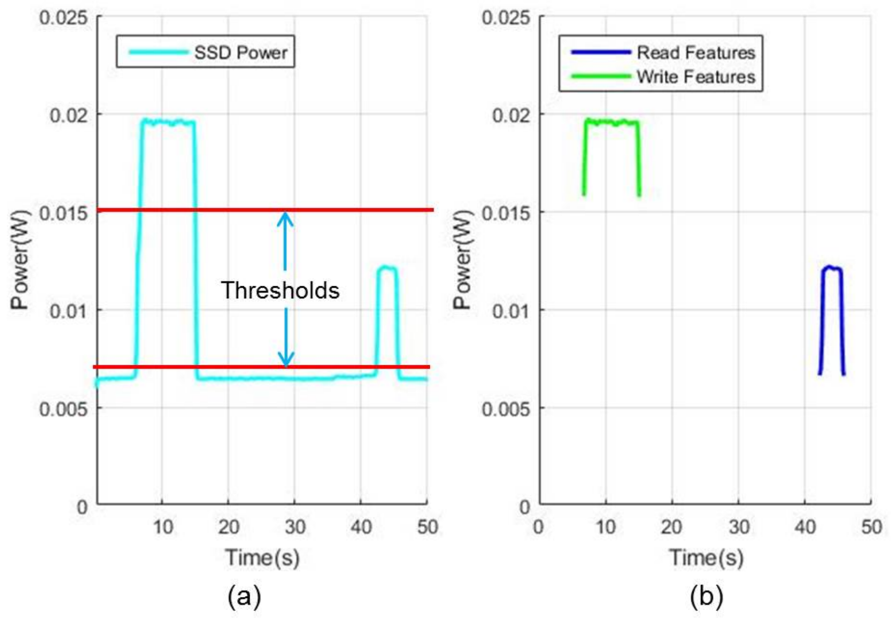

Figure 3.6. Power Plot of 1.0-GB File Write then 1.0-GB File Read in (a) and Plot of 1.0-GB File Write then 1.0-GB File Read Features in (b)

The process for extracting features is executed for the entirety of the inputted power trace. Once the features are extracted and deemed to be significant, the algorithm methodically checks for associated write and read operations.

3.3.2 Associating Read and Write Features

A vital step in the process to identify the occurrence of an encryption operation is to find a read and a write feature that are related. Using steps one and three presented in Figure 3.4, we use the assertion that the size of the file read from secondary memory to RAM in step 1 is the same size as the file written back to secondary memory in step 3 as a method for classifying the existence of a relationship. This notion leads to the idea that the read and write feature associated with encryption is consistently correlated with all file sizes being encrypted. To address this concept, the read and write features are integrated to find the total power used to perform that operation. The encryption-detecting algorithm uses a linear function generated from experimental data to check if the energy of the read and write features extracted from the SSD power trace are correlated in a manner that suggests the potential presence of an encryption operation. Specifically, in Figure 3.7 we see the correlation between the integrated values of the associated read and write feature.

After the feature extraction, the algorithm starts to compare the read and write features. In the research process, we used both a linear and degree-two relationship to compare features. The additional degree fit did not add any additional classification ability, so a simple linear comparison was implemented. Beginning with the first obtained read feature, we compared write operations to the linear function of best fit to discern their relationship. Nonetheless, not all extracted write features were used to compare against each read feature. Only write operations occurring within 90 s after the time of the read feature are compared with the linear model. The time limitation ensures that the only write operations considered are ones that could be the result of the read feature being considered. If no correlation is found between a read and write feature, then the algorithm moves on to consider another pairing of these two feature types. If a relationship is identified, as shown in Figure 3.3, the algorithm moves on to attempt to associate a CPU feature to potentially predict the occurrence of an encryption operation.

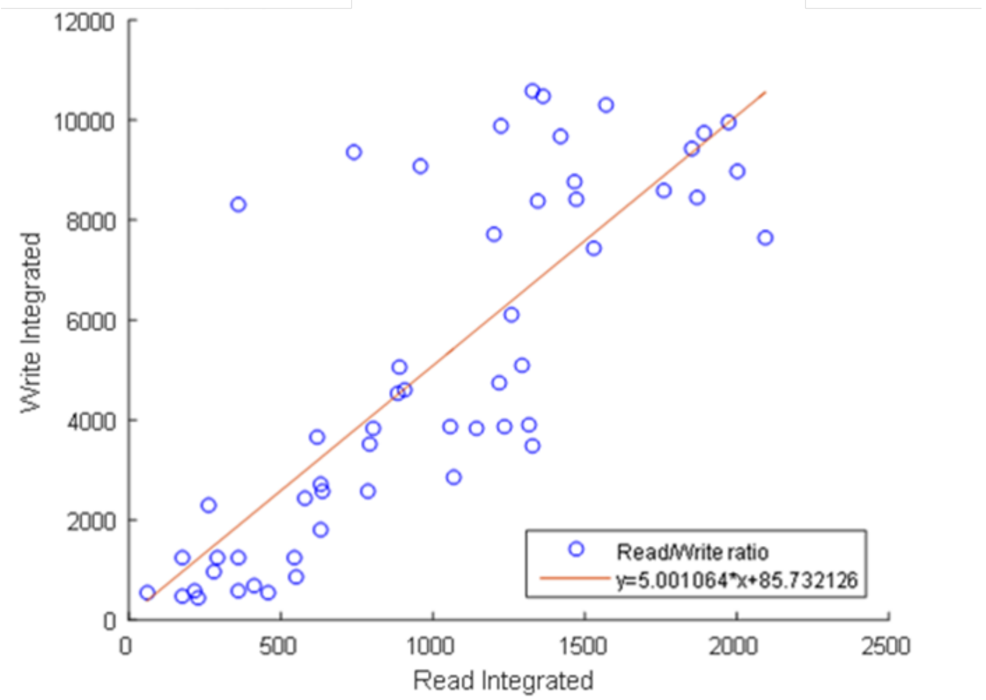

Figure 3.7. Read and Corresponding Write Power for Encryption

3.3.3 Associating CPU Features

The final step in identifying the presence of an encryption operation is to verify that a CPU feature can be associated with the already related read and write features. The importance of this step in the algorithm was highlighted when analyzing operations that contained similar steps to that of an encryption operation. For example, a file copy operation contains both the read operation of the file to RAM from secondary memory and the write operations from RAM to secondary memory. The impact on the SSDs power consumption is remarkably similar to that of an encryption operation; however, the distinguishing factor is seen in the CPUs power trace. Since, an encryption operation has extensive computations to make, the CPU has a significant increase in power consumption as compared to a simple file copy operation.

The importance of using the CPU power consumption can be seen by comparing Figure 3.8 (a) and (b). The power plots of the encryption operation and copy operation are similar except for the impact on the CPU power trace. We see that the encryption operation causes the CPU to consume a great deal more power. With a large enough threshold for identifying CPU features, operations such as a file copy will fail to elicit a corresponding CPU feature to the associated read and write features.

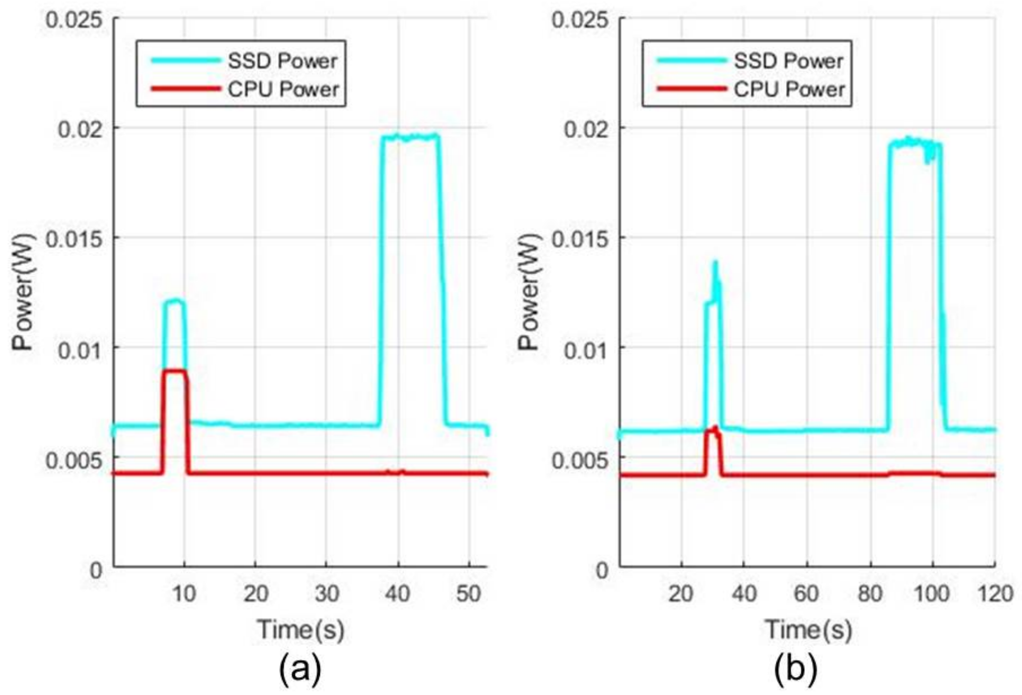

Figure 3.8. 1.0-GB File Encryption Power Plot in (a) and 1.0-GB File Copy Power Plot in (b)

The implementation of checking for a related CPU feature is fairly simple. After identifying related read and write operations, the algorithm looks for a CPU feature within 5.0 s of the read feature. Normally, the CPU feature is seen collocated in time with the read feature; however, in implementation the association window was expanded to 5.0 s to account for processor delay. If the CPU feature is of similar size to the read feature, then an encryption operation has been detected. Since the algorithm requires the information of the write feature which occurs last in the sequence, encryption is detected precisely at the end of the related write feature. For example, in Figure 3.9 we see the features extracted for a 1.0-GB file encryption operation and the point at which the algorithm identifies the encryption operation occurring.

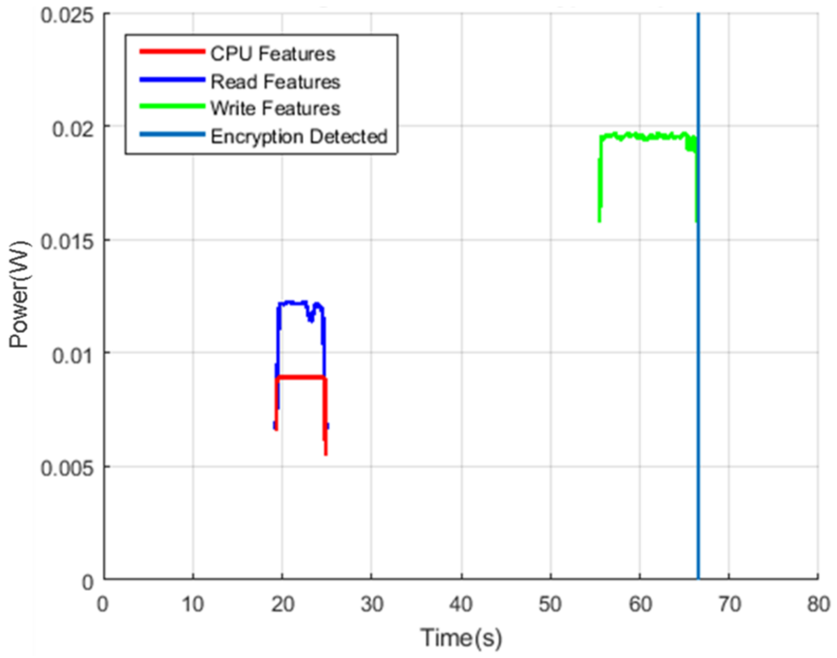

Figure 3.9. 1.0-GB File Encryption Detection

3.4 Summary

In this chapter, an algorithm was presented that uses the power consumption of a computer's SSD and CPU to detect the execution of encryption operations. The algorithm uses thresholds to extract features from the power traces. Once features are extracted from the SSD, they are compared using a linear relationship to determine the possibility of encryption. After this comparison is completed, features from the CPU are used to validate the presence of encryption. In the next chapter, the results of the proposed algorithm are discussed including testing with Wannacry ransomware.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4: Results and Analysis

Testing of the encryption detecting algorithm was conducted throughout the design phase, yielding a product with a high degree of accuracy for larger encryption operations. Each of the steps discussed in Chapter 3 was tested individually with encryption operations and with other similar actions. This process ensured that each module performed as expected and improved the overall performance of the algorithm. Finally, this culminated with an evaluation of the performance of the algorithm with detection of encryption operations and ransomware.

4.1 Encryption Detection

The design purpose of the algorithm is to detect encryption operations occurring on a computer through analyzing the power consumption of the CPU and SSD. The initial experiments were deigned to determine the detection rate of encryption of a range of file sizes. The goal of these experiments was to determine the upper and lower limits for the sizes of encrypted data that could be detected. Additionally, the algorithm was tested against power traces that held similar sized operations along with encryption. This test demonstrated the algorithm's ability to avoid false positives and determine the overall accuracy of the process. Finally, an experiment was performed with the detection of ransomware. Specifically, a malware created in 2017 called Wannacry [1] was analyzed and used to demonstrate the effectiveness of the detection algorithm.

4.1.1 Experimental Setup

In order to gather the power consumption of the CPU and SSD on a computer, a setup was designed to measure and store the corresponding data. The entire system is automated through the use of a data gathering program that synchronizes execution of operations with a data recorder through the use of the computer's serial port. The overall setup is shown in Figure 4.1.

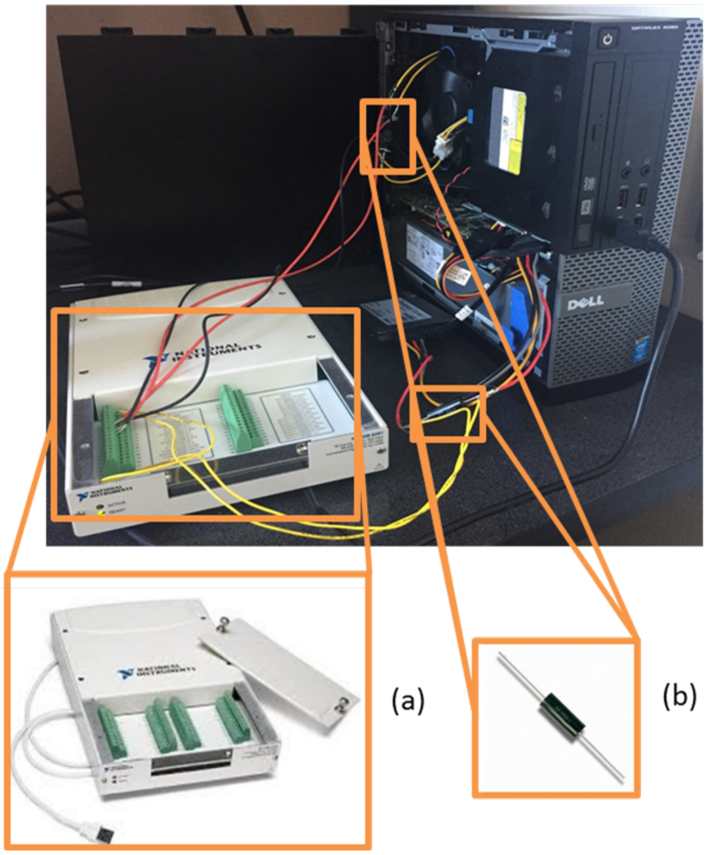

Figure 4.1. Experimental Setup

Computer Setup

The test bed computer system is an Intel Core i7-4790 with 16 GB of RAM running Ubuntu 18.04 LTS. The setup contains two drives, the first is a 240 GB Seagate 600 Pro SSD that contains the operating system, and the second is a 64 GB Crucial M4 SSD where the tests occur. Having two drives with one specifically dedicated for testing ensures that all power fluctuations are a direct result of the testing commands as opposed to the OS. This is accomplished because the files for the OS are stored on the Seagate SSD and, when accessed, have no impact on the test bed SSD because of their physical separation.

SSD and CPU Monitoring

First, a method was chosen to collect the power being consumed by the SSD and CPU. The three options that were considered was using a Rogowski coil to sample the current being drawn by the components, the use of small-value precision resistors, or a current probe. The small-value resistors were chosen to be placed in series with the power supply and can be seen in (b) of Figure 4.1. Specifically, 0.1-Ω resistors with a tolerance of 1.0% and a power rating of 5.0 W were used. In past research from [10], current sensing resistors were used with great effect. In this setup, the SSD's power supply was accessed by placing a precision resistor in series with the 5.0-V wire within the Serial Advanced Technology Attachment (SATA) power cable. The CPU's power was gathered by splicing both 12.0-V power cords in Advanced Technology eXtended (ATX) 12V four pin power supply cord and placing the precision resistors in series. Once the resistors were placed in series with the power supply to both the SSD and CPU, a data acquisition device was connected to the ends of the resistors in order to monitor their voltage. Specifically, a National Instruments USB-6281 multifuction I/O device shown in (a) of Figure 4.1, was used to sample the voltage across the resistors with 18 bits of precision and at a rate of 50 ksamples/s. The data recorder was triggered externally through a USB-to-Serial connection from the test computer to coordinate the start and stop times of the data collection experiments.

4.2 Detection Rate

To determine if the proposed algorithm can successfully predict encryption, a method was developed to test the limits of the process in terms of size of detectable encryption operations. Specifically, a range of file sizes were selected, and power samples of those files being encrypted were collected. For this experiment the file sizes used are shown in Table 4.1.

Table 4.1. Number of Encryption Data Recordings

File Size (MB)	**50**	**100**	**250**	**500**	**750**	**1000**	**2500**	**5000**
Number of Encryption Samples	46	46	46	46	46	46	46	46

In order to gather power samples of encryption operations, a program was created to encrypt files using AES encryption in a block cipher mode. This means that each block of bits encrypted is used to initialize the next block of bits to be encrypted. An advantage of this method is that multiple blocks of the same plain text result in different cipher texts and as a result, the cipher text are more resistant to frequency-analysis attacks. Block cipher mode is used by various known ransomwares. An automated executable, displayed in Appendix A.3.3, was created to use this program as well as coordinate the data collection of the power consumption of the CPU and SSD. Furthermore, this executable performs an encryption of the file sizes shown in Table 4.1 and then performs a 20-GB file write. The 20-GB file write ensures that the RAM does not keep a copy of the encrypted data so that future operations have to interact with the SSD. This large file write was necessary because in previous attempts to collect data the read operations did not occur when multiple operations were executed on the same file. This is a result of the file being still stored in RAM for sequential operations using the same file. In order to prevent this issue, the entire space on the RAM had to be overwritten. Each of the mentioned file sizes were encrypted 46 times, and their CPU and SSD power traces were recorded. The encryption detection algorithm was then applied to each sample, and the results for detection are displayed in Figure 4.2 and Table 4.2.

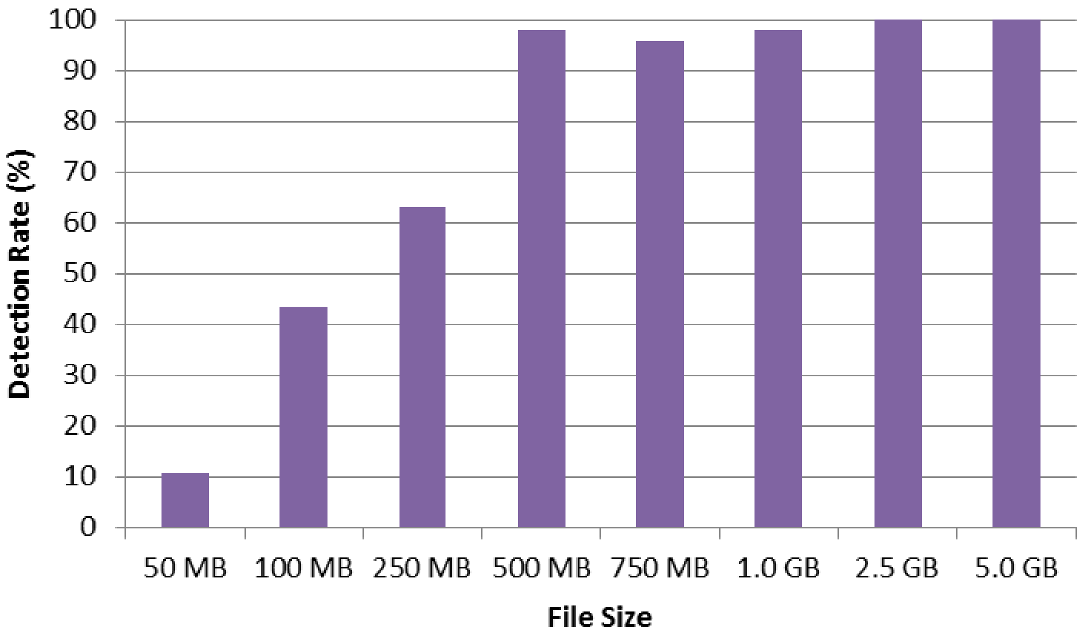

Figure 4.2. Encryption Detection Rate for File Sizes

Table 4.2. Detection Percentage for Encryption Operations

	File Size(MB)							
	50	**100**	**250**	**500**	**750**	**1000**	**2500**	**5000**
Detection Rate(%)	10.8	43.5	63	97.8	95.7	97.8	100	100

As we can see from the detection rates displayed in Table 4.2, the algorithm was able to successfully detect encryption over 95 percent of the time for file sizes of 500 MB or larger. For file sizes 250 MB and smaller, there is a decline in detection rate as file size decreases. The downtrend in performance is a result of a decrease power consumption impact on the SSD and CPU as file size decreases. An example of this decreased power impact is illustrated in (a) of Figure 4.3 where the read step in the encryption operation fails to elicit a significant response by the algorithm. The failure to extract a read feature due to a lack of impact on the power consumption is shown in (b) of Figure 4.3.

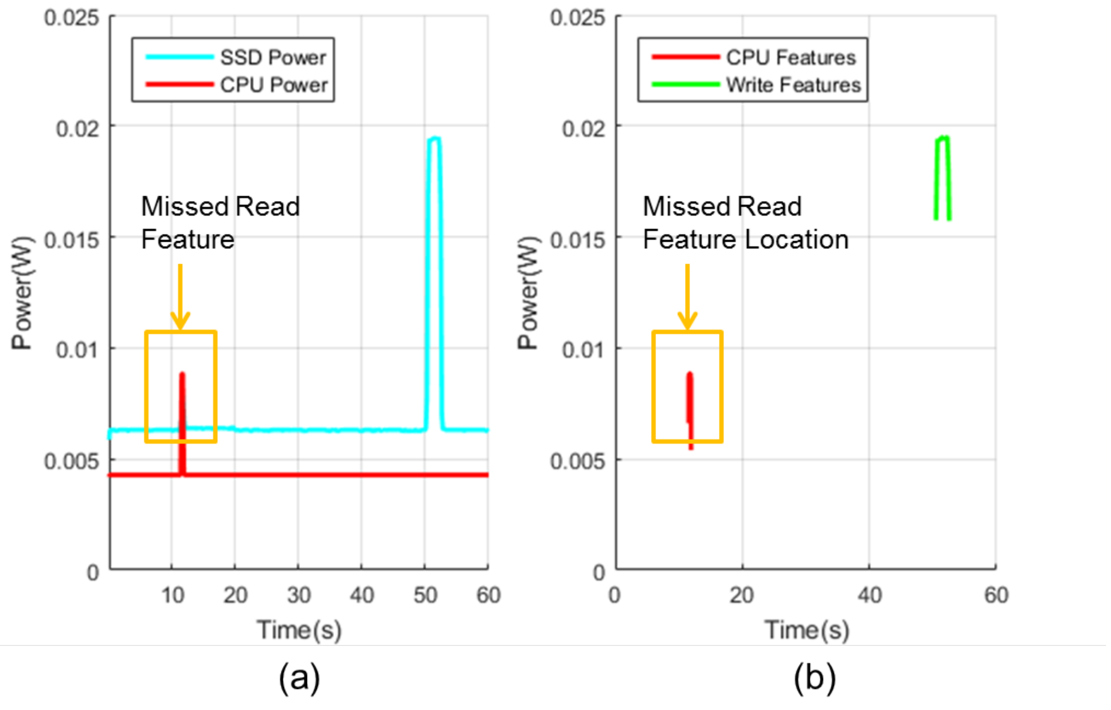

Figure 4.3. Example of Missed Read Feature for a 100 MB Encryption Operation

This trend affects the algorithm in two ways. First, as stated in Chapter 3, a median filter is used to help reduce noise. If the feature size in smaller than the window used in the median filter, the power increase is eliminated in the filter output. Secondly, as the power impact of the encryption operations is reduced, the relationship between the integrated values of the read and write features varies to a greater degree. As a result, the linear function used to compare the two features may not be able to overcome the greater variation for the read and write operation relationship and does not classify the operation as encryption. To overcome these shortfalls, a higher sampling rate would help alleviate the impact of the stated issues.

The results of this experiment indicate that encryption operations of 500 MB and larger can be detected by the proposed algorithm. Today, with the average computer carrying at least a 500 GB hard drive, detecting an encryption operation from ransomware the size of 500 MB leaves a large portion of the data yet to be modified. After establishing that the detection of encryption operations above 500MB are feasible, an experiment to see how well the algorithm performed in a noisy environment was devised.

4.3 Testing in a Noisy Environment

For a binary classifier to be considered successful, it must be able to accurately predict target events while not miss-classifying operations. In the previous section, the encryption detection algorithm showed its ability to classify encryption operations, and in this section the algorithm is tested to determine its capability to classify true positive and true negatives. Specifically, an experiment was designed to randomly select 18 operations with power impacts similar to encryption and two actual encryption operations and test the algorithms classification ability. These were file copy operations of the same size as the tested encryption operations. This experiment was conducted 20 times for the file sizes 50 MB, 100 MB, 250 MB, 500 MB, 750 MB, 1.0 GB, 2.5 GB, and 5.0 GB. The results of these experiments are illustrated with a confusion matrix displayed in Tables 4.3- 4.10.

Table 4.3. Confusion Matrix of 50-MB Encryption Operations in Noisy Environment

n=400	**Predicted: Encryption**	**Predicted: No Encryption**	
Actual: Encryption	8	32	40
Actual: No Encryption	0	360	360
	8	392	

Table 4.4. Confusion Matrix of 100-MB Encryption Operations in Noisy Environment

n=400	**Predicted: Encryption**	**Predicted: No Encryption**	
Actual: Encryption	17	23	40
Actual: No Encryption	0	360	360
	17	383	

Table 4.5. Confusion Matrix of 250-MB Encryption Operations in Noisy Environment

n=400	**Predicted: Encryption**	**Predicted: No Encryption**	
Actual: Encryption	29	11	40
Actual: No Encryption	4	356	360
	33	367	

Table 4.6. Confusion Matrix of 500-MB Encryption Operations in Noisy Environment

n=400	**Predicted: Encryption**	**Predicted: No Encryption**	
Actual: Encryption	36	4	40
Actual: No Encryption	2	358	360
	38	362	

Table 4.7. Confusion Matrix of 750-MB Encryption Operations in Noisy Environment

n=400	**Predicted: Encryption**	**Predicted: No Encryption**	
Actual: Encryption	37	3	40
Actual: No Encryption	5	355	360
	42	358	

Table 4.8. Confusion Matrix of 1.0-GB Encryption Operations in Noisy Environment

n=400	**Predicted: Encryption**	**Predicted: No Encryption**	
Actual: Encryption	38	2	40
Actual: No Encryption	2	358	360
	40	360	

Table 4.9. Confusion Matrix of 2.5-GB Encryption Operations in Noisy Environment

n=400	**Predicted: Encryption**	**Predicted: No Encryption**	
Actual: Encryption	40	0	40
Actual: No Encryption	0	360	360
	40	360	

Table 4.10. Confusion Matrix of 5.0-GB Encryption Operations in Noisy Environment

n=400	**Predicted: Encryption**	**Predicted: No Encryption**	
Actual: Encryption	40	0	40
Actual: No Encryption	0	360	360
	40	360	

Table 4.11. F score for Test in Noisy Environment

	File Size (MB)							
	50	**100**	**250**	**500**	**750**	**1000**	**2500**	**5000**
Recall	0.2	0.425	0.725	0.9	0.925	0.95	1	1
Precision	1	1	0.879	0.947	0.881	0.95	1	1
F score	0.333	0.596	0.794	0.923	0.902	0.95	1	1

When analyzing a confusion matrix, ideal results in this situation are large numbers in the diagonal, corresponding to true positives and true negatives. The confusion matrices for the encryption operations of 500 MB and larger all have very large values in the diagonal for true positives and true negatives. As a result the encryption detecting algorithm shows its robustness in still identifying encryption in a high activity environment.

In order to numerically determine the success of the algorithm, a statistic called the F score was calculated, displayed in Table 4.11. The F score is calculated using two other statistics called recall and precision. Precision is the number of correct positive predictions divided by the total number of positive predictions. A high precision shows that a classifier, when predicting a true positive, is usually right. Recall is the amount of correct positive predictions divided by the number of samples that should have been predicted positive. A large value for recall shows that a classifier is able to return most relevant results or true positives. The F score calculation is computed as [42]

$$Fscore = 2/((1/Recall) + (1/Precision)), \tag{4.1}$$

where the resulting score is a combination of recall and precision. As we see from the Table 4.11, the algorithm's ability to detect encryption operations is shown by the recall statistic. As file size increases, the algorithm is able to increasingly detect the encryption operation. The precision statistic reveals that when the algorithm classifies an operation as an encryption operation, it is highly likely that the operation is encryption. Due to the reduced impact on the power consumption at smaller file sizes, file copy operations do not produce features that indicate encryption; however, as file sizes increase, the features extracted from file copies can be confused with encryption in some cases. This relationship explains the trend of a high precision value decreasing slightly from 50 MB to 750 MB as shown in Table 4.11; however, this trend reverses and increases from file sizes 750 MB to 5.0 GB due to the more consistent feature sizes seen in larger files. This enables the algorithm to better identify encryption operations. In Table 4.11 is an illustration that the F score and the recall statistic have the same general trend of starting low and increasing to one, demonstrating that the effectiveness of the algorithm increases as file sizes escalate and eventually plateauing at a file size of 500 MB. As a result, the algorithm can be considered extremely effective for file sizes of 500 MB and larger.

4.4 Testing Against Wannacry Ransomware

The ultimate goal of this research was to create a method for ransomware detection based on analysis of the power consumption of the SSD and CPU. In the past sections, the proposed algorithm was demonstrated to have a high degree of accuracy for encryption operations above 500 MB. With the assertion that detecting encryption can identify the execution of crypto ransomware, a final experiment was designed to apply the encryption detection algorithm to a sample of Wannacry ransomware. The purpose behind this experiment is to demonstrate that an encryption detecting algorithm is effective in identifying the execution of ransomware. With the same hardware setup used in the previous experiments, a 50 GB Windows 7 virtual machine (VM) was run on the 64 GB secondary storage SSD. Within this VM, a directory of 14 GB of random text files was created to simulate a user's data. The ransomware executable was then run while the power consumption of the computer's SSD and CPU was collected, this is displayed in Figure 4.4. The algorithm's extracted features and locations or detected encryption are shown in Figure 4.5.

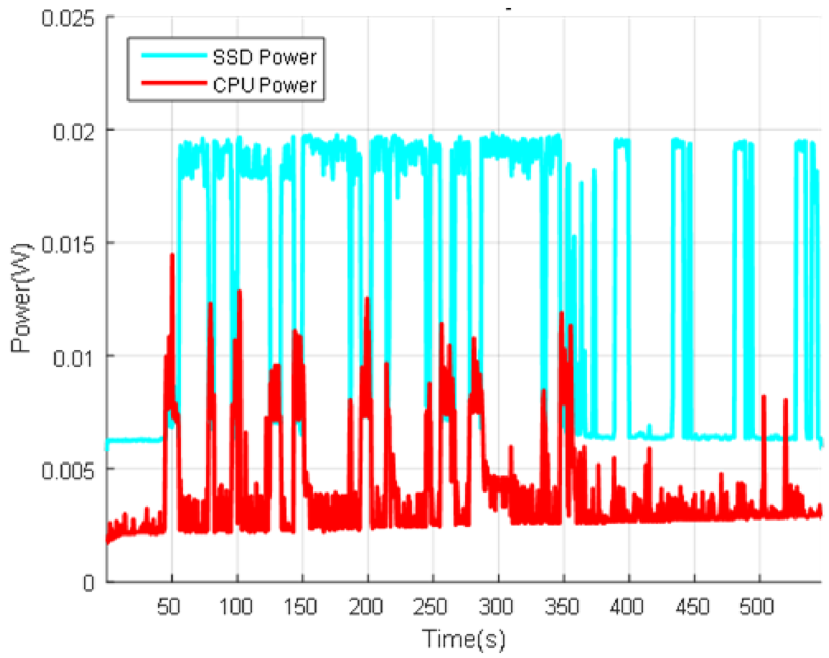

Figure 4.4. Wannacry Ransomware Attack Power Plot

The power impact of the encryption operations occurring as a result of the ransomware's process to encrypt a targets data is illustrated in Figure 4.4. The features extracted from the ransomware's power trace correspond to a previous assertion that encryption operations follow the procedure of performing a read operation, encrypting that data and then subsequently executing a write operation. The repetition of this pattern is displayed Figure 4.5, where read features occur at the same time as CPU features and afterwards a write feature occurs. The proposed algorithm clearly identifies this process with vertical blue line

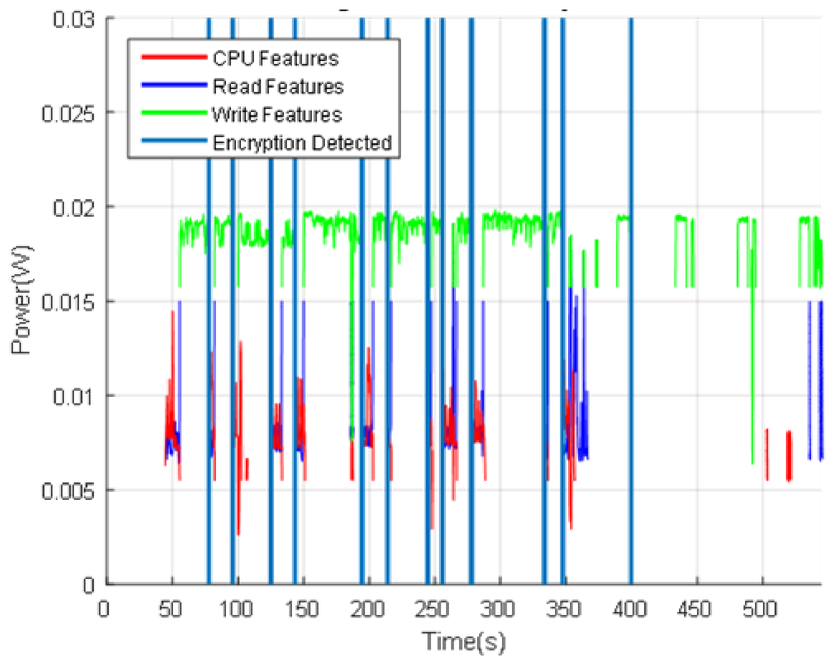

Figure 4.5. Wannacry Ransomware Attack Features

representing the detection of encryption in illustrated in Figure 4.5. The experiment was conducted ten times, and in each case the proposed encryption detection algorithm clearly identified the presence of encryption operations within the power traces gathered from the SSD and CPU.

4.5 Summary

In this chapter, several experiments were discussed that involved calculating the performance of the encryption detecting algorithm. Using many samples of encryption for a range of file sizes, we demonstrated a detection rate for encryption to exceed 95 percent for files sizes 500 MB and greater. In addition, the algorithm was exposed to a noisy environment and established a high degree of accuracy for identifying encryption operations. The best performance was seen in file sizes greater than 500 MB due to the consistent noticeable impact on the power traces of the SSD and CPU for larger file sizes. On application to a sample of Wannacry ransomware, the algorithm detected multiple instances of encryption in every test run. These results clearly show that crypto ransomware can be identified by power consumption of a computer's SSD and CPU.

CHAPTER 5:
Conclusions and Recommendations

5.1 Summary

With global digital interconnectivity ever increasing in the modern age, new challenges for maintaining secure communications and networks occur constantly. In particular, cyber criminals are continually in the process of developing malicious software to compromise legitimate organizations. Recently, the state of malware has seen the emergence of ransomware as the most prevalent threat to devices and networks. With millions of dollars in damages caused each year by ransomware, measures for detection and prevention are of great importance to institutions worldwide. Previous research has suggested the use of signature detection methods, honeypot techniques, entropy, and machine learning algorithms as a way to detect ransomware. Unfortunately these suggested solutions lack consistency, fail to protect against new versions of malware, and have large delays between malware creations and eventual implementation of counter measures. To provide a remedy to these shortcomings, in this research we proposed an algorithm that focuses on detecting the universal characteristic of all crypto ransomwares, encryption. The proposed method uses power analysis techniques to classify the steps of encryption and correlate them to identify when encryption is occurring on a computer.

To detect encryption by analyzing the power consumption, an initial understanding of the steps in an encryption operation is required. For the purpose of this research, encryption was broken down into the following three steps: file read from secondary memory, encryption of data in RAM, and file write back to secondary memory. From these steps we concluded that the SSD and CPU components would have large impacts on their power consumption. In addition, past research has shown that read and write operations can be identified by evaluating the power traces of SSDs [10]. As a result, an automated testing setup was designed and created to monitor the power consumption of these parts while operations could be executed on the computer.

With a data collection process for the power consumption of a computer's SSD and CPU, an algorithm was devised for the detection of the previously mentioned steps for encryption. This algorithm identifies read and write operations from the SSD's power trace and CPU activity from the CPU's power trace. Additionally, a classifier uses a function to compare the energy usage of extracted read and write operations to verify the execution of an encryption operation. An experiment was devised to test the accuracy of the encryption detecting algorithm. A test with varying size of files undergoing encryption revealed that the algorithm is able to detect encryption operations 500 MB and larger with greater than 95 percent accuracy. Smaller file sizes have a reduced consistent impact on the power consumption of the SSD and CPU which decreases the accuracy of the algorithm; however, these results are not concerning since ransomware attacks typically encrypt gigabytes of information. To further test the design, an experiment was devised to apply the algorithm to a recent sample of ransomware called Wannacry. In all runs, the proposed algorithm predicted encryption several times. These results indicate that crypto ransomware can be detected through power-analysis techniques.

The original contribution in this research is an algorithm and test setup capable of detecting encryption. When applying this algorithm to ransomware, due to the nature of the malware, encryption processes are detected. When such processes are observed, it is possible to stop the computer processes until verified by the user. As a result, ransomware activity can be halted before continuing to encrypt data and possibly preventing further damage in data loss. Stopping encryption early in the ransomware's process limits the amount of data being no longer accessible by the user while saving most of the files on a computer.

5.2 Future Work

5.2.1 Identifying Smaller Encryption Operations

With the current setup, the encryption detection algorithm is only able to accurately detect encryption operations of sizes 500 MB and larger. As discussed in Chapter 4, the current implementation struggles to overcome the issue of smaller operations having a reduced power consumption impact. To account for this reality, improvements in the sampling rate could pick up the smaller power impacts. Additionally, a different model can be implemented to check for read and write operations relationship being indicative of an encryption operation. The current method uses a linear regression with an acceptable error region to identify a correlation; however, with further investigation, a different method of classification can be implement to improve the algorithm's ability to associate read and write operations at the lower magnitudes seen in smaller file sizes.

Other methods for comparing the features extracted may have a higher degree of performance. At the early stages of this research, using the length of the CPU, read, and write features was considered as a method of comparison instead of using integrated values. This method was never fully investigated. Additionally, in [43] frequency analysis of SSD power consumption was used as a method to determine characteristics of the onboard file system. Perhaps a similar method could be used an avenue to extract additional features from the power traces and, in turn, reveal another method of classification. Furthermore, another power consumption source such as the RAM component may yield more identifying features. Since each of the three steps of encryption identified in this research interact with RAM, more information can be extracted from this component. With the main issue for smaller encryption files being their reduced interaction with the current collection methods, any of the new suggested methods can help improve the current method's performance.

5.2.2 User Authentication from Computer Power Consumption

Industries throughout the world have issues with network and digital security. Similar to the problem of identifying malicious code such as ransomware, another vital issue is verifying user and user activity on computers and networks. As a result there have been many proposed methods for building a profile for users based on network activity and even physical movements such as the way a computer mouse is handled [44]. In a similar application, a computers power consumption can be used to provide user identification. When the built profile does correspond to current behavior, it may signal the use of an unauthorized user or a malicious program executing operations. The future work aspect of this topic is the creation and implementation of a method that extracts features from the power consumption of a computer and uses the information as a method for user authentication.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX: Power Plots of Data

A.1 Power Plots

The following figures are power plots of the encryption samples that were gathered to test the detection rate of the proposed algorithm

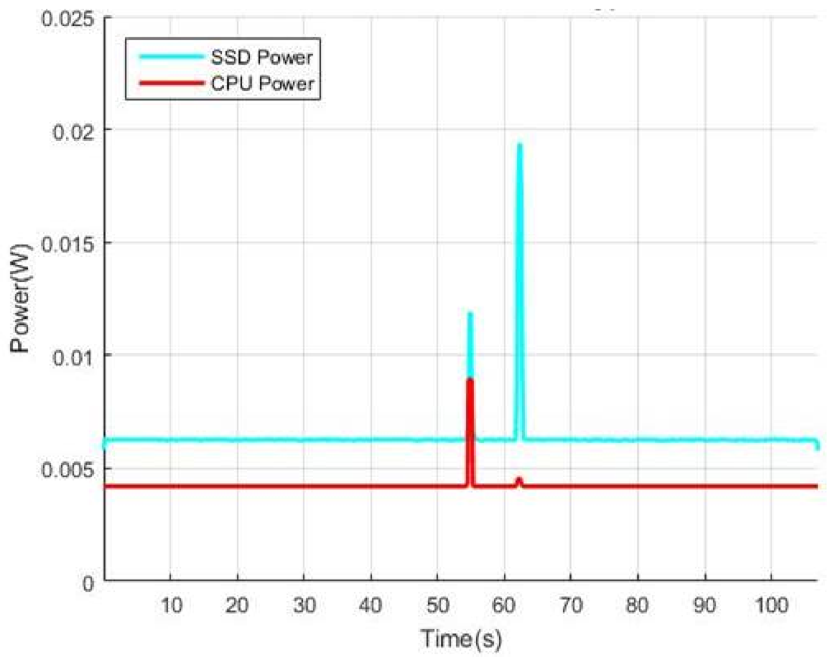

Figure A.1. Power Plot of a 50-MB File Encryption

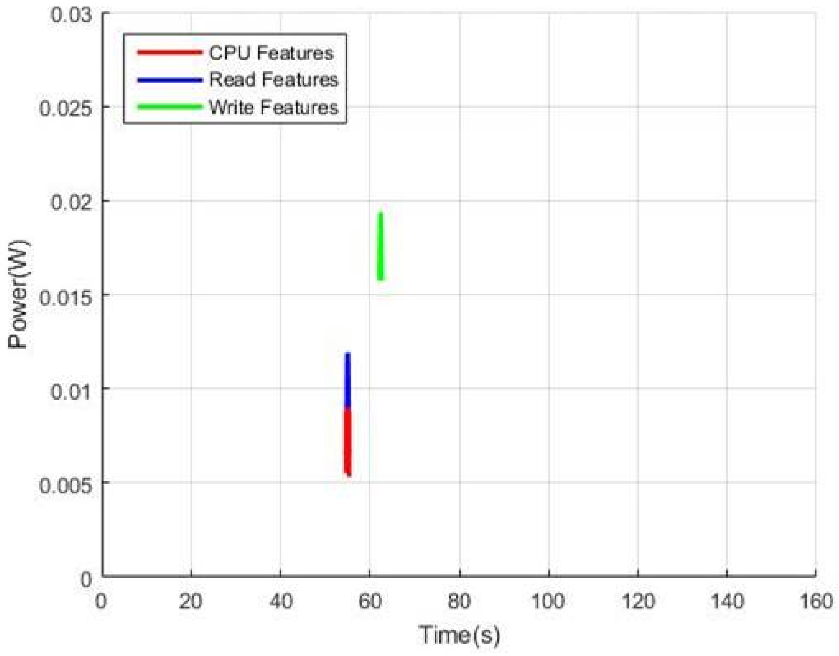

Figure A.2. Feature Plot of a 50-MB File Encryption

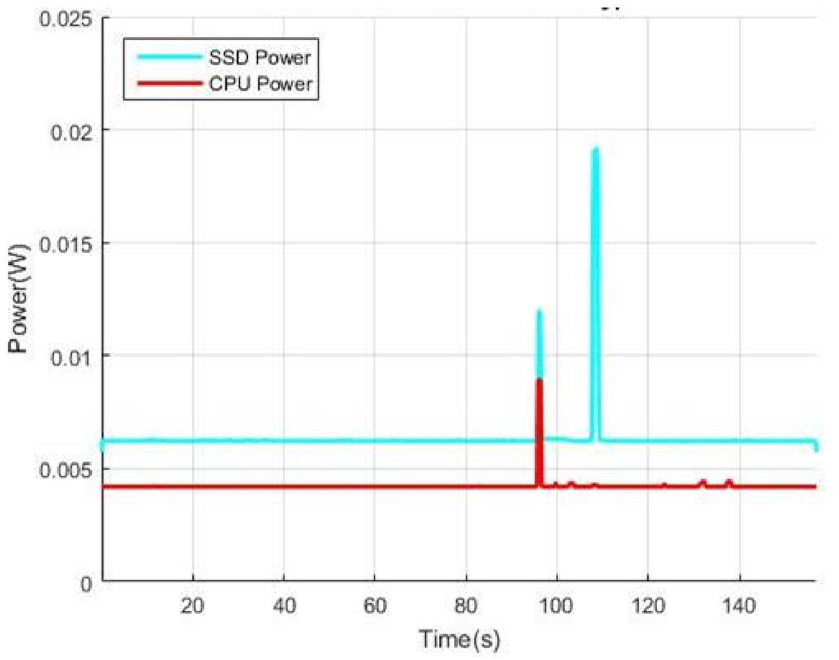

Figure A.3. Power Plot of a 100-MB File Encryption

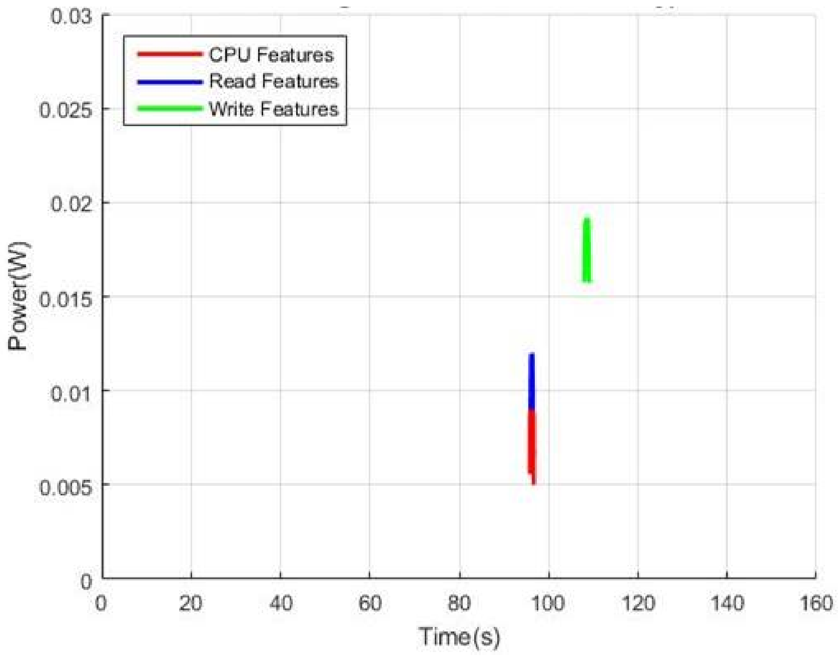

Figure A.4. Feature Plot of a 100-MB File Encryption

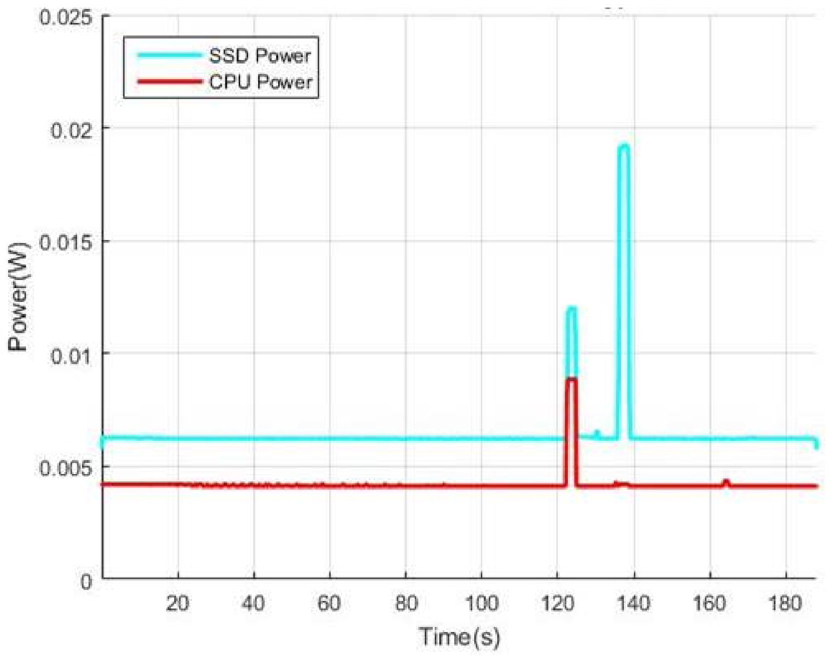

Figure A.5. Power Plot of a 250-MB File Encryption

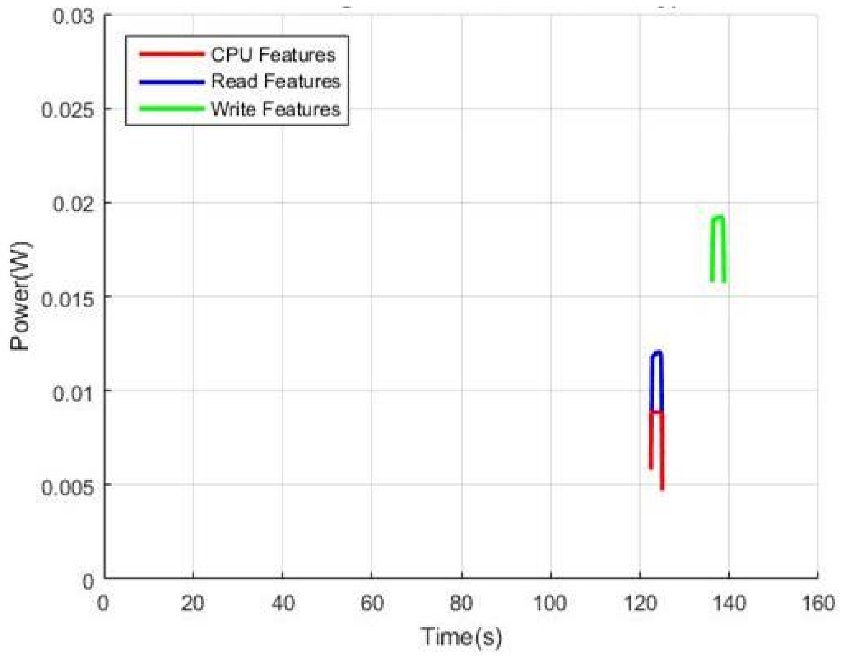

Figure A.6. Feature Plot of a 250-MB File Encryption

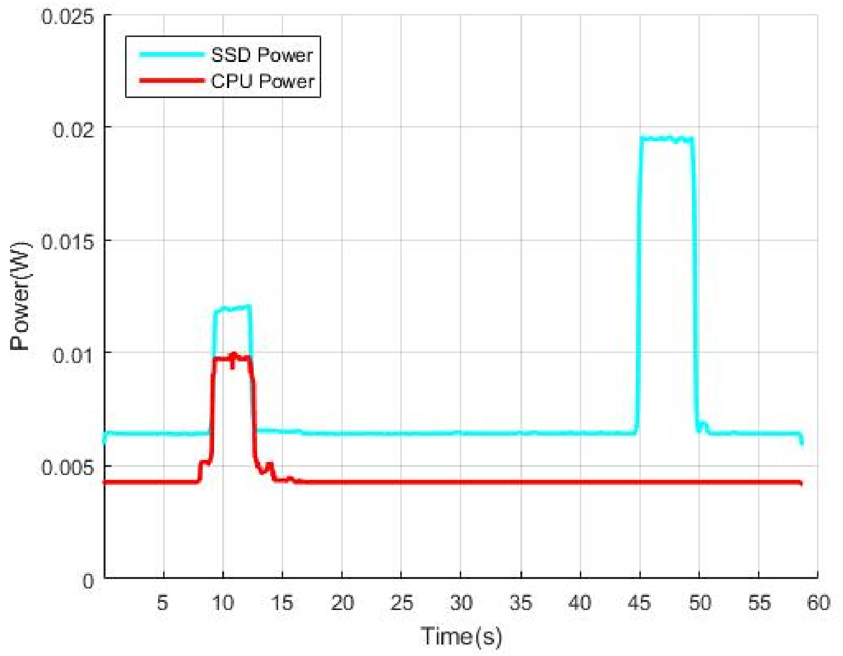

Figure A.7. Power Plot of a 500-MB File Encryption

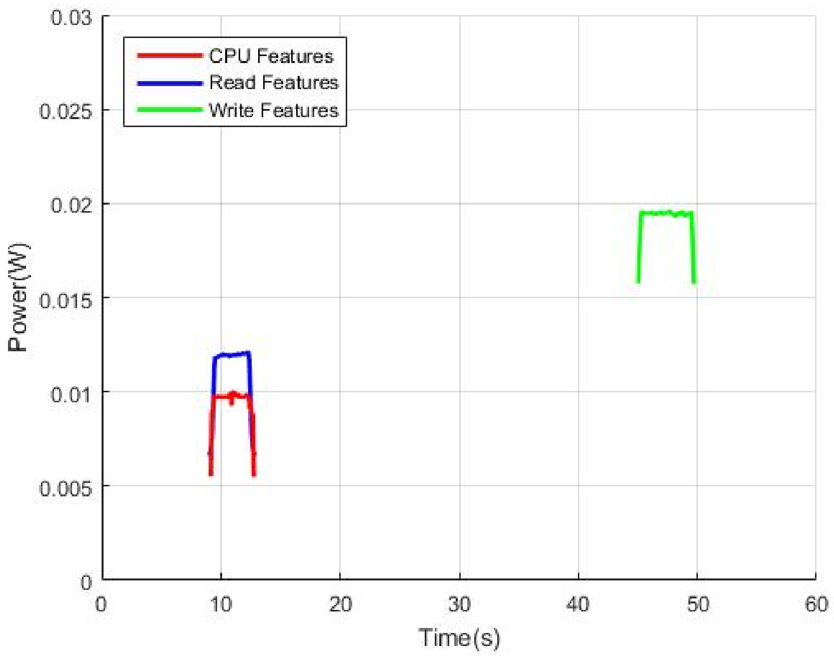

Figure A.8. Feature Plot of a 500-MB File Encryption

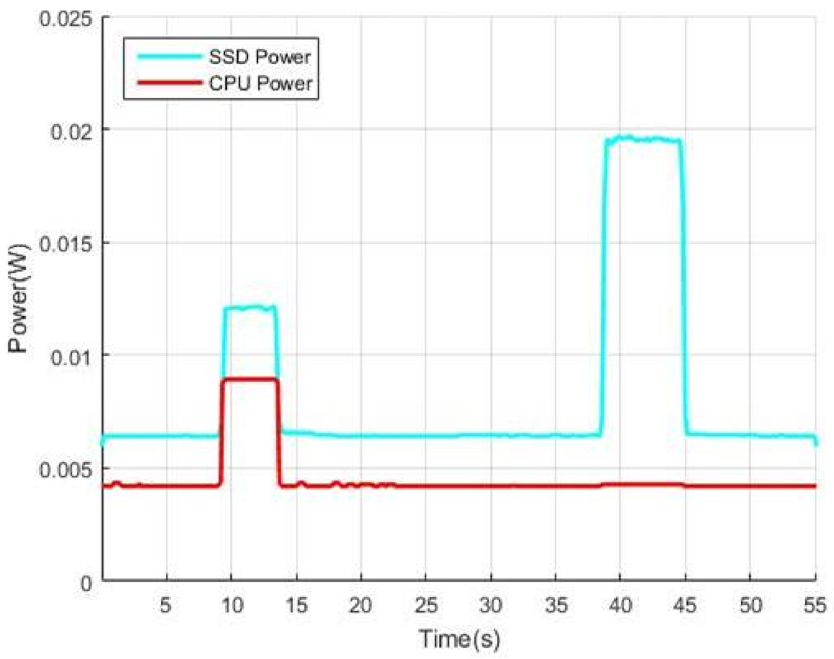

Figure A.9. Power Plot of a 750-MB File Encryption

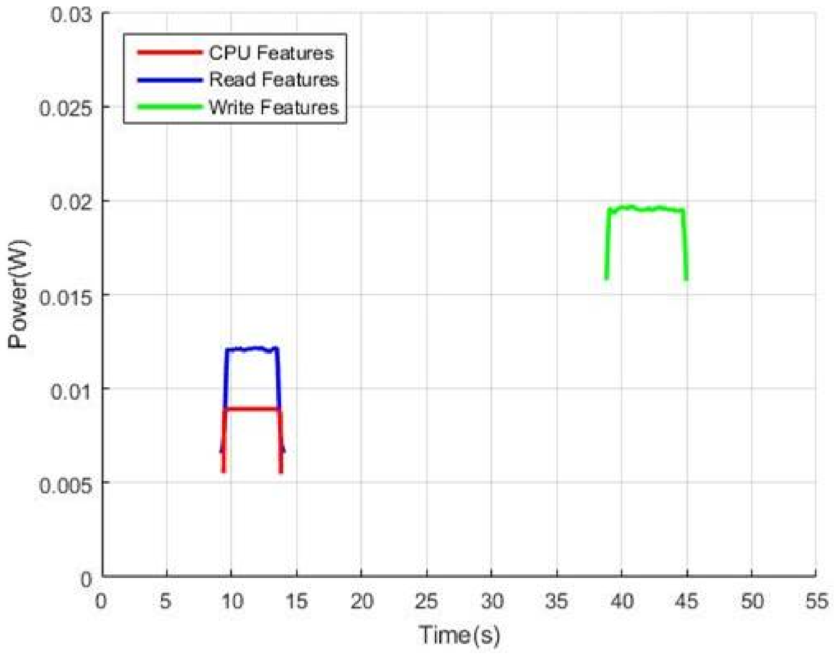

Figure A.10. Feature Plot of a 750-MB File Encryption

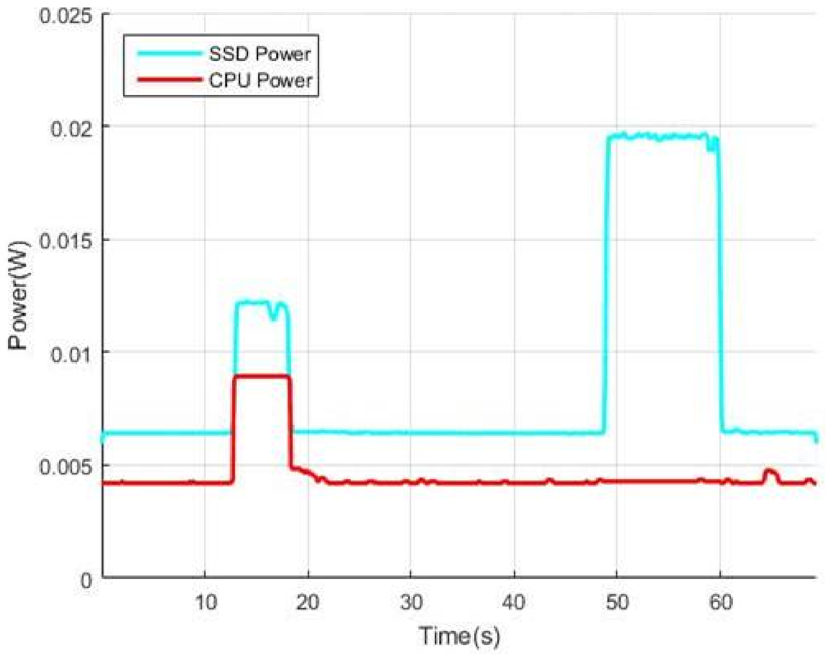

Figure A.11. Power Plot of a 1.0-GB File Encryption

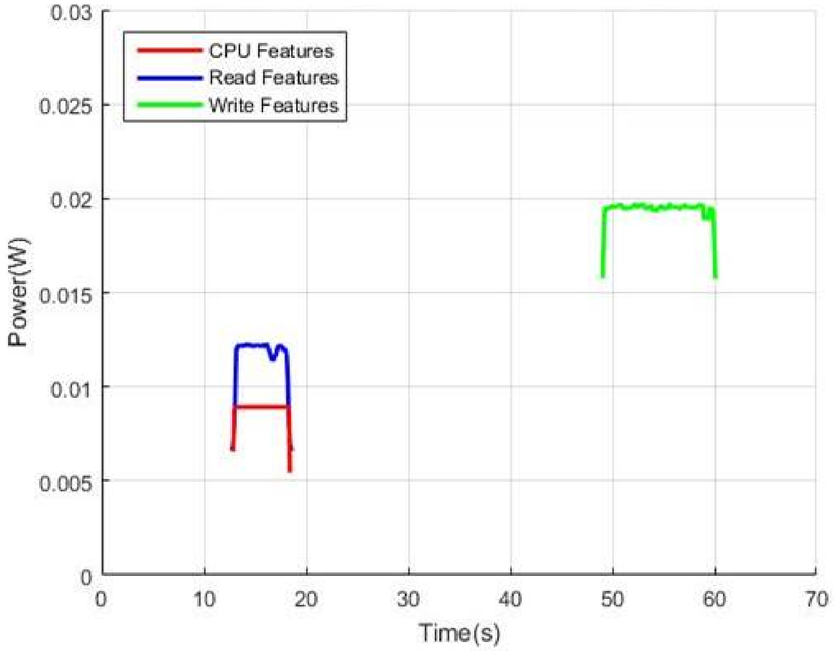

Figure A.12. Feature Plot of a 1.0-GB File Encryption

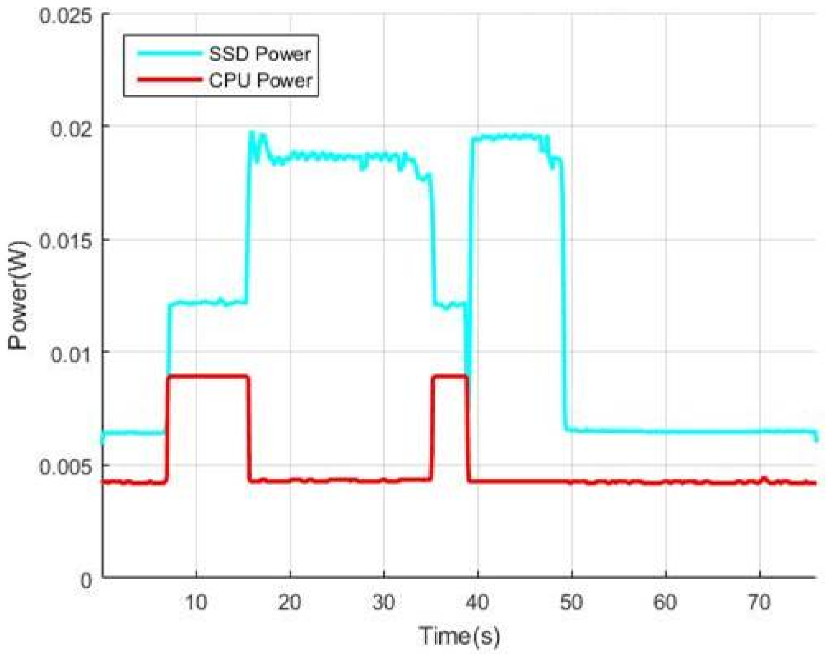

Figure A.13. Power Plot of a 2.5-GB File Encryption

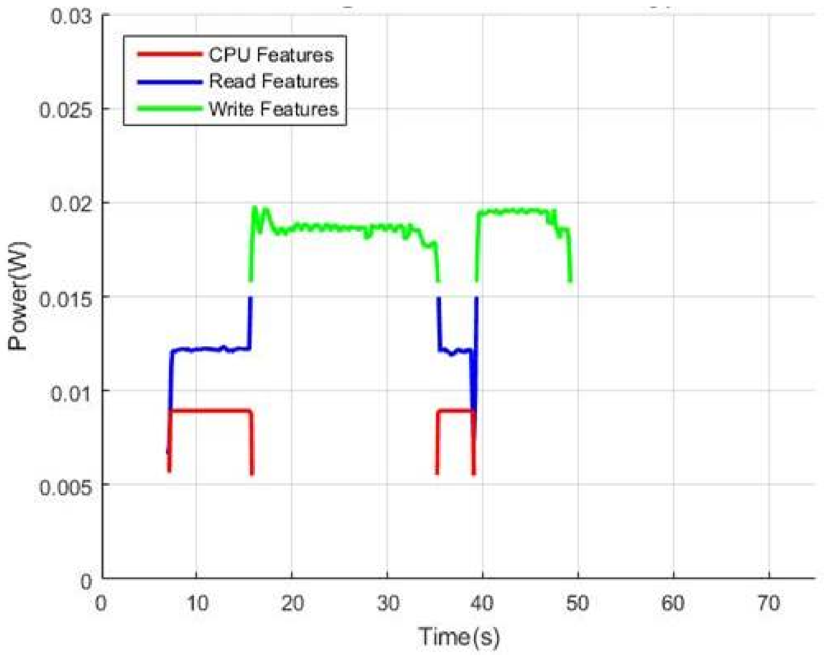

Figure A.14. Feature Plot of a 2.5-GB File Encryption

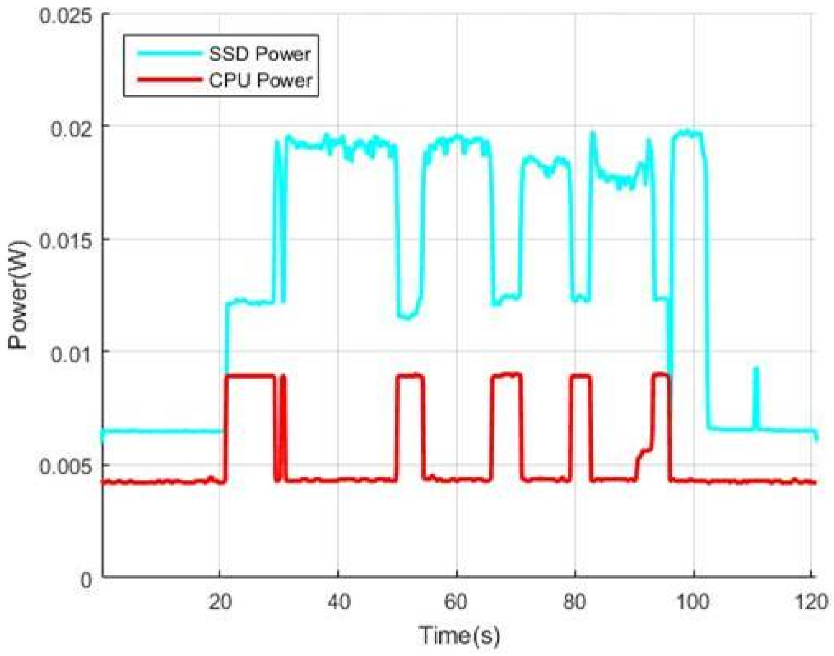

Figure A.15. Power Plot of a 5.0-GB File Encryption

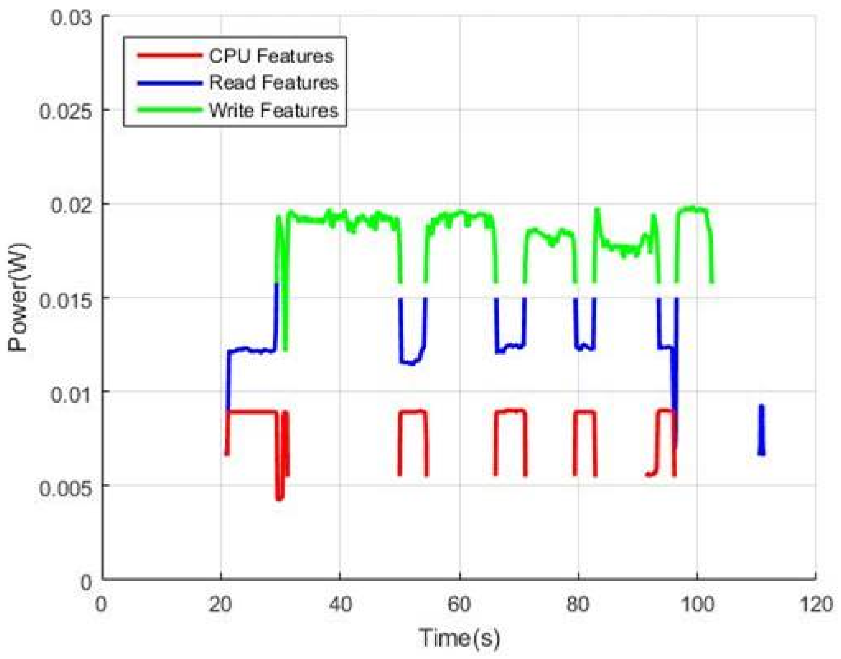

Figure A.16. Feature Plot of a 5.0-GB File Encryption

A.2 MATLAB Code for Classification Algorithm

A.2.1 Main Code Block for Classification Algorithm

```
clear all
close all

%Specify file to be tested and load data
file='5GB2';
loc=strcat('H:\MyDocs\transfer3\',file);
load(strcat(loc,'/Alldata.mat'));
CPUpower=data{1};
SSDpower=data{2};

%Get Read features
[ featureRead,featureReadIntegrated ]=getReadFeatures(SSDpower);
%Get Write features
[ featureWrite,featureWriteIntegrated ]=getWriteFeatures(SSDpower);
%Get CPU features
[ featureCPU,featureCPUIntegrated ] = getCPUFeatures( CPUpower );
```

```matlab

%Calculate sizes of read features
Readsize=1;
for i=2:2:length(featureRead)
     temp=featureRead(i)-featureRead(i-1);
     Readsize=[Readsize,temp];
end
Readsize(1)=[];

%Calculate sizes of write features
Writesize=1;
for i=2:2:length(featureWrite)
     temp=featureWrite(i)-featureWrite(i-1);
     Writesize=[Writesize,temp];
end
Writesize(1)=[];

%Filter the power signals to look beter for plots
SSDpower=medfilt1(SSDpower,10000);
CPUpower=medfilt1(CPUpower,10000);

detectEncryption=[];
for i=1:length(featureReadIntegrated)
    for k=1:length( featureWriteIntegrated)
         clear temp

        %Percent error accepted for association of read and write
        %feature size
          percentError=67;

         %Equation to associated read and write features based on
    integrated size
                 tempp1=(0)*featureReadIntegrated(i)^(2)+(5.0011)*
    featureReadIntegrated(i)-85.7321;
                 tempp1=((featureWriteIntegrated(k)-tempp1)/tempp1)*100

         %Equation to associated read and write features based on length
         %tempp1=(0)*Readsize(i)^(2)+(5.0011)*Readsize(i)-85.7321;
         %tempp1=((Writesize(k)-tempp1)/tempp1)*100

        if (abs(tempp1) <percentError)
            %test for deatures being near in time
            temp2=featureWrite(2*k-1)-featureRead(2*i-1);
            if (temp2 <1400000 && temp2 >0)
                %read and write feature are correlated to indicate
                %encryption
                disp('yes encryption for read and write')
                for m=1:length(featureCPUIntegrated)
                    temp3=abs(featureCPU(2*m-1)-featureRead(2*i-1));
                    if temp3<50000
                        %Encryption is detected by associating a read,
                        %write and CPU feature
                          disp('yes encryption for read and write and cpu')
                          detectEncryption=[detectEncryption,featureWrite(2*
    k)]
                    end
                end
            end
        end
    end
end

%Figure to plot features and loactions of detected encryption
try
figure
t=1:length(SSDpower);
t=t./18000;
hold on

     %Plot read features
for i=1:2:length(featureRead)
     h2=plot((featureRead(i):featureRead(i+1))/18000,SSDpower(featureRead
    (i):featureRead(i+1))*.5,'b');
     tmp=length(featureRead)-i;
     set(h2,'linewidth',2);
     if tmp ==1
         break;
     end
end
     %Plot CPU features
```

```
for i=1:2:length(featureCPU)
    h1=plot((featureCPU(i):featureCPU(i+1))/18000,CPUpower(featureCPU(i)
   :featureCPU(i+1)),'r');
    tmp=length(featureCPU)-i;
    set(h1,'linewidth',2);
    if tmp ==1
        break;
    end
end
   %Plot write features
for i=1:2:length(featureWrite)
    h3=plot((featureWrite(i):featureWrite(i+1))/18000,SSDpower(
   featureWrite(i):featureWrite(i+1))*.5,'g');
    tmp=length(featureWrite)-i;
    set(h3,'linewidth',2);
    if tmp ==1
        break;
    end
end

    %Ploting locations where encryption was detected
for i=1: length(detectEncryption)
    h4=line([detectEncryption(i)/18000 detectEncryption(i)/18000],[0
   .045]);
     set(h4,'linewidth',2);
end
grid on
title(strcat('Feature recognition of: ',' 5GB File Encryption'));
xlabel('Time(s)')
ylabel('Power(W)')
axis([0 120 0 .03])
try
legend([h1,h2,h3,h4],'CPU Features','Read Features','Write Features','
   Encryption Detected','Location','northwest');
catch
    try
        legend([h1,h2,h3],'CPU Features','Read Features','Write Features
   ','Location','northwest');
    catch
        try
```

```
            legend([h1,h3],'CPU Features','Write Features','Location','
    northwest');
        catch
            try
                legend([h1],'CPU Features','Location','northwest');
            catch
            end
        end
    end
end
hold off
catch
end
    %Plot just power consumption of SSD and CPU
figure
hold on
t=1:length(SSDpower(2000:end-2000));
t=t./18000;
h4=plot(t,SSDpower(2000:end-2000)*.5,'c')
set(h4,'linewidth',2);
h5=plot(t,CPUpower(2000:end-2000),'r')
set(h5,'linewidth',2);
grid on
title(strcat('Power Plot of: ',' 5GB File Encryption'));
xlabel('Time(s)')
ylabel('Power(W)')
legend('SSD Power','CPU Power','Location','northwest');
hold off
axis([-inf inf 0 .025])
```

A.2.2 Function for Extracting Read Features

```
function [ featureRead,featureReadIntegrated ] = getReadFeatures(
    SSDpower )
    featureRead=0;
    featureReadIntegrated=0;

    %Median filter to get rid of noise
    SSDfilt=medfilt1(SSDpower,10000);
    SSDread=SSDfilt;
```

```

%Use thresholds
index = find(SSDfilt<.0132 | SSDfilt>.03);
for i=1:length(index)
    SSDread(index(i))=0;
end
SSDread=medfilt1(SSDread,3000);

%value of idle state
medianofsig=.01238;

%Look for contiguous features
FlagFeature=0;
temp=1;
for i=1:length(SSDread)
    if SSDread(i)>0;
        %feature
        if FlagFeature ==0
            %start of feature
            temp=i;

            FlagFeature =1;
        else
            FlagFeature =1;
        end
    else
        if FlagFeature ==0
            %no feature
            temp=i;

            FlagFeature =0;
        else
            %end of feature
            FlagFeature =0;
            featureRead=[featureRead,temp,i];
        end
    end
end
featureRead(1)=[];
featureRead1=1;
```

```
try
    featureRead1=featureRead(1);
end

%Combine features close to each other
for i=2:2:length(featureRead)
    tmp=length(featureRead)-i;
    if tmp ==0
        break;
    end
    temp=featureRead(i+1)-featureRead(i);
     if temp<40000

     else
         featureRead1=[featureRead1,featureRead(i),featureRead(i+1)
];
     end
    tmp=length(featureRead)-i;
    if tmp ==0
        break;
    end
end
try
    featureRead1=[featureRead1,featureRead(length(featureRead))];
end
clear featureRead
featureRead=featureRead1;

%Extract feature integrated size
try
    for i=1:2:length(featureRead)
        if i==1
            featureReadIntegrated=sum(SSDread(featureRead(i):
featureRead(i+1)))-medianofsig*(featureRead(i+1)-featureRead(i));
        else
            featureReadIntegrated=[featureReadIntegrated,sum(SSDread(
featureRead(i):featureRead(i+1)))-medianofsig*(featureRead(i+1)-
featureRead(i))];
        end
        tmp=length(featureRead)-i;
```

```
                if tmp ==1
                    break;
                end
        end

        %features must exceed  a certain size
        del=0;
        featureRead1=0;
        featureReadIntegrated1=0;
        for i=1:length(featureReadIntegrated)

            if  featureReadIntegrated(i)<18
                %delete index
            else
                featureRead1=[featureRead1,featureRead(i*2-1),featureRead(i*2)];
                featureReadIntegrated1=[featureReadIntegrated1,featureReadIntegrated(i)];
            end
        end
        featureRead1(1)=[];
        featureReadIntegrated1(1)=[];
        clear featureRead featureReadIntegrated
        featureRead=featureRead1;
        featureReadIntegrated=featureReadIntegrated1;
    end
end
```

A.2.3 Function for Extracting Write Features

```
function [ featureWrite,featureWriteIntegrated ] = getWriteFeatures(SSDpower )
    featureWrite=0;
    featureWriteIntegrated=0;

    %Median filter to get rid of noise
    SSDfilt=medfilt1(SSDpower,10000);

    %Use thresholds
    SSDwrite=SSDfilt;
```

```
index = find(SSDfilt<.0315);
for i=1:length(index)
    SSDwrite(index(i))=0;
end
SSDwrite=medfilt1(SSDwrite,3000);

%value of idle state
medianofsig=.01238;

%Look for contiguous features
edgeUP=0;
edgeDown=0;
FlagFeature=0;
temp=1;
for i=1:length(SSDwrite)
    if SSDwrite(i)>0;
        %feature
        if FlagFeature ==0
            %start of feature
            temp=i;

            FlagFeature =1;
        else
            FlagFeature =1;
        end
    else
        if FlagFeature ==0
            %no feature
            temp=i;

            FlagFeature =0;
        else
            %end of feature
            FlagFeature =0;
            featureWrite=[featureWrite,temp,i];
        end
    end
end
featureWrite(1)=[];
featureWrite1=1;
```

```
try
    featureWrite1=featureWrite(1);
end

%Combine features close to each other
if length(featureWrite)>2
    for i=2:2:length(featureWrite)
        temp=featureWrite(i+1)-featureWrite(i);
        if temp<40000

        else
            featureWrite1=[featureWrite1,featureWrite(i),
featureWrite(i+1)];
        end
        tmp=length(featureWrite)-i;
        if tmp ==2
            break;
        end
        if tmp ==0
            break;
        end
    end
end
try
    featureWrite1=[featureWrite1,featureWrite(length(featureWrite))
];
end
clear featureWrite
featureWrite=featureWrite1;

%Extract feature integrated size
try
    if length(featureWrite)>1
        for i=1:2:length(featureWrite)
            if i==1
                featureWriteIntegrated=sum(SSDwrite(featureWrite(i):
featureWrite(i+1)))-medianofsig*(featureWrite(i+1)-featureWrite(i));
            else
                featureWriteIntegrated=[featureWriteIntegrated,sum(
SSDwrite(featureWrite(i):featureWrite(i+1)))-medianofsig*(
        featureWrite(i+1)-featureWrite(i))];
                end
                tmp=length(featureWrite)-i;
                 if tmp ==1
                     break;
                 end
            end
        end

        %features must exceed  a certain size
        del=0;
        featureWrite1=0;
        featureWriteIntegrated1=0;
        for i=1:length(featureWriteIntegrated)

            if  featureWriteIntegrated(i)<100
                 %delete index
            else
                featureWrite1=[featureWrite1 ,featureWrite(i*2-1),
        featureWrite(i*2)];
                featureWriteIntegrated1=[featureWriteIntegrated1 ,
        featureWriteIntegrated(i)];
            end
        end
        featureWrite1(1)=[];
        featureWriteIntegrated1(1)=[];
        clear featureWrite featureWriteIntegrated
        featureWrite=featureWrite1;
        featureWriteIntegrated=featureWriteIntegrated1;
    end
end
```

A.2.4 Function for Extracting CPU Features

```
function [ featureCPU,featureCPUIntegrated ] = getCPUFeatures( CPUpower
    )
     featureCPU=0;
     featureCPUIntegrated=0;

    %Median filter to get rid of noise
```

```
CPUfilt=medfilt1(CPUpower,10000);

%Use thresholds
CPUsignal=CPUfilt;
index = find(CPUfilt<.0055);
for i=1:length(index)
    CPUsignal(index(i))=0;
end
CPUsignal=medfilt1(CPUsignal,3000);

%value of idle state
medianofsig=.004201;

%Look for contiguous features
FlagFeature=0;
temp=1;
for i=1:length(CPUsignal)
    if CPUsignal(i)>0;
        %feature
        if FlagFeature ==0
            %start of feature
            temp=i;

            FlagFeature =1;
        else
            FlagFeature =1;
        end
    else
        if FlagFeature ==0
            %no feature
            temp=i;

            FlagFeature =0;
        else
            %end of feature
            FlagFeature =0;
            featureCPU=[featureCPU,temp,i];
        end
    end
end
```

```
featureCPU(1)=[];

%Combine features close to each other
for i=2:2:length(featureCPU)
    tmp=length(featureCPU)-i;
    if tmp ==0
        break;
    end
    temp=featureCPU(i+1)-featureCPU(i);
    if temp<30000
        featureCPU(i+1)=[];
        featureCPU(i)=[];
    end
    tmp=length(featureCPU)-i;
    if tmp ==0
        break;
    end
end

%Extract feature integrated size
for i=1:2:length(featureCPU)
    if i==1
        featureCPUIntegrated=sum(CPUsignal(featureCPU(i):featureCPU(i
+1)))-medianofsig*(featureCPU(i+1)-featureCPU(i));
    else
        featureCPUIntegrated=[featureCPUIntegrated,sum(CPUsignal(
featureCPU(i):featureCPU(i+1)))-medianofsig*(featureCPU(i+1)-
featureCPU(i))];
    end
    tmp=length(featureCPU)-i;
    if tmp ==1
        break;
    end
end

%features must exceed  a certain size
del=0;
featureCPU1=0;
featureCPUIntegrated1=0;
for i=1:length(featureCPUIntegrated)
```

```

        if  featureCPUIntegrated(i)<20
             %delete index
        else
            featureCPU1=[featureCPU1,featureCPU(i*2-1),featureCPU(i*2)];
            featureCPUIntegrated1=[featureCPUIntegrated1,
    featureCPUIntegrated(i)];
        end
    end
    featureCPU1(1)=[];
    featureCPUIntegrated1(1)=[];
    clear featureCPU featureCPUIntegrated
    featureCPU=featureCPU1;
    featureCPUIntegrated=featureCPUIntegrated1;

end
```

A.3 Python Code

A.3.1 Code for Generating Random Text Files

```

from random import choice
from string import ascii_uppercase
import string

#size of random text file to be created
sizeinMB=750

kb=sizeinMB*1000
bytes=kb*1000
filename= str(sizeinMB)
filename='move from/'+filename+'MB.txt'

#get random characters
s=''.join(choice(string.hexdigits) for i in range(100))
print(s)

f= open(filename,"w+")
```

```
looptimes=bytes/100

for x in range(0,looptimes):
  s=''.join(choice(string.hexdigits) for i in range(100))
  f.write(s)
f.close()
```

A.3.2 Code for Encrpting Files in a Location

```
import glob
import time
import os, random, struct
from Crypto.Cipher import AES

#AES encryption function using Cipher Block Chaining
def encrypt_fileAES(key, in_filename, out_filename=None, chunksize
    =64*1024):
    if not out_filename:
        out_filename = in_filename + '.enc'

    iv = os.urandom(16)
    encryptor = AES.new(key ,AES.MODE_CBC, iv)
    filesize = os.path.getsize(in_filename)

    with open(in_filename, 'rb') as infile:
        with open(out_filename, 'wb') as outfile:
            outfile.write(struct.pack('<Q', filesize))
            outfile.write(iv)

            while True:
                chunk = infile.read(chunksize)
                if len(chunk) == 0:
                    break
                elif len(chunk) % 16 != 0:
                    chunk += b' ' * (16 - len(chunk) % 16)

                outfile.write(encryptor.encrypt(chunk))

#Decryption function
def decrypt_fileAES(key, in_filename, out_filename=None, chunksize
    =24*1024):

    if not out_filename:
        out_filename = os.path.splitext(in_filename)[0]

    with open(in_filename, 'rb') as infile:
        origsize = struct.unpack('<Q', infile.read(struct.calcsize('Q'))
    )[0]
        iv = infile.read(16)
        decryptor = AES.new(key, AES.MODE_CBC, iv)

        with open(out_filename, 'wb') as outfile:
            while True:
                chunk = infile.read(chunksize)
                if len(chunk) == 0:
                    break
                outfile.write(decryptor.decrypt(chunk))

            outfile.truncate(origsize)

key = 'key'

#Directory to be Encrypted
startPath = '/path'

q=1

if q==1:
  for subdir, dirs, files in os.walk(startPath):
    for file in files:
      print('Encrypting> ' + file)
      print(os.path.join(subdir, file))
      encrypt_fileAES(key, os.path.join(subdir, file))
      os.remove(os.path.join(subdir, file))
else:
  for subdir, dirs, files in os.walk(startPath):
    for file in files:
      filename=os.path.join(subdir, file)
      fname, ext = os.path.splitext(filename)
      if (ext == '.enc'):
        print('Decrypting> ' + file)
        print(os.path.join(subdir, file))
        decrypt_fileAES(key, os.path.join(subdir, file))
        os.remove(os.path.join(subdir, file))
```

A.3.3 Code for Executing Encryption Operations 50MB-5GB

```
import glob
import time
import os, random, struct
from Crypto.Cipher import AES

def encrypt_fileAES(key, in_filename, out_filename=None, chunksize
    =64*1024):

    if not out_filename:
        out_filename = in_filename + 't'

    iv = os.urandom(16)
    encryptor = AES.new(key ,AES.MODE_CBC, iv)
    filesize = os.path.getsize(in_filename)

    with open(in_filename, 'rb') as infile:
        with open(out_filename, 'wb') as outfile:
            outfile.write(struct.pack('<Q', filesize))
            outfile.write(iv)

            while True:
                chunk = infile.read(chunksize)
                if len(chunk) == 0:
                    break
                elif len(chunk) % 16 != 0:
                    chunk += b' ' * (16 - len(chunk) % 16)

                outfile.write(encryptor.encrypt(chunk))

def decrypt_fileAES(key, in_filename, out_filename=None, chunksize
    =24*1024):

    if not out_filename:
        out_filename = os.path.splitext(in_filename)[0]

    with open(in_filename, 'rb') as infile:
        origsize = struct.unpack('<Q', infile.read(struct.calcsize('Q')))[0]
        iv = infile.read(16)
        decryptor = AES.new(key, AES.MODE_CBC, iv)

        with open(out_filename, 'wb') as outfile:
            while True:
                chunk = infile.read(chunksize)
                if len(chunk) == 0:
                    break
                outfile.write(decryptor.decrypt(chunk))

            outfile.truncate(origsize)

key = 'This is a key123'

delay=600


for x in range(1,40):

  Start=time.time()

  time.sleep(100)
  startPath = '/media/jake/e0931aa2-d501-41c4-95ef-d84bfa2d1737/Encrpyt/50MB'

  for subdir, dirs, files in os.walk(startPath):
    for file in files:
      print('Encrypting> ' + file)
      print(os.path.join(subdir, file))
      encrypt_fileAES(key, os.path.join(subdir, file))
      os.remove(os.path.join(subdir, file))
  time.sleep(50)

  #need to fill up ram
  startPath = '/media/jake/e0931aa2-d501-41c4-95ef-d84bfa2d1737/Encrpyt/
```

```
    ram'

  for subdir, dirs, files in os.walk(startPath):
    for file in files:
      print('Encrypting> ' + file)
      print(os.path.join(subdir, file))
      encrypt_fileAES(key, os.path.join(subdir, file))
      os.remove(os.path.join(subdir, file))

  End=time.time()
  print('Takes this many seconds for run')
  print(End-Start)

  temp=End-Start
  while temp <delay:
    End=time.time()
    temp=End-Start
    print(temp)

for x in range(1,40):
  Start=time.time()

  time.sleep(100)
  startPath = '/media/jake/e0931aa2-d501-41c4-95ef-d84bfa2d1737/Encrpyt
    /100MB'

  for subdir, dirs, files in os.walk(startPath):
    for file in files:
      print('Encrypting> ' + file)
      print(os.path.join(subdir, file))
      encrypt_fileAES(key, os.path.join(subdir, file))
      os.remove(os.path.join(subdir, file))
  time.sleep(50)
  #need to fill up ram
  startPath = '/media/jake/e0931aa2-d501-41c4-95ef-d84bfa2d1737/Encrpyt/
    ram'

  for subdir, dirs, files in os.walk(startPath):
    for file in files:
      print('Encrypting> ' + file)
      print(os.path.join(subdir, file))
      encrypt_fileAES(key, os.path.join(subdir, file))
      os.remove(os.path.join(subdir, file))

  End=time.time()
  print('Takes this many seconds for run')
  print(End-Start)

  temp=End-Start
  while temp <delay:
    End=time.time()
    temp=End-Start
    print(temp)

for x in range(1,40):
  Start=time.time()

  time.sleep(100)
  startPath = '/media/jake/e0931aa2-d501-41c4-95ef-d84bfa2d1737/Encrpyt
    /250MB'

  for subdir, dirs, files in os.walk(startPath):
    for file in files:
      print('Encrypting> ' + file)
      print(os.path.join(subdir, file))
      encrypt_fileAES(key, os.path.join(subdir, file))
      os.remove(os.path.join(subdir, file))
  time.sleep(50)
  #need to fill up ram
  startPath = '/media/jake/e0931aa2-d501-41c4-95ef-d84bfa2d1737/Encrpyt/
    ram'

  for subdir, dirs, files in os.walk(startPath):
    for file in files:
      print('Encrypting> ' + file)
      print(os.path.join(subdir, file))
      encrypt_fileAES(key, os.path.join(subdir, file))
      os.remove(os.path.join(subdir, file))

  End=time.time()
  print('Takes this many seconds for run')
  print(End-Start)

  temp=End-Start
  while temp <delay:
    End=time.time()
    temp=End-Start
    print(temp)

for x in range(1,40):
  Start=time.time()

  time.sleep(100)
  startPath = '/media/jake/e0931aa2-d501-41c4-95ef-d84bfa2d1737/Encrpyt
   /500MB'

  for subdir, dirs, files in os.walk(startPath):
    for file in files:
      print('Encrypting> ' + file)
      print(os.path.join(subdir, file))
      encrypt_fileAES(key, os.path.join(subdir, file))
      os.remove(os.path.join(subdir, file))
  time.sleep(50)
  #need to fill up ram
  startPath = '/media/jake/e0931aa2-d501-41c4-95ef-d84bfa2d1737/Encrpyt/
   ram'

  for subdir, dirs, files in os.walk(startPath):
    for file in files:
      print('Encrypting> ' + file)
      print(os.path.join(subdir, file))
      encrypt_fileAES(key, os.path.join(subdir, file))
      os.remove(os.path.join(subdir, file))

  End=time.time()
  print('Takes this many seconds for run')
  print(End-Start)

  temp=End-Start
  while temp <delay:
```

```python
    End=time.time()
    temp=End-Start
    print(temp)


for x in range(1,40):
  Start=time.time()

  time.sleep(100)
  startPath = '/media/jake/e0931aa2-d501-41c4-95ef-d84bfa2d1737/Encrpyt/1000MB'

  for subdir, dirs, files in os.walk(startPath):
    for file in files:
      print('Encrypting> ' + file)
      print(os.path.join(subdir, file))
      encrypt_fileAES(key, os.path.join(subdir, file))
      os.remove(os.path.join(subdir, file))
  time.sleep(50)
  #need to fill up ram
  startPath = '/media/jake/e0931aa2-d501-41c4-95ef-d84bfa2d1737/Encrpyt/ram'

  for subdir, dirs, files in os.walk(startPath):
    for file in files:
      print('Encrypting> ' + file)
      print(os.path.join(subdir, file))
      encrypt_fileAES(key, os.path.join(subdir, file))
      os.remove(os.path.join(subdir, file))

  End=time.time()
  print('Takes this many seconds for run')
  print(End-Start)

  temp=End-Start
  while temp <delay:
    End=time.time()
    temp=End-Start
    print(temp)

```

```
for x in range(1,40):
  Start=time.time()

  time.sleep(100)
  startPath = '/media/jake/e0931aa2-d501-41c4-95ef-d84bfa2d1737/Encrpyt
    /2500MB'

  for subdir, dirs, files in os.walk(startPath):
    for file in files:
      print('Encrypting> ' + file)
      print(os.path.join(subdir, file))
      encrypt_fileAES(key, os.path.join(subdir, file))
      os.remove(os.path.join(subdir, file))
  time.sleep(50)
  #need to fill up ram
  startPath = '/media/jake/e0931aa2-d501-41c4-95ef-d84bfa2d1737/Encrpyt/
    ram'

  for subdir, dirs, files in os.walk(startPath):
    for file in files:
      print('Encrypting> ' + file)
      print(os.path.join(subdir, file))
      encrypt_fileAES(key, os.path.join(subdir, file))
      os.remove(os.path.join(subdir, file))

  End=time.time()
  print('Takes this many seconds for run')
  print(End-Start)

  temp=End-Start
  while temp <delay:
    End=time.time()
    temp=End-Start
    print(temp)

for x in range(1,40):
  Start=time.time()

  time.sleep(100)
  startPath = '/media/jake/e0931aa2-d501-41c4-95ef-d84bfa2d1737/Encrpyt
```

```
    /5000MB'

  for subdir, dirs, files in os.walk(startPath):
    for file in files:
      print('Encrypting> ' + file)
      print(os.path.join(subdir, file))
      encrypt_fileAES(key, os.path.join(subdir, file))
      os.remove(os.path.join(subdir, file))
  time.sleep(50)
  #need to fill up ram
  startPath = '/media/jake/e0931aa2-d501-41c4-95ef-d84bfa2d1737/Encrpyt/
    ram'

  for subdir, dirs, files in os.walk(startPath):
    for file in files:
      print('Encrypting> ' + file)
      print(os.path.join(subdir, file))
      encrypt_fileAES(key, os.path.join(subdir, file))
      os.remove(os.path.join(subdir, file))

  End=time.time()
  print('Takes this many seconds for run')
  print(End-Start)

  temp=End-Start
  while temp <delay:
    End=time.time()
    temp=End-Start
    print(temp)

print('Test Done')
print(time.time())
```

A.3.4 Code for Executing Move Operations 50MB-5GB

```
import glob
import time
import shutil
import os, random, struct
from Crypto.Cipher import AES
```

```python

def encrypt_fileAES(key, in_filename, out_filename=None, chunksize=64*1024):
    if not out_filename:
        out_filename = in_filename + 't'

    iv = os.urandom(16)
    encryptor = AES.new(key ,AES.MODE_CBC, iv)
    filesize = os.path.getsize(in_filename)

    with open(in_filename, 'rb') as infile:
        with open(out_filename, 'wb') as outfile:
            outfile.write(struct.pack('<Q', filesize))
            outfile.write(iv)

            while True:
                chunk = infile.read(chunksize)
                if len(chunk) == 0:
                    break
                elif len(chunk) % 16 != 0:
                    chunk += b' ' * (16 - len(chunk) % 16)

                outfile.write(encryptor.encrypt(chunk))

def decrypt_fileAES(key, in_filename, out_filename=None, chunksize=24*1024):

    if not out_filename:
        out_filename = os.path.splitext(in_filename)[0]

    with open(in_filename, 'rb') as infile:
        origsize = struct.unpack('<Q', infile.read(struct.calcsize('Q')))[0]
        iv = infile.read(16)
        decryptor = AES.new(key, AES.MODE_CBC, iv)

        with open(out_filename, 'wb') as outfile:
            while True:
                chunk = infile.read(chunksize)
                if len(chunk) == 0:
                    break
                outfile.write(decryptor.decrypt(chunk))

            outfile.truncate(origsize)

key = 'This is a key123'

delay=900


for x in range(1,90):
    Start=time.time()

    time.sleep(100)
    time.sleep(10)
    startPath = '/media/jake/e0931aa2-d501-41c4-95ef-d84bfa2d1737/movefrom
        /50MB/50MB.txt'
    endPath = '/media/jake/e0931aa2-d501-41c4-95ef-d84bfa2d1737/moveto/50
        MB/50MB.txt'

    shutil.copyfile(startPath,endPath)
    time.sleep(60)
    os.remove('/media/jake/e0931aa2-d501-41c4-95ef-d84bfa2d1737/moveto/50
        MB/50MB.txt')

    time.sleep(10)
    startPath = '/media/jake/e0931aa2-d501-41c4-95ef-d84bfa2d1737/movefrom
        /100MB/100MB.txt'
    endPath = '/media/jake/e0931aa2-d501-41c4-95ef-d84bfa2d1737/moveto/100
        MB/100MB.txt'

    shutil.copyfile(startPath,endPath)
    time.sleep(60)
    os.remove('/media/jake/e0931aa2-d501-41c4-95ef-d84bfa2d1737/moveto/100
        MB/100MB.txt')

    time.sleep(10)
    startPath = '/media/jake/e0931aa2-d501-41c4-95ef-d84bfa2d1737/movefrom
        /250MB/250MB.txt'
    endPath = '/media/jake/e0931aa2-d501-41c4-95ef-d84bfa2d1737/moveto/250
    MB/250MB.txt'


    shutil.copyfile(startPath,endPath)
    time.sleep(60)
    os.remove('/media/jake/e0931aa2-d501-41c4-95ef-d84bfa2d1737/moveto/250
    MB/250MB.txt')

    time.sleep(10)
    startPath = '/media/jake/e0931aa2-d501-41c4-95ef-d84bfa2d1737/movefrom
    /500MB/500MB.txt'
    endPath = '/media/jake/e0931aa2-d501-41c4-95ef-d84bfa2d1737/moveto/500
    MB/500MB.txt'

    shutil.copyfile(startPath,endPath)
    time.sleep(60)
    os.remove('/media/jake/e0931aa2-d501-41c4-95ef-d84bfa2d1737/moveto/500
    MB/500MB.txt')

    time.sleep(10)
    startPath = '/media/jake/e0931aa2-d501-41c4-95ef-d84bfa2d1737/movefrom
    /750MB/750MB.txt'
    endPath = '/media/jake/e0931aa2-d501-41c4-95ef-d84bfa2d1737/moveto/750
    MB/750MB.txt'

    shutil.copyfile(startPath,endPath)
    time.sleep(60)
    os.remove('/media/jake/e0931aa2-d501-41c4-95ef-d84bfa2d1737/moveto/750
    MB/750MB.txt')
    time.sleep(10)
    startPath = '/media/jake/e0931aa2-d501-41c4-95ef-d84bfa2d1737/movefrom
    /1000MB/1000MB.txt'
    endPath = '/media/jake/e0931aa2-d501-41c4-95ef-d84bfa2d1737/moveto
    /1000MB/1000MB.txt'

    shutil.copyfile(startPath,endPath)
    time.sleep(60)
    os.remove('/media/jake/e0931aa2-d501-41c4-95ef-d84bfa2d1737/moveto
    /1000MB/1000MB.txt')
    time.sleep(10)
```

```
startPath = '/media/jake/e0931aa2-d501-41c4-95ef-d84bfa2d1737/movefrom
  /2500MB/2500MB.txt'
endPath = '/media/jake/e0931aa2-d501-41c4-95ef-d84bfa2d1737/moveto
  /2500MB/2500MB.txt'

shutil.copyfile(startPath,endPath)
time.sleep(60)
os.remove('/media/jake/e0931aa2-d501-41c4-95ef-d84bfa2d1737/moveto
  /2500MB/2500MB.txt')
time.sleep(10)
time.sleep(10)
startPath = '/media/jake/e0931aa2-d501-41c4-95ef-d84bfa2d1737/movefrom
  /5000MB/5000MB.txt'
endPath = '/media/jake/e0931aa2-d501-41c4-95ef-d84bfa2d1737/moveto
  /5000MB/5000MB.txt'

shutil.copyfile(startPath,endPath)
time.sleep(60)
os.remove('/media/jake/e0931aa2-d501-41c4-95ef-d84bfa2d1737/moveto
  /5000MB/5000MB.txt')
time.sleep(10)




#need to fill up ram
startPath = '/media/jake/e0931aa2-d501-41c4-95ef-d84bfa2d1737/Encrpyt/
  ram/'

for subdir, dirs, files in os.walk(startPath):
  for file in files:
    print('Encrypting> ' + file)
    print(os.path.join(subdir, file))
    encrypt_fileAES(key, os.path.join(subdir, file))
    os.remove(os.path.join(subdir, file))

End=time.time()
print('Takes this many seconds for run')
print(End-Start)

```

```
    temp=End-Start
    while temp <delay:
        End=time.time()
        temp=End-Start
        print(temp)

print('Test Done')
print(time.time())
```

A.4 Labview Code

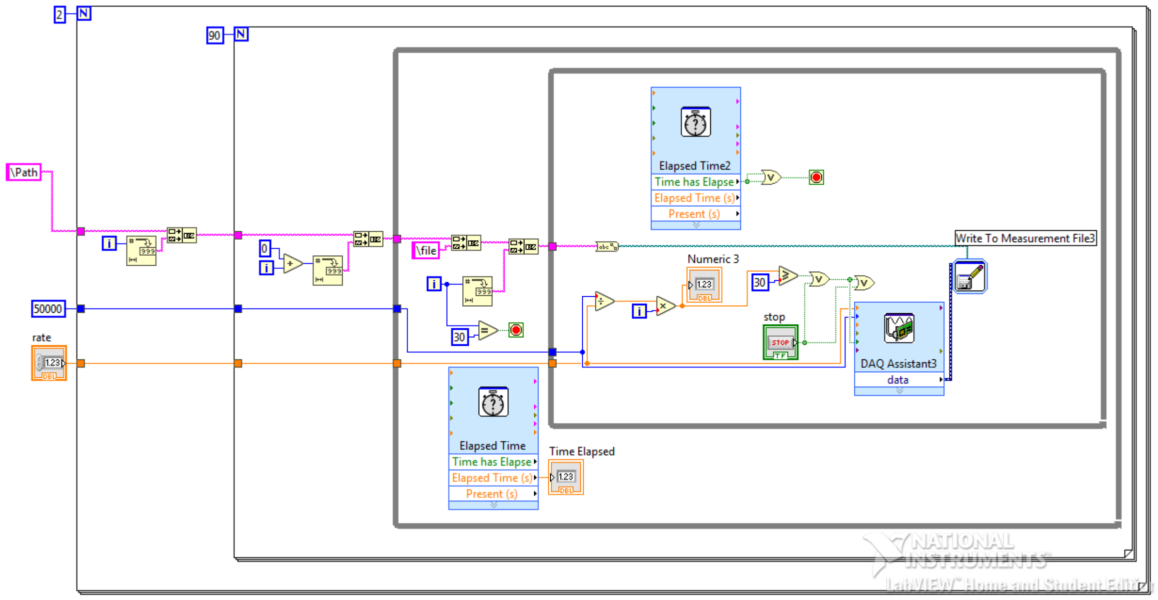

Figure A.17. Labview Block Diagram for Data Collection

THIS PAGE INTENTIONALLY LEFT BLANK

List of References

[1] S. Mohurle and M. Patil, “A brief study of wannacry threat: Ransomware attack 2017,” *International Journal*, vol. 8, no. 5, pp. 1938–1940, 2017.

[2] N. Hampton and Z. A. Baig, “Ransomware: Emergence of the cyber-extortion menace,” in *13th Australian Information Security Management Conference*, Perth, Western Australia, 2015, pp. 47–56.

[3] A. Bhardwaj, V. Avasthi, H. Sastry, and G. Subrahmanyam, “Ransomware digital extortion: a rising new age threat,” *Indian Journal of Science and Technology*, vol. 9, no. 14, pp. 1–5, 2016.

[4] K. Savage, P. Coogan, and H. Lau, “The evolution of ransomware, symantec security response,” Symantec Corporation, Mountain View, CA, August 2015.

[5] N. Idika and A. P. Mathur. (2007). A survey of malware detection techniques. Purdue University. West Lafayette, IN. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.75.4594&rep=rep1&type=pdf

[6] C. Moore, “Detecting ransomware with honeypot techniques,” in *Cybersecurity and Cyberforensics Conference*, Aug 2016, pp. 77–81.

[7] J. Z. Kolter and M. A. Maloof, “Learning to detect and classify malicious executables in the wild,” *Journal of Machine Learning Research*, vol. 7, pp. 2721–2744, Dec 2006.

[8] A. Moradi, A. Barenghi, T. Kasper, and C. Paar, “On the vulnerability of FPGA bitstream encryption against power analysis attacks: Extracting keys from xilinx virtex-ii FPGAs,” in *Proceedings of the 18th ACM conference on Computer and communications security*, 2011, pp. 111–124.

[9] J. Shey, R. Rakvic, H. Ngo, O. Walker, T. Tedesso, J. A. Blanco, and K. Fairbanks, “Inferring trimming activity of solid-state drives based on energy consumption,” in *IEEE Instrumentation and Measurement Technology Conference*, 2016, pp. 1–6.

[10] J. Canclini, J. McMasters, J. Shey, O. Walker, R. Rakvic, H. Ngo, and K. D. Fairbanks, “Inferring read and write operations of solid-state drives based on energy consumption,” in *IEEE 7th Ubiquitous Computing, Electronics Mobile Communication Conference*, Oct 2016, pp. 1–6.

[11] N. Lord. (2017, March). A history of ransomware attacks: The biggest and worst ransomware attacks of all time. *Digital Guardian*. [Online]. Available: https://digitalguardian.com/blog/history-ransomware-attacks-biggest-and-worst-ransomware-attacks-all-time

[12] M. Paquet-Clouston, B. Haslhofer, and B. Dupont, "Ransomware payments in the bitcoin ecosystem," *17th Annual Workshop on the Economics of Information Security*, April 2018.

[13] R. Richardson and M. North, "Ransomware: Evolution, mitigation and prevention," *International Management Review*, vol. 13, no. 1, pp. 10–21, 2017.

[14] S. C. Hsiao and D. Y. Kao, "The static analysis of wannacry ransomware," in *20th International Conference on Advanced Communication Technology*, Feb 2018, pp. 153–158.

[15] M. Cova, C. Kruegel, and G. Vigna, "Detection and analysis of drive-by-download attacks and malicious javascript code," in *Proceedings of the 19th international conference on World wide web*, 2010, pp. 281–290.

[16] H. Zuhair and A. Selamat, "Phishing classification models: Issues and perspectives," in *IEEE Conference on Open Systems*, Nov 2017, pp. 26–31.

[17] V. Kotov and F. Massacci, "Anatomy of exploit kits," in *International Symposium on Engineering Secure Software and Systems*. Springer, 2013, pp. 181–196.

[18] S. Chadha and U. Kumar, "Ransomware: Let's fight back!" in *International Conference on Computing, Communication and Automation*, May 2017, pp. 925–930.

[19] K. Ganame, M. A. Allaire, G. Zagdene, and O. Boudar, "Network behavioral analysis for zero-day malware detection–a case study," in *International Conference on Intelligent, Secure, and Dependable Systems in Distributed and Cloud Environments*. Springer, 2017, pp. 169–181.

[20] D. Gonzalez and T. Hayajneh, "Detection and prevention of crypto-ransomware," in *IEEE 8th Annual Ubiquitous Computing, Electronics and Mobile Communication Conference*, Oct 2017, pp. 472–478.

[21] D. Loesche. (2017, June). Cyberattacks against the US government up 1,300% since 2006. [Online]. Available: https://www.statista.com/chart/10045/new-malware-specimen-and-share-of-windows-based-malware/

[22] Z. Bazrafshan, H. Hashemi, S. M. H. Fard, and A. Hamzeh, "A survey on heuristic malware detection techniques," in *The 5th Conference on Information and Knowledge Technology*. IEEE, May 2013, pp. 113–120.

[23] S. Bhattacharyya. (2016, June). Ransomware makes up small share of growing malware threat. Statista. [Online]. Available: http://www.thefiscaltimes.com/2016/06/22/Cyberattacks-Against-US-Government-1300-2006

[24] J. Jackson. (2018, Mar). User behavior analytics (UBA) & ransomware analytics. IT Knowledge Exchange. [Online]. Available: https://itknowledgeexchange.techtarget.com/virtual-ciso/user-behavior-analytics-uba-ransomware-analytics/

[25] What are hash functions. Learn cryptography. [Online]. Available: https://learncryptography.com/hash-functions/what-are-hash-functions

[26] N. Scaife, H. Carter, P. Traynor, and K. R. B. Butler, "Cryptolock (and drop it): Stopping ransomware attacks on user data," in *IEEE 36th International Conference on Distributed Computing Systems*, June 2016, pp. 303–312.

[27] J. Kornblum, "Identifying almost identical files using context triggered piecewise hashing," in *Proceedings of the Digital Forensic Research Conference*. Elsevier, 2006, vol. 3, pp. 91–97.

[28] C. B. Paul, "Entropy-based file type identification and partitioning," M.S. thesis, Naval Postgraduate School, Monterey, California, 2017.

[29] S. K. Shaukat and V. J. Ribeiro, "Ransomwall: A layered defense system against cryptographic ransomware attacks using machine learning," in *IEEE 10th International Conference on Communication Systems & Networks*, 2018, pp. 356–363.

[30] D. Sgandurra, L. Muñoz-González, R. Mohsen, and E. C. Lupu, "Automated dynamic analysis of ransomware: Benefits, limitations and use for detection," *arXiv preprint arXiv:1609.03020*, 2016.

[31] G. Bonetti, M. Viglione, A. Frossi, F. Maggi, and S. Zanero, "A comprehensive black-box methodology for testing the forensic characteristics of solid-state drives," in *Proceedings of the 29th Annual Computer Security Applications Conference*, New Orleans, Louisiana, 2013, pp. 269–278.

[32] R. Bez, E. Camerlenghi, A. Modelli, and A. Visconti, "Introduction to flash memory," *Proceedings of the IEEE*, vol. 91, no. 4, pp. 489–502, April 2003.

[33] S. Larrivee. (2015, May). Solid state drives 101: Everything you ever wanted to know. Cactus Technologies. [Online]. Available: https://www.cactus-tech.com/resources/blog/details/solid-state-drive-primer-7-controller-architecture-basic-overview

[34] Y. Zhou and D. Feng, "Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing." IACR Cryptology ePrint Archive, p. 388, 2005.

[35] M. Y. C. Wei, L. M. Grupp, F. E. Spada, and S. Swanson, "Reliably erasing data from flash-based solid state drives." in *Proceedings of the 9th USENIX conference on File and stroage technologies*, 2011, vol. 11, p. 8.

[36] A. Adamov and A. Carlsson, "The state of ransomware. trends and mitigation techniques," in *IEEE East-West Design Test Symposium*, Sept 2017, pp. 1–8.

[37] H. Alanazi, B. B. Zaidan, A. A. Zaidan, H. A. Jalab, M. Shabbir, Y. Al-Nabhani *et al.*, "New comparative study between des, 3des and aes within nine factors," *Journal of Computing*, vol. 2, no. 3, pp. 65–67, March 2010.

[38] V. Lomne, A. Dehaboui, P. Maurine, L. Torres, and M. Robert, "Side channel attacks," in *Security trends for FPGAS*, B. Badrignans, Ed. London: Springer, 2011, pp. 47–72.

[39] J. Von Neumann, *The Computer and the Brain*. New Haven, Connecticut: Yale University Press, 2012.

[40] 3. Features of a Von Neumann architecture. (2008, May). Teach ICT. [Online]. Available: http://www.teach-ict.com/as_as_computing/ocr/H447/F453/3_3_3/vonn_neuman/miniweb/pg3.htm

[41] L. Rabiner, M. Sambur, and C. Schmidt, "Applications of a nonlinear smoothing algorithm to speech processing," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 23, no. 6, pp. 552–557, 1975.

[42] M. Sokolova, N. Japkowicz, and S. Szpakowicz, "Beyond accuracy, f-score and roc: a family of discriminant measures for performance evaluation," in *Advances in Artificial Intelligence*. Springer, 2006, pp. 1015–1021.

[43] J. Melton, R. Rakvic, J. Shey, H. Ngo, O. Walker, J. Blanco, D. Brown, L. McDowell, and K. Fairbanks, "Inferring file system of solid state drives based on current consumption," presented at IEEE CYBER Technology in Automation, Control, and Intelligent Systems Conference, Waikiki Beach, Hawaii, 2017.

[44] B. A. Anima, M. Jasim, K. A. Rahman, and M. Hasanuzzaman, "User authentication based on mouse movement data using normalized features," in *IEEE 19th International Conference on Computer and Information Technology*, 2016, pp. 399–404.

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California