ONLINE MALWARE DETECTION IN CLOUD AUTO-SCALING SYSTEMS USING PERFORMANCE METRICS

by

MAHMOUD ABDELSALAM, M.Sc.

DISSERTATION
Presented to the Graduate Faculty of
The University of Texas at San Antonio
In Partial Fulfillment
Of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

COMMITTEE MEMBERS:
Prof. Ravi Sandhu, Ph. D., Co-Chair
Dr. Ram Krishnan, Ph.D., Co-Chair
Dr. Matthew Gibson, Ph.D.
Dr. Murtuza Jadliwala, Ph.D.
Dr. Gregory White, Ph.D.

THE UNIVERSITY OF TEXAS AT SAN ANTONIO
College of Sciences
Department of Computer Science
December 2018

DEDICATION

I would like to dedicate this dissertation to my beloved family.

ACKNOWLEDGEMENTS

My PhD journey has certainly been a great learning experience, on many different levels, and there are many people who have had an impact and who deserve to be sincerely thanked.

Thanks to my supervisors, Prof. Ravi Sandhu and Dr. Ram Krishnan, for their diligent efforts and close mentoring. Their vast knowledge, wisdom, continuous guidance and support of my research have helped shape the contributions in this dissertation. This dissertation would not have been accomplished without their inspiration and encouragement.

I would also like to thank my Ph.D committee members Dr. Mathew Gibson, Dr. Murtuza Jadliwala, and Dr. Gregory White for their time and insightful comments.

I would like to thank my dear wife, Elham, and my lovely daughter, Dana, who have been with me all these years and have made them the best years of my life. I am indebted to my parents, Abdelsalam and Shokran, for all their love and blessings, and imbibing the values of life in me. I appreciate all their sacrifices and value the lessons of life they have taught me growing up. I am really grateful to my brothers, Mohamed and Khaled, and my sister, Ghada, for their continuous and unparalleled love, help and support.

I would also like to express my gratitude to members of the ICS, Farhan Patwa, James Benson, Suzanne Tanaka and Lisa Ho for their help and motivation. I am thankful to my colleagues at UTSA, Maanak Gupta, Smriti Bhatt, Moustafa Saleh and Sajad Khorsandroo and others for their companionship, useful discussions and ideas.

Finally, thanks to NSF (CNS-1553696) and DoD (DoD W911NF-15-1-0518) for supporting this research.

December 2018

ONLINE MALWARE DETECTION IN CLOUD AUTO-SCALING SYSTEMS USING PERFORMANCE METRICS

Mahmoud Abdelsalam, Ph.D.
The University of Texas at San Antonio, 2018

Supervising Professors: Prof. Ravi Sandhu, Ph. D. and Dr. Ram Krishnan, Ph.D.

Cloud computing is becoming increasingly popular among organizations. The Infrastructure as a Service (IaaS) cloud computing model has become an attractive solution because of the ability of reducing costs and improving resource utilization. Such cloud services are expected to be always available and reliable as per the Service Level Agreements (SLA) between the cloud service providers (CSPs) and their customers. Cloud ecosystems have also become attractive targets to attackers because of the massive amount of data residing on the cloud as well as the massive processing power that can be recruited for malicious intent. Thus, security is a very critical task in cloud ecosystems and the need of continuous security monitoring in the cloud is mandatory for detecting malicious activities.

This dissertation addresses the problem of online malware detection in cloud auto-scaling systems using performance metrics. First, we review the current state-of-the-art malware detection techniques in general with a focus on techniques that target cloud IaaS, specifically virtual machines (VMs). We find that malware detection techniques that target VMs lack taking advantage of cloud unique characteristics. Those techniques can be applied to VMs as well as stand alone servers with nothing specific about cloud.

We then propose a malware detection framework that leverages cloud unique characteristics (i.e. auto-scaling) using black-box features (performance metrics), where data are collected from outside the VMs by the hypervisor in an auto-scaling scenario (e.g. three-tier web architecture with scalability in place). Our approach assumes no prior knowledge of the installed applications on the VMs. In this work, a modified version of sequential K-means clustering algorithm is used to group similar VMs based on workloads (e.g. applications servers, web servers and database servers are

three different groups). Then, malware is detected as anomalies when one VM of the same group exceeds a certain threshold.

Despite showing that highly active malware (e.g. ransomware) can be effectively detected by inspecting the performance and resource utilization metrics of VMs as a black-box, this approach is not as effective for detecting malware that maintains a low profile of resource utilization. Accordingly, we propose a white-box approach (where data are obtained from inside the VMs by either the hypervisor or pre-installed agents) for detecting such malware using 2d and 3d Convolutional Neural Networks (CNN). 3d CNN classifiers are introduced to partially mitigate the underestimated mislabeling problem.

The developed white-box approach achieved good results; however, it works only for single VMs. To leverage auto-scalability, we extended the previous approach to handle multiple VMs and introduced a new approach based on paired samples to accommodate for correlations between VMs.

We evaluate the proposed approaches on synthetic data collected from our OpenStack (a popular open-source cloud IaaS software) testbed based on a standard 3-tier web architecture with the ability to scale-up (when multiple copies of the server are spawned) and scale-down (where the number of copies are reduced) on demand.

TABLE OF CONTENTS

Vita

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AIS Artificial Immune Systems

ANN Artificial Neural Networks

CNN Convolutional Neural Networks

CSP Cloud Service Provider

DBSCAN Density-based Spatial Clustering of Applications with Noise

EDoS Economic Denial of Sustainability

FN False Negative

FP False Positive

GA Genetic Algorithm

GRU Gated Recurrent Units

IaaS Infrastructure as a Service

IDF Inverse Document Frequency

IDS Intrusion Detection System

KNN K-nearest Neighbors

LSTM Long Short Term Memory

ML Machine Learning

MLP Multi Layer Perceptron

NIDS Network Intrusion Detection System

NIST National Institute of Standards and Technology

PaaS Platform as a Service

RVSM Robust support vector machine

SaaS Software as a Service

SOM Self Organizing Map

TN True Negative

TP True Positive

VM Virtual Machine

CHAPTER 1: INTRODUCTION

Organizations increasingly utilize cloud services such as Infrastructure as a Service (IaaS) where virtualized IT infrastructure is offered on demand by cloud providers. A major challenge for cloud providers is the security of virtual infrastructures that are provided to their customers. In particular, a key concern is whether virtual machines (VMs) in the data center are performing tasks that are not expected of those machines. Given the scale of data centers, *continuous* security monitoring of the virtual assets is essential to detect malicious behavior.

Cloud infrastructure has become increasingly prone to novel attacks and malware [21, 31, 33, 34, 39, 90]. One of the most prevalent threats to cloud is malware. Cloud malware injection [34] is a threat where an attacker injects a malware to manipulate the victim's Virtual Machine (VM). Due to the nature of the cloud and automatic provisioning, a large number of VMs are often similarly configured. One example is when a web server scales-out due to increase in demand and scales-in when the demand goes down. This means that the attack that compromised one of the VMs is highly likely to compromise many of the other similar VMs. The attacker can inject a botware to be used for creating a botnet due to a large number of VMs available in scaling scenarios. As a result, the need for malware detection in VMs is critical. Consequentially, this dissertation addresses the following questions:

Given an organization (known as a cloud tenant) that utilizes cloud services (i.e. VMs), what mechanisms are available for online malware detection (also known as real-time malware detection) in the utilized VMs using machine learning? How can auto-scaling, a cloud's unique characteristic, be leveraged for online malware detection in cloud?

1.1 Motivation

Cloud data centers are widely used for a range of always-on services across all the cloud deployment models (public, private, and hybrid). These data centers are used as an infrastructure for resource pooling to multiple cloud tenants (cloud customers). Cloud tenants' VMs need to be

secure and resilient in the face of the new attacks that are introduced, and will continue to be introduced, because of the virtualization and cloud ecosystem nature. The following scenarios are some of the cloud new threats.

Exploited system vulnerabilities. These are not new, but they've become a bigger issue with the advent of multi-tenancy in cloud computing. Tenants share memory, databases, and other resources in close proximity to one another, which creates new attack surfaces. Although attacks on system vulnerabilities can be mitigated by existing mechanisms (e.g. vulnerability scanners), it is still an issue that is yet to be completely solved.

Configuration vulnerabilities. Due to the nature of the cloud and automatic provisioning, a large number of the VMs are similarly configured, hence many VMs have the same vulnerabilities or misconfiguration. For example, a configuration script is used to spawn new VMs based on workload (i.e., auto-scaling). This increase the chance that malware can infect many VMs as well as sets an outstanding opportunity for attackers to target cloud systems.

Insider threats. Utilizing cloud resources means that a customer's data security is as good as the utilized cloud security. Insider threats can be a way of malware infection (no matter intentionally or unintentionally). These can arise in many ways such as through a current or former employee or a system administrator. The intent can go from data theft to revenge. For example, an administrator who intentionally injects malware to steal sensitive customer data.

Compromised credentials. This can happen in many different ways from weak passwords to leaving passwords in the open. For example, many developers make the mistake of embedding credentials and cryptographic keys in source code and leaving them in public-facing repositories e.g., GitHub.

The increased vulnerability surface acts as an entry point to large amount of malware, which in turn is the initial step in launching large scale attacks such as DDoS, phishing or spamming. As a result, the need for malware detection methods in cloud has become a necessity.

Most malware detection methods falls under two major techniques: static and dynamic analysis. In static analysis, the aim to examine the executable/binary file itself before it actually runs on

the system. Two main approaches are used for static analysis. First, analysis can directly be done on the binary file. One of the simplest forms of static analysis is extracting parts of the binary file as features (n-grams). Second, executable can be disassembled or reverse engineered (convert from binary to Assembly code) using dis-assemblers to get the actual code. Detection of malware takes place on the actual code using different techniques. Then different machine learning techniques can be applied for either of the aforementioned cases.

Although static analysis approaches are fast, cheap and, in fact, very effective, most sophisticated malware can evade these methods by embedding syntactic code errors that will confuse dis-assemblers but that will still function during actual execution. Polymorphic malware is able to change and evolve while preserving code semantics. Nearly every malware is currently using obfuscation, where binary and textual data is unreadable or hard to understand. Packing is an obfuscation technique for evading static analysis approaches, where a malware is modified using a run-time compression (or encryption) program.

Although, most sophisticated malware evade static analysis methods, their true malicious intentions remains the same. For example, a malware that intends to steal secret keys of a target system will continue to do so even if it is obfuscated. As a result, the need for behavior based detection methods is critical.

Dynamic analysis approaches can help overcoming some of the static analysis drawbacks since they rely on monitoring the behavior as opposed to static inspection. In dynamic analysis methods, a malware executes, usually, in a closed environment (virtual machine, sandbox or emulator) and its activities are monitored for few minutes. For infection prevention of the deliberately running malware, a new clean environment is created/restored for each malware sample.

Although, dynamic analysis methods shows significant potential, they can be evaded in many ways. First, delayed execution is a technique where a malware doesn't show its malicious activities for a long period of time which will deem the analysis process useless. Increasing the analysis time is impractical and also ineffective since the malware can always increase its sleeping time as well. Second, most malware tries to detect the presence of a sandbox or an emulator and,

once discovered, ceases its malicious activities. Third, most malware will not show any malicious activities if there is no internet connection, since it usually tries to connect to its command and control center.

Static and dynamic analysis are prevention mechanism in that they don't actually run the executable on the target system until they analyze and determine that it's benign. Given the mentioned drawbacks of static and dynamic analysis as well as the aforementioned new threats in cloud systems, it is a fact that malware will successfully execute on target VMs. The need for online malware detection has become a necessity for cloud infrastructures.

As opposed to static and dynamic analysis, where a suspicious executable needs to be analyzed, online malware detection is concerned about the whole system at all times. This overcome malware not showing its malicious activities, since once it starts its malicious activities, it should be detected and stopped. Fewer research has been done on online malware detection and even fewer has been done for cloud specifically. Most of prior work employs modifications to traditional malware detection techniques for a single VM.

In this dissertation, we are motivated by two facts.

- Due to the aforementioned drawbacks of static and dynamic analysis, prevention mechanisms in place can be evaded.

- Due to the aforementioned new cloud threats, malware has found new ways of getting into cloud VMs, bypassing any prevention mechanism.

Consequently, a fair assumption is that malware will always get into and execute in cloud infrastructures. To that end, due to the nature of cloud systems, we focus on online malware detection for cloud infrastructures. Unlike current research, we keep in mind the entire cloud context as opposed to a traditional physical stand-alone system. In addition, we leverage auto-scaling for online malware detection in cloud.

1.2 Problem and Thesis Statement

Malware is a critical threat to cloud and there will always be a way for the malware to infect cloud infrastructures. Online malware detection using machine learning has been applied to stand-alone systems; however, few research has specifically addressed the cloud. Consequentially, there is a lack of practical online malware detection techniques that are specifically tailored to cloud. Such techniques should rely on cloud unique characteristics and should thoroughly be investigated.

Cloud unique characteristic "auto-scaling" can effectively be utilized for online malware detection within a single-tenant's virtual resources, in black-box and white-box granularity using performance metrics.

1.2.1 Scope and Terminology

Organizations that utilize cloud services can be any company or person that is a cloud customer. They are also called cloud tenants. Online malware detection refers to behavior based techniques that provide ongoing real-time monitoring for VMs in the cloud.

The work in this dissertation focuses on detecting malware within a single-tenant's VMs. Although categorizing the type of malware is a very interesting problem, it is out of scope of this dissertation. Malware in the form of advanced persistent threats (APT) is also out of scope of this dissertation. This dissertation focuses only on data-driven online (real-time) malware detection using machine learning. Other malware detection techniques not using machine learning are out of scope.

1.2.2 Assumptions

In this dissertation we make the following assumptions:

- Cloud infrastructures will be increasingly subjected to new classes of attacks and, in turn, new ways of malware injection will be available. As a result, traditional static and dynamic based malware detection techniques will be insufficient to prevent the execution of malware. More generic online malware detection techniques, for malware that bypasses traditional in

place defense mechanisms (e.g. anti-viruses), are necessary.

- We make the assumption that VMs which are configured by the same configuration script (e.g., web-servers VMs are all spawned using the same script) should behave similarly without any significant deviations in their behaviors.

- Malware infection doesn't occur in all VMs at the same time. For example, VMs that belong to the same group/cluster (e.g., identically configured web-servers VMs) will not be infected at exactly the same time.

1.3 Key Contributions

We developed a cloud security monitoring framework for anomaly detection in cloud IaaS. We did so by leveraging the *auto-scaling* cloud characteristic and using a modified sequential K-means clustering algorithm to group similar VMs. We applied the algorithm on a defined set of black-box metrics (i.e. external collected metrics) that deal with the resource usage of VMs. The framework proved to be successful against Economic Denial of Sustainability (EDoS) attacks and malware that maintains high-profile behavior (i.e. resource-intensive activities such as ransomware) of resource utilization.

The aforementioned approach is not as effective for detecting malware that maintains low-profile behavior of resource utilization. As a result, we introduced and discussed an effective malware detection approach in cloud infrastructure using Convolutional Neural Networks (CNN), a deep learning approach. We initially employed a standard 2d CNN by training on metadata available for each of the processes in a virtual machine obtained by means of the hypervisor. We enhanced the CNN classifier accuracy by using a novel 3d CNN (where an input is a collection of samples over a time interval), which greatly helped reduce mislabelled samples during data collection and training. Our experiments are performed on data collected by running various malware (mostly Trojans and Rootkits) on VMs. The malware used in our experiments are randomly selected. This reduces the selection bias of known-to-be highly active malware for easy detection.

We demonstrated that our 2d CNN model reaches an accuracy of $\simeq 79\%$, and our 3d CNN model significantly improves the accuracy to $\simeq 90\%$.

Further, we extended and built upon the white-box approach to leverage the existence of multiple VMs in an auto-scaling scenario. First, we introduced Multiple VMs Single Samples (MVSS) method which is similar to the approach used in Chapter 4 but targets multiple VMs using single samples. MVSS achieved good results with an accuracy of $\simeq 90\%$. Then, inspired by the duplicate questions problem, we introduced Multiple VMs Paired Samples (MVPS) which targets multiple VMs using paired samples. MVPS takes the previous approach a step forward by pairing samples from multiple VMs which helps in finding correlations between the VMs. MVPS showed a substantial improvement over MVSS with an accuracy of $\simeq 96.9\%$.

1.4 Related Publication

Online malware detection using VMs black-box performance metrics based on the work in Chapter 3 was accepted for publication in *IEEE CLOUD 2017*.

- Mahmoud Abdelsalam, Ram Krishnan, Yufei Huang and Ravi Sandhu *"Clustering-Based IaaS Cloud Monitoring"*, In Proceedings 10th IEEE International Conference on Cloud Computing (CLOUD), Honolulu, Hawaii, June 25-30, 2017, 8 pages.

Overcoming the limitation of using VMs black-box performance metrics, a white-box (process-level) performance metrics approach using CNN based on the work in Chapter 4 was accepted for publication in *IEEE CLOUD 2018*.

- Mahmoud Abdelsalam, Ram Krishnan, and Ravi Sandhu *"Malware Detection in Cloud Infrastructures using Convolutional Neural Networks"*, In Proceedings 11th IEEE International Conference on Cloud Computing (CLOUD), San Francisco, CA, July 2-7, 2018, 8 pages.

1.5 Dissertation Organization

In this chapter, the motivation behind of this work, the problem statement, the objectives and key contributions are illustrated. The rest of the dissertation is organized as follows: Chapter 2 gives a brief background on the major cloud threats and machine learning techniques. It also talks about the current state of the art related work on malware detection using machine learning categorized in static and dynamic analysis. Then, it layouts the few research work on machine learning based malware detection that specifically target cloud. Chapter 3 gives a detailed description on the use of black-box features for anomaly detection in the cloud while leveraging auto-scaling characteristic. It targets highly active malware (e.g, ransomware). A developed framework and testbed are presented as well as testing and evaluation. Chapter 4 discusses the feasibility of using CNN and process-level performance metrics for online malware detection. It points out the general mislabeling problem and its solutions using 3d CNN. It targets malware that maintains a low-profile through-out its execution life. Chapter 5 extends the previous work by targeting multiple VMs as opposed to a single VM. It also introduces a new approach based on paired samples. Finally, Chapter 6 summarizes and concludes the dissertation and gives an overview of the potential future work.

CHAPTER 2: BACKGROUND AND RELATED WORK

2.1 Cloud Computing Security and Threats

Cloud computing is a broad term which includes the on-demand delivery of computing power, storage, networks, applications, and other IT resources through a cloud services platform with a pay-as-you-go pricing model. There are several definitions of cloud computing in the literature [30, 81]. In this dissertation we follow the standard definition provided by NIST [56]:

> "Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models."

Most cloud definitions, including NIST's, include system levels. Cloud security monitoring takes place at each of the following levels:

- Physical/Server: Includes computer hardware (e.g. physical machines, networks and other devices).
- Infrastructure as a Service (IaaS): Cloud infrastructure such as Virtual machines, networks, and storage.
- Platform as a Service (PaaS): Platform services such as run-time environments.
- Software as a Service (SaaS): Cloud applications over the Internet typically without the need of software installation or the clients.

Online malware detection on the physical level, which is is usually data centers made of clusters, can use the traditional distributed system techniques. However, malware detection on the other

virtualized layers needs more sophisticated frameworks since each layer has different characteristics. There are two visions [57] regarding cloud security: a) Client vision: where the client (tenant) can monitor only resources that belongs to him, without considering the whole cloud resources. b) Cloud service provider (CSP) vision: where the CSP can monitor the whole cloud system with outside access to clients' resources information. We need to consider such visions when dealing with cloud malware detection because of many reasons including privacy concerns of which data is allowed to be access by the CSP and which resources are available to be used by the tenant.

Cloud platforms have become very attractive to customers due to their essential characteristics (such as rapid elasticity, on-demand self-service, migration, controlled measured service and resource pooling). With its increased adoption by many organizations, cloud has become more attractive to attackers as well. For example, a large number of VMs are spawned daily on the cloud which can be used by attackers to create bot-nets. Understanding these threats and vulnerabilities is the first step toward better cloud security.

In [21,31,33,34,39,90], the authors discuss the vulnerabilities that exist due to cloud properties. One of the most prevalent threats to cloud is malware. Cloud malware injection [34] is a threat where an attacker injects a malware to manipulate the victim's VM. This applies to stand-alone systems as well as clouds. However, due to the nature of the cloud and automatic provisioning, a large number of the VMs are similarly configured. One example is when a web server scales-up due to increase in demand and scales-down when the demand goes down. This means that the attack that compromised one of the VMs is highly likely to compromise many of the other VMs. For example, an attacker can inject a bot-ware to use it for creating a bot-net due to the large number of VMs.

One of the characteristics of the cloud is the pay-as-you-go pricing model. This also introduced new threats. Economic Denial of Sustainability (EDoS) [75] is a threat where the attacker tries to manipulate the pricing model either by generating traffic not high enough to be detected as a Denial of Service (DoS) attack but high enough to cost the customer substantial financial loss or by somehow getting unauthorized access to the customer's management interface and creating many

VMs that remain dormant (to be undetectable) which also costs money.

VM co-residence is another threat due to the characteristic of multi-tenancy in clouds. VM co-residency means that multiple independent customers' VMs may be placed on the same physical server. This introduce certain issues such as cross-VM attack [67] which allows for shared memory attacks or side-channels attacks. VM image manipulation [32] is another threat where the attacker manipulates a victim's image which used to create new VMs.

Additionally, DoS/Flooding attacks seem to be applicable to normal systems as well as VMs in cloud. However a new class of DoS attacks exist because of the co-residency of VMs. These types of DoS attacks are called Indirect Denial of Service [39]. In this attack an attacker can launch a DoS attack on another VM that doesn't belong to the victim but is on the same physical server. Most probably, this leads to scaling-up and automatic provisioning of other VMs which consumes the resources on the physical server. As a side effect, the victim's VM (i.e. the co-resident VM) is affected by this DoS attack as well.

2.2 Machine Learning

Machine learning has been used as a behavioral-based anomaly detection technique to learn about the VMs behavior and in turn detect anomalies (abnormal behavior). Some techniques use labeled training data such as supervised learning whereas others assume no prior knowledge of the data such as unsupervised learning.

2.2.1 Supervised and Unsupervised Learning

Supervised learning: having an input and output variables x and y, respectively, an algorithm is used to learn the mapping function $y = f(x)$. The goal is to approximate the mapping function so that when a new input x is given, the algorithm predicts the output value y. In simple words, the process takes training data samples, creates a generalization, and assumes that any future data will follow that generalization. Supervised learning is sometimes referred to as inductive learning because it goes from specific examples to more generalized rules. Supervised learning is mainly

classified into two categories of problems: *regression* and *classification*. Regression problems is when the output variable is a real value, such as "weight", while classification problems is when the output variable is a category such as "color".

Unsupervised learning: having an input data x without any corresponding output and the goal is to derive the underlying structure in the data in order to learn more about the data. Unlike supervised learning algorithms, unsupervised learning algorithms are supposed to discover interesting structure of the data on their own without a teacher to correct them. There are mainly two main categories of problems in unsupervised learning: *clustering* and *associative*. Clustering problems is when the goal is to group data samples with similar features together. Associative problems is when the goal is to discover rules that describe large portion of the data. For example, most customers that tend to buy X also buy Y.

2.2.2 Machine Learning Models and Techniques

Artificial Neural Networks (ANNs) are computational model which is based on large number of interconnected artificial neurons [36]. They are analogous to a human brain. The input data activates neurons in the first layer which in turn produces output as input data of the second layer. Each layer passes its output as an input to the next layer. Middle layers are called hidden layer. In the case of classification problem, the final output is a label to the classified category. Self Organizing Map (SOM) [44] is an unsupervised model that is very commonly applied to anomaly detection. Although it falls in the category of ANNs, it is also considered to be a clustering technique, which produce lower dimensional (typically two-dimensional) representations of multi-dimensional data, which is called a map.

A **Bayesian network** is a probabilistic graphical model that represents a set of random variables and their relationships (conditional dependencies) by a directed acyclic graph (DAG) [60]. Nodes maintain the states of random variables and the conditional probability form (if applicable). Edges represent conditional dependencies. Nodes that are not connected to each other represent variables that are conditionally independent of each other. Each node has probability function that takes

a set of values for the node's parent variables as input, and gives the probability of the variable represented by the node.

Clustering is an unsupervised technique to group similar data samples into clusters. There are several techniques [11, 51] for clustering the input data. K-means is a centroid based clustering technique where data points are grouped based on their distance to each other. Each cluster is represented by its centroid which is a mean vector that includes all dimensions. Density-based spatial clustering of applications with noise (DBSCAN) [29] is a density based clustering models group the points that are closely packed together (points with many nearby neighbors), while marking (as outliers) points that lie in low-density regions. K-nearest neighbors (K-NN) is an instance-based learning (sometimes referred to as lazy learning) algorithm. It is used in classification problems where the classification of points is determined by the k nearest neighbors of that data point.

Decision tree is a tree-list structure that builds classification or regression models. The leaves represents the classification while the branches represents the features that lead to those classifications. The final result is a tree with decision nodes and leaf nodes. The most commonly and widely used decision trees algorithms are ID3 [65], C4.5 [66, 68] and CART [69].

In simple words, a **Support Vector Machine (SVM)** [13, 18, 78] is a classification technique where given labeled training data (supervised learning), the algorithm outputs an optimal hyperplane which categorizes new examples. SVM is a classifier formally defined by a separating hyperplane of the feature space into two classes in which the distance between the hyperplane and closest data points of each class is as far as possible. When data are not labeled (i.e. supervised learning is not possible), an unsupervised learning approach can be used. Support vector clustering [9] attempts to find natural clustering of the data to groups and map new data to these groups.

Detecting anomalies doesn't necessary means detecting malicious activities. Anomalies can be divided into two categorizes: benign and malicious. Benign anomalies usually deal with system failures (e.g. unresponsive server) while malicious anomalies deal with attacks (e.g. malware). Unsupervised techniques is more suitable and more widely used for anomaly detection, since labeling thousands, or even millions, of records is a hard laborious task and mislabeling can occur.

SVMs are supervised learning algorithms which have been used increasingly for anomaly detection. One of the primary benefits of SVMs is that they learn very effectively from high dimensional data [12] and they are trained very quickly compared with Multi Layer Perceptrons (MLPs) [59, 77]. Most SVM algorithms are binary classifiers (e.g., normal and anomalous data). However, there are other SVM algorithms, which have been proposed, support multi-class learning [19,26]. Furthermore, one-class SVM was proposed by [71] and has been used in many papers.

Robust support vector machines (RVSMs) [76], a variation of SVM, was used in [37] for anomaly detection. The study used the 1998 DARPA BSM data set collected at MIT's Lincoln Labs. It showed a good classification performance in the presence of noise with 75% accuracy and no false alarms, and 100% accuracy with a 3% false alarms. [74] developed a framework for intrusion detection in network traffic. A combination of SOM, Genetic Algorithms (GAs), and SVM was used. SOM was used for packet profiling, GAs was used for features selection, and data were classified using the enhanced SVM.

One of the most used techniques, which falls under the umbrella of anomaly detection, is Intrusion Detection Systems (IDS). The work in [50] gave a comprehensive overview of the approaches for IDS. Mainly, it is categorized into two approaches: signature-based and anomaly-based. Signature-based approach is an effective method for known attacks, however it is not effective to detect unknown attacks or even a variant of a known attack since it depends on a signature of known attacks. Behavioral-based anomaly detection approaches are effective against new attacks; However, they suffer from accuracy and false alarms rate. In Chapter 3, a behavioral-based anomaly detection approach for online malware detection is used for VMs that belongs to a single tenant.

The work in [17] gives a summary of the different techniques for anomaly detection. In this work, a framework for choosing the right anomaly detection technique is presented. The key components include the research area, nature of data, data labels, anomaly type, application domain and output. These components must be determined carefully based on the situation. Although more than one anomaly detection technique can be suitable for a single situation, each technique

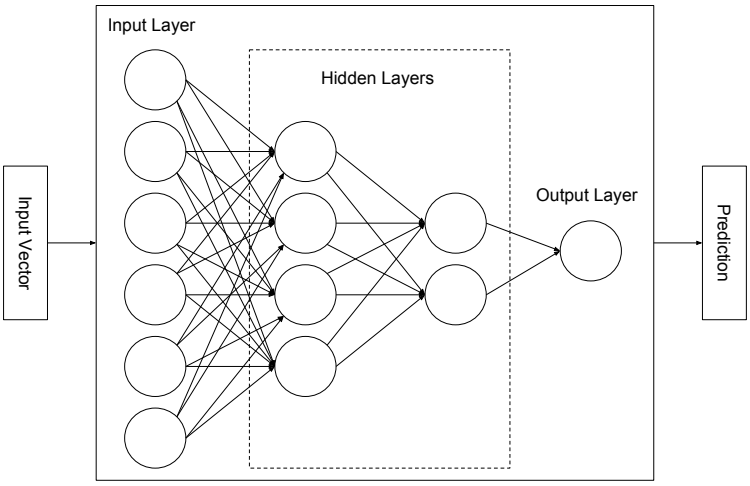


Figure 2.1: Abstract of a feed forward deep neural network architecture

has its own drawbacks. The work in [17] also categorizes the anomaly detection techniques into classification based, clustering based, nearest neighbor based and statistical based. Statistical techniques [85, 86] for anomaly detection are being used, but suffer performance overhead due to their complexity, lack of scalability and the need of prior knowledge.

Figure 2.1 depicts a traditional **deep neural network (DNN)** architecture. The leftmost layer of a DNN is called the input layer, and the rightmost layer the output layer (which has only one node in this figure). The middle layers of nodes are called the hidden layers.

Hidden Unit is the most basic component in DNNs. It is a computational unit that takes an input feature vector $x = [x_1, x_2, ..., x_n]$ and outputs $y = f(\sum_{i=1}^{n}(W_i x_i + b))$, where b is the bias, W_i is the weights matrix, and $f : \mathbb{R} \rightarrow \mathbb{R}$ is referred to as the activation function. The activation function, usually denoted by $f(.)$, determines the unit's output and introduces non-linearity in the neural network. Without a non-linear activation function, no matter how many hidden layers the network has, the network will behave like a single-layer perceptron, because, at the end, summing these layers gives another linear function.

Hidden Layer is the highest-level component in DNNs. Each layer consists of many of the aforementioned hidden units. Typically, a DNN consists of n layers where layer 1 is the input layer and layer n is the output layer. A hidden layer usually receives the previous hidden layer's output vector as its input and outputs a new feature vector $y_l = (W_l y_{l-1} + b_l)$ as input for the next

layer, where y_l is the output feature vector and W_l, b_l are the weights matrix and the bias for layer l. The advantage of having multiple layers is that they add levels of abstraction that cannot be as simply contained within a single layer.

Input/Output Layers are the left most and right most layers of a DNN, respectively. Mostly, DNNs are used for classification problems, so the input layer takes an input vector which represents the object to be classified and is passed to the hidden layers for processing. Depending on the problem, the output layer deals with the transformed vectors sent by the last of the hidden layers. In case of classification, the output layer transforms the output (received from the last hidden layer) to a probability distribution representing the estimated probabilities that an object belongs to each class.

2.3 Machine Learning Based Malware Detection

This section provides an abbreviated introduction to the major machine learning based malware detection techniques. Although we only focus on malware detection in cloud infrastructures, we review general malware detection approaches using machine learning because approaches that work for stand-alone systems can as well be applied to cloud systems. Figure 2.2 gives a broad overview of the machine learning based malware detection techniques and the extracted features. Generally, malware detection techniques falls under one of the two categories: online malware detection and file classification.

In file classification approaches, an executable binary file is given and the task is to classify whether it is a malware or not by running it (usually in an isolated environment) and observing its behavior. These approaches are used for malware prevention. Once an executable is classified as benign, then it actually run on the operating system without further monitoring. The majority of malware file classification techniques are further categorized under one of the two approaches: static analysis (where an executable is analyzed without actually running it) or dynamic analysis (where an executable run and its behavior is analyzed). Mainly, static analysis techniques make use of three major categories of features: Binary N-grams, Control Flow Graphs (CFGs) and Static

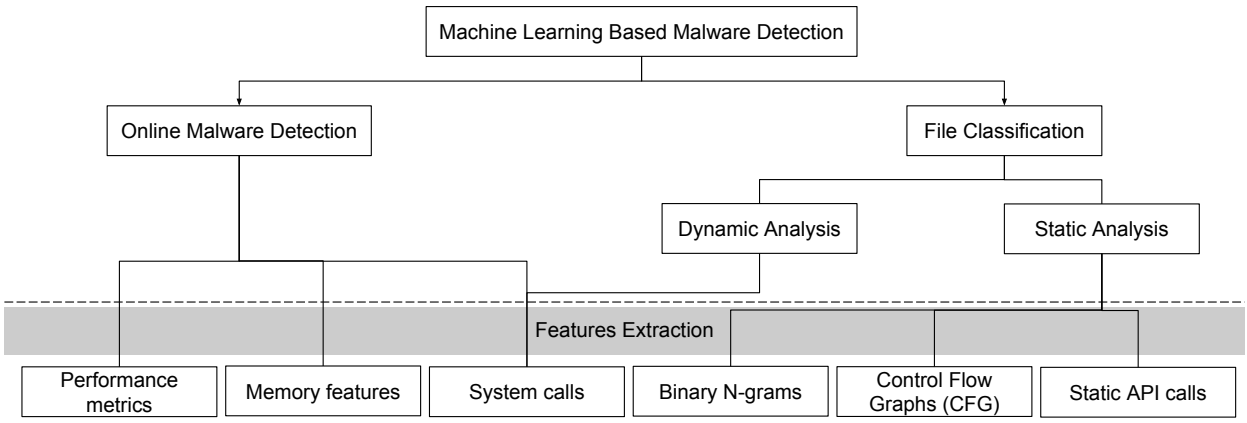


Figure 2.2: Categorization of machine learning based malware detection methods and used features

features/Disassembling, while dynamic analysis techniques make use of system/API calls features.

On the other hand, in online detection approaches, the whole system is under continuous online monitoring for the presence of malware. They focus on more dynamic and time series features such as performance metrics, memory features or system/API calls. Such approaches are more expensive in terms of performance; however, they mitigate drawbacks such as detecting malware in an already checked benign application that got infected later on.

2.3.1 Malware File Classification

Static Analysis

During static analysis, no execution of executables/binary files takes place. It is the process of analyzing executables by examining their code without actually executing them. There are two approaches used for static analysis. First, an executable file can be disassembled or reverse engineered using disassemblers to get the actual code. Then detection of malware takes place on the actual code. Most sophisticated malware can evade this method by embedding syntactic code errors that will confuse disassemblers but that will still function during actual execution. Second, analysis can be done directly on a binary file format. For example, one of the simplest forms of static analysis, is extracting parts of the binary file as features (n-grams). Then ML techniques are

used to find malicious patterns.

In the work done by Tahan et al. [79], the approach is to remove n-grams that are known to be benign. For example, a worm that distributes itself via emails contains code to send an email which is benign in many applications, so removing these segments from the file, while comparing what is left to known malicious segments is a valid approach. The paper used different ML techniques including ANN and Decision Trees. The works in [2, 45, 73] are similar but use different ML algorithms, where the slight differences mostly lie in how to filter out common n-grams.

Saxe and Berlin [70] use four types of features extracted (and constructed) from executable binary files: contextual byte features, portable executable (PE) import features, string 2d histogram features and PE metadata features. These features are fed to a four layers deep neural network and then a score calibration model is used to determine the probability of an executable being malware. Similarly, [72] uses deep learning approach for malware static analysis.

Other research [27, 28, 40] disassemble the executables and extract features based on control flow graphs or API/function calls names. The extracted features are used as input to different ML techniques.

Most sophisticated malware has polymorphic nature. It constantly changes and evolves to evade anti-virus software. Evolution of the malware can be done in different ways such as compression and encryption (code obfuscation techniques). Such sophisticated malware obstructs the effectiveness of static approaches. Although, due to such obfuscation techniques, malware changes, the essential function usually remains the same. For example, a ransomware that intends to encrypt a victim's machine will continue to perform such function even though its signature changes. Dynamic analysis approaches can help overcome some of the static analysis drawbacks since they rely on monitoring the behavior as opposed to static inspection.

Dynamic Analysis

In dynamic analysis, the executable is executed, typically, in an isolated environment (e.g., sandbox or VM) and information is gathered during execution (e.g., system calls, memory accesses or

network communications).

Dahl et al. [22] use dynamic analysis for malware files classification by extracting API calls features. Malware run in a lightweight VM and hundreds of thousands of features are extracted to be used in a deep learning technique. Random projection is used to further reduce the dimensionality of the input space. This approach is for files classification which is not naturally suitable for online malware detection.

Huang and Stokes [38] extend Dahl's work by using multi-task learning (where a set of network layers is shared between learning tasks) using Deep Neural Networks (DNN) for malware detection and malware family classification of binary files. They focused on comparing and showing the improvement of using deep learning techniques as opposed to shallow neural networks. Data are extracted from dynamic analysis done by a light weight anti-malware engine. The collected data used as features are divided into two: a sequence of API call events and their parameters and a sequence of null-terminated objects extracted from system memory.

Athiwaratkun et al. [6] use a light weight emulator (usually used by anti-viruses) to capture and log system calls done by an executable before actually running an executable on a Windows operating system. Every logged system call is mapped to one of 114 defined high level system calls, which, in turn, are enumerated and translated to integer sequences (from 0 to 113). Several DL models including LSTM and GRU based language models which works on the sequences of system calls. In addition, they proposed a character-level CNN model, where each character represents an event.

Similarly, Kirat et al. [43] use system calls features in their work. They target evasive malware (which completely changes behavior based on the underlying environment). Their system work by automatically extracting evasive signatures of a malware by using bioinformatics-inspired algorithms (data mining) and leveraging data flow analysis techniques. System calls sequences are collected by running the malware in different environments then system calls alignment takes place. Then inverse document frequency (IDF is a measure of whether a term is common or rare across all documents, usually used in information retrieval) is used to filter out common execution

events.

Agrawal et al. [4] uses API calls made by a portable executable (PE) file. Malware files are executed in an isolated environment with no internet access and event sequences are logged as event sequences. End-to-end learning models are used based on Long Short Term Memory (LSTM). To handle extreme long sequences, Convoluted Partitioning of Long Sequences is used where the entire input sequence is split into chunks and each is processed by CNNs in a recurrent way. LSTM is used to capture the sequential data while CNN is used to extract significant event occurrences within the entire large sequences.

2.3.2 Online Malware Detection

System calls features are used for online malware detection. Research based on system calls is proposed in [23, 25, 53], while other research [5, 64] focuses on using API calls features.

The work by Tobiyama et al. [80] applies deep learning for malware detection using process API calls log information. First, LSTM is used to extract features and then CNN is given these features as input. The downside of this work is using a sandbox to monitor processes. In most cases, malware will detect the presence of a sandbox and hide its true behavior. Also, this deals with single data sample without considering that malware can be benign at certain times and malicious at others.

Some research [61, 91] focuses on using memory features for online malware detection. Xu et al. [91] propose a hardware assisted malware detection approach based on virtual memory access patterns caused by the malware. Their assumption is that, in order for the malware to work, it needs to modify/change control flow and/or data structures which, in turn, leaves finger prints in memory. The features extracted are represented in histograms of memory accesses and are used as an input to different ML techniques. This work focused on the detection of rootkits and memory corruption malware. One model is trained for each application which can be quite expensive.

Performance metrics are used for malware detection. A valid assumption is that benign programs have multiple common patterns of resource utilization that keeps repeating. Another, as-

sumption is that malware that belongs to the same family, regardless of code variations in most cases, will perform similar malicious tasks which will impact the system's performance.

Demme et al. [24] examine the feasibility of using performance counters for malware detection. Many precautions are taken to avoid mislabeling of the data collected. These includes: contamination, where malware infect subsequent runs of the data collection experiments and network connectivity, where malware stop showing malicious activities if there is no internet connection. It's actually hard to say when a malware truly shows it's malicious activities, so labeling all data after malware injection in an experiment as malicious is, in fact, inaccurate. There is no possible feasible solution (except human experts) to correctly label all data samples, so data pollution is present in most online malware detection experiments. The study use the performance counters as input to ML techniques, including K-Nearest Neighbors (KNN) and Decision Trees.

2.4 Cloud Malware Detection

Few research has addressed the problem of cloud IaaS malware detection since many of the stand-alone approaches works for cloud single VMs as well. Most, if not all, of the cloud-specific malware detection techniques falls under the online malware detection category (which includes anomaly detection approaches). Furthermore, they all focus on extracting features from the hypervisor since it adds another security layer.

Dawson et al. [23] focus on rootkits and intercept system calls through the hypervisor to be used as features. Their system call analysis is based on a non linear phase-space algorithm to detect anomalous system behavior. Evaluation is based on the dissimilarity among phase-space graphs over time.

Wang [84] introduced Entropy based Anomaly Testing (EbAT) an online analysis system of multiple system-level metrics (e.g. CPU utilization and memory utilization) for anomaly detection. The proposed system used a light-weight analysis approach and showed a good potential in detection accuracy and monitoring scalability. However, the evaluation used didn't show pragmatic and realistic cloud scenarios.

Azmandian et al. [7] propose an anomaly detection approach where all features are extracted directly from the hypervisor. Various performance metrics are collected per process (e.g., disk i/o, network i/o) and unsupervised machine learning techniques like K-NN and Local Outlier Factor (LOF) are used.

Classification of VMs is used for anomaly detection. Pannu et al. [62] propose an adaptive anomaly detection system for cloud IaaS. They focus mainly on various faults within the cloud infrastructure. Although this work is not directly addressing malware, such technique is valid for malware detection since malware can cause faults in VMs, thus worth mentioning. It used a realistic testbed experimentation comprising 362-node cloud in a university campus. The results showed a good potential with over 87% of anomaly detection sensitivity. One of the drawbacks of this work lies within using two-class SVM. It suffered from data imbalance problem which led to several false classification of new anomalies.

The work by Watson et al. [87] is similar to [62] but directly addresses detecting malicious behavior in the cloud. It tried to overcome the drawbacks in [62] by using one class Support Vector Machine (SVM) for detection of malware in cloud infrastructure. The approach gathers features at the system and network levels. The system level features are gathered per process which includes (memory usage, memory usage peak, number of threads and number of handles). The network level features are gathered using CAIDA's CoralReef[1] tool. The study shows high accuracy results. However, gathering features per process is a very exhausting and intrusive operation. Having thousands of VMs with hundreds of processes running can have a significant performance degradation. Furthermore, the study uses known-to-be highly active malware that easily skew the system's resource utilization (e.g., by forking many processes).

[1]CoralReef Suite: https://www.caida.org/tools/ measurement/coralreef/

CHAPTER 3: BLACK-BOX ONLINE MALWARE DETECTION IN CLOUD IAAS

3.1 Introduction

Cloud systems are becoming more complex due to the large number of services, resources and customers (tenants) involved. Cloud tenants' resource usage ranges from few VMs to hundreds or even thousands of VMs. For example Hadoop systems are sometimes deployed over thousands of VMs. Hence monitoring VMs for security has become a critical task for not only the cloud providers but also their tenants. Many security aspects need to be monitored within the cloud. For example, if services do not work properly, it may affect the Service Level Agreement (SLA) fulfillment as well as the tenants' security. Tenants can be malicious and affect other tenants' systems via co-resident attacks. Users accessing tenant hosted resources (e.g., web application hosted on a VM) can be malicious. All of these scenarios need to be systematically monitored. In this chapter, first, we provide a systematic way of presenting the monitoring points in the cloud. Then we focus our scope to provide a way of monitoring cloud infrastructure as a service (IaaS) using clustering of VMs based on resource usage and interaction. Most anomaly detection approaches are based on a single/multiple VM(s) without leveraging the cloud properties as a whole system. As a matter of fact, the behavior that is considered anomalous for one VM might not be anomalous for another VM. In practice, a cloud system has many tenants and each tenant has many VMs. Typically, those VMs are not randomly create, instead they are systematically created (e.g., scalability policy) with each group of VMs doing a specific job. In this chapter, we consider the holistic view of the cloud rather than single/multiple VMs. This chapter makes the following contributions:

- We systematically identify the monitoring points in cloud IaaS.
- We propose an approach for single-tenant's VMs clustering to detect anomalous behavior in scale-up and scale-down scenarios. First, we identify the most important VM features to be used in clustering. Based on this, we use a modified sequential K-means (variation of

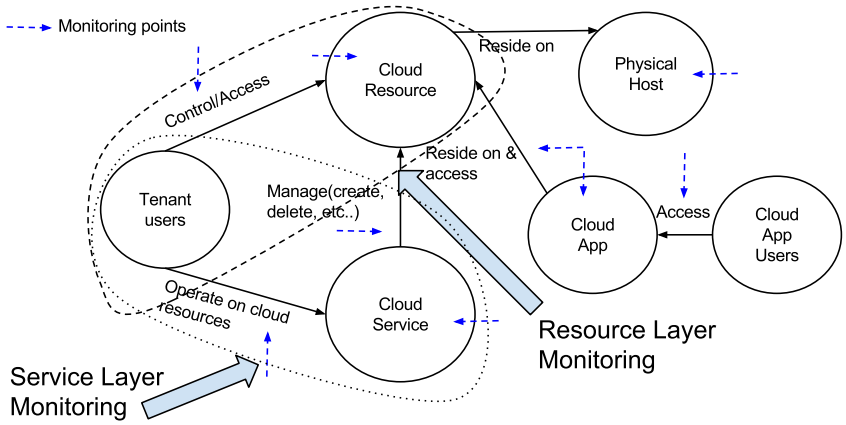


Figure 3.1: Cloud Monitoring Points

K-means [54]) clustering algorithm.

To the best of our knowledge, this is the first clustering approach to consider profiling VMs with respect to each other in order to detect anomalous behavior.

The rest of the chapter is organized as follows. Section 3.2 gives an overview of cloud monitoring and explains the different cloud monitoring points. Section 3.3 describes the clustering techniques and its usage for the work in this chapter. Section 3.4 provides an overview of the proposed framework and explains the methodology for implementing the framework. Section 3.5 explains the OpenStack-based testbed setup. Section 3.6 discusses the experiments we performed on an OpenStack-based testbed and their results. Finally, Section 3.7 concludes our findings and gives some directions for future work.

3.2 Cloud Monitoring Overview

Figure 3.1 illustrates the various monitoring points in cloud IaaS scenario. The figure represents the interaction between different entities in a typical cloud environment. Wherever there is interaction, there is a security risk since one or both of the two ends on the interaction can be malicious. Typically, cloud customers (tenants) interact with the cloud services in order to create new resources or manage them, or interact with their resource directly for internal configurations (e.g., ssh into a VM). Cloud tenants host web applications which can be accessed by cloud application

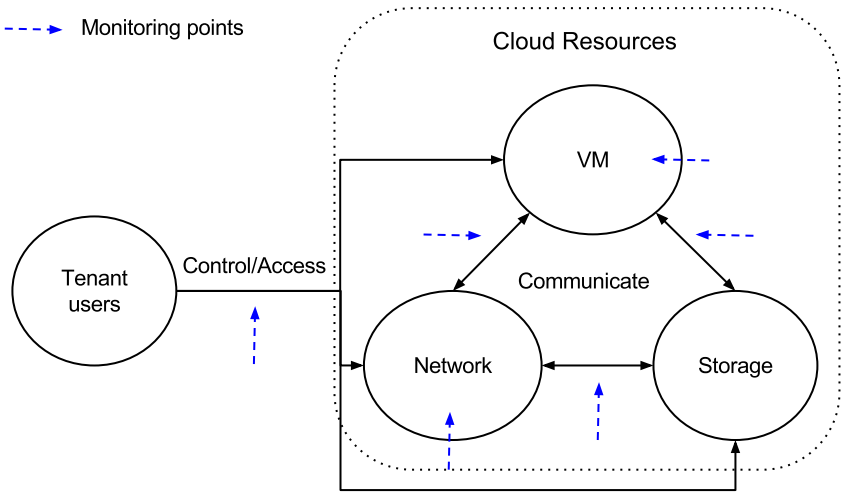


Figure 3.2: Resource Layer Monitoring

users. These cloud applications are hosted on the cloud resources (VMs) and can interact with other resources. Therefore, if a cloud application user successfully hacked or injected malicious software into a cloud web application, the interaction between the application and the resources could be malicious. The fact that many VMs are placed on the same physical host also has security risks, for example co-resident attacks [67]. We categorize cloud security monitoring as follows.

1. Resource layer monitoring which is concerned about the monitoring of resources created by the cloud services. For example, VM monitoring for malicious behavior or network resources traffic monitoring.

2. Service layer monitoring which is concerned about the monitoring of cloud services such as usage patterns by tenants.

Figure 3.2 shows a closer look at the resource layer monitoring. It shows, in a simple way, the cloud resources and their interactions. In this chapter, we focus on monitoring VMs at the resource-layer. The techniques developed are applicable to other resources in cloud such as virtual routers and storage.

Figure 3.3 shows a closer look at the service layer monitoring. Although it is not the focus of this chapter, it is essential to shed some light over it. Pattern recognition can be used to find anomalous pattern within the usage of cloud service by tenants. The services communicate internally. For

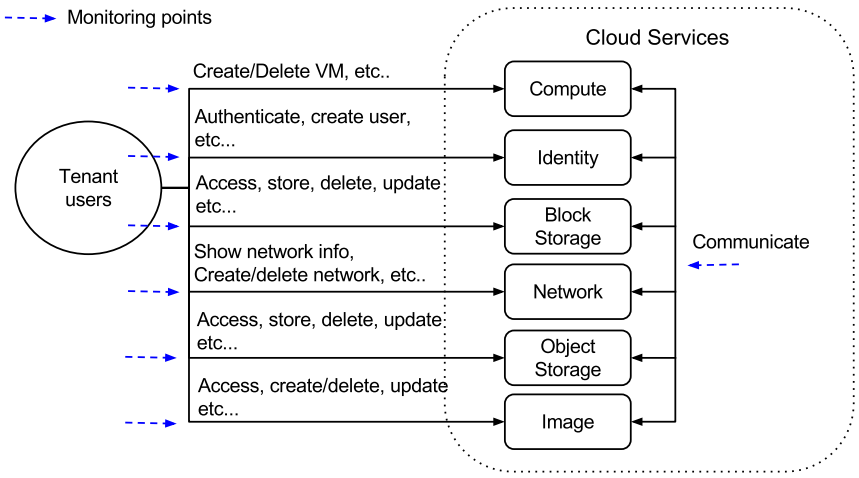


Figure 3.3: Service Layer Monitoring

example, a tenant can create a VM by sending a command to the compute service. However, the compute service needs to communicate with the network service to allocate IP. It also needs to communicate with the image service to create the VM among many other communications. In order to make sure the cloud services are working properly, a monitoring service can check if the steps of a request are always the same. This is left to future work for additional investigation.

3.3 Clustering

Clustering is a technique to group similar data samples into clusters. One of the main assumptions, which is essential in using clustering for anomaly detection, is that the number of normal data samples is far greater than the number of anomalies. We believe that this is true in the security domain since having anomalies is not the usual case in any system. We refer to anomalies as any abnormal behavior such as, for example, malicious behavior (due to system breach or malware) or system failure. Clustering techniques are not as effective if anomalies create a cluster by themselves. False anomalies remain a big challenge to clustering techniques. We discuss about reducing the number of false anomalies in section 3.4.5.

3.3.1 K-means

K-means clustering algorithm is one of the most popular clustering algorithms due to its performance and simplicity. It groups data samples based on their feature values into k clusters. Data samples that belongs to the same cluster have similar feature values. Knowing the best k value remains a challenge, although there are some proposed approaches [63]. Our framework is intended to be practical; hence to be used by cloud customers (tenants) in real scenarios. We assume that the tenants at the least know what type of VMs they are having. For example, a tenant is hosting a web application on the cloud and is using a three-tier web architecture (web servers, application servers, and database servers). Therefore, there are three clusters, so $k = 3$ is an input to the monitoring system. Here are the steps of K-means clustering:

1. Set the number of clusters (k).
2. Initialize k centroids/means (by guessing their initial values or randomly choosing them).
3. For each data sample compute the distance to all centroids and assign it to the closest centroid.
4. Modify the centroids based on the new data sample.
5. Go to step 3 until each of the centroid values do not change.

The Euclidean distance is used to compute the distance between a data sample and the centroids. It is defined as:

$$dis(x, c) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

where x and c are vectors of quantitative features of the data sample and the centroid respectively.

3.3.2 Sequential K-means

Since our framework is meant to be as practical as possible, there are a few assumptions that need to be addressed:

- The data is time series, meaning that we have one data sample at a time.
- A training phase is infeasible. Each cloud customer needs to have a training phase (each scenario has completely different data) and this can be impractical.

Since we are dealing with time series data, sequential K-means is used. It is a variation of the original K-means clustering algorithm. In sequential K-means, the data are infinite and comes one sample at a time which is convenient for our case. Another difference between K-means and Sequential K-means is that K-means iterate over the same data samples many times until the centroid values doesn't change anymore while Sequential K-means doesn't. Listing 3.1 shows the pseudo code of the sequential K-means algorithm.

```
make initial guesses for means (centroids) m_1, m_2, ..., m_k
set the counters n_1, n_2, ..., n_k to zero
until interrupted
  get the next sample x
  if m_i is closest to x
    increment ni
    replace m_i with m_i + (1/n_i) * (x - m_i)
  end_if
end_until
```

Listing 3.1: Sequential K-means

3.4 Framework Overview

This section provides an overview of the proposed framework as well as a description of the methodology to detect anomalies using modified sequential K-means. First, we define the fea-

Table 3.1: Black-box Virtual machines features/metrics

Metric	Description	Unit
CPU util	Average CPU utilization	%
Memory usage	Volume of RAM used by the VM from the amount of its allocated memory	MB
Memory resident	Volume of RAM used by the VM on the physical machine	MB
Disk read requests	Rate of disk read requests	rate/s
Disk write requests	Rate of disk write requests	rate/s
Disk read bytes	Rate of disk read bytes	rate/s
Disk write bytes	Rate of disk write bytes	rate/s
Network outgoing bytes	Rate of network outgoing bytes	rate/s
Network incoming bytes	Rate of network incoming bytes	rate/s

tures of the VMs. These features will be collected and used for clustering. Then, the features are normalized since they are not of the same scale. Normalization is done based on the Min-Max approach. Lastly, a real-time clustering (modified sequential K-means) is applied and anomalies are detected based on the specified threshold.

3.4.1 Features Definition

VMs features are usually divided in two categories: *inside* and *outside*. For the sake of practicality, we assume no prior knowledge of any information inside the VMs. Our framework deals with the VMs as a blackbox. Table 3.1 shows the outside features selected to be collected for every monitored virtual machine. (Note that the features used in this work are just a selection for illustration purposes–in practice, a lot more features are available.)

3.4.2 Features Normalization

Clustering algorithms can be very sensitive to data scales (more weight goes to features with higher values). Since data samples are not of the same scale, thus data normalization is needed. We used a simple data normalization technique called Min-Max. Min-Max normalization is a technique where you can fit the data with a pre-defined boundary. Min-Max normalization is simply defined

as:

$$A' = \frac{A - minValue_A}{maxValue_A - A}$$

Pre-defining the $maxValue_A$ can be tricky for time series data. Thus, we employ a Min-Max normalization based on a fixed-size sliding window.

3.4.3 Modified Sequential K-means

As stated in section 3.3, one challenge to using clustering is the number of false positives. However, to reduce the rate of false positives, we add stabilizing time parameter. *Stabilizing time* is an input parameter which represents the time to wait until each newly created VM is booted up and configured. For example, if a new VM is created by the scaling policy, the cloud system may need to tie it to a load-balancer, boot-up the operating system, install a web-server and do other configurations. The framework shouldn't monitor the web-server until it is completely up and running as intended.

Each data sample that belongs to a VM will be clustered according to the clustering algorithm used. Therefore, a data sample from a particular VM can belong to a cluster x at one time while another data sample from the same VM can belong to cluster y at another time. The clustering algorithm will not report this as an anomaly. For example, a data sample from a web-server VM should not belong to a DB servers cluster. This should be reported as an anomaly. Since, we assume no prior information about each VM function (because many VMs can be created automatically by some policy, e.g., scaling) and cannot decide if a data sample really belongs to a particular cluster or not, this presents a problem. We overcome this problem by slightly modifying the clustering algorithm by adding a new parameter *assigning time*. This parameter represents the time needed for the monitoring system to make sure that a VM belongs to a certain cluster. Once this is done, all the data samples to come for a particular VM will be compared to its assigned cluster. For example, let's assume there are three clusters (web servers, application servers and database servers). VM x is newly created. For the first m minutes, x's data samples is compared and counted to all

the three clusters. The cluster with the maximum number of data samples is the cluster that is assigned to VM x. After that, x's data samples are compared only to its assigned cluster to check for anomalies as well as updating the cluster's information. Listing 3.2 shows pseudo code of the modified sequential K-means.

```
make initial guesses for means (centroids) m_1, m_2, ..., m_k
set the counters n_1, n_2, ..., n_k to zero
counter j //Number of current VMs
set assigningTime[1..j] to z minutes
def VMClusters[1..j] //VM i belongs to cluster[1..k]
set VMClusterCounts[1..j][1..k] to zero
until interrupted
  get the next sample x
  get VM v //x sample belongs to VM v
  if m_i is closest to x
    increment n_i
    replace m_i with m_i + (1/n_i) * (x - m_i)
  end_if
  if x is not assigned to any cluster
    //If time end assign it to a cluster
    if assigningTime[v] <= 0
      set VMClusters[v] to index of max(VMClusterCounts[v])
    else
      if m_i is closest to x
        increment VMClusterCounts[v][i]
      end_if
      decrease assigningTime[v]
    end_if
  end_if
end_until
```

Listing 3.2: Modified Sequential K-means

It is worth mentioning that one of the most critical aspects of cloud monitoring is the complexity of the monitoring system in place. The modified sequential k-means still iterates once over any data sample, which results in a linear complexity.

3.4.4 Anomaly Detection

A sample is considered an anomaly if it's far from its centroid by a threshold. There are two parameters that need to be set up by the cloud customer. *Anomaly threshold* ($x\%$) and *Anomaly number* (y/min). Anomalies are detected based on x. If a particular sample is off by $x\%$ from its assigned centroid, then it is marked as an anomaly. Once, there are y anomalies per minute, an alarm will be raised. This is done to reduce the number of false alarms due to behavior fluctuations.

3.4.5 Parameters tuning

The framework has some important parameters that need to be set up as accurate as possible since they affect the clustering algorithm as well as the anomaly detection. These parameters are given by the cloud customer because they differ based on each scenario.

Clustering parameters: *stabilizing time*(s). This parameter is very dependent on the scenario because it is affected by the VM assigned resources, the operating system boot time and the inside configurations. This parameter can be easily set by having the VM signal when the booting and configuration are done.

Anomaly detection parameter: *anomaly threshold*(t). While increasing the threshold reduces the number of false alarms, it also increases the chance of not detecting real anomalies. On the other hand, while decreasing the threshold increases the number of false alarms, it also decreases chance of anomalies getting through undetected. Thus, tuning these parameters are very important.

Normalization parameter: *window size*(w). This parameter represents the window size in which the sliding-window Min-Max normalization should keep in record. On one hand, keeping many

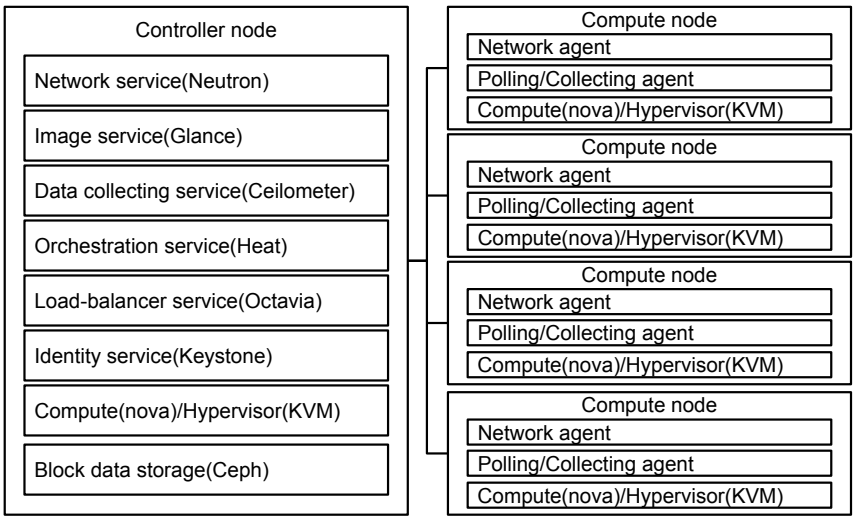


Figure 3.4: Testbed Setup

history samples might affect the new data samples that started shifting towards different values. On the other hand, keeping few history samples will distort the data samples during the normalization stage.

Automatic parameter tuning is a real challenge that we plan to investigate further in the future work.

3.5 Experiments Setup

3.5.1 Testbed Environment

The cloud testbed used in this work is OpenStack [1] which is a major cloud orchestration software used by many cloud providers. Figure 3.4 shows the setup of the cloud testbed. The testbed composed of five nodes. One controller node is responsible for services such as the dashboard, storage, network, identity, and compute. Four compute nodes are responsible just for the compute service. The compute nodes also contains network agents as well as samples collecting agents.

[1] Openstack website. https://www.openstack.org/

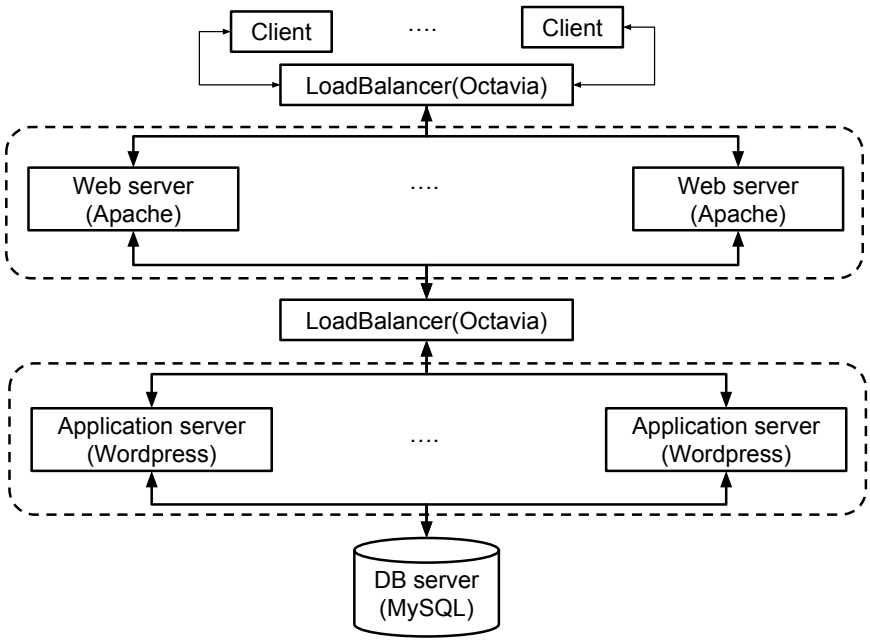


Figure 3.5: 3-tier web application

3.5.2 Use Case Application

In order to simulate a real environment as much as possible, a three-tier web application, a common cloud architecture according to AWS[2], is built as a use case. A three-tier web application is an application program that is organized into three major parts, where each tier can be hosted on one or more different places. In our case, these places are the cloud nodes hosting the compute services.

Figure 3.5 shows the three-tier web application built on top of our testbed. The application used for this work is Wordpress[3], a major open-source content management system (CMS) based on PHP and MySQL. In a typical three-tier web application, a web server hosts the static pages, an application server hosts the application logic, and a database server store the data. The work flow typically is:

1. Web server receives a request from a client.

2. Web server replies back if it is a static page request that doesn't need computation in the application logic.

[2]Amazon architecture references. https://aws.amazon.com/architecture/
[3]Wordpress website. https://wordpress.org/

3. If not 2, it sends the request to an application server.

4. Application server receives the request.

5. Application server accesses the database server if it needs stored data and replies back to the web server.

6. Web server replies back to the client.

Separating the application into three tiers allows for the concurrent development and configuration of the three different tiers. It also allows for the scaling of the three-tiers separately. In our case, scaling up and down is enabled for the web and application servers but not the database server. The 3-tier web application used for this work utilizes two load balancers. A web server load balancer which is responsible for distributing the requests on all the web servers and an application server load balancer which is responsible for distributing the requests on all the application servers.

3.5.3 Traffic Generation

Most literature uses the Poisson process for generating traffic because of its simplicity. It is still applicable in many cases, however it was proven to be inaccurate for Internet traffic. Internet traffic was proven to be of self-similar nature [20, 49, 82, 89]. All the experiments are conducted twice based on two traffic generation models: *Poisson process* and *ON/OFF Pareto*. A multi-process program is built, acting as the concurrent users, to send requests to the web-servers' load-balancer. The simulation parameters are as follows:

- Generator: On/Off Pareto, Poisson
- Number of concurrent clients: 50
- Requests arrival rate/hour: 3600
- Type of requests: GET and POST(randomly generated)

The On/Off Pareto input parameters are set according to the NS2[4] tool defaults. The amount of traffic is chosen to stress the VMs to trigger the scalability policy. The scale up policy is set to scale up whenever the CPU utilization average of a specific tier (app-server tier or web-server tier) is above 70% and scale down when the CPU load average is below 30%.

3.6 Results and Discussion

Anomaly injection is randomized along two dimensions - time of injection, and magnitude of the anomaly. We explore the effectiveness of our framework based on three use cases. In each use case, Poisson process and On/Off Pareto traffic generation models are used. All the experiments' duration is one hour.

Evaluation methodology. We use four metrics to evaluate the effectiveness and applicability of our approach [55].

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

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

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

$$Fscore = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

When the system detects an anomaly, it is considered a *positive* outcome. When the system doesn't detect an anomaly, the outcome is *negative*. Therefore:

1. TP: anomaly occurred and was successfully reported.

2. FP: anomaly didn't occur and was reported.

3. TN: anomaly didn't occur and was not reported.

[4]NS2 tool manual. http://www.isi.edu/nsnam/ns/doc/node509.html

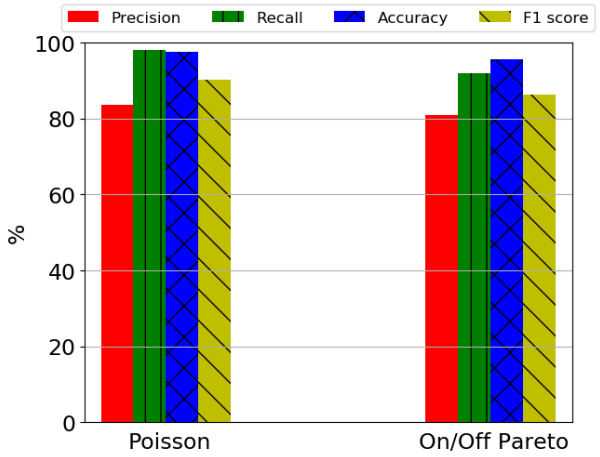


Figure 3.6: Injected anomalies detection with tuned classifier

4. FN: anomaly occurred and was not reported.

Precision refers to the number of data samples, detected as anomalies, that are actually true anomalous samples. On the other hand, recall refers to the percentage of correctly detected anomalies based on the total number of anomalies of the data samples.

3.6.1 Injected Anomalies

The effectiveness of the framework is being tested by injecting anomalies in randomly chosen VMs. Injected anomalies are *cpu, memory* and *disk intensive*.

Figure 3.6 shows the detection results of the experiment. The results show that the detector performs similarly well on both traffic models with accuracies over 90% but the precision suffers an amount of FPs due to the nature of the fluctuated traffic load and resource usage.

3.6.2 EDoS

Not all threats intensively use resources. As a matter of fact, the EDoS [75] attack tries to avoid the intensive resource usage by keeping low profile. One form of EDoS is to create some VMs while remaining dormant and idle (can be done by stealing the credentials of one of the cloud tenants who has authority to create VMs). Such attacks tries to waste resources in way that is not obvious to the tenants. EDoS was simulated by randomly injecting VMs, with the only outcome during the

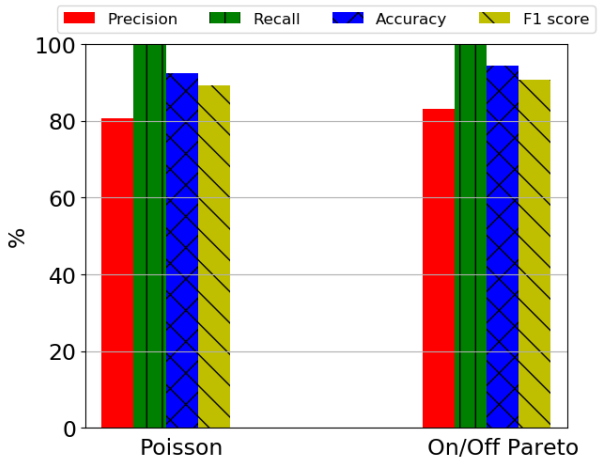


Figure 3.7: EDoS detection with tunned classifier

life-time of each injected anomalous VM being TPs or FNs. The injected VMs remain dormant and keep low resource usage profile.

Figure 3.7 shows the detection evaluation performance for this experiment. It is clear that the detector is effective against this kind of EDoS attack since the injected VMs have very different profiles than the 3-tiers. The results show a loss of precision due to detected FPs. After investigation, it was due to spawning high load of VMs (by injecting and scaling). The cloud was unresponsive which prevented any traffic load from reaching the VMs resulting in sudden drop in resource usage. As such, this is not caused by poor detection. In fact, this helps security administrators detecting times where their system is down.

3.6.3 Ransomware

Ransomware has become very popular since 2016. Netskope[5] quarterly cloud report states that 43.7% of the cloud malware types detected in cloud apps are common ransomware delivery vehicles. Ransomware basically encrypt various files on victim's hard drives before asking for a ransom to get the files decrypted. KillDisk linux-variant ransomware is used for experiments. The samples were obtained from VirusTotal[6]. The 1 hour experiment is divided in two phases: normal phase (first 20 minutes) and malicious phase (last 40 minutes). The malicious phase is the time after the

[5]Netskope website. https://www.netskope.com
[6]VirusTotal website. https://www.virustotal.com

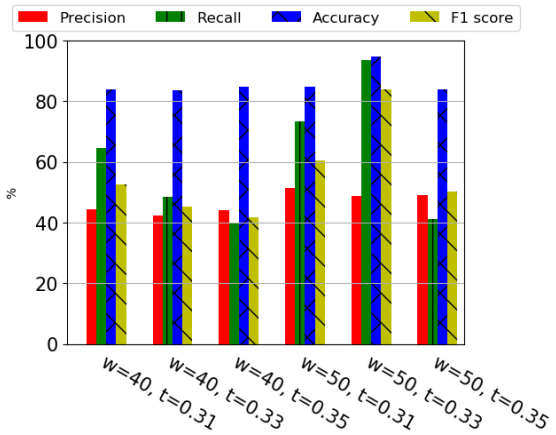


Figure 3.8: KillDisk ransomware detection - Poisson

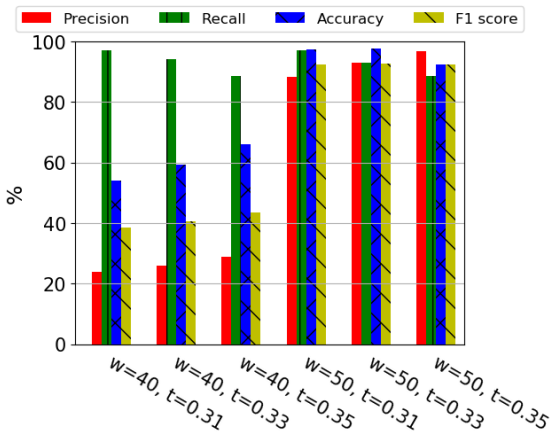


Figure 3.9: KillDisk ransomware detection - On/Off Pareto

ransomware is injected, with the only valid outcome being TPs or FNs. The ransomware is injected into a random application server due to the fact that applications are more likely vulnerable than the widely used web servers (eg. Apache or Nginx).

The results of this experiment are shown in Figure 3.8 and Figure 3.9 where the bars are produced by calculating the performance metrics for each set of modified sequential k-means specific parameters. The most two critical parameters are chosen for optimal ($w = 50$, $t = 0.33$) and near optimal results. In the case of Poisson traffic, the detector suffers FPs. On the other hand, in the case of On/Off Pareto traffic, it is clear from the results that the detector is effective with detection performance of overall more than 90%.

3.7 Conclusion

In this chapter, we investigated clustering-based cloud monitoring for detecting anomalies in scale-up and down scenarios in IaaS cloud. In order to have a better security in the cloud, all of the cloud monitoring points have to be covered. We proposed a framework to cover a subset of the cloud monitoring points. The framework uses clustering for IaaS monitoring in the cloud. It uses a modified version of the Sequential K-means algorithm to overcome the problem of high false alarm rate when using clustering. The results showed how the framework can detect anomalies in three scenarios. The experiments show that parameter tuning is a very important issue and is very

dependent on the use case.

The reason we focused on one category of malware (ransomware) is the growing concern of activity of ransomeware as well as its noticeable behavior and thus it is applicable for detection by our framework. One limitation of our framework is that it is vulnerable to low-profile anomalies and malware. Detecting those types of anomalies has proven to be hard using only resource usage metrics. Another limitation is that an expert is necessary for parameter tuning which, in some cases, is unavailable/unaffordable for some cloud tenants.

In the future, we plan to investigate various issues. First, dynamic parameter tuning, which is essential for getting more accurate results. Second, we plan to experiment on different architectures other than the 3-tier web architecture. Lastly, we plan to use and compare different machine learning algorithms for anomaly detection.

CHAPTER 4: MALWARE DETECTION IN CLOUD INFRASTRUCTURES USING CNN

4.1 Introduction

In Chapter 3, we showed that malware detection can be effectively performed by inspecting the performance and resource utilization metrics of VMs as a black-box. Although the approach works well with highly active malware (e.g. ransomware), it is not as effective for detecting malware that maintains a low-profile of resource utilization. In this chapter, we develop a novel and effective technique to detect such low-profile malware that utilizes minimal system resources, by inspecting raw, fine-grained meta-data of each process in a VM.

As stated in Chapter 2, two major approaches have been explored for malware detection in the current literature: static analysis, where malware code is analyzed without running it, and dynamic analysis, where a malware is executed and its behavior observed in order to detect it. The pros and cons of these approaches for malware detection are well understood.

In this chapter, we introduce and discuss a malware detection approach using Deep Learning (DL). We demonstrate the applicability of using a 2d CNN for malware detection through the utilization of raw, process behavior (performance metrics) data. Our approach falls under dynamic analysis. However, unlike most prior works that utilize machine learning (ML) in dynamic analysis to classify malware files, we use it for online malware detection. Note that the approach introduced in this chapter is general and is not confined to the use of CNN. The choice of using CNN is due to its simplicity and training speed as opposed to other DL architectures such as Recurrent Neural Networks (RNNs). Applying and comparing different ML approaches is left to future work.

One of the biggest challenges in employing ML for malware detection is the **mislabeling problem**. This is because, during the training phase, there is no guarantee that a malware exhibited malicious behavior. While some malware start performing malicious activities immediately after infecting a machine, a reasonably sophisticated malware starts off as a process and idles until some

condition is met (e.g., a command from its remote owner), which can occur at any time. In particular, such a condition may never occur during the training phase for the malware to activate. However, this issue is rarely addressed in existing literature except for the work in [24], which recognizes this issue. The authors stated that this problem can pollute the training and testing data; however, since there is no way around it, they had to make the assumption that it is all right to label all the data as malicious after a malware execution takes place. In other words, the assumption is that malware will always show malicious activity at all times.

We follow their assumption in this work but not to the fullest. Consider a more common scenario when a malware periodically (e.g., every 1 minute) performs malicious activities such as stealing and sending some information to its Command and Control servers (C&Cs). Now the malware is surely conducting a malicious behavior but only periodically. As a result, if a malware is run for 15 minutes and we collect a data sample every 10 seconds (total of 90 samples), all the collected data samples will be labeled as malicious whereas in fact only 15 of them are malicious. This will cause a **mislabeling problem** during the training phase.

To mitigate this problem, we refine the above assumption by assuming that a malware will show malicious activity within a time window. The underlying rationale is that while there is no way to know for sure that a malware ever exhibited malicious behavior during the training phase, it is more practical to consider a sliding window of time during which malicious behavior is exhibited instead of assuming that all data samples collected after malware injection indicate malicious activity. This increases the probability of correctly labeling our samples. Toward this end, we develop a 3d CNN classifier which takes a 3d input matrix containing multiple samples over a time window. In summary, the contributions of this chapter are two-fold:

- We develop an effective approach for detecting malware by learning behavior from fine-grained and raw process meta-data that are available directly from the hypervisor. The approach we develop is resistant to the aforementioned mislabeling problem.

- We demonstrate the effectiveness of this approach by first developing a standard 2d CNN model that does not incorporate the time window, and then comparing it with a newly de-

veloped 3d CNN model that significantly improves detection accuracy mainly due to the employment of a time window as the third dimension, thereby mitigating the mislabeling problem.

To the best of our knowledge, our work is the first to apply 2d and 3d CNN on raw performance metrics of processes, which can be easily obtained through the hypervisor layer. This is critical if a cloud service provider were to offer such a malware detection service. Since the approach we propose does not require an agent to run within VMs, we avoid any major privacy and security concern for cloud tenants.

The remainder of the chapter is organized as follows. Section 4.2 outlines the methodology including the architecture of the CNN models used. Section 4.3 describes the experiments setup and results. Section 4.4 gives a discussion about some of the important limitations and possible mitigations. Section 4.5 summarizes and concludes this chapter.

4.2 Methodology

This section provides an overview of the methodology used for malware detection in VMs using CNN.

4.2.1 Convolutional Neural Network

CNN is a type of DL that has been applied to images analysis and classification. One advantage of CNN is that it requires little pre-processing as compared to similar image classification algorithms since it works on raw data. It acts as a feature extractor which is very convenient since feature selection in most cases requires human experts.

Figure 4.1 shows the architectural overview of a CNN. Much like deep neural networks, CNN consists of input and output layers and multiple hidden layers. A *Convolutional layer* applies a convolution operation on the input matrix and passes the output to the next layer. A convolution operates on two inputs: feature map (input matrix) and convolution kernel (works as a filter) and outputs another image. The kernel is used to filter out certain information from the feature map

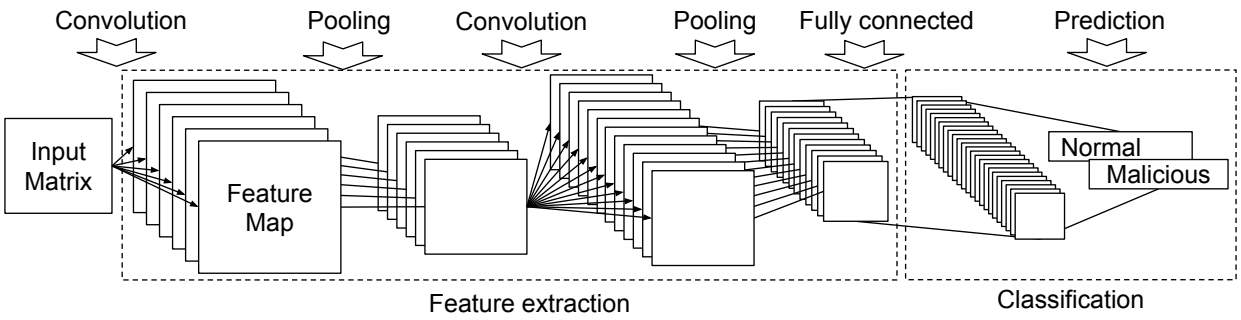


Figure 4.1: CNN overview

and discard other information. In other words, a convolution operation uses multiple kernels where each kernel is responsible to extract and focus on a piece of information (e.g., one kernel might filter edge information). Usually, a convolutional layer is followed by a *Pooling layer* which takes the output of the convolutional layer as input. Pooling is an operation in which it down samples the feature maps received from the convolutional layer. It works by taking a certain area of the input and reduces it to a single value. For example, max pooling uses the maximum value from certain area, while average pooling uses the average value. Convolutional and pooling layers are followed by fully connected layers, which connect every neuron in one layer to every neuron in the next layer.

4.2.2 Process Performance Metrics

In this work, we use performance metrics as a way of defining a process behavior. Table 4.1 shows metrics that are selected to be collected for the VMs. Selected metrics are for the purpose of showing the effectiveness of our approach; in practice, many more metrics are available. For the sake of practicality, we assume no prior knowledge of any additional information other than the metrics we collect in Table 4.1.

4.2.3 CNN Input

We represent each sample as an image (2d matrix) which will be the input to the CNN. Consider a sample X_t at a particular time t, that records n features (performance metrics) per process for m

Table 4.1: Virtual machines process-level performance metrics

Metric Category	**Description**
Status	Process status
CPU information	CPU usage percent, CPU times in user space, CPU times in system/kernel space, CPU times of children processes in user space, CPU times of children processes in system space.
Context switches	Number of context switches voluntary, Number of context switches involuntary
IO counters	Number of read requests, Number of write requests, Number of read bytes, Number of written bytes, Number of read chars, Number of written chars
Memory information	Amount of memory swapped out to disk, Proportional set size (PSS), Resident set size (RSS), Unique set size (USS), Virtual memory size (VMS), Number of dirty pages, Amount of physical memory, text resident set (TRS), Memory used by shared libraries, memory that with other processes
Threads	Number of used threads
File descriptors	Number of opened file descriptors
Network information	Number of received bytes, Number of sent bytes

processes in a VM, such that:

$$\mathbf{X}_t = \begin{bmatrix} & f_1 & f_2 & \dots & f_n \\ p_1 & \vdots & \vdots & \dots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ p_m & \vdots & \vdots & \dots & \vdots \end{bmatrix}$$

Note that a CNN requires the same process to remain in the same row in each sample. For example, a process with PID 1 that resides in the first row of the matrix must remain in the first row across all upcoming samples. The CNN in computer vision takes fixed-size images as inputs, so the number of features (n) and processes (m) must be predetermined. The number of features is easily determined since we have a fixed number of collected features represented in Table 4.1 (28 in our case). On the other hand, determining the number of processes is not as easy since the processes are dynamic in nature. In highly active systems (e.g., web or app server), many processes get created and killed to handle client requests based on the workload.

A process is defined by a process identification number (PID) which is assigned by the OS. In a Linux based OS (used in our experiments), PID numbers will increase to a maximum system-

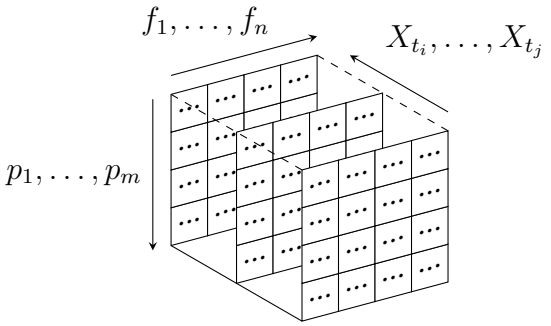
dependent limit and then wrap around (recycle). The kernel will not reuse a PID before this wrap-around occurs.[1] The limit (maximum number of PIDs) is defined in `/proc/sys/kernel/pid_max` which is usually 32k. This number presents a problem because a matrix of $32k \times 28$ is a huge input matrix. Also having too many variables in the input requires a large number of input in any neural network. Limiting the max number of processes to a lower value and depending on the concept of wrap-around will not solve the problem because of many reasons. First, the reason the max number of processes is set to a very large number (i.e. 32k) is that it can confuse the kernel if the value is too small and wraps around too often, not to mention that it is hard to determine the appropriate number before hand. Second, there is no guarantee that, for instance, a process with a PID 1000 at time t_1 is going to be the same process at time t_{100}. Considering the wrap-around concept, this process might have been killed and a new different process could be assigned the same PID later on. This can cause inaccurate results by the CNN since an important requirement is that the same processes remain in the same rows at all times.

To solve these problems, instead of defining a process by it's PID, we define a process, referred to as **unique process**, by a 3-tuple: process name, command line used to run process, and the hash of the process binary file (if applicable). In cases where the same application (e.g., apache web server) forks multiple child processes (with the same name, cmd, and originated binary), we aggregate these processes by taking the average of their performance metrics. This also helps in smoothing the fluctuations of processes that have similar functions. In all of our experiments none of the VMs had more than 100 unique processes; however, for practicality, we set the maximum number of unique processes to 120 to accommodate for newly created unique processes. Any unavailable unique process (due to termination) at a particular time is padded with zero-values. In the rest of the chapter, the term process and unique process are used interchangeably, where both refer to unique process.

The 3d CNN model input includes multiple samples over a time window. The input matrix is

[1] Linux Manual. http://man7.org/linux/man-pages/man5/proc.5.html

$$\mathbf{X}_{t_{ij}} =$$



where $\mathbf{X}_{t_{ij}}$ is the 3d input matrix containing samples from time t_i to t_j. As stated in section 4.1, we use a 3d CNN model to enhance the results by capturing patterns over a small time window which in turn helps in mitigating the mislabeling problem.

4.3 Experiment Setup and Results

In this section, first, we present the CNN model used in this work as well as the data preprocessing step. Second, we review our experimental setup. Then, we provide the results to illustrate that a 2d CNN can be effective in detecting low-profile malware using per-process performance metrics. Lastly, we show how using a 3d CNN can improve the results by attempting to solve the mislabeling problem.

4.3.1 Preprocessing

It is essential to CNN to have scaled data input for faster convergence and better accuracy results. A standard approach is to rescale the data to have a mean of 0 and standard deviation of 1. It is done in a per feature fashion. Given a set of features $F = \{f_1, f_i, ..., f_n\}$ and a set of samples $X = \{x_1, x_j, ..., x_t\}$, it is defined as $x^{(f_i)}_{j_{standardized}} = (x^{(f_i)}_j - \mu^{(f_i)})/\sigma^{(f_i)}$, where $x^{(f_i)}_j$ is a vector of values corresponding to feature f_i in the jth input sample, and $\mu^{(f_i)}$, $\sigma^{(f_i)}$ are respectively the mean and the standard deviation of values corresponding to feature f_i across all samples in set X. The same two sets of $\mu^{(f_i)}$ and $\sigma^{(f_i)}$ (obtained from the training dataset) are used for standardizing the validation and testing datasets.

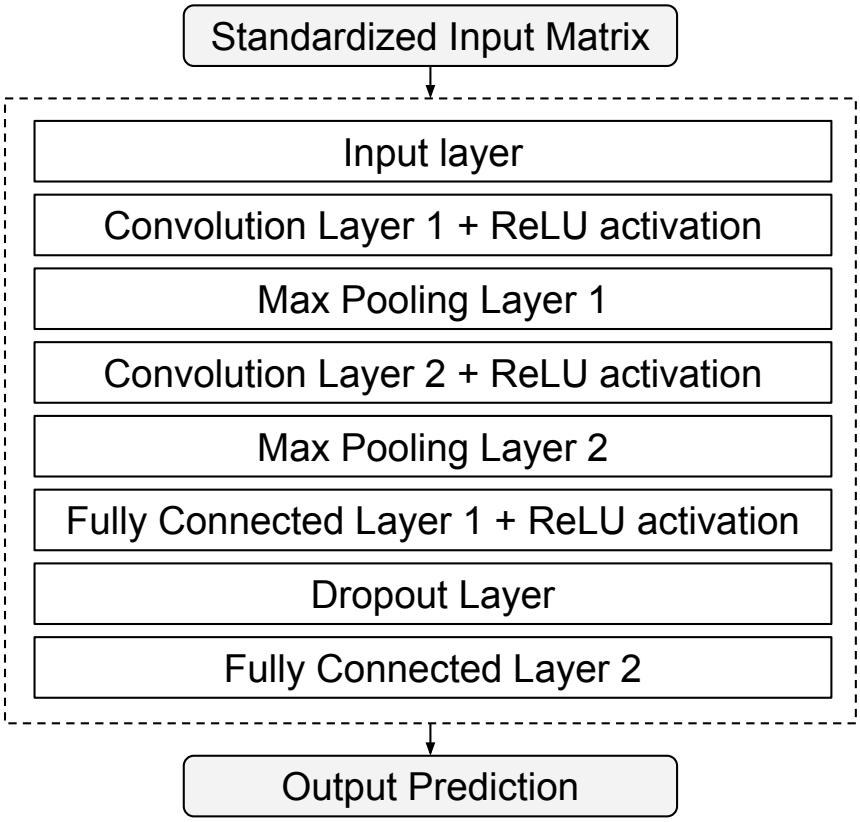


Figure 4.2: Proposed CNN Model

4.3.2 CNN Model Architecture

Figure 4.2 shows the CNN model used in this work. It consists of 8 layers. First, the input layer which is basically received as the input matrix. Second, a convolutional layer which receives a $d \times 120 \times 28$ standardized matrix, representing samples in a particular time window, where d is the depth of the input matrix and 120×28 is the length of the 2d matrices representing the number processes and features, respectively. Then, it performs a convolutional operation with 32 kernels of size $d \times 5 \times 5$ with zero-padded ending. The results of this layer are 32 feature maps of size $d \times 120 \times 28$. Third, a max pooling layer of size $2 \times 2 \times 2$ which down size each dimension by a magnitude of 2, resulting in a 32 feature maps of size $d/2 \times 60 \times 14$. The fourth and fifth layer are replicates of layer two and three so the output of the max pool layer 2 is 64 feature maps of size $d/4 \times 30 \times 7$. The last 3 layers are a fully connected layer with size of 1024, a dropout layer described below, and, last, another fully connected layer with size of 2 denoting the classification

probability of a malicious or benign VM sample. Note that the model doesn't classify malicious or benign processes but rather the VM as a whole which means there is no way to know which process is malicious.

To reduce over fitting, we use a dropout [35] layer after the first fully connected layer, since it is shown in previous work [83] that dropout regularization works well with fully connected layers.

Rectified linear unit (ReLU), a simple and fast activation function, is simply defined as $f(x) = max(0, x)$. It turned out that ReLU (which is used in our work) works better in practice than the other activation functions as well as it's several times faster in training as stated in [46].

The model is trained using back-propagation for Adam Optimizer [42], a stochastic gradient descent that automatically adapt the learning rate. The optimizer works on minimizing the loss function. We use the mean cross entropy as a loss function. The model is also trained using mini-batches which is not reflected in the layers described above.

The described CNN model is used for both 2d CNN and 3d CNN except the former has one less dimension (i.e. the depth d of the input matrix is 1). The CNN structure used in this work is considered to be shallow as opposed to models such as GoogleNet and LeNet due to the limit of the experiments we could perform in our lab which, in turn, led to lack of large data sets. Experimenting in a larger scale and comparing different CNN models is left to future work.

4.3.3 Parameters Tuning

Parameters tuning is a very challenging problem in ML in general. It helps choosing the set of parameters that yields the best classification accuracy. A common approach, used for most of the parameters in our work, is grid search, where we define (based on our knowledge) bounds for each parameter and try all the combinations that yields the best classification accuracy during the validation phase. Other approaches can be more practical such as random search [10]. In our case, the set of important parameters are as follows. **Dropout**. Dropout is a regularization technique that turns neurons on/off in each layer to force them to go through different path. This operation improves generalization of the network and prevents over-fitting. We set this parameter to 0.5 [8].

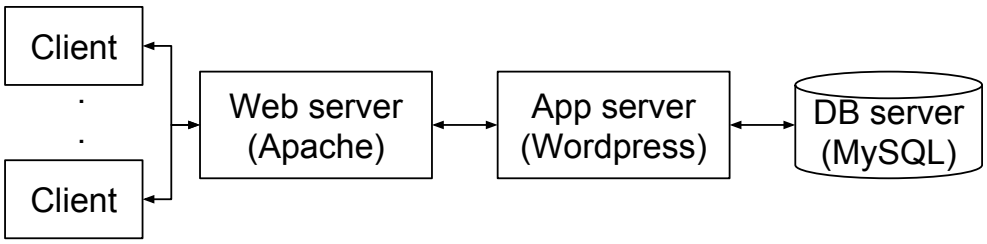


Figure 4.3: 3-tier web architecture

Learning rate. This determines how fast we move toward the optimal weights in our network. If this parameter is very large, it will skip optimal values. On the other hand, if it is too small, it will take too much time to converge to the optimal values, and it may get stuck in local minima. Typically, a stochastic gradient descend uses decay learning rate to slow down the learning rate as it moves forward. AdamOptimizer (used in our study), adapts the learning rate automatically, however the adaptation maximum ceiling is defined by our learning rate parameter. If we set it very large, it will give the optimizer more room to adapt which can be problematic in some cases. The values we found reasonable during our experiments lies between $1e-3$ and $1e-5$. **Mini-batch size**. As CNN is using mini-batches to learn, we define the bounds of our mini-batches sizes between 10 and 30. Going lower or higher proved to decrease the accuracy.

4.3.4 Experimental Setup

Our experiments were conducted on Openstack[2] (a major open-source cloud orchestration software). To simulate a real world scenario, we used a 3-tier web architecture (one of the most common cloud architectures according to Amazon[3]). Note that our work is not confined to the 3-tier web architecture use case used in the experiments since our approach relies on learning the behavior of processes in VMs. This means that learning approach of processes behavior would remain the same regardless the architecture in place. Figure 4.3 shows the setup used to conduct our experiments on Openstack. A 3-tier web architecture, typically, consists of 3 separate tiers: web, application and database server. In our case, we used Apache as a web server, Wordpress[4]

[2]Openstack website. https://www.openstack.org/
[3]Amazon architecture references. https://aws.amazon.com/architecture/
[4]Wordpress website. https://wordpress.org/

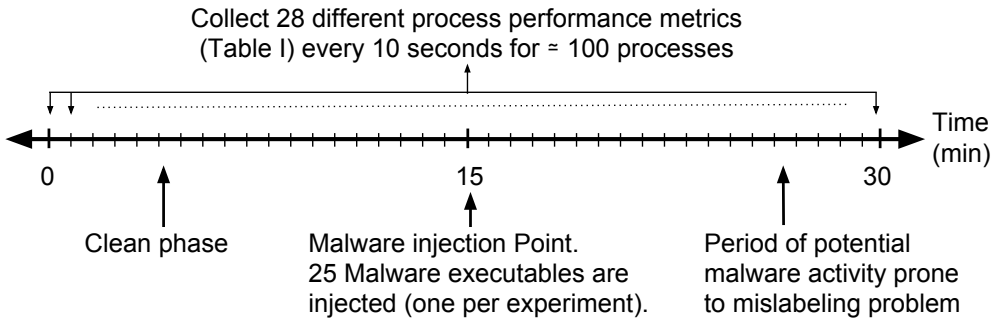


Figure 4.4: Data collection overview

(a major open-source content management system) that utilizes PHP as an application server and MySQL as a database server.

According to [49], Internet traffic is of self-similar nature. Thus, we built a multi-process traffic generator (set to the NS2[5] default parameters values), based on ON/OFF Pareto distribution, to generate traffic for our experiments.

Figure 4.4 shows an overview of the data collection process. Each of our experiments was 30 minutes long. The aforementioned 3-tier architecture was created from known-to-be clean images. The first 15 minutes is the normal phase, where no malicious activity takes place, and is followed by 15 minutes of malicious phase, where a single malware is injected and executed in the application server. Few normal processes were injected during the normal phase to check the effectiveness of our approach in handling false positives. The malware was injected in the application server VM because most vulnerabilities, typically, lies in the application side.

The image used for spawning VMs is Ubuntu 16.04 which was modified to include a data collection agent. Data was collected at 10-second intervals in a JSON object. We refer to each of the collected objects at a particular time as a sample. For simplicity, we included an agent inside VMs to collect data; however, data collection could also be done through Virtual Machine Introspection (VMI) since similar metrics [7, 87] could be collected from the hypervisor.

The 25 malware binaries[6] used were randomly obtained from VirusTotal[7]. They mainly belong to 3 classes: Rootkits, Trojans and Backdoors and have unique SHA-256 hashes.

[5]NS2 tool manual. http://www.isi.edu/nsnam/ns/doc/node509.html
[6]https://github.com/mahmoudaslan/researchrepo/blob/master/malwarehashes
[7]VirusTotal website. https://www.virustotal.com

Most malware check for connection to their C&Cs, otherwise, they remain idle. Many researches (on malware dynamic analysis) use sandboxes or VMs in a controlled environment which can cause hindrance to the malware. To accommodate for this problem, all of our VMs are connected to the Internet outside of firewalls to prevent any intervention. To avoid data pollution, experiments were totally independent and all VMs used for one run were completely destroyed before the next run because malware can infect other VMs and possibly pollute subsequent runs.

We collected samples at 10 seconds intervals for 30 minutes duration, so we have a total of $\simeq$ 180 samples per experiment and $\simeq$ 4500 samples in total.

4.3.5 Evaluation

We use four metrics [55] to evaluate our CNN classifiers:

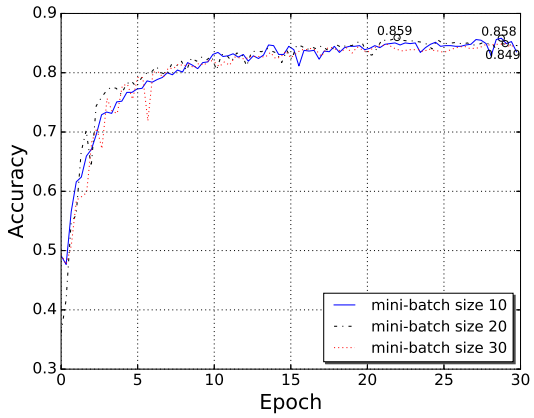
$$Precision = \frac{TP}{TP + FP}$$

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

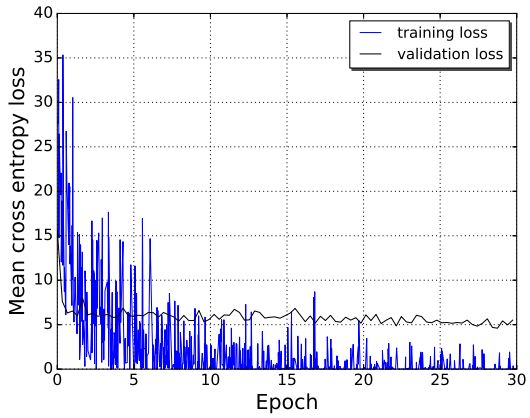
$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

$$Fscore = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

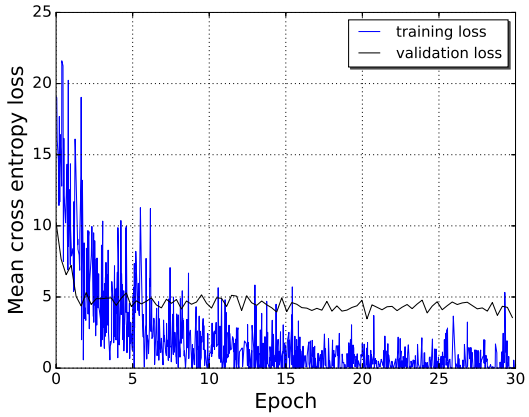
Precision is the number of correct malware predictions. Recall is the number of correct malware predictions over the number of true malicious samples. Accuracy is the measure of correct classification. F score is the harmonic mean of precision and recall. True Positive (TP) refers to malicious activity that occurred and was correctly predicted. False Positive (FP) refers to malicious activity that did not occur but was wrongly predicted. True Negative (TN) refers to malicious activity that did not occur and was correctly predicted. False Negative (FN) refers to malicious activity that occurred but was wrongly predicted.



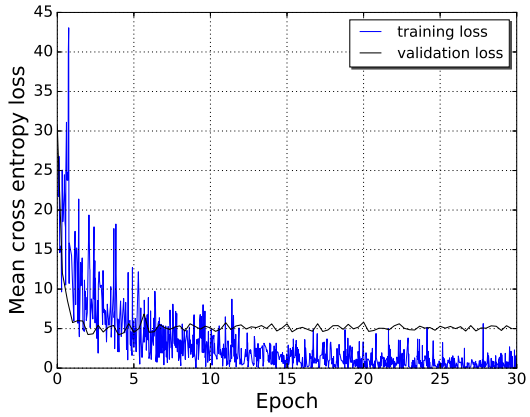
(a) Accuracy of 2d CNN. mini-batch sizes of 10, 20 and 30



(b) Mean cross entropy loss of 2d CNN. mini-batch size = 10



(c) Mean cross entropy loss of 2d CNN. mini-batch size = 20



(d) Mean cross entropy loss of 2d CNN. mini-batch size = 30

Figure 4.5: 2d CNN trained with different mini-batch sizes. Optimized with learning rate of 1e-5 for AdamOptimizer

4.3.6 2d CNN Results

The data collected are divided into 3 sets: training, validation and testing sets with the percentages of 60%, 20%, and 20% respectively. The split is done on the number of experiments. For example, the 25 experiments (each using a different malware) is split to 15, 5 and 5 respectively. This means that the validation and testing phases are exposed to unknown malware. Training data is used to train the CNN models. Then, the validation set is used as a way to tune the parameters of the CNN. Once we get the highest validation accuracy for a model with specific set of parameters, we use

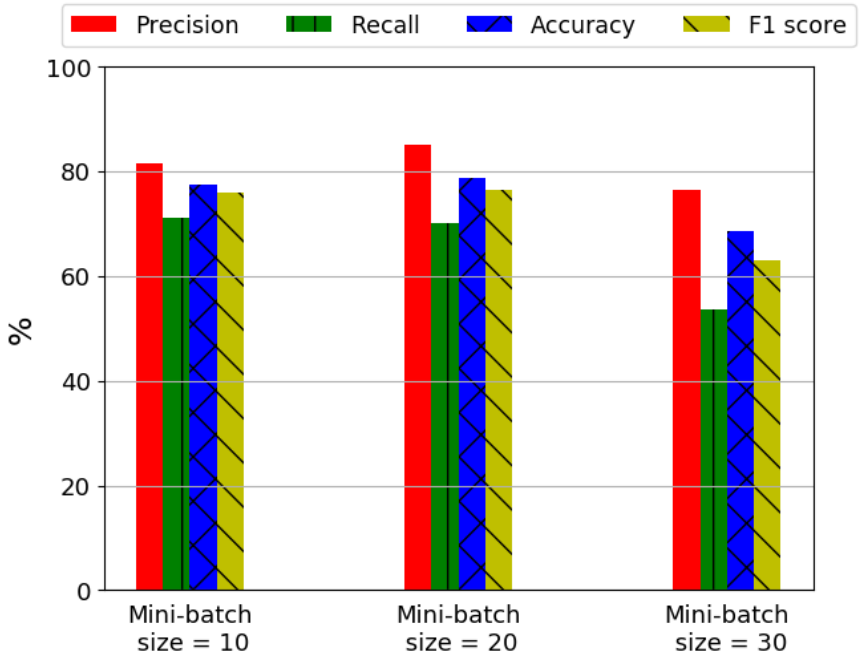


Figure 4.6: 2d CNN classifiers results

the testing set to test the chosen model (optimized classifier). The classifiers were trained for 30 epochs as it turned out, in our case, that there was no extra gain of accuracy or decrease in mean cross entropy loss afterwards.

We only show results for classifiers using learning rate of $1e-5$ because they showed the highest accuracy and lowest mean cross entropy loss. Figure 4.5 shows three trained classifiers based on different mini-batch sizes of 10, 20, and 30. Figures 5.3a shows the accuracy the three 2d classifiers, and similarly, Figures 5.3b, 5.3c and 5.3d show the mean cross entropy for mini-batch size of 10, 20 and 30, respectively.

In general, the results show that using mini-batch size of 20 yields the highest accuracy of 85.9% during validation.

Figure 4.6 shows the results of the 4 evaluation metrics. The CNN classifier with mini-batch of 20 shows the highest results when it is evaluated on the testing data set, while the classifier with mini-batch size of 30 shows the lowest (larger mini-batch sizes can lose generalization [41]); however, there is a drop in the overall performance of the classifiers on the testing data set where the highest accuracy is $\simeq 79\%$.

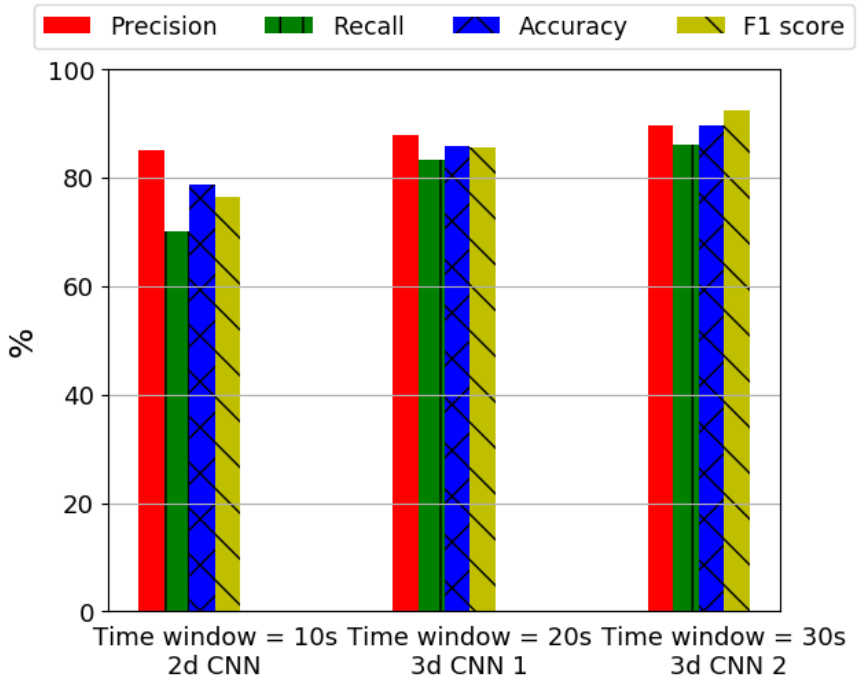


Figure 4.7: Optimized 2d and 3d CNN classifiers results. 3d CNN classifiers are best optimized with learning rate of 1e-4 as well as with 20 and 30 mini-batch sizes, respectively.

4.3.7 3d CNN Results

The 3d CNN classifiers take time-windowed input. Samples inside this time window represent the depth of the input matrix. In fact, the 2d CNN is a special case of the 3d CNN where the depth is 1. Our experiments is done on 2 time-windows: 20 and 30 seconds. Since data is collected in 10 seconds intervals, a 10 seconds time window means 1 data sample and, similarly, 20 and 30 seconds time windows means 2 and 3 data samples, respectively.

Figure 4.7 shows a comparison of the performance metrics of the 2d and the newly tested 3d classifiers. These results are based on the testing data set. We refer to the classifiers as shown in Figure 4.7: 2d CNN, 3d CNN 1 (20 seconds time window) and 3d CNN 2 (30 seconds time window). The results showed significant improvement of using 3d CNN 1 and 3d CNN 2. The accuracy of 3d CNN 1 and 3d CNN 2 classifiers jumped to $\simeq 86\%$ and $\simeq 90\%$, respectively, as opposed to the 2d CNN classifier accuracy of $\simeq 79\%$.

4.4 Discussion

In this section, we discuss some relevant issues in our approach and some possible improvements for future work.

Accuracy drop between validation and test. The 2d CNN classifiers showed a drop of accuracy from $\simeq 86\%$ (validation dataset) to $\simeq 79\%$ (testing dataset). Similarly, a drop of accuracy also happened during 3d CNN classifiers evaluation (from $\simeq 97\%$ to $\simeq 90\%$ and $\simeq 89\%$ to $\simeq 86\%$). Although it might seem normal considering the validation set is biased since it is used for parameters tuning, one reason is that the malware included in the testing data set (after manual examination) is shown to have more different behavior than ones included in the training and validation set. Note also that malware which apparently has the same purpose can have different behavior which can confuse classifiers that uses malware classes information. For example, one Trojan we analyzed opens a back-door and remains idle, while another opens a back-door, steals and sends system information over the Internet. In our experiments, we randomly selected our malware from few classes (trojans, rootkits, etc..) to completely unbias our experiments.

Mislabeling problem. Using 3d CNN, we improved the mislabeling problem stated in Section 4.1. Figure 4.8 shows a behavior of a malware for just 1 metric. The spike in the figure shows the time when the malware first booted up. Then, the malware keeps idle for specific time not performing any malicious activity. Labeling all samples corresponding to the benign area shown in the figure will pollute the data because the classifier learns that these actions are malicious while in fact they are not. On the other hand, when the malware steals and sends data over the Internet, samples should be labeled as malicious. Differentiating between those two actions is not possible unless it is seen by human experts. In most cases, researches take the risk of this kind of pollution because there is no way around it. A partial solution is to take both the shown areas as one sample and state that during this time window a malicious activity has happened. This is essentially what our 3d CNN classifiers are trying to do by decreasing the number of mislabeled samples as well as capturing patterns over a small time window. In theory, the larger the window the better; however, a very large time window would need a large amount of data , as well as it would act as a window

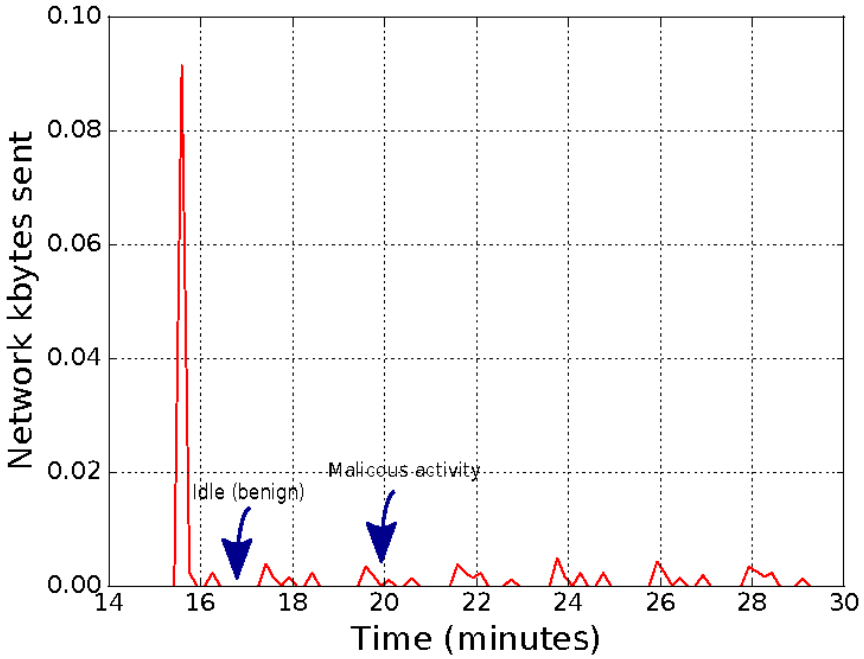


Figure 4.8: Malware behavior of the *network sent kBs* metric.

of opportunity for the malware to maliciously act before detection and possible mitigation.

3d CNN's need for data. We experimented on two time windows (20 and 30 seconds) due to the limited amount of data. Increasing the time window (meaning increasing the depth of the input matrix), needs to stack multiple data samples together. Trying to experiment with 40 seconds time window and above caused dramatic decrease in accuracy because the CNNs did not have enough data to converge and learn properly. Using 3d CNN showed significant improvement with a very short time window, so having large enough data can further improve the results.

Processes and metrics ordering. One advantage of CNN is that it takes into account the spatial structure of the data; however, in our case, it seems that the input lacks spatial structure across columns and rows of the matrix. For example, if we substituted feature f_1 column with feature f_2 column, it is still going to represent the input. On the other hand, in the case of a normal 2d image, this substitution will distort the image. The same situation is true with the rows of our input matrices when, for example, substituting process p_1 row with process p_2 row. Note that correlations might exist between the features (e.g. when CPU percent goes up, memory usage goes up as well); however, we did not use this information in our work. We believe that obtaining

correlation information about the features to be used in ordering our input matrices might help with getting better results. It is true for processes as well, although it is not as easy because of the processes' dynamic nature and the possibility of newly created processes during testing time.

4.5 Conclusion

In this work, we introduced a malware detection method for VMs using 2d CNN model by utilizing performance metrics. Results showed a reasonable accuracy of $\simeq 79\%$ on the testing dataset. We noted the problem of mislabeling and we improved the performance by introducing 3d CNN model which uses samples over a time-window. It adds a 3rd dimension (depth) to the 2d input matrix representing the samples inside the defined time window. Results showed a significant improvement of accuracy of $\simeq 90\%$ for 3d CNN 2 classifier which is practically acceptable.

In the future, we plan to dedicate a pre-training step to evaluate the effectiveness of ordering the processes and features in the input matrix. We also plan to increase the scale of our experiments by using more malware binaries which will allow evaluating different time-window sizes for the 3d CNN models as well as using deeper CNN models.

CHAPTER 5: ONLINE MALWARE DETECTION USING SHALLOW CONVOLUTIONAL NEURAL NETWORKS IN CLOUD AUTO-SCALING SYSTEMS

5.1 Introduction

Cloud computing characteristics [56] enable novel attacks and malware [21, 31, 33, 34, 39, 90]. In particular, cloud has become a major target for malware developers since a large number of Virtual Machines (VMs) are similarly configured. Automatic provisioning and auto configuration tools have led to the widespread use of auto-scaling, where VMs scale-in/out on demand. Applications utilizing auto-scaling architectures[1] is one of the most prevalent in cloud. As a result, a malware that infects one VM can be easily reused to infect other VMs that are similarly configured or imaged. To that end, cloud has become a very interesting target to most attackers.

In malware analysis, files are scanned before execution on the actual system either through static or dynamic analysis. Once an executable/application is deemed to be benign, it executes on the system without further monitoring. Such methods often fall short in the case of cloud malware injection [34], a threat where an attacker injects a malware to manipulate the victim's VMs, because the initial scan is usually bypassed or malware is injected into an already scanned benign application. Consequentially, the need for online malware detection, where you continuously monitor the whole system for malware, has become a necessity.

Few works [1,23,62,84,87] exist in the domain of *online* malware detection in cloud. Typically, machine learning is used in online malware detection. First, a set of system features are selected and used to build a model. Then, this model is used for malware detection. Some works use system calls while others use performance metrics. Although such works target cloud systems in some sense, there is no real difference between standard online malware detection methods and cloud-specific methods except in the features selected for machine learning, where cloud-specific

[1] Amazon architecture references. https://aws.amazon.com/architecture/

methods restrict the selection of features to those that can only be fetched through the hypervisor. One can argue that such works focus on malware detection in VMs running on a hypervisor.

However, what makes cloud computing powerful is the novel characteristics that they support [56] such as on-demand self-service, resource pooling and rapid elasticity via auto-scaling. In this chapter, we explore malware detection approaches that can leverage specific cloud characteristics. In particular, we focus on auto-scaling. The high-level idea is that in an auto-scaling scenario, where multiple VMs are spawned based on demand, each of those VMs is typically a replica. This means the "behavior" of those VMs need to closely correspond with each other. If a malware were to be injected online into one of those VMs, the infected VM's behavior will likely deviate at some point in time. Our work seeks to detect such deviations when they occur. A sophisticated attacker can attempt to simultaneously inject malware into multiple VMs, which could induce similar behavior across those VMs, and thereby escape our detection mechanisms. This is an interesting challenge and we plan this for future work. This chapter focuses on malware detection when exactly one of the VMs in an auto-scaling environment is compromised.

In Chapter 3, we showed that malware can be effectively detected using black-box VM-level performance and resource utilization metrics (such as CPU and memory utilization). Although, the work showed promising results for highly active malware (e.g., ransomware), it is not as effective for low-profile malware that would not impact black-box level resource utilization significantly. Subsequently, in Chapter 3, we introduced a CNN based online malware detection method for low-profile malware. This work utilized resource utilization metrics for various processes within a VM. The method was able to detect low-profile malware with accuracy of $\simeq 90\%$. Although, this work yielded good results, it targeted a single VM much like other related works.

Unlike our prior work and other related works, this chapter targets malware detection when multiple VMs are running, while leveraging specific cloud characteristics such as auto-scaling.

In terms of the approach, first, we introduce and discuss a cloud-specific online malware detection approach. It applies 2d CNN, a deep learning approach, for online malware detection by utilizing system process-level performance metrics. A 2d input matrix/image is represented as the

unique processes $\times$ *selected features*. We assume that similarly configured VMs should have similar behavior, so we train a single model for VMs that belongs to the same group such as the group of application servers in a 3-tier auto-scaling web architecture of web servers, application servers and database servers. Next, we introduce a new approach that leverages auto-scaling. Here, we consider correlations between multiple VMs by pairing samples from pairs of those VMs. Samples collected at the same time from multiple VMs are paired and fed into CNN as a single sample.

CNN is chosen because of its simplicity and training speed as opposed to other deep learning approach (e.g. Recurrent Neural Networks). Also, for the sake of practicality, we show that even a shallow CNN (LeNet-5) trained only for a few epochs can be effective for online malware detection. In summary the contributions of this chapter are two-fold:

- We introduce a 2d CNN based online malware detection approach for *multiple* VMs.
- We improve 2d CNN by introducing a new approach by pairing samples from different VMs to accommodate for correlations between those VMs.

To the best of our knowledge, our work is the first to focus on leveraging cloud-specific characteristics for online malware detection. The remainder of this chapter is organized as follows. Section 5.2 explains the key intuition about the idea presented in this chapter. Section 5.3 outlines the methodology including the architecture of the CNN models used.Section 5.4 describes the experiments setup and results. Finally, Section 5.5 summarizes and concludes this chapter.

5.2 Key Intuition

In classification-based process-level online malware detection methods, a machine learning model is trained on benign and malicious samples of processes where the goal is to classify a new input sample. The data collection phase, usually, works by running a VM for some time (benign phase) and then injecting a malware (malicious phase) while logging the required data. This is referred to as a single run. The data set includes multiple runs with same/different malware which is later divided into training and test data sets. In other words, given sample X at time t (X_t), the task is

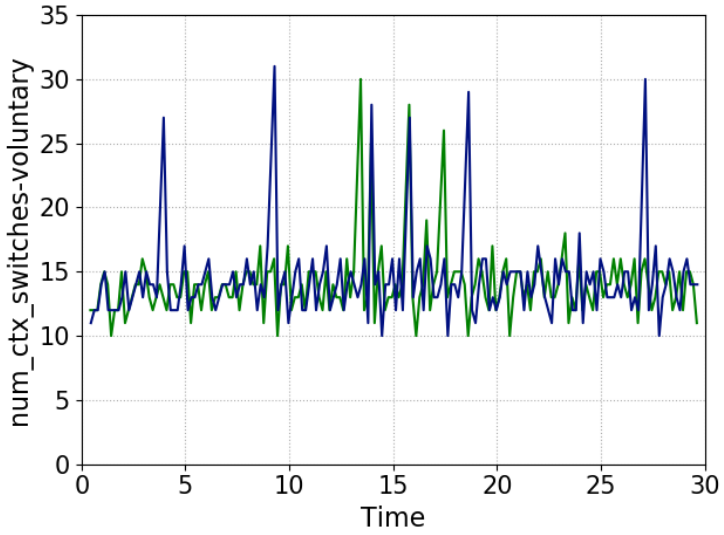


Figure 5.1: Number of used voluntarily context switches over 30 minutes for two different experiment runs of the same unique process.

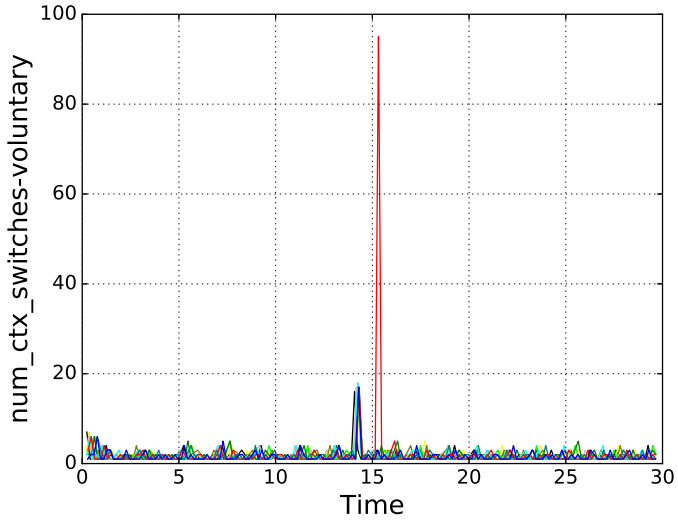


Figure 5.2: Number of used voluntarily context switches over 30 minutes for one experiment run of 10 VMs in an auto-scaling scenario. Red denotes a VM with an injected malware.

to compare X_t to previously seen samples of the training data set. For a single run, we deal with individual samples of a single VM. Thus, we refer to this approach as Single VM Single Sample (SVSS).

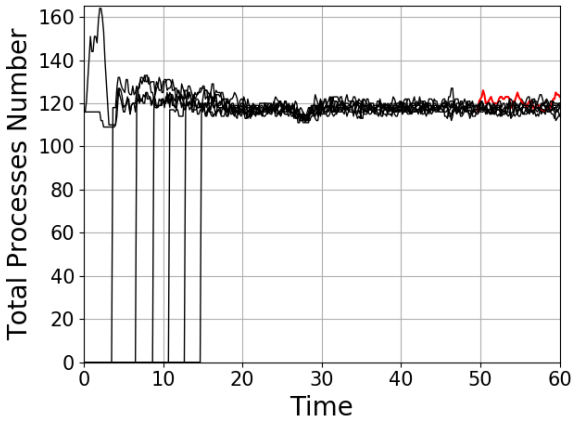
SVSS can work in an auto-scaling scenario where we have a trained model for each auto-scaling tier; however, input samples will lose some information. Note that multiple runs of a single

VM is not the same as multiple VMs running at the same time. The reason depends mostly on the architecture in place. If a VM has some effects over another VM, then input samples from single VM in multiple runs will lose this information. To that end, we extend SVSS and build an auto-scaling testbed where we can learn from multiple VMs running at the same time. We refer to it as Multiple VMs Single Sample (MVSS).

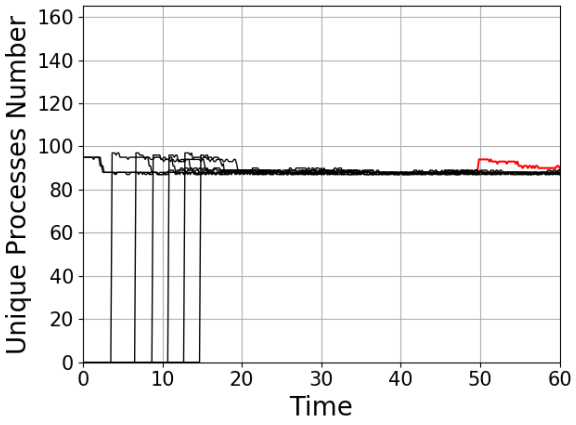
The MVSS approach, however, has a disadvantage in the context of process-level performance metrics. Processes have a very dynamic nature, meaning spikes are always happening. These spikes are mostly due to sudden events or traffic surges. For example, Figure 5.1 shows two different runs of the same process for the number of voluntary context switches. No malware is running inside either of the two VMs. During the training phase, two patterns will be learned, a smooth recurring up and down pattern and a pattern where there can be some spikes. During the testing phase, if either pattern is seen, it will be regarded as benign.

On the other hand, Figure 5.2 shows one run of the same unique process in 10 VMs (belongs to the same group of VMs in an auto-scaling scenario). VMs are running at the same time in an auto-scaling scenario. The red colored process belongs to a VM where a malware was injected. There are two major spikes in the figure. The first spike happened in the same unique process of all the VMs. If one of the processes did not have that spike and it was classified as benign, it might be a misclassification since such spike should happen to all VMs at the same time. The second spike is caused by the malware injected. MVSS and SVSS will lose such correlations between VMs since they learn from individual samples regardless of the scenario.

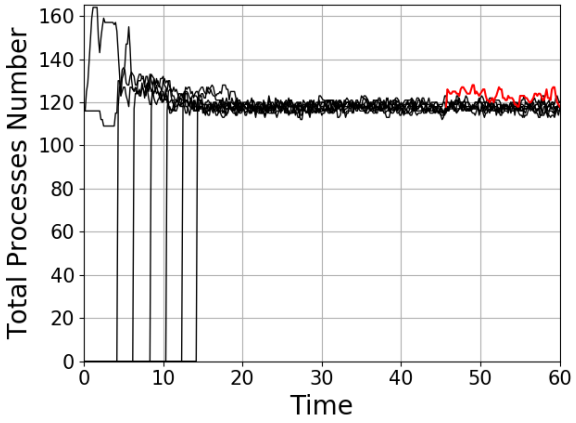
Consequentially, we introduce a new approach where the correlation of multiple VMs is utilized by pairing samples (at the same time). In other words, given sample X of VM vm_i at time t ($X_{vm_{it}}$), the idea is to compare $X_{vm_{it}}$ to previously seen paired samples of multiple VMs. We refer to this approach as Multiple VMs Paired Samples (MVPS).



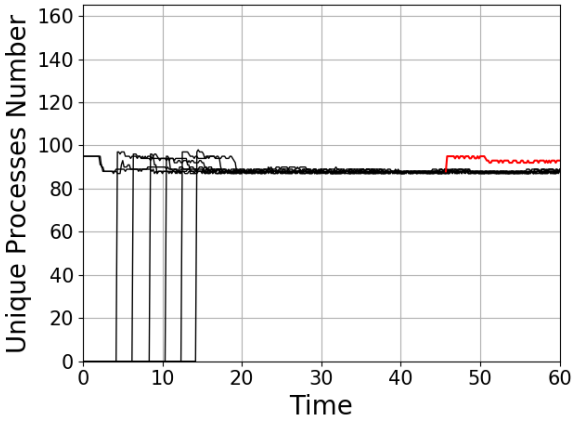
(a) Total number of processes



(b) Number of unique processes



(c) Total number of processes



(d) Number of unique processes

Figure 5.3: Total number of standard processes versus the number of unique processes in VMs running at the same time in an auto-scaling scenario. Red portions represents the VM where a malware started executing.

5.3 Methodology

This section provides an overview of the methodology used in this work. Just like the work in Chapter 4, we use per process-level performance data as features (depicted in Table 4.1). We also define a unique process which is identified by three elements: process name (name), command line used to execute the process (cmd), and hash of binary executables (if applicable, md5 hash).

Beside the reasons mentioned in 4.2.3, unique processes help in smoothing the number of processes in a highly active server because most malware create new unique processes since malware

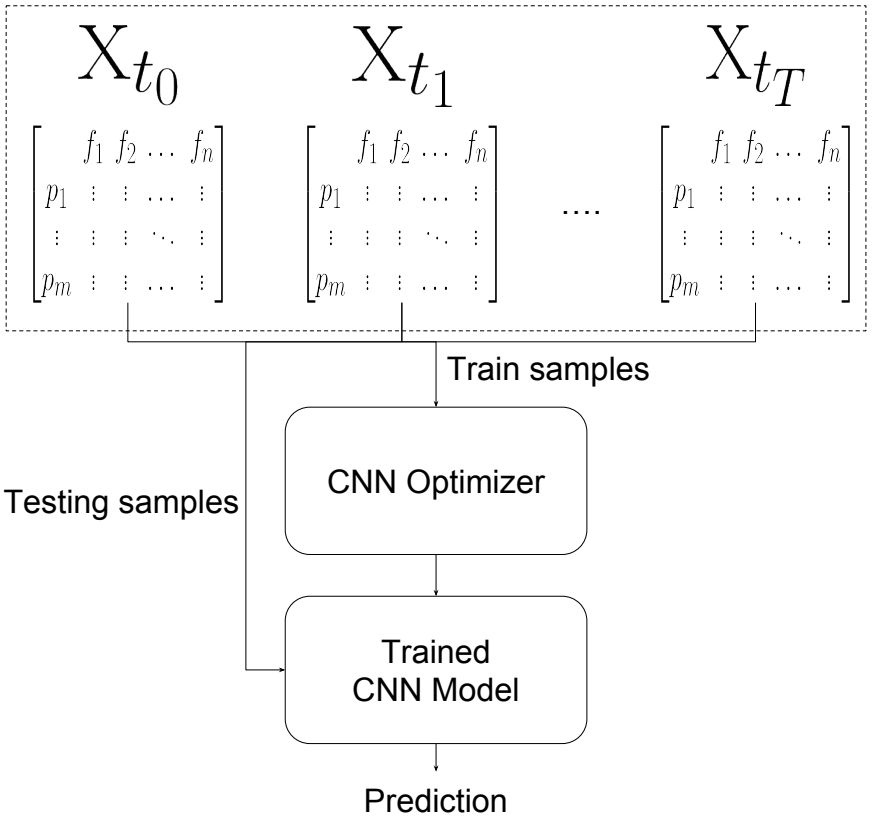


Figure 5.4: Single VMs Single Samples (MVSS)

typically run different applications than the existing processes. Figure 5.3 shows two different experiments (each with a different malware) where the total number of processes are compared to the number of unique processes. Red portions are the start time of malware execution. As shown in the figure, the total number of processes in such a highly active VM doesn't help much in revealing the malware behavior as opposed to the unique processes case. Note that throughout this chapter the terms process and unique process are used interchangeably where both are referring to unique process.

5.3.1 Malware Detection in Multiple VMs using Single Samples (MVSS)

Online malware detection in a single VM (SVSS, used in Chapter 4) is shown in Figure 5.4, where we have samples X_{t_k}, where X is a data sample collected at time t_k during one experiment. Samples from many runs are collected and are fed to the CNN optimizer where the learning process

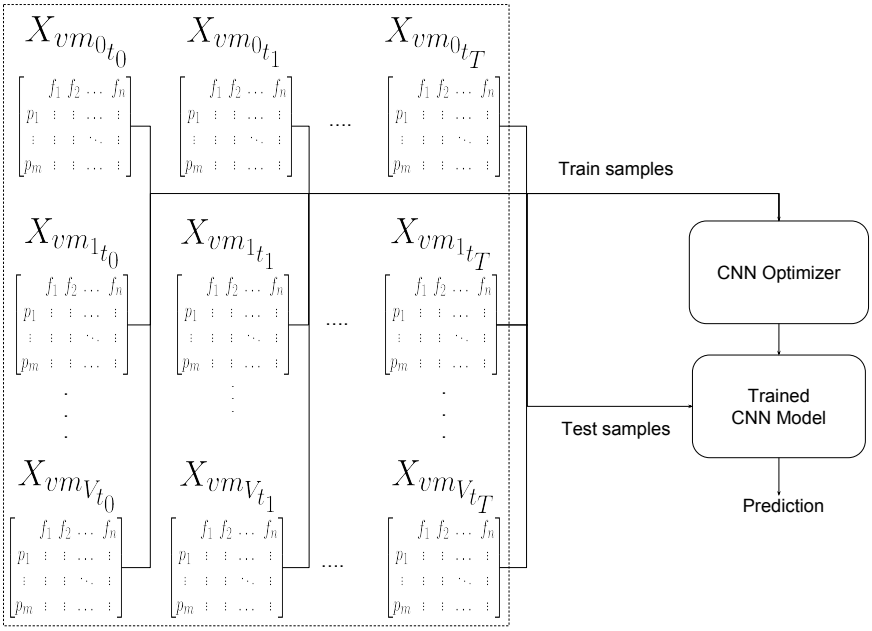


Figure 5.5: Multiple VMs Single Samples (MVSS)

takes place. Then the trained CNN model is used for predictions.

In this chapter we target multiple VMs in an auto-scaling scenario. Figure 5.5 shows the approach used to handle multiple VMs. In fact, we use the same simple approach used in SVSS except in the auto-scaling scenario we have samples $X_{vm_{i t_k}}$ from multiple VMs running at the same time, where X is a sample of VM vm_i at time t_k. Similarly, these samples are used to train the CNN model which is, in turn, used in predictions.

5.3.2 Malware Detection in Multiple VMs using Paired Samples (MVPS)

The MVPS approach is inspired by the duplicate questions detection problem in online Q&A forums like Stack Overflow and Quora. The problem focuses on determining semantic equivalence between pairs of questions. It is a simple yet complicated binary classification problem where two questions Q_1 and Q_2 are given and the task is to determine whether they are duplicates or not. Note that the two questions are not exactly the same, but semantically equivalent.

Based on the aforementioned assumption that VMs belong to the same group should behave similarly, we use the same analogy to tackle our problem. To that end, we change the formalization

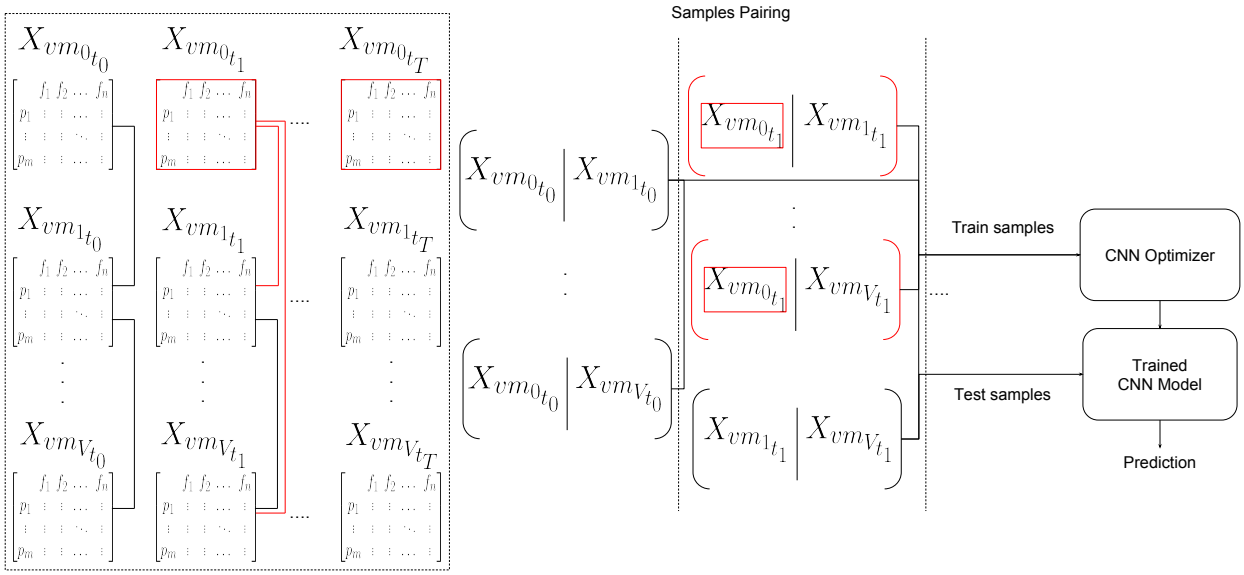


Figure 5.6: Multiple VMs Paired Samples (MVPS)

of our problem by using the same duplicate questions problem concept except, in our case, we are given two samples $X_{vm_{i t_k}}$ and $X_{vm_{j t_k}}$, where $X_{vm_{i t_k}}$ is a 2d matrix (picture in CNN terminology) that belongs to vm_i at time t_k and $X_{vm_{j t_k}}$ is a 2d matrix that belongs to vm_j at the same time t_k. Figure 5.6 shows the approach used for pairing samples. Our goal is to find whether $X_{vm_{i t_k}}$ and $X_{vm_{j t_k}}$ are duplicates (similar). This is done by pairing the two samples as an input to CNN. Two samples are considered similar if they are benign, whereas two samples are considered not similar if either one of them is malicious (red bordered samples are malicious).

By pairing samples, we are actually taking into account the correlations between samples of different VMs at the same time as well as the history of samples (previously seen patterns). The pairing method works in an auto-scaling scenario where there are at least two VMs of the same group. Note that it is important that we only pair samples of the same time as pairing samples of different times might have completely different values. Also, note that we only inject malware in a single VM, so we don't have multiple infected VMs in a single experiment run.

Pairing all samples is a very time consuming operation as the number of samples will be squared. In addition, that will introduce a class imbalance problem since we are only infecting a single VM. Although, practically, this may not be the case in real scenarios, we believe that in-

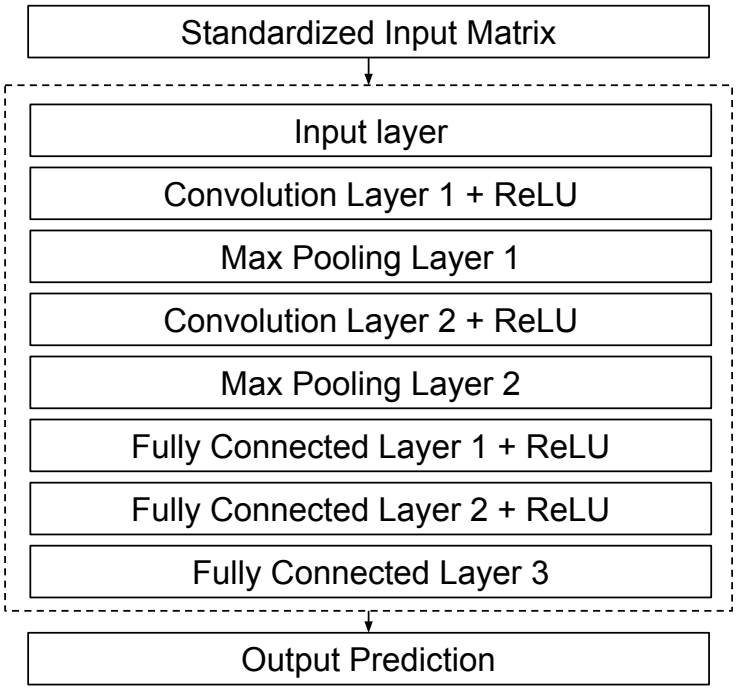


Figure 5.7: CNN Model (LeNet-5)

fecting multiple VMs is hard to occur at the exact same time and that a malware needs sometime to reconnaissance and infect other similarly configured VMs. Consequently, as shown in Figure 5.6, we pair a malicious sample with all benign samples from other machines at a particular time. On the other hand, we pair each benign sample sequentially with a sample from the next VM.

5.4 Experiment Setup and Results

In this section, first, we present the CNN model used in this work. Second, we briefly review our experimental setup. Then, we provide the results of using the MVSS approach. Lastly, we show that using the samples pairing MVPS approach can significantly improve the results.

5.4.1 CNN Model Architecture

A deep CNN model would require considerably larger processing power. In reality, this might not be affordable. For the sake of practicality, we chose to work with a shallow CNN. We show that even a shallow CNN can achieve near optimal results in our pairing approach. Figure 5.7 shows

the CNN model used in this work. We chose LeNet-5 [48] CNN model. Although, it is currently by no means one of the state-of-the-art CNN models, its shallowness makes it one of the best candidates in practice. Note that in the context of online malware detection, the model might need to be trained multiple times based on the deployed workloads in place. For example, a 3-tier web architecture and a Hadoop architecture might need different trained models.

As mentioned in Section 4.3.1, the CNN model receives a standardized 2d matrix. Lenet-5 CNN consists of 7 layers (excluding the input layer). The input layer is a 2d matrix of 120×45 (120×90 for MVPS), representing a sample of maximum 120 processes and 45 features. Empty processes rows are padded with zeros. The first layer is a convolutional layer with 32 kernels of size 5×5 with zero padding ending. This results in 32 feature maps of size 120×45. The second layer is a max pool layer of size 2 $\times$2 in which down size each dimension by a magnitude of 2, resulting in 32 feature maps of size 60×23 (60×45 for MVPS). The third layer, another convolutional layer with 64 kernels of size 60×23, is followed by a max pool layer which results in 64 down sized feature maps of size 30×12 (30×23 for MVPS). The fifth and sixth layers are fully connected layers of size 1024 and 512, respectively. The last layer is another fully connected layer of size 2, representing the prediction class (malicious or benign).

ReLU activation is used after every convolutional and fully connected layer (excluding the last fully connected layer). Adam Optimizer, a stochastic gradient descent with automatic learning rate adaptation, is used to train the model. Adam Optimizer learning rate is a maximum change threshold to control how fast the learning process can be (set to $1e-5$). The optimizer works by minimizing the loss function (mean cross entropy). Random grid search is used to tune the CNN parameters (e.g., mini batch size).

5.4.2 Experimental Setup

Our experiments testbed setup is similar to that of Chapter 4. Our experiments were conducted on Openstack and a 3-tier web architecture[1], with auto-scaling enabled on the web and application server layers. Traffic was generated based on ON/OFF Pareto distribution.

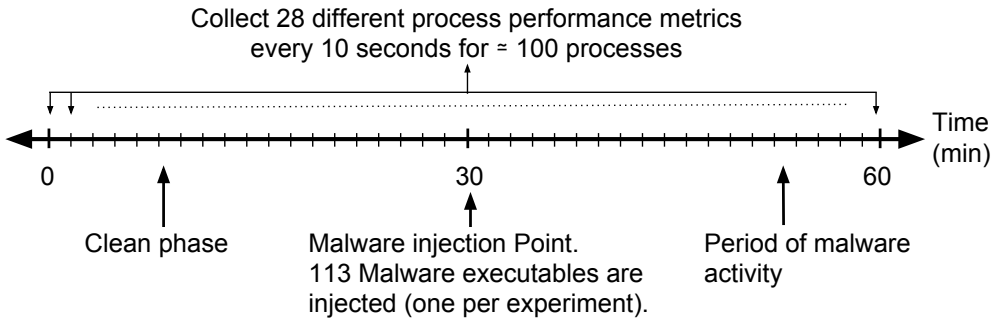


Figure 5.8: Data collection overview

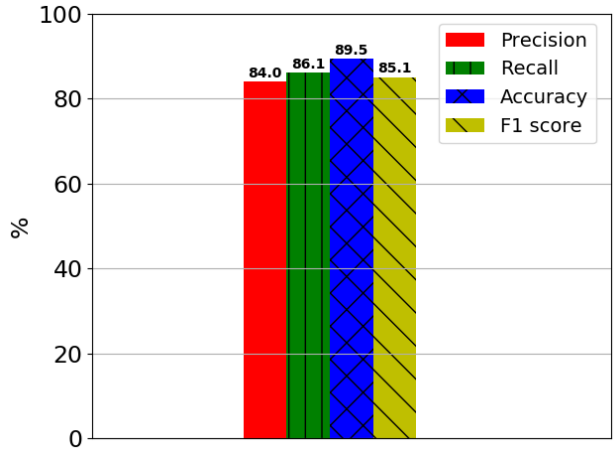


Figure 5.9: Optimized MVSS CNN classifier results

The data collection process is shown in Figure 5.8. Each of our experiments was 1 hour long. The first 30 minutes are the clean phase. The second 30 minutes are malicious phase where a malware is injected. A set of 113 malware were used each for a different experiment. All firewalls were disabled and an internet connection was provided to avoid any hindrance to the malware malicious intentions. Samples were collected at 10 seconds intervals, so during a single experiment 360 samples were collected.

5.4.3 MVSS and MVPS Results

Like most standard machine learning classification problems, data was split into three sets: training (60%), validation (20%) and testing (20%) sets. We split on the number of experiments. The 113 experiments were split to 67, 23 and 23 respectively. This makes sure that validation and testing phases are exposed to unseen malware. After training the model on the training set, validation

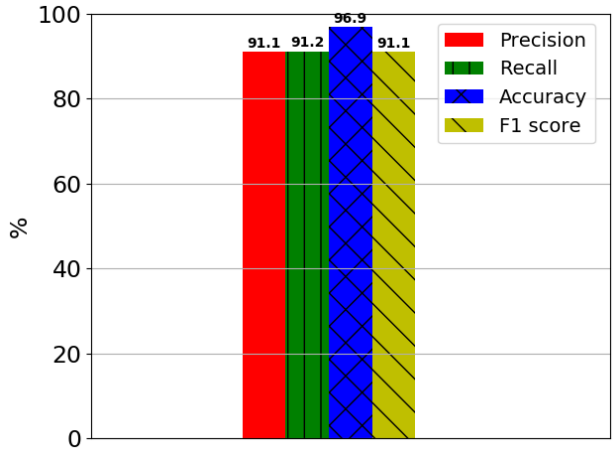


Figure 5.10: Optimized MVPS CNN classifier results

set is used to tune the model parameters as well as choosing the highest accuracy model. The model is evaluated against the validation set after each epoch and the highest accuracy model is chosen. Then the testing set is used to test the chosen model (optimized classifier). For the sake of practicality, we chose a shallow CNN model (LeNet-5) so it can be trained fast with as few epochs as possible.

Figure 5.9 shows the results of MVSS optimized classifier. The optimized classifier yields accuracy of $\simeq 90\%$ while precision, recall and fscore are $\simeq 85\%$ on the test data set. This approach achieved good results compared to the similar simple 2d CNN approach in 4.3.6. There are two reasons for this improvement. First, increasing the number of data (113 malware experiments as opposed to 25). Second, using data from multiple VMs as opposed to a single VM; however, we still had to filter part of the data to balance our data sets (ratio of benign to malicious samples).

Figure 5.10 shows the results of MVPS optimized classifier. There is a significant increase in the four evaluation metrics when compared to the MVSS classifier. The optimized chosen MVPS classifier had a highest accuracy of $\simeq 98.2\%$ during the validation phase. It yielded a $\simeq 96.9\%$ accuracy on the test data set. Fscore, recall and precision all jumped to $\simeq 91\%$ on the test data set. The main reason for this high improvement is that the MVPS approach finds correlations between the multiple VMs running at the same time which is very beneficial in an auto-scaling scenario.

In both cases, mini-batch size of size 64 and learning rate of $1e-5$ yielded the best results.

Note that we don't use a dropout layer (usually used to avoid over-fitting) since it is not useful when dealing with a shallow CNN trained for only a few epochs.

5.5 Conclusion

In this chapter, we introduced an online malware detection approach to leverage the behavior correlation between multiple VMs in an auto-scaling scenario. The approaches introduced used 2d CNN for malware detection. First, we introduced the MVSS method which targets multiple VMs using single individual samples. MVSS achieved good results with an accuracy of $\simeq 90\%$. Then, we introduced MVPS which targets multiple VMs using paired samples. MVPS takes the previous approach a step forward by pairing samples from multiple VMs which helps in finding correlations between the VMs. MVPS showed a considerable improvement over MVSS with an accuracy of $\simeq 96.9\%$. In the future, we plan to use different use case scenarios such as Hadoop and Containers. We also plan to perform an analysis using different CNN models architecture. Finally, we plan to develop techniques to handle the situation when multiple VMs are infected simultaneously by an attacker.

CHAPTER 6: CONCLUSION AND FUTURE WORK

As cloud continues to emerge and offer new concepts and capabilities, new attacks will get advantage of them. In order to keep up with the cycle of threats and mitigation that never ends, we need to leverage these new capabilities for securing the cloud.

The goal of this dissertation is to provide a comprehensive framework for cloud security monitoring by leveraging cloud essential characteristics (e.g., on-demand self-service) and bridge the gap between cloud security and machine learning (ML). We particularly focus on the Infrastructure as a Service (IaaS) layer in the cloud.

6.1 Summary of Contributions

We focused on the auto-scalability feature in the cloud as a starting point for leveraging cloud characteristics. As a result a fundamental assumption is made: *VMs that belong to the same group (e.g. webservers group in a 3-tier web architecture) should behave similarly with only small deviations.* This assumption is true to a great extent since in practice auto-scaling is done based on the same configuration script, thus VMs (intended for the same purpose) that was spawned from the same configuration scripts should behave similarly.

In Chapter 3, we developed an online anomaly detection system for cloud IaaS. It works by clustering the VMs of a single tenant using performance black-box metrics. Black-box features assumes only metrics about the VM as a black-box with no prior knowledge of what's inside a VM. A 3-tier web architecture was used as a use case where there are 3 tiers (groups) of VMs: web servers, application servers and database servers with auto-scaling in place according to VMs' resource usage. We showed that threats like ransomware and EDoS can be effectively detected by inspecting the performance and resource utilization metrics of VMs as a black-box.

Although the approach in Chapter 3 works well with highly active malware (e.g. ransomware), it is not as effective for detecting malware that maintains a low-profile of resource utilization. In Chapter 4, we developed an effective approach for detecting malware by learning behavior from

fine-grained and raw process meta-data that are available directly from the hypervisor. The approach develop is resistant (to great extent) to the aforementioned mislabeling problem. To mitigate this problem, we refine the above assumption by assuming that a malware will show malicious activity within a time window. We demonstrate the effectiveness of this approach by first developing a standard 2d Convolutional Neural Network (CNN) model that does not incorporate the time window, and then comparing it with a newly developed 3d CNN model that significantly improves detection accuracy mainly due to the employment of a time window as the third dimension, thereby mitigating the mislabeling problem. The approach was tested against unseen set of malware and proved to be effective with our 2d CNN model reaches an accuracy of 79% and our 3d CNN model significantly improves the accuracy to 90%.

The aforementioned approach considers a single VM. In order to leverage the auto-scalability characteristic for better cloud security, we extended the previous work by introducing Multiple VMs Single Sample (MVSS) and Multiple VMs Paired Samples (MVPS). MVSS is just a simple extension of the work in Chapter 4 that accepts input from multiple VMs then randomly balance the data set to avoid class imbalance problems. It achieved an accuracy of $\simeq 90\%$. MVPS greatly improved the situation by taking full advantage of auto-scalability. It considers correlations between running VMs by pairing samples of different VMs at a particular time. MVPS achieved an accuracy of $\simeq 96.9\%$.

6.2 Future Directions

There are two directions regarding future research: present day and futuristic research. For present day research, the work done can be extended by applying and testing multiple architectures (e.g., Hadoop systems or containers). It can also include implementing different ML models and doing a thorough comparison between them. Another direction can be investigating more cloud characteristics and leveraging them for cloud security.

Although, present day research is very important as it's more on the practical side, futuristic radical ideas are very challenging and more interesting. Interesting futuristic research can be

summarized in two questions:

1. Can we automatically evaluate a security monitoring system?

Evaluation of cloud security monitoring frameworks or anomaly detection techniques in general is a very difficult and challenging task due to several reason. First, it is always difficult to get data from real world organizations due to privacy issues. Second, even with real world data, labeling the data as malicious, normal, or system failures can be very expensive task since it is always done by human experts. In fact, in most cases it is infeasible because of the cost and the enormous amount of time needed. Third, *concept drift*, which states that the constant change of traffic as well as other environmental changes can not only introduces new types of anomalies (e.g. attacks) but can also change the definition of *"normal"* behavior. Thus, automatic evaluation for anomaly detection techniques is still an open question. Our focus is on developing an automatic evaluation process for cloud specific security monitoring framework. Some work tried to propose semi-solutions. For example, [47] proposed a way of evaluating unlabeled data. The work was done by inspecting user profiles and comparing the activity during an intrusion to the activity during normal use. The drawback of this approach is that it is very specific to a class of attacks where a user is malicious.

This problem can be investigated by defining set of metrics for automatic evaluation. One promising approach is to look at this problem as a reinforcement learning problem. Reinforcement learning problems mostly lies in the area of robotics and computer programs that try to learn playing games. Certain actions are taken by the robot and a reward is given based on these actions in a specific environment. In this domain, it may be easier defining actions (e.g. rules of playing a game) and reward (e.g. win or loss the game). The robot is surely getting a feedback based on these rewards. On the other hand, in our case, defining the actions that a cloud security monitoring can take and rewards based on its actions is a challenging problem. Actions can be, not limited to, killing or suspending a VM or a process inside a particular VM, while rewards can be given based impact these actions had on performance or the distance of data points of particular VM (e.g. an actions caused VM data points to be closer to its cluster's centroid).

2. As systems evolve over time, can we develop security monitoring agents with the ability to adapt and evolve?

The aforementioned *concept drift* is a serious problem with any anomaly detection system as the environment in which the anomaly detection system lies tends to change overtime. What is supposed to be normal data points can change and become anomalies at some point in the future. Some ideas to deal with concept drift were proposed in [88] which relies on using a fixed or dynamic timing window to choose a range of instances and then train a new classifier, while others proposed adaptive anomaly detection (AADs) (e.g. [52]) based on detecting substantial changes within the data and update AADs accordingly. ADDs are promising approaches however the major drawback lies in the built-in forgetting mechanism [58]. They tend to adapt to the new changes and forget about the past. The need for adapting while keeping history intact can be potential solution.

We believe that the answer to this question greatly depends on the first question. If we successfully and practically have a way to automatically evaluate security monitoring agents, we can employ different technique to achieve our goal. Assuming we have an answer to the first question, we can use the evolution concept where we have an initial generation of security monitoring agents (each is slightly different) and they evolve over time. Evolution can be done by using genetic algorithms (GAs) which are mainly used in optimizations problems. In a GA, an initial population is chosen and individuals (in our case security monitoring agents) evolve in the next generation. Deciding which individuals should survive in the next generation is based on a defined *Fitness* function. The automatic evaluation process can be used as a fitness function. The next generation of agents is selected based on concepts like: mutation (where a subset of an agent's chromosomes are randomly changed), crossover (where an agent chromosomes are constructed by taking more than one parent chromosomes), and selection (where an agent is selected to live as it is in the next generation). This approach allows us to overcome the built-in mechanism problem because some agents in the population will keep the old chromosomes intact while being selected to the next generations. It also allows us to have multiple agents, where each agent is specialized in a particular anomaly class/type, and those agents can evolve and adapt to new unseen anomalies. Our ultimate

goal is to end up with a number of security monitoring agents where each agent specialize in a particular anomaly class (e.g. malware, ransomware, system failures or DDoS).

BIBLIOGRAPHY

[1] Mahmoud Abdelsalam, Ram Krishnan, and Ravi Sandhu. Clustering-based IaaS cloud monitoring. In *10th IEEE CLOUD*. IEEE, 2017.

[2] Tony Abou-Assaleh and et al. N-gram-based detection of new malicious code. In *COMPSAC*, volume 2. IEEE, 2004.

[3] Giuseppe Aceto, Alessio Botta, Walter De Donato, and Antonio Pescapè. Cloud monitoring: A survey. *Computer Networks*, 57:2093–2115, 2013.

[4] Rakshit Agrawal, Jack W Stokes, Mady Marinescu, and Karthik Selvaraj. Robust neural malware detection models for emulation sequence learning. *arXiv preprint arXiv:1806.10741*, 2018.

[5] Mamoun Alazab, Sitalakshmi Venkatraman, Paul Watters, and Moutaz Alazab. Zero-day malware detection based on supervised learning algorithms of api call signatures. In *Proceedings of the Ninth Australasian Data Mining Conference-Volume 121*, pages 171–182. Australian Computer Society, Inc., 2011.

[6] Ben Athiwaratkun and Jack W Stokes. Malware classification with LSTM and GRU language models and a character-level cnn. In *ICASSP*. IEEE, 2017.

[7] Fatemeh Azmandian, Micha Moffie, Malak Alshawabkeh, Jennifer Dy, Javed Aslam, and David Kaeli. Virtual machine monitor-based lightweight intrusion detection. *ACM SIGOPS Operating Systems Review*, 45, 2011.

[8] Pierre Baldi and Peter J Sadowski. Understanding dropout. In *Advances in Neural Information Processing Systems*, 2013.

[9] Asa Ben-Hur, David Horn, Hava T Siegelmann, and Vladimir Vapnik. Support vector clustering. *Journal of machine learning research*, 2:125–137, 2001.

[10] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13, 2012.

[11] Pavel Berkhin. A survey of clustering data mining techniques. In *Grouping multidimensional data*, pages 25–71. Springer, 2006.

[12] Bernhard E Boser, Isabelle M Guyon, and Vladimir N Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 144–152. ACM, 1992.

[13] Christopher JC Burges. A tutorial on support vector machines for pattern recognition. *Data mining and knowledge discovery*, 2:121–167, 1998.

[14] Claudia Canali and Riccardo Lancellotti. Automated clustering of virtual machines based on correlation of resource usage. *Communications Software and Systems*, 8:102–109, 2012.

[15] Claudia Canali and Riccardo Lancellotti. Automated clustering of vms for scalable cloud monitoring and management. In *Software, 20th SoftCOM, 2012*, pages 1–5. IEEE, 2012.

[16] Claudia Canali and Riccardo Lancellotti. Automatic virtual machine clustering based on bhattacharyya distance for multi-cloud systems. In *Proc. of MultiCloud*, pages 45–52. ACM, 2013.

[17] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41:15, 2009.

[18] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20:273–297, 1995.

[19] Koby Crammer and Yoram Singer. On the learnability and design of output codes for multiclass problems. *Machine learning*, 47:201–233, 2002.

[20] Mark E Crovella and Azer Bestavros. Self-similarity in world wide web traffic: Evidence and possible causes. *IEEE/ACM Transactions on networking*, 5, 1997.

[21] Kamal Dahbur, Bassil Mohammad, and Ahmad Bisher Tarakji. A survey of risks, threats and vulnerabilities in cloud computing. In *ISWSA*, 2011.

[22] George E Dahl, Jack W Stokes, Li Deng, and Dong Yu. Large-scale malware classification using random projections and neural networks. In *ICASSP*. IEEE, 2013.

[23] Joel A Dawson, Jeffrey T McDonald, Lee Hively, Todd R Andel, Mark Yampolskiy, and Charles Hubbard. Phase space detection of virtual machine cyber events through hypervisor-level system call analysis. In *Data Intelligence and Security (ICDIS), 2018 1st International Conference on*, pages 159–167. IEEE, 2018.

[24] John Demme and et al. On the feasibility of online malware detection with performance counters. In *ACM SIGARCH Computer Architecture News*, volume 41. ACM, 2013.

[25] Gianluca Dini, Fabio Martinelli, Andrea Saracino, and Daniele Sgandurra. Madam: a multi-level anomaly detector for android malware. In *International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security*, pages 240–253. Springer, 2012.

[26] Kai-Bo Duan and S Sathiya Keerthi. Which is the best multiclass svm method? an empirical study. In *International Workshop on Multiple Classifier Systems*, pages 278–285. Springer, 2005.

[27] Mojtaba Eskandari and Sattar Hashemi. Ecfgm: enriched control flow graph miner for unknown vicious infected code detection. *Journal in Computer Virology*, 8:99–108, 2012.

[28] Mojtaba Eskandari and Sattar Hashemi. A graph mining approach for detecting unknown malwares. *Journal of Visual Languages & Computing*, 23:154–162, 2012.

[29] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.

[30] Ian Foster, Yong Zhao, Ioan Raicu, and Shiyong Lu. Cloud computing and grid computing 360-degree compared. In *Grid Computing Environments Workshop, 2008. GCE'08*, pages 1–10. Ieee, 2008.

[31] Ali Gholami and Erwin Laure. Security and privacy of sensitive data in cloud computing: a survey of recent developments. *arXiv preprint arXiv:1601.01498*, 2016.

[32] Daniel Gonzales, Jeremy Kaplan, Evan Saltzman, Zev Winkelman, and Dulani Woods. Cloud-trust-a security assessment model for infrastructure as a service (iaas) clouds. 2015.

[33] Bernd Grobauer, Tobias Walloschek, and Elmar Stocker. Understanding cloud computing vulnerabilities. *IEEE Security & Privacy*, 9, 2011.

[34] Nils Gruschka and Meiko Jensen. Attack surfaces: A taxonomy for attacks on cloud services. In *IEEE CLOUD*, pages 276–279, 2010.

[35] Geoffrey E Hinton and et al. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[36] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2:359–366, 1989.

[37] Wenjie Hu, Yihua Liao, and V Rao Vemuri. Robust support vector machines for anomaly detection in computer security. In *ICMLA*, pages 168–174, 2003.

[38] Wenyi Huang and Jack W Stokes. MtNet: a multi-task neural network for dynamic malware classification. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016.

[39] Meiko Jensen, Jörg Schwenk, Nils Gruschka, and Luigi Lo Iacono. On technical security issues in cloud computing. In *IEEE CLOUD*, 2009.

[40] Zeliang Kan, Haoyu Wang, Guoai Xu, Yao Guo, and Xiangqun Chen. Towards light-weight deep learning based malware detection. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, pages 600–609. IEEE, 2018.

[41] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.

[42] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[43] Dhilung Kirat and Giovanni Vigna. Malgene: Automatic extraction of malware analysis evasion signature. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 769–780. ACM, 2015.

[44] Teuvo Kohonen. The self-organizing map. *Neurocomputing*, 21:1–6, 1998.

[45] J Zico Kolter and Marcus A Maloof. Learning to detect and classify malicious executables in the wild. *Journal of Machine Learning Research*, 7, 2006.

[46] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[47] Terran Lane, Carla E Brodley, et al. Sequence matching and learning in anomaly detection for computer security. In *AAAI Workshop: AI Approaches to Fraud Detection and Risk Management*, pages 43–49, 1997.

[48] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[49] Will E Leland, Murad S Taqqu, Walter Willinger, and Daniel V Wilson. On the self-similar nature of ethernet traffic (extended version). *IEEE/ACM Transactions on networking*, 2, 1994.

[50] Hung-Jen Liao, Chun-Hung Richard Lin, Ying-Chih Lin, and Kuang-Yuan Tung. Intrusion detection system: A comprehensive review. *Journal of Network and Computer Applications*, 36:16–24, 2013.

[51] T Warren Liao. Clustering of time series data a survey. *Pattern recognition*, 38:1857–1874, 2005.

[52] Yihua Liao, V Rao Vemuri, and Alejandro Pasos. Adaptive anomaly detection with evolving connectionist systems. *Journal of Network and Computer Applications*, 30:60–80, 2007.

[53] Patrick Luckett, J Todd McDonald, and Joel Dawson. Neural network analysis of system call timing for rootkit detection. In *2016 Cybersecurity Symposium (CYBERSEC)*, pages 1–6. IEEE, 2016.

[54] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proc. of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA., 1967.

[55] Oded Maimon and Lior Rokach. *Data Mining and Knowledge Discovery Handbook*, volume 2. Springer, 2005.

[56] Peter Mell, Tim Grance, et al. The nist definition of cloud computing. 2011.

[57] Jesús Montes, Alberto Sánchez, Bunjamin Memishi, María S Pérez, and Gabriel Antoniu. Gmone: A complete approach to cloud monitoring. *Future Generation Computer Systems*, 29:2026–2040, 2013.

[58] Michael D Muhlbaier and Robi Polikar. An ensemble approach for incremental learning in nonstationary environments. In *International Workshop on Multiple Classifier Systems*, pages 490–500. Springer, 2007.

[59] Srinivas Mukkamala, Guadalupe Janoski, and Andrew Sung. Intrusion detection using neural networks and support vector machines. In *Neural Networks, 2002. IJCNN'02. Proceedings of the 2002 International Joint Conference on*, volume 2, pages 1702–1707. IEEE, 2002.

[60] Thomas Dyhre Nielsen and Finn Verner Jensen. *Bayesian networks and decision graphs*. Springer Science & Business Media, 2009.

[61] Meltem Ozsoy, Caleb Donovick, Iakov Gorelik, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Malware-aware processors: A framework for efficient online malware detection. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 651–661. IEEE, 2015.

[62] Husanbir S Pannu, Jianguo Liu, and Song Fu. Aad: Adaptive anomaly detection system for cloud computing infrastructures. In *Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium on*, pages 396–397. IEEE, 2012.

[63] Duc Truong Pham, Stefan S Dimov, and CD Nguyen. Selection of k in k-means clustering. *Proc. of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science*, 219:103–119, 2005.

[64] Radu S Pirscoveanu, Steven S Hansen, Thor MT Larsen, Matija Stevanovic, Jens Myrup Pedersen, and Alexandre Czech. Analysis of malware behavior: Type classification using machine learning. In *Cyber Situational Awareness, Data Analytics and Assessment (CyberSA), 2015 International Conference on*, pages 1–7. IEEE, 2015.

[65] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1:81–106, 1986.

[66] J Ross Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014.

[67] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proc. of the 16th ACM CCS*, pages 199–212. ACM, 2009.

[68] Salvatore Ruggieri. Efficient c4. 5 [classification algorithm]. *IEEE transactions on knowledge and data engineering*, 14:438–444, 2002.

[69] S Rasoul Safavian and David Landgrebe. A survey of decision tree classifier methodology. *IEEE transactions on systems, man, and cybernetics*, 21:660–674, 1991.

[70] Joshua Saxe and Konstantin Berlin. Deep neural network based malware detection using two dimensional binary program features. In *10th MALWARE*. IEEE, 2015.

[71] Bernhard Schölkopf, John C Platt, John Shawe-Taylor, Alex J Smola, and Robert C Williamson. Estimating the support of a high-dimensional distribution. *Neural computation*, 13:1443–1471, 2001.

[72] Seonhee Seok and Howon Kim. Visualized malware classification based-on convolutional neural network. *Journal of the Korea Institute of Information Security and Cryptology*, 26, 2016.

[73] Asaf Shabtai and et al. Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey. *information security technical report*, 14, 2009.

[74] Taeshik Shon and Jongsub Moon. A hybrid machine learning approach to network anomaly detection. *Information Sciences*, 177:3799–3821, 2007.

[75] Gaurav Somani, Manoj Singh Gaur, and Dheeraj Sanghi. Ddos/edos attack in cloud: affecting everyone out there! In *Proc. of the 8th SIN*, pages 169–176. ACM, 2015.

[76] Qing Song, Wenjie Hu, and Wenfang Xie. Robust support vector machine with bullet hole image classification. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 32:440–448, 2002.

[77] Andrew H Sung and Srinivas Mukkamala. Identifying important features for intrusion detection using support vector machines and neural networks. In *Applications and the Internet, 2003. Proceedings. 2003 Symposium on*, pages 209–216. IEEE, 2003.

[78] Johan AK Suykens and Joos Vandewalle. Least squares support vector machine classifiers. *Neural processing letters*, 9:293–300, 1999.

[79] Gil Tahan, Lior Rokach, and Yuval Shahar. Mal-ID: Automatic malware detection using common segment analysis and meta-features. *Journal of Machine Learning Research*, 13, 2012.

[80] Shun Tobiyama, Yukiko Yamaguchi, Hajime Shimada, Tomonori Ikuse, and Takeshi Yagi. Malware detection with deep neural network using process behavior. In *COMPSAC*, volume 2. IEEE, 2016.

[81] Luis M Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, 39:50–55, 2008.

[82] Antoine Varet and Nicolas Larrieu. Realistic network traffic profile generation: Theory and practice. *Computer and Information Science*, 7, 2014.

[83] Li Wan, Matthew Zeiler, Sixin Zhang, Yann L Cun, and Rob Fergus. Regularization of neural networks using dropconnect. In *Proceedings of the 30th international conference on machine learning (ICML-13)*, 2013.

[84] Chengwei Wang. Ebat: online methods for detecting utility cloud anomalies. In *Proceedings of the 6th Middleware Doctoral Symposium*, page 4. ACM, 2009.

[85] Chengwei Wang, Vanish Talwar, Karsten Schwan, and Parthasarathy Ranganathan. Online detection of utility cloud anomalies using metric distributions. In *2010 IEEE NOMS 2010*, pages 96–103. IEEE, 2010.

[86] Chengwei Wang, Krishnamurthy Viswanathan, Lakshminarayan Choudur, Vanish Talwar, Wade Satterfield, and Karsten Schwan. Statistical techniques for online anomaly detection in data centers. In *12th IFIP/IEEE IM 2011*, pages 385–392. IEEE, 2011.

[87] Michael R Watson and et al. Malware detection in cloud computing infrastructures. *IEEE Transactions on Dependable and Secure Computing*, 13, 2016.

[88] Gerhard Widmer and Miroslav Kubat. Learning in the presence of concept drift and hidden contexts. *Machine learning*, 23:69–101, 1996.

[89] Michael Wilson. A historical view of network traffic models. *Unpublished survey paper. See http://www. arl. wustl. edu/ mlw2/classpubs/traffic models*, 2006.

[90] Zhifeng Xiao and Yang Xiao. Security and privacy in cloud computing. *IEEE Communications Surveys & Tutorials*, 15, 2013.

[91] Zhixing Xu, Sayak Ray, Pramod Subramanyan, and Sharad Malik. Malware detection using machine learning based analysis of virtual memory access patterns. In *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2017.

VITA

Mahmoud Abdelsalam was born in February, 1992 in Alexandria, Egypt. He received his B.Sc degree with a major in Computer Science from the Arab Academy for Science and Technology and Maritime Transport (AASTMT), Egypt. He subsequently entered the doctoral program in the Department of Computer Science at the University of Texas at San Antonio (UTSA) in Spring 2014. He joined the Institute for Cyber Security (ICS) at UTSA in Summer 2014 and started working with Prof. Ravi Sandhu and Dr. Ram Krishnan since 2015. He received his interim M.Sc degree in 2017. His research interests include security in cloud infrastructures. In particular, his focus is on developing cloud-specific online malware detection methods.