Journal of Computer Security 26 (2018) 589–613
DOI 10.3233/JCS-171060
IOS Press


# Identifying stealth malware using CPU power consumption and learning algorithms

Patrick Luckett [a,*], J. Todd McDonald [a], William B. Glisson [b], Ryan Benton [a], Joel Dawson [a] and Blair A. Doyle [a]

[a] *School of Computing, University of South Alabama, Mobile, Al, United States*
*E-mails: phl801@jagmail.southalabama.edu, jtmcdonald@southalabama.edu, rbenton@southalabama.edu, jad1324@jagmail.southalabama.edu, bad1524@jagmail.southalabama.edu*

[b] *Department of Computer Science, Sam Houston State University, Huntsville, TX, United States*
*E-mail: glisson@shsu.edu*

**Abstract.** With the increased assimilation of technology into all aspects of everyday life, rootkits pose a credible threat to individuals, corporations, and governments. Using various techniques, rootkits can infect systems and remain undetected for extended periods of time. This threat necessitates the careful consideration of real-time detection solutions. Behavioral detection techniques allow for the identification of rootkits with no previously recorded signatures. This research examines a variety of machine learning algorithms, including Nearest Neighbor, Decision Trees, Neural Networks, and Support Vector Machines, and proposes a behavioral detection method based on low yield CPU power consumption. The method is evaluated on Windows 7, Windows 10, Ubuntu Desktop, and Ubuntu Server operating systems along with employing four different rootkits. Relevant features within the data are calculated and the overall best performing algorithms are identified. A nested neural network is then applied that enables highly accurate data classification. Our results present a viable method of rootkit detection that can operate in real-time with minimal computational and space complexity.

Keywords: Rootkit, anomaly detection, machine learning



## 1. Introduction

Rootkits are malicious programs that acquire root privileges on a computer. Hooking techniques allow rootkits to hide within a system without detection from traditional security mechanisms [6]. As defined by the Microsoft developers network, a hook is "a point in the system message-handling mechanism where an application can install a subroutine to monitor the message traffic in the system and process certain types of messages before they reach the target window procedure" [38]. There are, generally, two reasons for deploying a rootkit. The first is establishing remote command and control of a system and the second is eavesdropping on a system [28]. Over time rootkits have evolved, and are still used in modern attacks. In 2015 it was revealed the Chinese cybercriminal group Winnti was using rootkit technology in their attacks [45]. In 2016, Check Point Mobile Threat Prevention identified the mobile rootkit HummingBad, which could install more than 50,000 fraudulent apps each day, and display 20 million malicious advertisements [9]. According to Kaspersky Security, in 2016 the third most prominent

*Corresponding author. E-mail: phl801@jagmail.southalabama.edu.

banking malware family (Trojan.Win32.Neurevt) employed rootkit technology [56]. Rootkits are also employed in advanced persistent threat scenarios, as demonstrated by Stuxnet. Stuxnet used a Windows rootkit, the first ever PLC rootkit, process injection, and code hooking to damage centrifuges at the Natanz uranium enrichment plant in Iran [20,60]. When coupling this information with the increasing integration of technology into the automotive world [8], the aviation industry [42], critical infrastructure and legal environments [4,7,40,62], it stresses the need to identify methods that will discover rootkits before they can cause damage.

Methods proposed for identifying rootkits generally fall into three categories [55]. The first category is signature based detection, which looks for previously recorded byte patterns of known rootkits [55]. While this method is very effective, it cannot detect rootkits that have not yet been identified and recorded. The second category is integrity detection which involves periodically checking the system for unauthorized changes to file systems and operating system components [55]. The last category is behavioral detection and the focus area for this research. Behavioral detection involves identifying the presence of a rootkit by collecting data on the behavior of the system under normal conditions, and then uses the data as a baseline to identify deviations [10]. This process typically involves the use of machine learning or statistical models. In fact, a recent article noted that machine learning is the primary method for almost all non-signature based detection techniques, including network intrusion, spam identification, fraudulent activity, and malicious activity in general [39].

The contributions of this research are as follows. First We demonstrate the performance of a rootkit detection solution using power measurements from the host CPU. The experimental results indicate that this method is on par or supersedes the current work in several aspects, including the accuracy of the model and the data requirements. The algorithm is effective using data measured at low frequency random time frames with only information on the initial, minimum, maximum, intervals, and average power measurement during that time frame. Further, we demonstrate that this method is capable of detecting both user and kernel level rootkits during normal operating conditions and under stress tests. Our method is tested on four different operating systems: Ubuntu Server, Ubuntu Desktop, Windows 7, and Windows 10. The data is analyzed and results are reported on a variety of different machine learning algorithms, including decision trees, nearest neighbor, support vector machines, neural networks, regression, and ensemble methods. Lastly, a model is created using nested neural networks and discrete representations of projected data that outperforms other methods. On average, this model achieved 9.4% higher area under the curve (AUC) and 17.2% higher accuracy than the traditional machine learning algorithms tested.

The remainder of the paper is structured as follows. Section 2 presents related work in rootkit detection and learning algorithms. Section 3 discusses the algorithms used in our analysis and Section 4 describes our data collection environment, as well as any preprocessing techniques used on the data once collected. Section 5 presents an analysis of the results and Section 6 discusses the implications and future work. The last section of the paper draws conclusions.

## 2. Related work

Machine learning techniques have become common place in many applications, including malware detection. The following subsections describe related work in rootkit detection given specific features.

### 2.1. System call behavior

Das et al. [13] proposed an online malware detection method which they named GuardOL. GuardOL uses system call patterns to classify malware. The frequency-centralized model considers how often critical system calls are used and builds features by clustering the calls based on a set of rules that capture the behavior of malicious code. Based on these features, a multilayer perceptron is trained and used to identify malicious activity at runtime. The data set used contained 472 samples, of which 56 were rootkits. Their model, which was implemented on an FPGA, could detect 46% of malware within 30% of its execution time, and 97% of malware all total.

Dini et al. [18] designed a method for detecting malware on smartphones they called Multi-level Anomaly Detector for Android Malware (MADAM). Their behavior based method intercepts critical system calls at the kernel level and records the number of occurrences of the calls over a period of time. Their one nearest neighbor classifier could detect 100% of the rootkits tested.

Hernandez et al. [27] note cyber-attacks pose a credible threat, citing Stuxnet, which used rootkit technology. They evaluated a method of event detection based on reconstructing the phase-space formed by serialized system call timing data. Their algorithm evaluates the dissimilarity among phase-space graphs over time and successfully indicated anomalous activity.

In our recent work [37], we also evaluated system call timing data for rootkit detection. Our analysis tested two neural network architectures: feed forward and recurrent. The recurrent neural network could correctly classify 97% of system calls as either infected or not infected with a rootkit (KBeast). This accuracy was obtained using only one feature (system call time) and no preprocessing of the data. In another work [14] we evaluated the same system calls using phase-space analysis.

### 2.2. API behavior

Pirscoveanu et al. [50] note that attackers have a strong incentive to increase the complexity of malicious code to improve obfuscation and decrease the likelihood of being detected by anti-virus software. They suggest a method of detection based on dynamic analysis of system behavior. The behaviors used in their research included DNS requests, accessed files, mutexes, registry keys and Windows API's. Using a Random Forest classifier, they could detect trojan malware at .98, adware at .95, and rootkits at .97 areas under the curve (AUC). They also note their classifier had a higher number of false negatives when classifying rootkits as opposed to other forms of malware.

Alazab et al. [1] compared eight classifiers for detecting rootkits based on API features. They tested the performance of Naive Bayes, $K$-Nearest Neighbor, Sequential Minimal Optimization with a Normalized PolyKernel, Sequential Minimal Optimization with a PolyKernel, Sequential Minimal Optimization with Puk, Sequential Minimal Optimization with a Radial Basis, Backpropagation Neural Networks, and J48 decision trees. Their results indicate Support Vector Machines with a normalized PolyKernel performed best (98.5% true rate), and the Neural Network using backpropagation performed the poorest.

### 2.3. Hooking behavior

Ramani et al. [52] proposed rootkit detection based on hooking. They evaluated nine classifiers to identify which performed the best. Of the nine classifiers, the most accurate was one proposed by Ramani and Jacob called the Correlation Bayes Algorithm. The correlation Bayes Algorithm yielded a classification accuracy of 87.4% using 10-fold cross validation. A Random Committee algorithm came in second with an accuracy of 86.2%, and third was Logistic Regression with 85.1%.

Lobo et al. [36] suggested a method for rootkit detection called Rootkit Behavioral Analysis and Classification System (RBACS). This method analyzes Windows process affected by rootkit hooking techniques and the API functions that were hooked. Their analysis used 78 rootkit samples obtained from the Offensive Computing website [46] that produced 11,159 inline function hooks. Using the Expectation Maximization (EM) and 10-Fold Cross Validation algorithms, their method could categorize the different rootkits into one of five categories based on labels assigned by antivirus software.

### 2.4. Other characteristics

Demme et al. [17] define malware as any software designed with the intention of compromising a system or the privacy of an individual or business. Their research uses behavioral characteristics from performance counters and machine learning classifiers such as $K$-Nearest Neighbor and Decision Trees to identify the presence of Android malware and Linux rootkits. Their method, which uses hardware modifications to support dynamic analysis, yielded an accuracy of nearly 90%.

Arslan et al. [3] performed a thorough review of machine learning methods used in the detection of mobile attacks, including rootkits. They cited over 20 published works that employed machine learning classifiers, including Support Vector Machines, Artificial Neural Networks, Random Forrest, Naive Bayes, $K$ Nearest Neighbors, and others to identify the presence of a threat. They note these methods are most advantageous when employed against malicious code that has not yet been identified and has no recorded signature. They also identified three phases of malware detection using machine learning: feature extraction, feature selection, and classification.

Dubey et al. [19] proposed a three-step learning algorithm using $K$ Means clustering, Bayesian classifiers, and Neural Networks with backpropagation training as a method for intrusion detection systems. Of the attacks they studied, some of them included User-To-Root attacks (U2R), where an attacker exploits vulnerabilities within a system to gain root access, which is similar to a rootkit. Using data acquired from the 1998 DARPA Intrusion Detection Evaluation program, they could correctly classify 98% of rootkit style attacks. Lakshmi et al. [35] also investigated the application of machine learning to intrusion detection systems based on User-To-Root attacks. Their analysis used a subset of the KDD CUP 99 data set, which contained 52 User-To-Root attacks, each with 41 attributes. After various preprocessing methods were employed, such as normalization and Principal Component Analysis, the data was tested on several different classifiers. Of the classifiers tested, the best results came from Rule Induction, Decision trees, and Naive Bayes.

### 2.5. Power characteristics

Shropshire [54] used statistical analysis of power measurements to identify hypervisors that had been compromised. He notes that while attackers can hide unauthorized activity by modifying specific components within a system, they cannot manipulate external measures of energy consumption. The proposed system, named PowerCheck, measures power consumed by a server using an external sensor, as well as the hypervisor's reports of CPU load, network activity, and memory use. It then identifies anomalous activity by comparing predicted consumption against the actual usage. The model could detect small discrepancies in energy consumption and had minimal false positives.

In our recent [15] work we utilized power supply voltage measurements collected using a multimeter and current clamp, then extract time-serial system dynamics through the application of a phase-space algorithm. Examination of phase-space graph features between nominal and infected data during stress

test on an Ubuntu Server OS were used to classify the state of the system. Our results indicate that the algorithm could successfully detect a rootkit through power measurement analysis. The current manuscript extends this work by evaluating different operating systems, different rootkits, different hardware, and a variety of learning algorithms.

## 3. Algorithms

The learning algorithms used in this research can be broken down into seven categories: tree based methods, regression, support vector machines, nearest neighbor, discriminant analysis, neural networks, and ensemble classifiers. A general description of these methods is provided in the following subsections. During training, all algorithms used 5-Fold cross validation, which is a method used to estimate how well a predictive model generalizes to different data sets [33]. This section provides a general background for readers not familiar with the above algorithms. Readers familiar with statistical and machine learning prediction and classification algorithms may skip this section.

### 3.1. Decision trees

Decision trees are hierarchical models that split data into homogenous groups based on recursively applied rules [51]. The training objective is to find a set of decision rules that can be used to predict a class label. Our research uses the Gini-Simpson index (1) of diversity [24]:

$$G(v_i) = 1 - \sum_{j=1}^{k} p_j^2 \tag{1}$$

Here, $v_i$ represents the possible values of a given attribute, and $p_j$ the fraction of points containing said attribute belonging to a set of classes $j \in \{1, \ldots, k\}$. The Gini-Simpson index represents the probability of a given class occurring at a given node in the decision tree, and is based on the cumulative proportion of a given unit. Various size trees with limitations on branching factors were evaluated in our analysis. The trees in our analysis labeled complex have a maximum branching factor of 100, the medium trees have a maximum branching factor of 50, and the simple trees have a maximum branching factor of 20.

### 3.2. Regression

Logistic regression is a special case of regression where the dependent variable is dichotomous. There are different methods of logistic regression for classification, including ordinary least square, weighted least square and maximum likelihood estimations [2]. Our analysis focuses on maximum likelihood estimation, which attempts to find a value for an unknown parameter to maximize a function. In this case the function corresponds to the likelihood function, which is the function produced by reversing the roles of parameters in the Probability Density Function (PDF). For binary classification $(+1, -1)$, the logistic regression function for the positive (2) and negative (3) class respectively are as follows:

$$P(C = +1|\bar{X}) = \frac{1}{1 + e^{-(\theta_0 + \sum_{i=1}^{d} \theta_i x_i)}} \tag{2}$$

$$P(C = -1|\bar{X}) = \frac{1}{1 + e^{(\theta_0 + \sum_{i=1}^{d} \theta_i x_i)}} \tag{3}$$

These equations read 'The probability of the class label $C$ being a binary value given a set of records $X$ and a set of parameters $\Theta$', where $\theta_0$ represents a bias term and $\theta_i$ is the weighting coefficient of each feature.

### 3.3. Support vector machines

Support Vector Machines, or SVMs, are a technique for classification and outlier detection. SVMs seek to find an optimal hyperplane to separate classes by maximizing the margin between the closest points among classes [41]. For a feature vector $\bar{X}$ where $\bar{X}_i$ corresponds to the $i$th training point, a vector $\bar{W}$ corresponding to the normal direction of the hyperplane, and a bias $b$, the hyperplane is calculated with the following formula (4):

$$\bar{W} \cdot \bar{X} + b = 0 \tag{4}$$

For binary classification, values greater than 0 belong to the positive class, and values less than 0 belong to the negative class. A hyperplane of a $p$-dimensional space is a flat affine (does not pass through origin) subspace consisting of $p - 1$ dimensions [31]. When perfect separation is not possible, points that occur in the incorrect class are assigned lower weights as to reduce their impact. This technique uses what are called soft margins rather than optimal margins. SVMs also employ kernel techniques to classify nonlinear data that is not easily separable by projecting the data into higher dimensions. The kernel functions of a SVM are a type of similarity function used to calculate decision boundaries in projected space [31]. Our analysis evaluates the performance of four support vector machines each with a different kernel: linear (5), quadratic (6), cubic (7), and Gaussian radial base (8).

$$f(x_1, \ldots, x_k) = b + a_1 x_1 + \cdots + a_k x_k \tag{5}$$

$$f(x) = ax^2 + bx + c \tag{6}$$

$$f(x) = ax^3 + bx^2 + cx + d \tag{7}$$

$$f\left(\|x - x_i\|\right) = e^{(\epsilon r)^2} \tag{8}$$

### 3.4. Nearest neighbor

The nearest neighbor classification method is one of the most well-known techniques in machine learning. Despite its simplicity, it is one of the most widely used algorithms in many aspects of data mining [34]. Data is divided into samples and is represented by an attribute vector. This vector contains features that describe the sample [25]. Classification is achieved based on the similarity or distance function used in the algorithm. The accuracy of the $K$-Nearest Neighbor algorithm is strongly dependent on the choice of similarity or distance function [58]. In this research, we evaluated 3 distance functions: Euclidean (9), Minkowski (10), and Cosine (11).

$$d(p, q) = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2} \tag{9}$$

$$d(p,q) = \sqrt[m]{\sum_{i=1}^{n}(q_i - p_i)^m} \tag{10}$$

$$\cos(\theta) = \frac{\sum_{i=1}^{n}(q_i p_i)}{\sqrt{\sum_{i=1}^{n} q_i^2}\sqrt{\sum_{i=1}^{n} p_i^2}} \tag{11}$$

Here, $Q$ and $P$ represent feature vectors. In our analysis, all nearest neighbor algorithms used $k = 10$. For Minkowski distance, the parameter $m$ was assigned a value of five, which is equal to the number of dimensions in the feature vectors.

### 3.5. Discriminant analysis

Discriminant analysis seeks to determine which variables or features discriminate between two or more groups. It generally assumes the observations in each class are generated by a probability distribution specific to said class [22]. Our analysis considers two types of discriminant analysis: linear and quadratic. Linear discriminant analysis assumes a multivariate normal distribution and is used when the variance–covariance matrix does not depend on the sampled population. Classification rules are based on a Linear Score Function of the population means for each population. Quadratic discriminant analysis follows a similar method, but is used for heterogeneous variance–covariance matrices [22]. The score functions for Linear (12) and Quadratic (13) Discriminant analysis are as follows:

$$S_i^L(x) = d_{i0} + \sum_{i=1}^{p} d_{ij} x_j + \log p_i \tag{12}$$

$$S_i^Q(x) = -\frac{1}{2}\log|\sigma_i| - \frac{1}{2}(x - \bar{x})'\sigma_i^{-1}(x - \bar{x}) + \log p_i \tag{13}$$

### 3.6. Ensemble classifiers

Ensemble classification methods evaluated in our analysis include Boosted Trees and Bagged Trees. The method of Bagged Trees takes small bootstrap samples, which are data sets consisting of randomly drawn instances of the original data set [24], and builds a decision tree. Then, the process is repeated several times, building many different trees from many different sets of data. Classification is based on the average predictions from all trees. Boosting originated from the AdaBoost algorithm [23]. Boosting uses a collection of weak classifiers to yield one strong classifier [16]. The ensemble tree methods have a maximum branching factor of 20 and use 100 learners.

### 3.7. Neural networks

Neural networks have been widely used for function approximation, classification, clustering, and dynamic system modeling [61]. There are generally three aspects of a neural network: the architecture, training method, and transfer function. Network architectures include feed forward, cascade, recurrent, and many others. Some examples of training methods are scaled conjugate gradient descent, Levenberg–Marquardt, and Bayesian regularization backpropagation. There are also many types of transfer functions. Of those, the most common are the sigmoid functions. The general layout for a sigmoid transfer

function is:

$$s(x_i) = \frac{\alpha_i}{1 + e^{-\beta_i x_i}} + \epsilon_i \tag{14}$$

Specific examples of transfer functions in this format include the logistic and hyperbolic tangent function. The neural network used in our analysis was a feed forward network with one input layer, three hidden layers, and an output layer. The input layer consisted of five neurons corresponding to the five points in each feature vector, the hidden layers consisted of 31 neurons (30 neurons and one bias), and the output layer was a single neuron. The first hidden layer used a radial base transfer function, while the next two hidden layers used a hyperbolic tangent transfer function. The output layer used a linear step function. The network was trained using the Levenberg–Marquardt algorithm (LMA), which is a method for solving non-linear least squares. As part of our proposed method we also employ self organizing maps. Self Organizing Maps (SOM) are a type of unsupervised neural network that clusters data by projecting the input on-to a 2-dimensional grid based on similarity and topology of the feature vectors [57]. Our analysis employed a SOM with a four-by-four ($4 \times 4$) grid.

## 4. Methodology

The following subsections describe the equipment, hardware, and software used in our analysis. It also describes the steps involved in data collection.

### *4.1. Data collection equipment*

Power data was collected and recorded using a Fluke 289 Multimeter with a Fluke i30 AC/DC current clamp. The multimeter records changes in power readings given a certain threshold. The default threshold of the Multimeter is 4% and was not adjusted.

### *4.2. Software*

Our analysis focused on three operating systems. The first operating system was an Ubuntu 14.04 Server. Data was collected from the operating system prior to infection and after infection. The rootkit used on the Ubuntu operating system was KBeast [29]. KBeast is a Loadable Kernel Module rootkit capable of hiding itself, files, directories, and processes. It is also able to log key strokes and prevent removal of files and deletion of itself. KBeast is compiled using the setup script and then loaded into kernel space.

The second operating system was Windows 10 Pro. The rootkit used on the Windows 10 operating system was FUTo [46]. According to the authors, FUTo can manipulate the PspCidTable without using any function calls, and uses Direct Kernel Object Manipulation (DKOM) to hide certain objects. The FUTo file came with a prebuilt executable, and the system was infected by running the executable as an administrator.

The third operating system was Ubuntu Desktop version 16.04.1. Ubuntu Desktop was infected with the Azazel rootkit [46]. Azazel is a user level rootkit capable of hiding files, directories, processes, creating backdoors, and avoiding detection. Azazel was compiled using the scripts provided at Packet Storm [47].

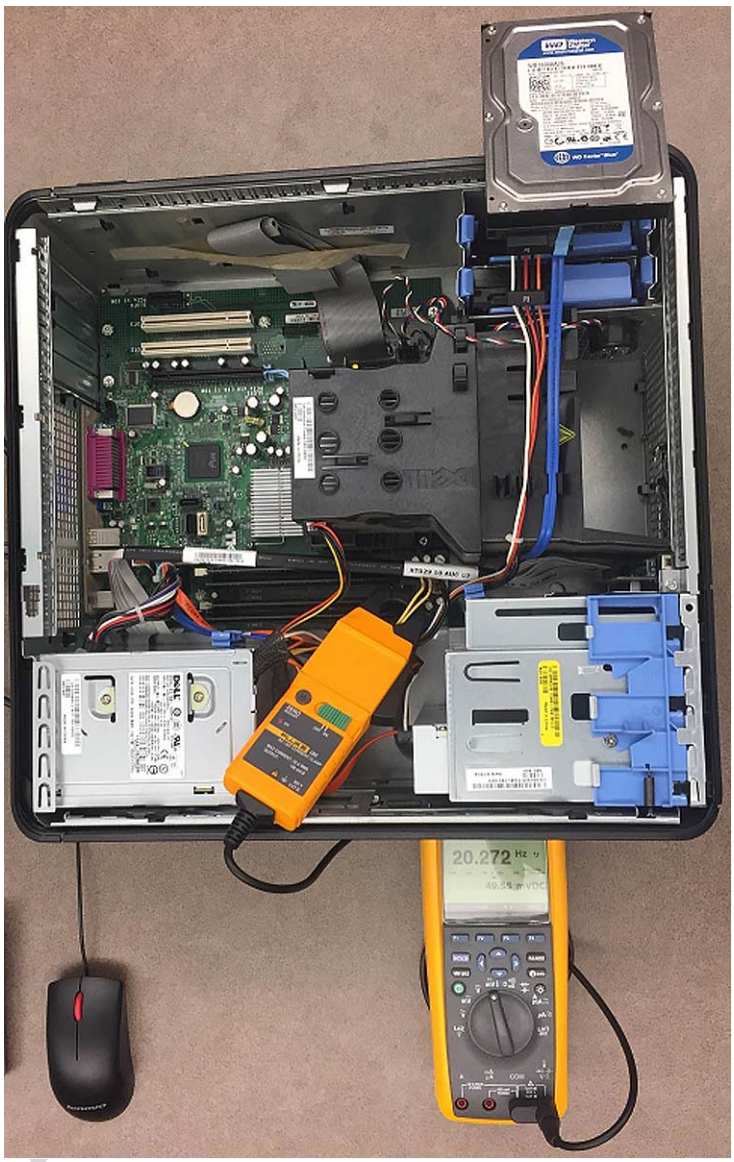



Fig. 1. Collection environment for power analysis.

The final operating system used for out of sample testing (Data Collection 3) was Windows 7 Professional with Service Pack 1. The rootkit was a version of Windows NT rootkit found at [48].

Data analysis was performed using Matlab version R2016A. Specific tool boxes used were the Neural Network Toolbox, Classification Toolbox, and the Statistics and Machine Learning Toolbox.

### *4.3. Computer hardware*

Figure 1 depicts the collection environment for the power analysis. The current clamps were placed around the two yellow power cords going from the motherboard to the CPU. The first computer used in our analysis was a Dell Optiplex 755 with an Intel Core 2 Duo CPU. The hard drive running the Ubuntu

14.04 Server operating system was a Western Digital WD1600AAJS with 160 gigabytes of memory. The hard drive running Windows 10 and Ubuntu Desktop was a western digital WD2500LPLX with 250 gigabytes of memory. Hard drives were erased using a Diskology Disk Jockey Pro Forensic Edition. The Disk Jockey was also used to copy the clean version of Ubuntu Server back to the hard drive. A clean version of both Windows 10 Pro and Ubuntu Desktop were reinstalled using a bootable image stored on a SanDisk Ultra USB 3.0.

For out of sample testing (Data Collection 3), we used a Dell Optiplex 790 with an Intel Core i3-2130 CPU with 3.40 GHz. The hard drive used was a Western Digital WD2500LPLX 250GB.

### *4.4. Data collection*

This section describes the different data sets collected for analysis. For clarity, the data sets are separated and labeled. Each data set is composed of eight collections. First, two collections are performed on the uninfected system. Then the system is infected with the appropriate rootkit and two more collections are performed. Then the hard drives are erased, the appropriate operating system is reinstalled, and the process is repeated.

#### *4.4.1. Data collection 1*

Data collection 1 was collected using the Fluke 289 Multimeter and Fluke i30 AC/DC current clamp. Again, the data instances were vectors of the following features: initial reading above/below threshold, duration of reading, minimum reading in the interval, maximum reading in the interval, average reading in the interval. The number of instances produced by a single collection is based on the number of times the values go above or below the threshold. This data collection process followed the following steps:

(1) Place the power clamp around the CPU power cords;
(2) Start recording on the Multimeter;
(3) Power on the Computer and log-in;
(4) Create a directory in C;
(5) Create a text file in the new directory;
(6) Open the text file and type “This is a test for rootkit detection!”;
(7) Save the file;
(8) Return to the C directory;
(9) Delete the created directory;
(10) Power off the computer and stop the multimeter from recording.

This process was performed on the Windows 10, Ubuntu server, and Ubuntu desktop operating systems. This resulted in three data sets from collection type 1. After logging on-to the computer, we waited until the multimeter showed little variability before conducting the above steps. This was to ensure that the initial start-up programs did not contaminate the readings. On average, the recording would begin within 60 to 120 seconds after logging in.

#### *4.4.2. Data collection 2*

Rather than a manual process, a stress tested was performed on the computer while the multimeter was collecting data. This data collection process followed the following steps:

(1) Place the power clamp around the CPU power cords;
(2) Power on the Computer and log in;
(3) Begin recording on the multimeter;

(4) Begin the stress test;
(5) After a given time frame stop the stress test and multimeter;
(6) Power off the computer.

The stress test lasted approximately five minutes. The stress test used on the Windows 10 operating system was HeavyLoad [30], and the stress test used on the Ubuntu operating systems was Stress-ng [32]. This process was performed on the Windows 10 and Ubuntu desktop operating systems. For this collection we were only concerned with the behavior of the system during the stress test. Therefore, the multimeter was not activated until after the boot sequence and at the same time the stress test was initiated. This is because the boot and shut down sequence would be approximately the same as the collection without the stress test (collection 1). This resulted in two data sets from collection type 2. Together, collection types 1 and 2 produced five data sets. The details of the collections are described in Table 1. We note the actual data set sizes may slightly vary due to some samples being unusable.

#### *4.4.3. Data collection 3*

Data collection 3 was utilized for out of sample testing. This data collection was the only collection which used the Windows 7 OS on the Dell Optiplex 790. The following procedure was followed when collecting the data:

(1) Place the power clamp around the CPU power cords;
(2) Start recording on the Multimeter;
(3) Power on the Computer and log-in;
(4) Open Notepad and type for 5 minutes, then close Notepad;
(5) Open MS Paint and draw/type for 5 minutes, then close;
(6) Open calculator and perform random calculations for 5 minutes, then close;
(7) Open Control Panel and navigate through various files for 5 minutes, then close;
(8) Open MS Paint and draw/type for 5 minutes, then close;
(9) Open Notepad and type for 5 minutes, then close;
(10) Open Word Pad and type for 5 minutes, then close;
(11) Open MS Paint and draw/type for 5 minutes, then close;
(12) Open Control Panel and navigate through various files for 5 minutes, then close;
(13) Open Word Pad and type for 5 minutes, then close;
(14) Power off the computer and stop the multimeter from recording.

A single clean and a single infected sample were gathered. The recordings lasted approximately 1 hour.

#### *4.4.4. Labeling and normalization*

Each data instance (feature vector) was assigned a class label of 1 for uninfected data and $-1$ for infected data. There were a total of five test sets and 5 corresponding training sets. Each training set contained four collections from a specific operating system either during a stress test or during a manual test and the corresponding test set contained four collections. The four collections were normalized independently of each other and merged to form one set for training and one for testing. The algorithms were trained on the individual instances of data. Because each collection followed the same method described above, each training/test set had approximately the same number of positive and negative class labels.

Much of the time and computational complexity involved in some machine learning algorithms, including those used in our analysis, is in the preprocessing of data. This process can become exponentially more difficult in real time and 'Big Data' scenarios. Therefore, one emphasis of this research is

Table 1
Description of individual data collections

| | Windows 10 | | | | | | | | Ubuntu Desktop | | | | | | | | Ubuntu Server | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Clean | | | | Infected | | | | Clean | | | | Infected | | | | Clean | | | | Infected | | | |
| | No Stress | | | | | | | | | | | | | | | | | | | | | | | |
| Collection | WC1 | WC2 | WC3 | WC4 | WI1 | WI2 | WI3 | WI4 | UC1 | UC2 | UC3 | UC4 | UI1 | UI2 | UI3 | UI4 | SC1 | SC2 | SC3 | SC4 | SI1 | SI2 | SI3 | SI4 |
| Number of Samples | 143 | 45 | 40 | 44 | 60 | 60 | 57 | 62 | 49 | 62 | 58 | 54 | 59 | 62 | 75 | 73 | 96 | 118 | 87 | 95 | 100 | 105 | 96 | 93 |
| Total | | | | 511 | | | | | | | | 492 | | | | | | | | 790 | | | | |
| | Stress | | | | | | | | | | | | | | | | | | | | | | | |
| Collection | WC1 | WC2 | WC3 | WC4 | WI1 | WI2 | WI3 | WI4 | UC1 | UC2 | UC3 | UC4 | UI1 | UI2 | UI3 | UI4 | – | – | – | – | – | – | – | – |
| Number of Samples | 189 | 165 | 160 | 184 | 148 | 183 | 160 | 162 | 119 | 121 | 113 | 83 | 82 | 134 | 136 | 133 | – | – | – | – | – | – | – | – |
| Total | | | | 1351 | | | | | | | | 921 | | | | | | | | – | | | | |

Table 2
Descriptive statistics of clean versus infected average power measurements on Windows 10 during stress test

| Status | Min | Q1 | Mean | Median | Q3 | Max |
|---|---|---|---|---|---|---|
| 1. Not Infected | 0 | .83 | .81 | .85 | .90 | .99 |
| 2. Infected | 0 | .93 | .90 | .94 | .96 | 1 |

to identify features and algorithms that need as little preprocessing as possible. To this end, the only preprocessing performed was standardization and normalization using the following formulas:

$$s(x_i) = \frac{x_i - mean}{\sigma} \tag{15}$$

$$s(x_i) = \frac{x_i - minimum}{maximum - minimum} \tag{16}$$

The first equation is the standard method of normalization that uses the mean and standard deviation. In general, the mean value of a data set provides a good measure of central tendency. This is under the assumption that the data set is reasonably large and there are no outliers. When these assumptions do not hold, the median will often yield better results. Table 2 is an example of the descriptive statistics for one data set. Both the infected and uninfected data had a mean and median that were relatively close. Because of this, and despite the fact that some data sets had outliers present, we still chose to normalize using the mean. The second equation scales all data points down to the range [0, 1]. After analysis, we found the Generalized Pareto Distribution was the closest fit, and in most cases the data were right skewed, as can be seen in Table 2. We note that during our experiments we evaluated both unnormalized and normalized data, and in every case the normalization improved the results.

A critical aspect of machine learning is estimating the importance and need of specific features. Including irrelevant features can, at best, slow training time and, at worst, reduce accuracy. In order to evaluate the features used in our analysis, we employed the RelieFF algorithm, which evaluates and weights the importance of features using a $K$ Nearest Neighbor method [53]. In our case, we chose $K$ to be 10. Figure 2 shows the results of RelieFF on a Windows 10 data set during a stress test. The fifth feature (minimum value during the interval) is assigned the highest weight, and is, therefore, considered the most important. The remaining features are all within a small range of importance. Because all weights were greater than zero, we choose to include all features in our analysis. The remaining data sets followed similar patterns.



## 5. Analysis

The following subsections describe the in sample and out of sample testing performed on the data.

### 5.1. In sample test

Table 3 presents the results of the analysis of data collection 1 from an Ubuntu Server operating system. The best performing algorithms were the neural networks and tree methods, and of the tree methods, the best were the ensemble tree methods. The nearest neighbor using Euclidean distance also performed well.

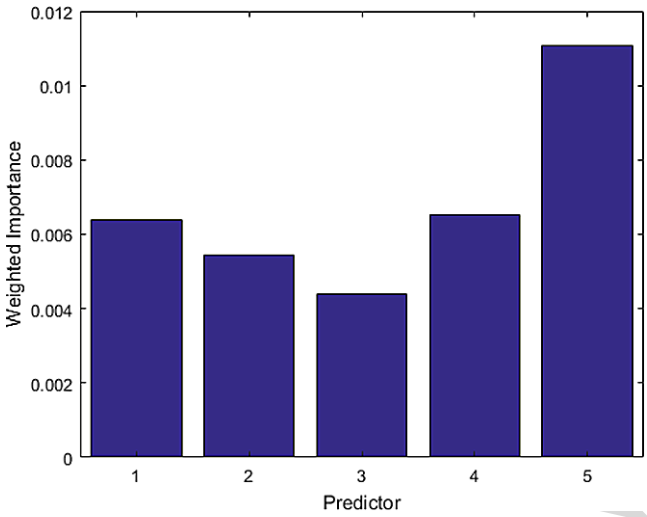



Fig. 2. Evaluation of predictor importance. 1 = Initial Reading, 2 = Interval Duration, 3 = Maximum, 4 = Average, 5 = Minimum.

Table 3
Analysis of Ubuntu server

| Method | Accuracy | AUC | Train Time (s) |
|---|---|---|---|
| 1. Complex Tree | 87 | .9 | 4 |
| 2. Medium tree | 87 | .93 | .7 |
| 3. Simple Tree | 78 | .76 | .6 |
| 4. LDA | 50 | .52 | 1 |
| 5. QDA | 48 | .49 | .77 |
| 6. Logistic Reg | 50 | .51 | 3 |
| 7. Linear SVM | 50 | .52 | 29 |
| 8. Quadratic SVM | 52 | .53 | 222 |
| 9. Cubic SVM | 50 | .5 | 171 |
| 10. Gaussian SVM | 75 | .82 | .9 |
| 11. NN-Euclidean | 84 | .84 | 1 |
| 12. NN-Minkowski | 78 | .86 | .57 |
| 13. NN-Cosine | 61 | 66.0 | .62 |
| 14. Boosted Tree | 88 | .95 | 8 |
| 15. Bagged Tree | 89 | .96 | 7 |
| 16. Neural network | 99 | .99 | 37 |



Table 4 illustrates the results of the analysis of data collection 2 from the Windows 10 operating system. Like the Ubuntu Server, the neural network, tree based methods, and ensemble tree methods performed the best. The support vector machines with linear and quadratic kernels also performed well, although the quadratic SVM required significantly more training time. The nearest neighbor algorithm using Euclidean distance also performed well.

Table 5 shows the results of the analysis on the Windows 10 operating system using the HeavyLoad stress test (data collection 2). The stress test was given one of the two cores and run as an administrator to allow reads and writes to the C drive. Again, the neural network and tree based methods performed

Table 4
Analysis of Windows 10

| Method | Accuracy | AUC | Train Time (s) |
|---|---|---|---|
| 1. Complex Tree | 81 | .83 | 6 |
| 2. Medium tree | 82 | .84 | 5 |
| 3. Simple Tree | 76 | .81 | 5 |
| 4. LDA | 55 | .48 | 8 |
| 5. QDA | 56 | .5 | 8 |
| 6. Logistic Reg | 59 | .49 | 8 |
| 7. Linear SVM | 85 | .83 | 9 |
| 8. Quadratic SVM | 87 | .88 | 119 |
| 9. Cubic SVM | 52 | .5 | 78 |
| 10. Gaussian SVM | 55 | .55 | 6 |
| 11. NN-Euclidean | 87 | .93 | 1 |
| 12. NN-Minkowski | 77 | .82 | 1 |
| 13. NN-Cosine | 82 | .88 | 2 |
| 14. Boosted Tree | 90 | .95 | 10 |
| 15. Bagged Tree | 89 | .96 | 10 |
| 16. Neural network | 93 | .95 | 27 |

Table 5
Analysis of Windows 10 with stress test

| Method | Accuracy | AUC | Train Time (s) |
|---|---|---|---|
| 1. Complex Tree | 80 | .88 | 2 |
| 2. Medium tree | 74 | .82 | 1 |
| 3. Simple Tree | 71 | .75 | 1 |
| 4. LDA | 60 | .66 | .92 |
| 5. QDA | 59 | .65 | .78 |
| 6. Logistic Reg | 69 | .74 | 1 |
| 7. Linear SVM | 68 | .74 | 55 |
| 8. Quadratic SVM | 57 | .75 | 724 |
| 9. Cubic SVM | 65 | .65 | 588 |
| 10. Gaussian SVM | 81 | .89 | 6 |
| 11. NN-Euclidean | 81 | .90 | 1 |
| 12. NN-Minkowski | 80 | .88 | 3 |
| 13. NN-Cosine | 76 | .85 | 3 |
| 14. Boosted Tree | 78 | .87 | 9 |
| 15. Bagged Tree | 85 | .94 | 8 |
| 16. Neural network | 88 | .93 | 181 |



the best. The support vector machine with a Gaussian kernel also performed well, but required considerably more training time. The nearest neighbor methods performed well, with the Euclidean method performing best.

Table 6 list the results of the analysis on Ubuntu Desktop for data collection 1. The neural network, boosted and bagged trees, and complex tree performed the best, with an average accuracy of approximately 82%. The remaining two tree-based methods performed reasonably well, as well as the nearest neighbor method with Euclidean distance. Surprisingly, all methods apart from linear and quadratic dis-

Table 6
Analysis of Ubuntu desktop

| Method | Accuracy | AUC | Train Time (s) |
|---|---|---|---|
| 1. Complex Tree | 82 | .86 | 2 |
| 2. Medium tree | 81 | .88 | 1 |
| 3. Simple Tree | 77 | .83 | 3 |
| 4. LDA | 51 | .49 | 3 |
| 5. QDA | 50 | .49 | 3 |
| 6. Logistic Reg | 51 | .48 | 4 |
| 7. Linear SVM | 52 | .48 | 90 |
| 8. Quadratic SVM | 55 | .51 | 269 |
| 9. Cubic SVM | 53 | .60 | 325 |
| 10. Gaussian SVM | 67 | .71 | 273 |
| 11. NN-Euclidean | 76 | .76 | 274 |
| 12. NN-Minkowski | 63 | .73 | 278 |
| 13. NN-Cosine | 63 | .69 | 272 |
| 14. Boosted Tree | 82 | .91 | 282 |
| 15. Bagged Tree | 84 | .93 | 288 |
| 16. Neural network | 83 | .90 | 39 |

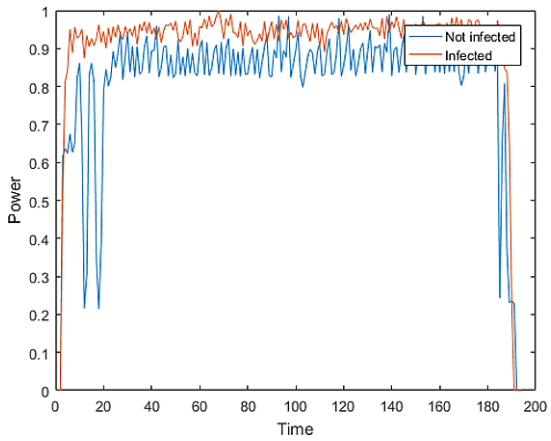
Table 7
Analysis of Ubuntu desktop with stress test

| Method | Accuracy | AUC | Train Time (s) |
|---|---|---|---|
| 1. Complex Tree | 85 | .88 | 2 |
| 2. Medium tree | 81 | .85 | 2 |
| 3. Simple Tree | 70 | .72 | 2 |
| 4. LDA | 58 | .64 | 2 |
| 5. QDA | 52 | .62 | 3 |
| 6. Logistic Reg | 88 | .87 | 6 |
| 7. Linear SVM | 91 | .89 | 58 |
| 8. Quadratic SVM | 86 | .94 | 390 |
| 9. Cubic SVM | 50 | .70 | 422 |
| 10. Gaussian SVM | 90 | .95 | 391 |
| 11. NN-Euclidean | 88 | .88 | 394 |
| 12. NN-Minkowski | 82 | .89 | 397 |
| 13. NN-Cosine | 86 | .93 | 397 |
| 14. Boosted Tree | 86 | .93 | 408 |
| 15. Bagged Tree | 90 | .96 | 415 |
| 16. Neural network | 86 | .93 | 48 |

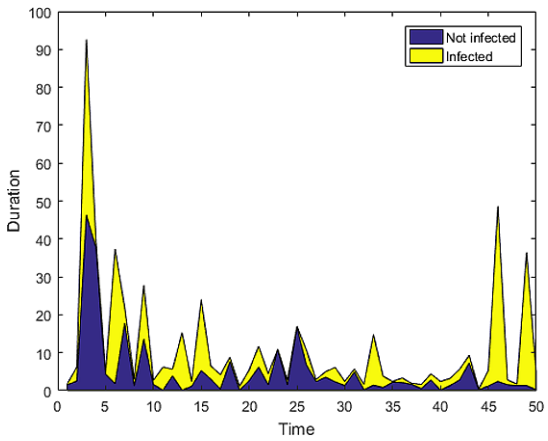


criminant analysis and the support vector machine with a cubic kernel performed well on the Ubuntu Desktop using the stress test (data collection 2), and the results are listed in Table 7.

Figure 3(a) is an example of the average power measurement reading per interval in one collection observation on Windows 10 Pro during a stress test after the previously discussed preprocessing techniques used in the analysis. The rootkit influenced the boot and shutdown procedures as well as caused a higher average reading throughout the stress test.




(a) Clean (lower line) versus infected (upper line) average power measurements on Windows 10 during stress test.




(b) Clean versus infected random interval duration on Ubuntu Desktop.

Fig. 3. Difference in clean and infected data.

Figure 3(b) shows the duration of the random time intervals for clean and infected Ubuntu Desktop. The readings for the infected operating system were higher than those of the uninfected system. The average interval for the uninfected reading was approximately 4.3, and 6.2 for the infected reading. The largest variation occurred during the boot up and shutdown procedure.

To identify the best performing algorithms for each test, we calculate an average score based on the order of performance of accuracy, AUC, and training time. As an example, given two algorithms, algorithm one has the best accuracy, the second-best AUC, and the second best training time. Therefore, the first algorithm would be $(1+2+2)/3 = 1.67$, and the second algorithm would be $(2+1+1)/3 = 1.33$. The second algorithm has the lower score, so it is considered better than the first. This is calculated across all 16 algorithms for each of the five data sets. Based on these calculations, the top five algorithms that performed best across all five data sets were the neural network, boosted trees, bagged trees, complex tree, and medium tree.

Intuitively, discriminant analysis methods did not perform well because they are somewhat dependent on data being approximately normal and covariance homogeneity [21]. We performed the Kolmogorov–Smirnov test for the null hypothesis that the data comes from a standard normal distribution and found said hypothesis to be rejected at a 5% significance level, indicating the data was not normally distributed. Decision trees and neural networks overcome these limitations by making no assumptions about the data. The intuition behind the nearest neighbor methods performing reasonably well is based on the clustering of the data. Figure 4 depicts the clustering of the average power values based on interval duration. The tight clusters with few outliers allow nearest neighbor methods to perform well.

While the accuracy and AUC of the five best performing algorithms showed promising results, it was anticipated that a method yielding an accuracy greater than 95% would be found. After the initial test were conducted, we attempted to improve the results using Principal Component Analysis (PCA). However, after PCA was performed on the data, there was no improvement in any of the models. We sought to represent the data in a way that would facilitate an increase in performance for any learning algorithm, as well as a learning algorithm that would be best suited for the specific data sets. The requirements sought from such a method were low computational costs to facilitate speed and low memory requirements.

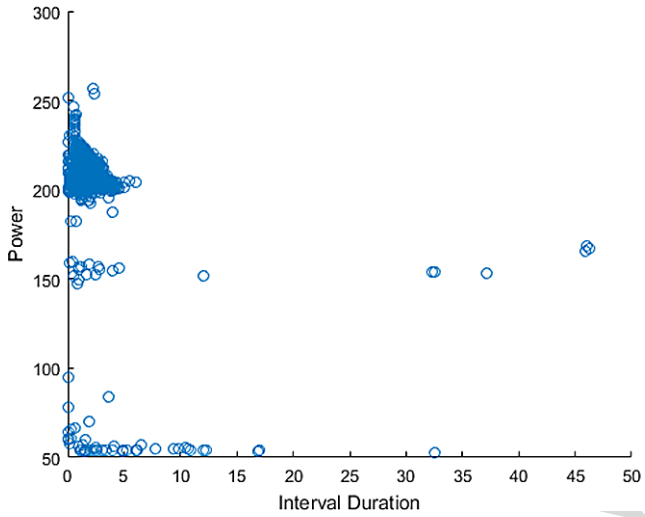



Fig. 4. Clustering of data.

A commonly used method for finding separability in data is by projecting the data into a higher dimension. This is based on Cover's Theorem [12], which states that given a set of data which is not linearly separable, one can transform it into a set that is linearly separable with high probability by projecting it into a higher dimensional space via a non-linear transformation provided the space is not densely populated. Kernel methods do this implicitly using specific functions. However, they depend on a kernel function that is generally not adaptable [5]. Still, it is also possible to explicitly project data into higher dimensions.

To project the data into higher dimensions, we employ the Self Organizing Map. As previously mentioned, SOMs are a type of unsupervised neural network that clusters data by projecting the input on-to a 2-dimensional grid based on similarity and topology [57]. The SOM algorithm was performed on each of our data sets using all five features. The SOM topology was a four-by-four hexagonal grid. The resulting output for each input vector was a 16-dimension vector containing a single one and 15 zeros. The one corresponds to the cluster the input was assigned to. This could be used to project the data into a higher dimension. By adding the vectors produced by the SOM to the already existing vectors, we would go from five dimensions to 21 dimensions. However, there are several issues with such a strategy. When projecting data into a higher dimension, it is possible to project too far, resulting in a decline in performance. Data that is separable in dimension D may not be separable in dimensions greater than D. Performance can also decline due to the curse of dimensionality and the cost of training the algorithms would be increased with the increase in dimensions. To overcome these issues, we treat the 16-dimension vector as a single binary number. We then translate the binary representations into the base 10 decimal representation. This yields a single unique value that represents a 16-dimension projection. The data was then normalized and feature scaled in the same manner as the original data.

To identify an optimal model for the data, we evaluated the existing results. The best performing learning algorithm, over all, was the neural network and the second best was the ensemble tree methods. Therefore, the algorithm we chose to use on the data to improve the performance was a type of hybrid ensemble neural network that we call a nested neural network. Nested neural networks consist of an ensemble of neural networks each of which is dedicated to learning a specific feature. The output of these networks is fed into a larger neural network that makes the final decision based on the results of

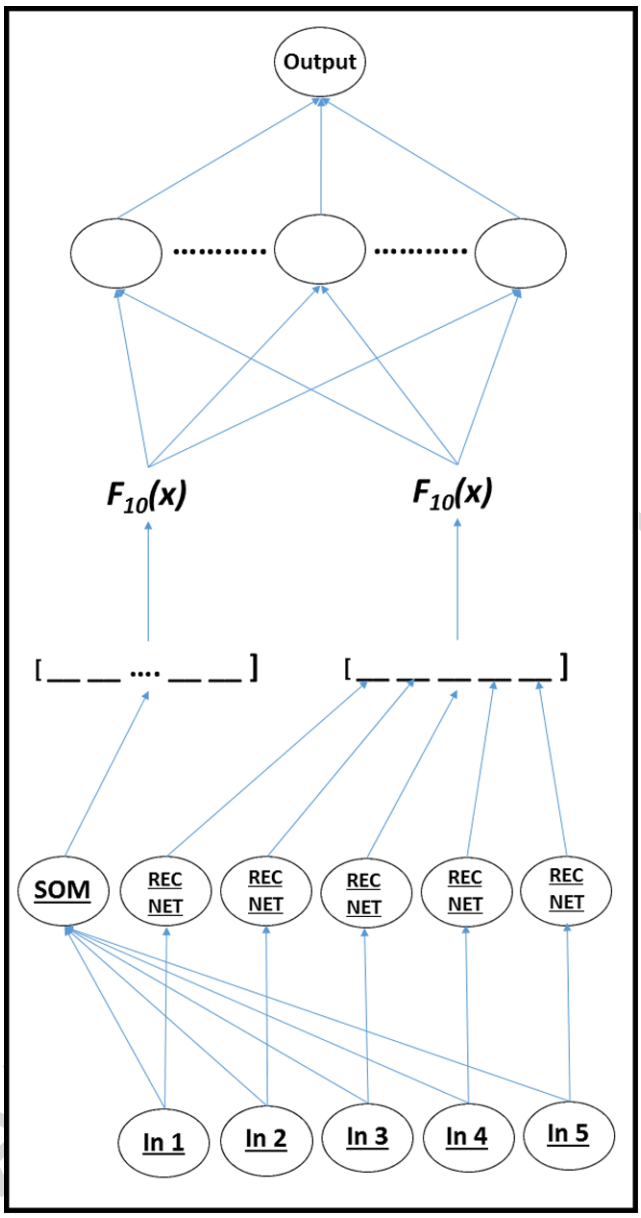



Fig. 5. Proposed ensemble nested network.

the smaller networks. This is inspired by biological studies showing that the human brain functions not as a single massive network, but as a group of small networks each dedicated to a specific task. There was a total of five smaller networks, each dedicated to its own feature. These neural networks were recurrent, meaning they contained feedback loops from the previous response into the current input. The output of these networks formed a binary vector. In the same manner as the SOM, the binary vector was converted to the base 10 decimal representation.

The nested network is depicted in Fig. 5. The first layer is the original normalized data. This data is then fed into the second layer, which contains the smaller specialized neural networks and the SOM.

Table 8
Average performance of top 5 algorithms and proposed model

| Method | Data Set 1 | | Data Set 2 | | Data Set 3 | | Data Set 4 | | Data Set 5 | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Accuracy | AUC | Accuracy | AUC | Accuracy | AUC | Accuracy | AUC | Accuracy | AUC |
| Complex Tree | 87 | .90 | 81 | .83 | 80 | .88 | 82 | .86 | 85 | .88 |
| Medium Tree | 87 | .93 | 82 | .84 | 74 | .82 | 81 | .88 | 81 | .85 |
| Boosted Tree | 88 | .95 | 90 | .95 | 78 | .87 | 82 | .91 | 86 | .93 |
| Bagged Tree | 89 | .96 | 89 | .96 | 85 | .94 | 84 | .93 | 90 | .96 |
| Neural Network | 99 | .99 | 93 | .95 | 89 | .93 | 83 | .90 | 86 | .93 |
| Nested Network | 99.8 | 1 | 99.9 | 1 | 99.9 | 1 | 99.8 | .99 | 99.5 | .99 |

The SOM received all the features of each vector, while the individual networks received only a single feature. Then, the two binary vectors are converted to the base 10 representation and fed into the final neural network.

The recurrent networks were trained by taking all training data sets and combining them into one large data set. Then, to improve generalization of the network, we generated extra data points by adding or subtracting random Gaussian noise to the existing points. The random Gaussian noise was in the range of plus or minus one tenth of the standard deviation. This created 10 extra points for every existing point in the original data. We also rounded the data points to two decimal places. The final neural network contained a single hidden layer with 20 neurons containing normalized radial base transfer functions. Table 8 depicts the results of the top 5 algorithms and the results of the proposed model for all five data sets. In all cases, the proposed model raised the accuracy to nearly 100%. On average, this model achieved 9.4% higher AUC and 17.2% higher accuracy than the traditional machine learning algorithms tested.

One important note is the test and training sets contained both clean and infected data, and there were approximately the same number of clean and infected samples (see Table 1). In a real-world scenario, the classifiers would be trained prior to use, and data would be collected and tested at the discretion of the user. Therefore, test sets would contain only clean data or only infected data. We performed tests to simulate such an environment using the proposed model. The algorithm was trained on both clean and infected data and tested on either clean or infected data. The algorithm made a prediction of either 1 or $-1$ for each instance. The predictions were then averaged to produce a score in the range $[-1, 1]$. The corresponding result represents the probability of being either infected or not infected. For example, a score of $-.67$ means the algorithm believes with a 67% confidence level that the system is infected with a rootkit. A score close to zero means the algorithm cannot make a meaningful prediction, and a score close to one means the algorithm believes the data is not infected. For our purposes, we use a hard threshold of $[-1, -.251]$ as infected, $[-.25, .25]$ as unknown, and $[.251, 1]$ as not infected. After training, the network was tested on a data set consisting of 200 uninfected points and returned a value of .91, correctly indicating uninfected data. We then tested against 200 infected points and returned a value of $-.93$, correctly indicating an infected data set. This not only illustrates a real-world implementation of our method, but also shows the algorithm can make meaningful predictions with very small data sets. Another important factor to consider is the false positive rate. It is reasonable to assume that in a real world setting there will be more clean data, and infected data would be considered an anomaly. Our model achieved a false positive rate of .015 on a point by point basis. We believe this is an acceptable rate, however in a real world setting one would not make judgments based on an individual point. Judgments would be made based on a group of consecutive points over a given time interval and evaluated in a manner like the above test. This is because if the computer is infected, then every data point will be

infected, not just an individual point. Therefore, the likelihood of false positives, both because of the low rate on a point to point basis, and evaluating multiple data points as above, would be highly unlikely.

To further evaluate the effectiveness and real world applications, we performed an analysis using the five top algorithms and the proposed model on two extra data sets. The first data set consisted of the three data sets under normal conditions combined into one and the second data set consisted of the two data sets using the stress test combined into one. The goal of this analysis was to see if the proposed model could not only differentiate between clean and infected data, but also data originating from different operating systems and from different hardware. Among the top five algorithms the average accuracy was 87% for the combined data set under normal conditions and, for the combined stress test data, an average accuracy of 85% was achieved. Our proposed model achieved 97% and 96% respectively.

### *5.2. Out of sample test*

A primary concern of any learning algorithm is overfitting. Overfitting occurs when an algorithm learns a specific training data set too precisely. This often leads to poor performance when new data is introduced. For the previous test discussed, several methods were employed to overcome overfitting, including cross validation on the classification models, hold out validation on the neural network models, rounding data points to lower decimal places, and generating data with random noise added. Still, validation on completely new data is need to identify how well the algorithm generalizes. To that end we performed one final test to evaluate the proposed model. Several aspects needed to be addressed to evaluate how well the model generalized, including 1) a different rootkit, 2) a different operating system, 3) different hardware, 4) different use case.

The nested network was trained and validated on the data from Windows 10 and Ubuntu desktop, and tested on the data from collection 3. A total of 322 data points were collected. Of those, 216 came from the infected operating system, and 106 came from the clean operating system. Of the 216 infected data points, the algorithm correctly identified 201 as infected, corresponding to a true positive rate of 93% and a false negative rate of 7%. Of the 106 clean data points, the algorithm correctly identified 83 as clean, corresponding to a true negative rate of 78% and a false positive rate of 22%. The overall accuracy of the model was 88%.

These results reflect several promising features of our proposed model. First, the algorithm can correctly classify anomalous data from an unknown rootkit. Because new malware is generated every day, the ability to generalize and identify unknown malware is required. Second, the accuracy of the algorithm is platform independent in both hardware and software, which would be required for any large scale or commercial use. Lastly, the algorithm performed well under a different use case than previously tested. This is necessary to accommodate systems with multiple users. In all, these results show that CPU power is both a reliable feature and able to generalize to many different situations.



## 6. Discussion and future work

While we have performed a number of successful test, the question still remains if this method could function in a real-world setting. Given the number of possible applications that could be running on a computer at any given time, modeling and simulating every possibility is not possible. In this research we intentionally analyzed data during boot, normal use, and shut down as a proof of concept. However, in a real-world setting, the data would likely be gathered during boot before other applications were loaded, and therefore not contaminate the data. This was the method used in our previous works [14,15,37],

which were successful. We are confident this method would work given the difference in the data we observed at boot (see Fig. 3a and b). An alternative would be to run a specific script during known down times and gather data then. Another possible application could be to monitor supervisory control and data acquisition (SCADA) systems. These systems have minimal third-party applications, do not update to different software often, and only do specific task which are already known. In any case, it would be necessary to gather data on the system and adapt the already trained learning algorithm to fit the specific system better in order to achieve accuracy $\geqslant 98\%$. This is clear given the slight decline in performance during our out of sample testing.

Future work will include further testing and tuning of various machine learning algorithms to attempt to achieve higher accuracy. Many current machine learning algorithms, such as deep learning, do not require preprocessing of data. This could alleviate the need for normalization, and therefore increase the effectiveness of real time monitoring. However, these methods often increase the amount of training time required.

One issue not addressed in this research is adaptive adversaries. Adaptive adversaries purposely manipulate data to compromise machine learning algorithms [11]. The algorithms can be attacked during training of the model parameters or during testing when classifying samples [26]. The mitigation of these adversarial inputs remains an open problem. A primary strength of using CPU power data features is that the data collection takes place at a base-level making it difficult to implement adaptive adversary techniques. In addition, this research implements a methodology that captures data via a side channel analysis that operates independently from the system being monitored to mitigate adversarial activities further. Often, the success of an attack depends on available resources [49]. For example, attack success can be dependent on system access and algorithm knowledge. A benefit of the side channel approach implemented in this research is that it can be run independently and offline from the monitored system. Hence, the attacker would not have access to I/O pairs. The success of the attack is also dependent on the attacker's knowledge of the algorithm. The results of this research demonstrate that the proposed method works on many different learning algorithms. Thus, using multiple models consecutively, or alternating models at random would decrease the likelihood of an attacker's success. Another approach is to design the models themselves to be robust to attacks, which recent research has proven successful [59]. Future research will investigate effective and efficient solutions that mitigate adaptive adversarial techniques along with the viability of other solutions like artificial intelligence concepts.

Another option would be to use methods that are time dependent, such as recurrent neural networks and autoregressive models to evaluate the entire data set at once. We also intended on evaluating other stress test, such as cross platform test that work on most operating systems.

Another interesting aspect would be to consider adding additional features to the data to attempt to improve performance. These could include other side channels, such as temperature fluctuation, power measurements from other parts of the computer, or sensor arrays measuring many aspects of the computer simultaneously. While other measures may slightly improve performance, it is likely the improvement would be menial. It has been shown that computers are complex nonlinear dynamical systems [44]. It has also been shown [43] that in practice, there is no advantage to adding more features when evaluating complex systems. This is assuming one chooses a single signal that can truly represent the state of the whole system, which we believe is captured by the CPU. A better solution would be to combine this method with signature based methods capable of identifying known rootkits and other malware, which will be evaluated in future work.

In this research we focus on rootkits as part of evaluating an initial hypothesis on whether power side channels can be used to detect malicious software execution. Our original hypothesis was that

rootkits, those that use system call hooking or alter normal system resource interfaces, by nature include more code than normal system calls or system access routines. Rootkits typically execute the original functionality of system/kernel level functions and then add additional code which is designed to hide their presence or provide exfiltration or system alteration. Additional code, theoretically, would translate to more machine code instructions, more CPU cycles, and thus more CPU power. General malware has a larger variety of functionality and potential usage and encompasses a much larger corpus of potential samples. Our future and next work in using this technique will be evaluation of general malware families. An external physical device will be implemented that can monitor the CPU to detect the presence of malware in near real time, and this device will be tested on real-world production servers for general evaluation.

## 7. Conclusion

This research presents a method of rootkit detection based on CPU power consumption. By measuring the power consumed by a CPU, we can detect the presence of malicious rootkits with high accuracy. The data sets used in the analysis were small enough to allow for fast training while not sacrificing reliability. Further, the data instances produced in the analysis were quite vague, providing little more than a minimum, maximum, average, and interval duration. Despite these restrictions, this method showed promising results. The results of the research demonstrate that this method can detect rootkits on four different operating systems, namely Windows 7, Windows 10, Ubuntu Desktop, and Ubuntu Server. It also shows the effectiveness of the proposed algorithm on different hardware, as well as four different rootkits, namely FUTo, KBeast, NT, and Azazel. This method can detect both user level and kernel level rootkits. The research results also demonstrate the detection capability on several different machine learning algorithms. Of those, the best performing algorithms were neural networks, decision trees, ensemble methods, and nearest neighbor methods. Furthermore, the capability of this method was demonstrated on two different tests, first under normal operating conditions and then during a stress tests. Lastly, a model inspired by biological studies of the human brain, called a nested neural network, could classify all data sets at 99% accuracy.



## References

[1] M. Alazab, S. Venkatraman, P. Watters and M. Alazab, Zero-day malware detection based on supervised learning algorithms of API call signatures, in: *Proceedings of the Ninth Australasian Data Mining Conference*, Vol. 121, Australian Computer Society, Inc., 2011, pp. 171–182.

[2] P.D. Allison, *Logistic Regression Using SAS: Theory and Application*, SAS Institute, 2012.

[3] B. Arslan, S. Gunduz and S. Sagiroglu, A review on mobile threats and machine learning based detection approaches, in: *Digital Forensic and Security (ISDFS), 2016 4th International Symposium on*, IEEE, 2016, pp. 7–13.

[4] K.J. Berman, W.B. Glisson and L.M. Glisson, Investigating the impact of global positioning system evidence, in: *System Sciences (HICSS), 2015 48th Hawaii International Conference on*, IEEE, 2015, pp. 5234–5243. doi:10.1109/HICSS.2015.618.

[5] P.P. Brahma, D. Wu and Y. She, Why deep learning works: A manifold disentanglement perspective, *IEEE transactions on neural networks and learning systems* **27**(10) (2016), 1997–2008. doi:10.1109/TNNLS.2015.2496947.

[6] R. Bray, D. Cid and A. Hay, *OSSEC Host-Based Intrusion Detection Guide*, Syngress, 2008.

[7] A. Brown, M. Yampolskiy, J. Gatlin and T. Andel, Legal aspects of protecting intellectual property in additive manufacturing, in: *Critical Infrastructure Protection X: 10th IFIP WG 11.10 International Conference, ICCIP 2016, Arlington, VA, USA, March 14–16, 2016, Revised Selected Papers 10*, Springer, 2016, pp. 63–79. doi:10.1007/978-3-319-48737-3_4.

[8] P. Carsten, T.R. Andel, M. Yampolskiy and J.T. McDonald, In-vehicle networks: Attacks, vulnerabilities, and proposed solutions, in: *Proceedings of the 10th Annual Cyber and Information Security Research Conference*, ACM, 2015, p. 1.
[9] Check Point, Humming Bad: A persistent mobile chain attack.
[10] F. Cohen, Computer viruses: Theory and experiments, *Computers & security* **6**(1) (1987), 22–35. doi:10.1016/0167-4048(87)90122-2.
[11] I. Corona, B. Biggio and D. Maiorca, AdversariaLib: An open-source library for the security evaluation of machine learning algorithms under attack, preprint, arXiv:1611.04786, 2016.
[12] T.M. Cover, Geometrical and statistical properties of systems of linear inequalities with applications in pattern recognition, *IEEE transactions on electronic computers* **3** (1965), 326–334. doi:10.1109/PGEC.1965.264137.
[13] S. Das, Y. Liu, W. Zhang and M. Chandramohan, Semantics-based online malware detection: Towards efficient real-time protection against malware, *IEEE transactions on information forensics and security* **11**(2) (2016), 289–302. doi:10.1109/TIFS.2015.2491300.
[14] J. Dawson, J.T. Mcdonald, L. Hively, T. Andel, M. Yampolskiy and C. Hubbard, Phase space detection of virtual machine cyber events through hypervisor-level system call analysis, in: *Proceedings of the International Conference on Data Intelligence and Security*, 2018.
[15] J. Dawson, J.T. Mcdonald, J. Shropshire, T. Andel, P. Luckett and L. Hively, Rootkit detection through phase-space analysis of power voltage measurements, in: *Proceedings of the 12th IEEE International Conference on Malicious and Unwanted Software*, 2017.
[16] G. De'Ath, Boosted trees for ecological modeling and prediction, *Ecology* **88**(1) (2007), 243–251. doi:10.1890/0012-9658(2007)88[243:BTFEMA]2.0.CO;2.
[17] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan and S. Stolfo, On the feasibility of online malware detection with performance counters, in: *ACM SIGARCH Computer Architecture News*, Vol. 41, ACM, 2013, pp. 559–570.
[18] G. Dini, F. Martinelli, A. Saracino and D. Sgandurra, MADAM: A multi-level anomaly detector for Android malware, in: *Computer Network Security*, 2012, pp. 240–253. doi:10.1007/978-3-642-33704-8_21.
[19] S. Dubey and J. Dubey, KBB: A hybrid method for intrusion detection, in: *Computer, Communication and Control (IC4), 2015 International Conference on*, IEEE, 2015, pp. 1–6.
[20] N. Falliere, L.O. Murchu and E. Chien, W32. stuxnet dossier, *White paper, Symantec Corp., Security Response* **5**(6) (2011).
[21] M.R. Feldesman, Classification trees as an alternative to linear discriminant analysis, *American Journal of Physical Anthropology* **119**(3) (2002), 257–275. doi:10.1002/ajpa.10102.
[22] C. Fraley and A.E. Raftery, Model-based clustering, discriminant analysis, and density estimation, *Journal of the American statistical Association* **97**(458) (2002), 611–631. doi:10.1198/016214502760047131.
[23] Y. Freund, R.E. Schapire et al., Experiments with a new boosting algorithm, in: *Icml*, Vol. 96, 1996, pp. 148–156.
[24] J. Friedman, T. Hastie and R. Tibshirani, *The Elements of Statistical Learning*, Vol. 1, Springer Series in Statistics, New York, 2001. doi:10.1007/978-0-387-21606-5.
[25] S. Garcia, J. Derrac, J. Cano and F. Herrera, Prototype selection for nearest neighbor classification: Taxonomy and empirical study, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **34**(3) (2012), 417–435. doi:10.1109/TPAMI.2011.142.
[26] K. Grosse, P. Manoharan, N. Papernot, M. Backes and P. McDaniel, On the (statistical) detection of adversarial examples, preprint, arXiv:1702.06280, 2017.
[27] J.M. Hernández, A. Ferber, S. Prowell and L. Hively, Phase-space detection of cyber events, in: *Proceedings of the 10th Annual Cyber and Information Security Research Conference*, ACM, 2015, p. 13.
[28] G. Hoglund and J. Butler, *Rootkits: Subverting the Windows Kernel*, Addison-Wesley Professional, 2006.
[29] IPSECS, Kbeast Rootkit, http://core.ipsecs.com/rootkit/kernel-rootkit/kbeast-v1/.
[30] Jam Software, HeavyLoad, https://www.jam-software.com/heavyload/.
[31] G. James, D. Witten, T. Hastie and R. Tibshirani, Support vector machines, in: *An Introduction to Statistical Learning*, Springer, 2013, pp. 337–372. doi:10.1007/978-1-4614-7138-7_9.
[32] C. King, Stress-ng, http://manpages.ubuntu.com/manpages/wily/en/man1/stress-ng.1.html.
[33] R. Kohavi et al., A study of cross-validation and bootstrap for accuracy estimation and model selection, in: *Ijcai*, Vol. 14, Stanford, CA, 1995, pp. 1137–1145.
[34] I. Kononenko and M. Kukar, *Machine Learning and Data Mining: Introduction to Principles and Algorithms*, Horwood Publishing, 2007. doi:10.1533/9780857099440.
[35] T.V.N. Lakshmi and V.K. Babu, Detection of user to root attacks using machine learning techniques, *International Journal of Advanced Engineering and Global Technology (IJAEGT)* **3**(3) (2015).
[36] D. Lobo, P. Watters and X. Wu, Rbacs: Rootkit behavioral analysis and classification system, in: *Knowledge Discovery and Data Mining, 2010. WKDD'10. Third International Conference on*, IEEE, 2010, pp. 75–80. doi:10.1109/WKDD.2010.23.

[37] P. Luckett, J.T. McDonald and J. Dawson, Neural network analysis of system call timing for rootkit detection, in: *Cybersecurity Symposium (CYBERSEC), 2016*, IEEE, 2016, pp. 1–6.
[38] K. Marsh, Win32 hooks, *Microsoft Developer Network*, 1993.
[39] P. McDaniel, N. Papernot and Z.B. Celik, Machine learning in adversarial settings, *IEEE Security & Privacy* **14**(3) (2016), 68–72. doi:10.1109/MSP.2016.51.
[40] J.E.R. McMillan, W.B. Glisson and M. Bromby, Investigating the increase in mobile phone evidence in criminal activities, in: *System Sciences (HICSS), 2013 46th Hawaii International Conference on*, IEEE, 2013, pp. 4900–4909. doi:10.1109/HICSS.2013.366.
[41] D. Meyer and F.T. Wien, Support vector machines, *R News* **1**(3) (2001), 23–26.
[42] D. Mink, A. Yasinsac, K.-K.R. Choo and W. Glisson, Next generation aircraft architecture and digital forensic, 2016.
[43] M.R. Muldoon, D.S. Broomhead and J.P. Huke, Delay reconstruction for multiprobe signals, in: *Exploiting Chaos in Signal Processing, IEE Colloquium on*, IET, 1994, pp. 3/1–3/6.
[44] T. Mytkowicz, A. Diwan and E. Bradley, Computer systems are dynamical systems, *Chaos: An Interdisciplinary Journal of Nonlinear Science* **19**(3) (2009), 033124.
[45] Y. Namestnikov, Kaspersky Security Bulletin 2015. Evolution of cyber threats in the corporate sector, 2015.
[46] Offensive Computing, http://www.offensivecomputing.net/.
[47] Packet Storm, https://packetstormsecurity.com/UNIX/penetration/rootkits.
[48] Packet Storm, NT Rootkit, 2001, https://packetstormsecurity.com/files/25071/_root_040.zip.html.
[49] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z.B. Celik and A. Swami, The limitations of deep learning in adversarial settings, in: *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*, IEEE, 2016, pp. 372–387. doi:10.1109/EuroSP.2016.36.
[50] R.S. Pirscoveanu, S.S. Hansen, T.M. Larsen, M. Stevanovic, J.M. Pedersen and A. Czech, Analysis of malware behavior: Type classification using machine learning, in: *Cyber Situational Awareness, Data Analytics and Assessment (CyberSA), 2015 International Conference on*, IEEE, 2015, pp. 1–7.
[51] B. Pradhan, A comparative study on the predictive ability of the decision tree, support vector machine and neuro-fuzzy models in landslide susceptibility mapping using GIS, *Computers & Geosciences* **51** (2013), 350–365. doi:10.1016/j.cageo.2012.08.023.
[52] R.G. Ramani, S.S. Kumar and S.G. Jacob, Rootkit (malicious code) prediction through data mining methods and techniques, in: *Computational Intelligence and Computing Research (ICCIC), 2013 IEEE International Conference on*, IEEE, 2013, pp. 1–5.
[53] M. Robnik-Šikonja and I. Kononenko, Theoretical and empirical analysis of ReliefF and RReliefF, *Machine learning* **53**(1–2) (2003), 23–69. doi:10.1023/A:1025667309714.
[54] J. Shropshire, Securing cloud infrastructure: Unobtrusive techniques for detecting hypervisor compromise, in: *ICCSM2015 – 3rd International Conference on Cloud Security and Management: ICCSM2015*, Academic Conferences and Publishing Limited, 2015, p. 86.
[55] S. Sparks and J. Butler, Shadow walker: Raising the bar for rootkit detection, *Black Hat Japan* **11**(63) (2005), 504–533.
[56] R. Unuchek, F. Sinitsyn, D. Parinov and V. Stolyarov, IT threat evolution Q1 2017. Statistics, 2017, https://securelist.com/it-threat-evolution-q1-2017-statistics/78475.
[57] J. Vesanto and E. Alhoniemi, Clustering of the self-organizing map, *IEEE Transactions on neural networks* **11**(3) (2000), 586–600. doi:10.1109/72.846731.
[58] Y. Xu, Q. Zhu, Z. Fan, M. Qiu, Y. Chen and H. Liu, Coarse to fine K nearest neighbor classifier, *Pattern recognition letters* **34**(9) (2013), 980–986. doi:10.1016/j.patrec.2013.01.028.
[59] V. Zantedeschi, M.-I. Nicolae and A. Rawat, Efficient defenses against adversarial attacks, in: *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, ACM, 2017, pp. 39–49.
[60] K. Zetter, An unprecedented look at Stuxnet, the world's first digital weapon, *Wired*, 2014.
[61] Y. Zhang, G. Tao and M. Chen, Adaptive neural network based control of noncanonical nonlinear systems, *IEEE transactions on neural networks and learning systems* **27**(9) (2016), 1864–1877. doi:10.1109/TNNLS.2015.2461001.
[62] M. Zhou, G. Fortino, W. Shen, J. Mitsugi, J. Jobin and R. Bhattacharyya, Guest editorial special section on advances and applications of Internet of things for smart automated systems, *IEEE Transactions on Automation Science and Engineering* **13**(3) (2016), 1225–1229. doi:10.1109/TASE.2016.2579538.