Securing Cloud Containers through Intrusion Detection and Remediation

Amr Sayed Omar Abed

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

T. Charles Clancy, Chair
Jeffrey H. Reed
Yaling Yang
Hesham A. Rakha
Mohamed M. Azab

August 1, 2017
Blacksburg, Virginia

Keywords: Security in Cloud Computing, Deep Learning, Intrusion Detection
Container Security, Behavior Modeling, Anomaly Detection
Copyright 2017, Amr S. Abed

Securing Cloud Containers through Intrusion Detection and Remediation

Amr Sayed Omar Abed

ABSTRACT

Linux containers are gaining increasing traction in both individual and industrial use. As these containers get integrated into mission-critical systems, real-time detection of malicious cyber attacks becomes a critical operational requirement. However, a little research has been conducted in this area.

This research introduces an anomaly-based intrusion detection and remediation system for container-based clouds. The introduced system monitors system calls between the container and the host server to passively detect malfeasance against applications running in cloud containers.

We started by applying a basic memory-based machine learning technique to model the container behavior. The same technique was also extended to learn the behavior of a distributed application running in a number of cloud-based containers. In addition to monitoring the behavior of each container independently, the system used prior knowledge for a more informed detection system.

We then studied the feasibility and effectiveness of applying a more sophisticated deep learning technique to the same problem. We used a recurrent neural network to model the container behavior.

We evaluated the system using a typical web application hosted in two containers, one for the front-end web server, and one for the back-end database server. The system has shown promising results for both of the machine learning techniques used.

Finally, we describe a number of incident handling and remediation techniques to be applied upon attack detection.

This work was partially funded by Northrop Grumman Corporation via a partnership agreement through S2ERC; an NSF Industry/University Cooperative Research Center.

Securing Cloud Containers through Intrusion Detection and Remediation

Amr Sayed Omar Abed

GENERAL AUDIENCE ABSTRACT

Cloud computing plays an important role in our daily lives today. Most of the online services and applications we use are hosted in a cloud environment. Examples include email, cloud storage, online booking systems, and many websites. Typically, a cloud environment would host many of those applications on a single host to maximize efficiency and minimize overhead. To achieve that, cloud service providers, such as Amazon Web Services and Google Cloud Platform, rely on virtual encapsulation environments, such as virtual machines and containers, to encapsulate and isolate applications from other applications running in the cloud.

One major concern usually raised when discussing cloud applications is the security of the application and the privacy of the data it handles, e.g. the files stored by the end users on their cloud storage. In addition to firewalls and traditional security measures that attempt to prevent an attack from affecting the application, intrusion detection systems (IDS) are usually used to detect when an application is affected by a successful attack that managed to escape the firewall. Many intrusion detection systems have been introduced to cloud applications using virtual machines, but almost none has been introduced to applications running in containers.

In this dissertation, we introduce an intrusion detection system to be deployed by cloud service providers to container-based cloud environments. The system uses machine learning techniques to learn the behavior of the application running in the container and detect when the behavior changes as an indication for a potential attack. Upon detection of the attack, the system applies one of three defense mechanisms to restore the running application to a safe state.

To my family

Acknowledgments

I would like to express my deepest appreciation to my advisor, Dr. Charles Clancy, for his persistent help and guidance, and for providing the perfect atmosphere to do my research. This dissertation would not have been possible without his continuous support.

Special thanks to Dr. David Levy, former associate director of the Hume Center, for initiating the research efforts that led to this dissertation. I would also like to thank my committee members, especially Dr. Mohamed Azab and Dr. Jeffrey Reed, for providing advice that helped guide my research in the right direction. I would also like to thank my former advisor, Dr. Jules White, and my interim advisor, Dr. Binoy Ravindran, for everything I learned from them.

I am grateful to members of the amazing team of the Hume Center. They have always been responsive and supportive whenever I needed their help. Special thanks to Leslie Sullivan, Deanna Clark, Kellye Richardson, Christie Thompson, Janet Murphy, Ryan Chase, Michael Fowler, and Dr. Kira Gantt.

I am also grateful to Dr. Sedki Riad for launching the VT-MENA program, which made it possible to pursue my Ph.D. at Virginia Tech. I also cannot forget Cynthia Hopkins and Ruth Athanson whose advising and assistance with administrative tasks made it easier to advance through my studies.

Finally, I cannot thank enough my parents and sisters for always being there for me, providing wise advice and emotional support when I needed it most. A special thank you to my wife, Manar, and my lovely daughter, Salma. I would not have made it without them.

Contents

1 Introduction **1**

1.1 Motivation . . . 2

1.1.1 Security in Cloud Computing . . . 2

1.1.2 Linux Containers as the Future of Cloud Virtualization . . . 3

1.1.3 Vulnerabilities of Linux Containers . . . 4

1.2 System Overview . . . 4

1.2.1 Mathematical Model . . . 4

1.2.2 Assumptions and Delimitations . . . 6

1.3 Research Objectives and Contributions . . . 7

1.3.1 Intrusion Detection for Linux Containers . . . 7

1.3.2 Applying Deep Learning for Intrusion Detection . . . 7

1.3.3 Resiliency of Container-based Applications . . . 8

2 Linux Containers **9**

2.1 Linux Containers versus Virtual Machines . . . 10

2.2 History of Linux Containers . . . 11

2.2.1 The chroot system call . . . 12

2.2.2 FreeBSD Jails & Linux VServer . . . 12

2.2.3 Solaris Zones . . . 12

2.2.4 OpenVZ . . . 13

2.2.5 Process Containers & Control Groups . . . 13

2.2.6 LXC . . . 13
2.2.7 Warden . . . 13
2.2.8 LMCTFY . . . 13
2.2.9 Docker & Rocket . . . 13
2.3 Security of Linux Containers . . . 14
2.3.1 Overview of Docker Container Security . . . 14
2.3.2 Current Efforts for Securing Linux Containers . . . 15

3 Intrusion Detection in the Cloud **16**
3.1 Taxonomy of Intrusion Detection Systems . . . 16
3.2 Evaluation of Intrusion Detection Systems . . . 17
3.2.1 True Positives, True Negatives, and False Positives . . . 17
3.2.2 Accuracy, Precision, and F-score . . . 17
3.2.3 Top-P accuracy . . . 18
3.3 Cloud-based Intrusion Detection Systems . . . 18
3.3.1 Monitoring System Calls for Intrusion Detection . . . 19
3.3.2 Monitoring Network Traffic for Intrusion Detection . . . 19
3.3.3 Monitoring Program Execution for Intrusion Detection . . . 20
3.3.4 Monitoring System Resources for Intrusion Detection . . . 20

4 Deep Learning **22**
4.1 Recurrent Neural Networks . . . 22
4.1.1 Unrolling RNN . . . 23
4.2 Long Short-Term Memory . . . 24
4.3 RNN Hyperparameter Selection . . . 25
4.3.1 Loss Function . . . 26
4.3.2 Learning Rate . . . 26
4.3.3 Batch Size . . . 26
4.3.4 Number of Training Iterations . . . 27

4.3.5 Momentum . . . 27
4.3.6 Number of Hidden Units . . . 27
4.3.7 Regularization Coefficient . . . 27
4.3.8 Activation Functions . . . 28
4.3.9 Initial Weights . . . 28
4.4 Using Deep Learning for Anomaly Detection . . . 28
4.4.1 Anomaly Detection using Recurrent Neural Networks . . . 28
4.4.2 Anomaly Detection using Autoencoders . . . 29

5 Memory-based Behavior Modeling **31**
5.1 System Call Monitoring . . . 31
5.2 Parsing Data from Behavior Log Files . . . 33
5.3 Real-Time Behavior Learning and Classification . . . 34
5.4 Evaluation using a single-server Application . . . 36
5.4.1 Generating normal workload . . . 36
5.4.2 Simulating malicious behavior . . . 37
5.4.3 Evaluation Metrics and Parameters . . . 37
5.4.4 Preliminary Results . . . 38

6 Collaborative Behavior Modeling for Distributed Applications **40**
6.1 Cloud Test Environment . . . 40
6.1.1 Environment Setup . . . 40
6.1.2 Network Configuration . . . 41
6.2 Utilizing Prior Knowledge for Enhancing Accuracy . . . 42
6.2.1 Comparing Container Behavior . . . 42
6.2.2 Aggregating databases of similar containers . . . 43

7 LSTM-based Behavior Modeling **45**
7.1 Network Configuration and Hyper-parameters . . . 45
7.2 Dataset Preparation . . . 49

7.3 Container Behavior Modeling 50

8 Test Case: Detecting Intrusion in SaaS Cloud Application 53

8.1 Cloud Test Environment 53

8.2 Threat Model 53

8.3 Experimental Results 55

8.3.1 Behavior Modeling and Classification using BoSC 55

8.3.2 Behavior Modeling and Classification using LSTM 58

8.4 Discussion 61

8.4.1 Memory-based versus Deep Learning Behavior Modeling 61

8.4.2 Comparing to other Cloud IDS 63

9 Incident Handling and Remediation 65

9.1 Checkpoint and Restore 66

9.2 Live Migration 67

9.3 Replication and Isolation 68

9.4 Prey-and-Predator Model 69

9.4.1 Mathematical Model 70

9.4.2 Simulation Scenario 71

9.4.3 Simulation Results 73

10 Conclusion and Future Directions 78

10.1 System Limitations 79

10.2 Future Research Directions 79

Publications 79

Bibliography 81

List of Figures

1.1 Main Modules of the Intrusion Detection and Remediation System 2

1.2 Simplified System Architecture and Logic Flow Diagram 5

2.1 Containers provide encapsulation environment for applications and dependencies 9

2.2 Containers provide a significantly more efficient alternative to virtual machines 10

2.3 History of Linux Containers 12

4.1 A sample fully-connected Recurrent Neural Network with one hidden layer . 23

4.2 Unfolded Recurrent Neural Network for better visualization 24

4.3 Architecture of a LSTM memory cell and gate units 25

5.1 Memory-based Behavior Modeling and Classification - Logic Flow Diagram and Architecture 32

5.2 Linux containers communicate with host kernel through system calls 33

5.3 Single-server Test Environment for the Intrusion Detection System 36

5.4 For constant detection threshold of 10 mismatches per epoch, the TPR is constant at 100%, while the FPR increases with the epoch size 39

5.5 For constant epoch size of 1000 system calls, raising the detection threshold helps decrease the FPR at the expense of increasing the TPR 39

6.1 Cloud Test Environment - Ambari cluster in Docker containers running on AWS EC2 instances with RHIDS installed on Ubuuntu OS 41

7.1 Logic Flow and System Architecture of the LSTM-based Intrusion Detection System 46

7.2 Data Preparation: raw data is indexed into an array D then reshaped into batch dispenser matrix B which then sliced into input and output matrices X_i and Y_i respectively 50
7.3 Learning rate decays by a factor of 2 every epoch after 4th epoch with initial value of 0.01 51
7.4 Training and validation perplexities while modeling container behavior using LSTM 52

8.1 A web application consisting of MySQL server and PHP server in two containers tested using Kali Linux running in a remote container 54
8.2 ROC curves for three different epoch sizes using BoSC 56
8.3 Accuracy measures for different values of detection threshold when using BoSC 57
8.4 Test perplexity for portion of the input trace 58
8.5 For normal ranges, the perplexity rapidly drops down to less than t_d 59
8.6 For malicious ranges, the perplexity typically exceeds t_d during the attack . . 59
8.7 ROC curve for using LSTM for behavior modeling and classification 60
8.8 Accuracy measures for different values of detection threshold when using LSTM 61
8.9 Comparing accuracy of BoSC and LSTM behavior modeling and classification 62

9.1 Checkpoint and Restore of a misbehaving container 67
9.2 Migration process 68
9.3 Replication and Isolation of Misbehaving containers 69
9.4 Survival probability of a static/mobile container at different number of hacked hosts (N) 74
9.5 Survival probability of a targeted mobile container at 7 hacked hosts and various attack detection times 74
9.6 Survival probability of a targeted static container at 7 hacked hosts and various attack detection times 75
9.7 Survival probability of a static/mobile targeted container at various increasing rates of logistic attack growth 76
9.8 Survival probability of a mobile targeted container at logistic attack growth with various attack detection time 76

9.9 Survival probability of a mobile targeted data container with different numbers of victim hosts (V) 77

List of Tables

2.1 Main differences between Linux containers and virtual machines 11

3.1 Summary of Cloud-based IDSs and their limitations 21

4.1 Parameters of 6 datasets used for anomaly detection using autoencoders . . . 30
4.2 Results for using Autoencoder network with one hidden layer on 6 datasets . 30

5.1 Sample entries of the Syscall-Index Lookup Table 33
5.2 Example of system call parsing 34
5.3 Sample entries of the Normal Behavior Database 35
5.4 Parameters used for automatic Load generation 37

6.1 Similarity matrix of 1 server and 2 agent containers 43
6.2 Similarity between aggregate database and original databases 43

7.1 Hyperparameters used for modeling container behavior using RNN 47

8.1 Accuracy measures for select values of epoch size and detection threshold . . 57
8.2 Accuracy measures for select values of epoch size and detection threshold . . 60
8.3 Comparing performance and accuracy of using LSTM versus BoSC 62
8.4 Summary of reported accuracy measures of Cloud IDS compared to our system 64

9.1 Prey-and-Predator Simulation Parameters 73

Chapter 1

Introduction

The evolution of cloud service providers (CSP), such as Amazon Web Services, Google Cloud Platform, and Microsoft Azure has encouraged many corporates and software startups to shift their business to the cloud. For the sake of portability, isolation, and fast deployment, a growing number of those industries choose to use Linux containers to encapsulate their applications for cloud deployment.

In this research, we present an anomaly-based intrusion detection and remediation system designed for container-based cloud environments. As shown in figure 1.1, the system consists of four main modules, namely monitoring, behavior modeling, classification, and remediation modules.

We started by applying a basic memory-based machine learning technique to learn the behavior of an application running in a standalone Linux container. The introduced system monitors system calls between the container and the host server for anomaly-detection based intrusion detection. The system was evaluated using a single-server database application.

The same technique was then extended to learn the behavior of a distributed application running within a number of cloud-based containers. The system started by remotely deploying the application to a number of containers running on a commercial cloud environment, monitoring the behavior of each container independently, and then by using prior knowledge for more informed detection system. For example, by comparing behavior of containers with similar applications, a shift in the behavior of one container from the other can indicate an intrusion that may not be detected by monitoring that single container. In addition, by aggregating normal-behavior databases from similar containers, false alarms can be reduced.

Moving forward, we studied the feasibility and effectiveness of applying a more sophisticated learning technique to the same problem. We used a Long Short-Term memory (LSTM) based Recurrent Neural Network (RNN) to learn the behavior of the containers. The system still relies on system calls only for behavior learning, and was tested using a typical web application hosted in two containers, one with a back-end MySQL server, and the other with

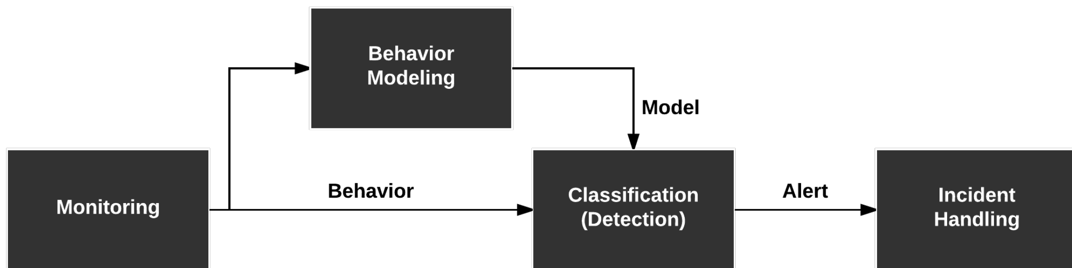

Figure 1.1: Main Modules of the Intrusion Detection and Remediation System

a front-end Apache webserver.

Finally, we discuss the integration of a number of resiliency techniques into the system as remediation techniques upon attack detection.

While this research mainly focuses on cloud containers, the same concepts introduced in this research can be simply tailored to support any other Linux-based system. Examples of such systems include, but not limited to, Internet of Things (IoT) devices, Software Defined Radios (SDR), Software Defined Networks (SDN), Autonomous Underwater Vehicle (AUV) controllers, and Unmanned Aerial Systems (UAS).

1.1 Motivation

Many companies and governments have favored to rely on cloud service providers (CSP) to maintain their online businesses rather than locally monitoring and maintaining them. Many more companies are planning to move to the cloud in 2017 as per a recent survey [1].

1.1.1 Security in Cloud Computing

While some characteristics of cloud computing, such as rapid elasticity and measured service, offers an optimized cost-effective service to the customers, other characteristics may raise security concerns.

One example is the on-demand self-service characteristic, which means that the user can provision computing capabilities automatically as needed without interaction with the cloud service provider (CSP). That is usually provided to the user through web service and management interfaces, such as the Amazon Web Service (AWS) command line interface (CLI), which raises the probability of unauthorized access to the management interface [2].

Another example is the multi-tenancy characteristic raises the concern of the data and pro-

cess visibility to other customers. Having their applications deal with highly sensitive data of their customers, corporations moving to the cloud need to make sure that the new environment is safe and secure for their applications and data. Moreover, knowing that their container-packed applications are typically expected to share the host operating system with applications from other customers further raises their security concern.

In such a multi-tenancy environment, the CSP hence needs to provide a level of trust in guaranteeing no malicious activity occurs on the host kernel or the guest applications that impacts their security.

The CSP is entitled by contractual means to monitor the behavior of containers running on the host kernel to provide safe environment for all hosted containers, and to protect the host kernel itself from the attack of a malicious container. However, providing information about the nature of the application running in the container or altering the container for monitoring purposes is usually undesirable, and more often impermissible, especially when critical applications are running inside the container. Such constraints mandate the use of an intrusion detection system (IDS) that does not interfere with the container structure or application.

1.1.2 Linux Containers as the Future of Cloud Virtualization

Linux containers, especially Docker [3], are also gaining increasing traction in that area as many CSPs and third-party tools are now providing native support of Docker containers. Examples include Amazon EC2 Container Service (ECS), Google Kubernetes, and Apache Mesos. Such tools and platforms provide a layer of resource management and scheduling over Docker. Moreover, recent release of Docker Enterprise Edition (EE) [1] would encourage more companies to adopt Docker containers when deploying their applications to the cloud.

In their 2014 whitepaper (Linux containers: Why they're in your future and what needs to happen first) [4], Cisco and RedHat compare Linux containers to hypervisors and conclude that "the time for Linux containers has arrived". They attribute that to a number of reasons including the lower cost of operation, improved resource utilization, faster provisioning, and greater application mobility.

A recent study [5] by Julian et. al. suggested that Docker containers are now ready for wider adoption in High Performance Computing (HPC) environments traditionally installed on bare metal and/or virtual machine environments.

[1] Docker EE - `https://www.docker.com/enterprise-edition`

1.1.3 Vulnerabilities of Linux Containers

As discussed in more detail in chapter 2, while Docker containers are "fairly secure" even with default configurations [6], some Docker options introduce security vulnerabilities.

With the use of control groups and security profiles applied to containers, the attack surface can be minimized [7]. However, attacks on mission-critical applications running within the container can still occur and can represent an attack vector to the host kernel itself [7]. As a result, understanding when the container has been compromised is of key interest, yet little research has been conducted in this area.

1.2 System Overview

In our system, we rely on the fact that Linux containers communicate with the host kernel and the outer world via system calls issued by the processes running within the container to the host kernel. Figure 1.2 summarizes the system operation.

To learn the behavior of the running container, we monitor those system calls logging them into a behavior log file to be processed by the behavior modeling module of the system. To monitor the system calls, we are using the strace-docker[2] background service running on the host. The service relies internally on Sysdig; an open-source tool that has native support for monitoring different aspects of Docker containers.

For the behavior modeling and classification modules, we are using two different versions deploying techniques from both ends of the machine learning spectrum. The first technique is the Bag of System Calls (BoSC) technique [8] which is a basic memory-based learning technique, while the other technique is a deep learning technique, where we use an LSTM-based RNN to learn the container behavior. Details of the BoSC variant is discussed in chapter 5 while chapter 7 covers the LSTM alternative. Chapter 8 summarizes the results of applying both techniques to a typical SaaS cloud application. Chapter 9 covers a number of approaches used by the resiliency module of the system upon attack detection.

1.2.1 Mathematical Model

Based on the framework defined in [9], we define the problem of detecting anomalies in the container behavior as follows:

Given a trace of system calls $S = [s_1, s_2, \ldots, s_n]$ where $\forall s_i \in S : s_i \in C$, where $C = \{c_1, c_2, \ldots, c_k\}$ is the complete set of system calls, detect subsequences $A \subset S$ that are anomalous with regard to the normal training set $N \subset S$.

[2] Available as open source at `https://github.com/amrabed/strace-dcoker`

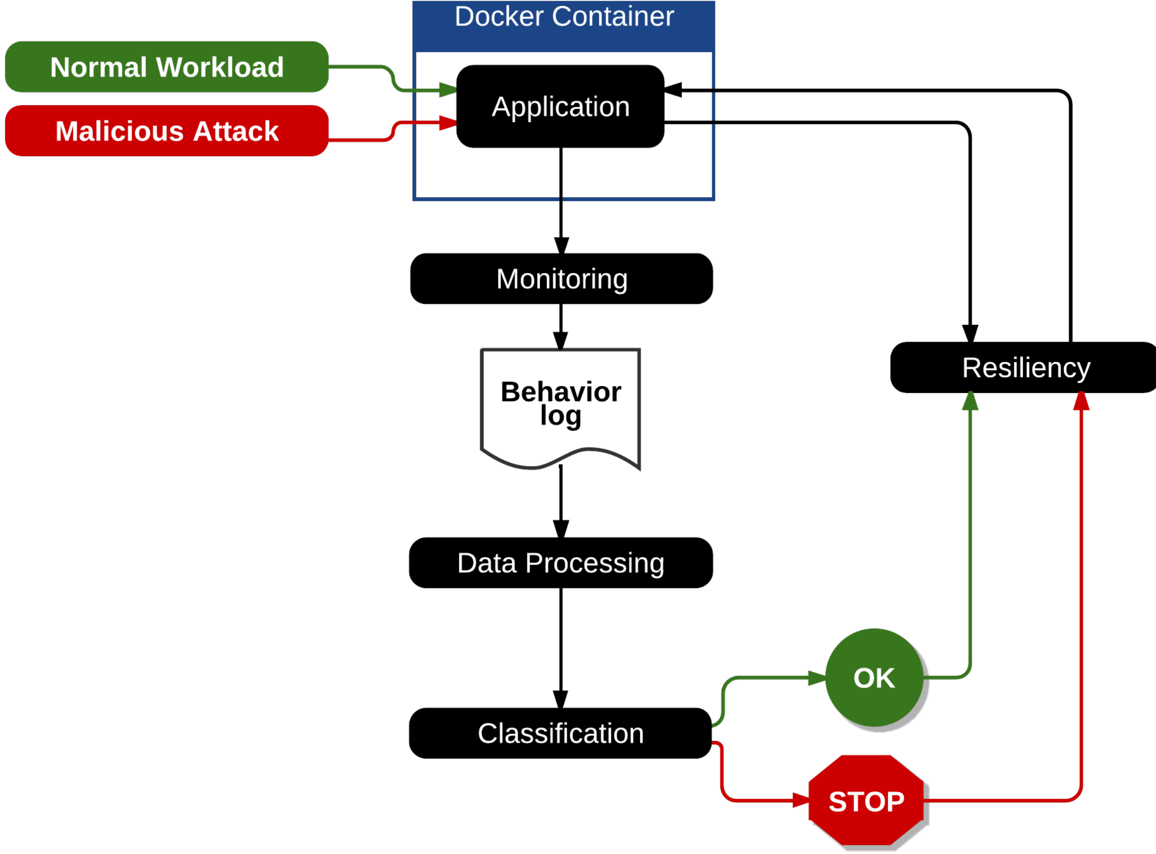

Figure 1.2: Simplified System Architecture and Logic Flow Diagram

To solve this problem, we are using two different approaches. One uses the Bag of System Calls (BoSC) which is a memory-based anomaly detection technique, while the other uses LSTM neural network to learn normal behavior.

Using BoSC

We define a sliding window filter $F_{bosc}(S) : S \mapsto E \subset 2^S$:

$$E = F_{bosc}(S) = \Big\{[s_1, s_2, \dots, s_w], [s_2, s_3, \dots, s_{w+1}], \dots, [s_{n-w}, s_{n-w+1}, \dots, s_n]\Big\}$$

where w is the sliding window length. For each sliding window $e_i \in E$, we define a BoSC function:

$$G(e_i) = \left[\sum_{s_i \in e_i} [s_i = c_1], \sum_{s_i \in e_i} [s_i = c_2], \dots, \sum_{s_i \in e_i} [s_i = c_k]\right]$$

where $[s_i = c_j]$ is 1 if s_i equals c_j and 0 otherwise.

The normal behavior database of the given trace of system calls can then be defined as:

$$db_S = \bigcup_{n_i \in F(N)} G(n_i)$$

A sequence $e_i \in E$ is then considered anomalous, i.e. $e_i \in A$, if $G(e_i) \notin db_S$. An anomaly signal is raised when the total number of anomalous sequences exceeds certain threshold t_d.

Using LSTM

Given a sequence S, build a model that predicts the next system call based on the previous system calls. Determine if each $s_i \in S$ is an anomaly or not based on how much it diverges from the value predicted by the model.

To train the model, we use two subsets of N, namely the training set $T \subset N$ and the validation set $V \subset N$ where $T \cap V = \emptyset$ and $T \cup V \subset N$.

We first define the loss function as the average negative log probability of the target system calls:

$$loss = -\frac{1}{k} \sum_{i=1}^{k} ln[p(c_i)] \tag{1.1}$$

During the model training, we minimize the average per-word perplexity, or simply *perplexity*. The perplexity is a measure commonly used in language modeling problems, which is defined as:

$$perplexity = e^{loss} = e^{-\frac{1}{k} \sum_{i=1}^{k} ln[p(c_i)]} \tag{1.2}$$

We then use the trained model to predict the next system call of the rest of the trace, and raise an anomaly signal if the number of mismatches exceeds the detection threshold t_d.

1.2.2 Assumptions and Delimitations

Our system is an anomaly-based intrusion detection as a service (IDaaS) to be deployed by cloud service providers (CSP) to monitor Linux containers initiated on their servers by their customers. Here are the assumptions and delimitations of our research:

- Due to the micro-service nature of Linux containers, a Linux container is not expected to shift its behavior during the course of the application.
- The system monitors all Linux containers hosted on the CSP servers where the system is deployed. Customers may not opt out from using the IDaaS deployed by the CSP to protect applications of other customers as well as their own cloud environment.

- The system monitors the behavior of applications running in Linux containers hosted on the cloud servers. Other applications running on the server, i.e., applications not running in Linux containers, are not monitored by our system, and hence an attack affecting only such applications without affecting guest containers may not be detected by our system.
- An attack initiated against a certain application running in one of the containers will be detected by the IDaaS instance monitoring that specific container, and may not reflect on other containers running on the same host.
- As an anomaly-based IDaaS, any significant behavior shift may be detected by the IDaaS as an intrusion. In the unlikely case of an expected shift of behavior, retraining of the IDaaS may be required, and is to be triggered by an administrator.
- A failure of the host system would result in a failure of the deployed IDaaS.

1.3 Research Objectives and Contributions

1.3.1 Intrusion Detection for Linux Containers

To the best of our knowledge, the introduced system is the first intrusion detection system introduced for securing Linux containers running in cloud environments. While many research efforts were directed toward securing virtual machines, a little research has been conducted in the area of securing Linux containers. The introduced system does not require any prior knowledge of the nature of the application inside the container, neither does it require any alteration to the container nor the host kernel, hence providing an opaque anomaly-based intrusion detection for cloud containers.

1.3.2 Applying Deep Learning for Intrusion Detection

Deep learning has already been applied to a wide variety of applications, including image recognition, fraud detection, language modeling and translation, and recommendation systems. Applying deep learning for anomaly detection purposes has already shown promising results [10, 11, 12]. In this research, we are using LSTM-based recurrent neural networks (RNN), to implement a more capable version of our anomaly detection based IDS. More details about using RNNs for learning container behavior is covered in chapter 7

1.3.3 Resiliency of Container-based Applications

In addition to intrusion detection, we describe a number of remediation techniques to recover an affected application upon attack detection. Chapter 9 gives more details about the remediation techniques used.

Dissertation Outline

The rest of the dissertation is organized as follows. Chapters 2 provides a background about Linux Containers. Chapter 3 gives a taxonomy and a summary of intrusion detection systems for cloud environments. Chapter 4 discusses recurrent neural networks (RNN) and their use in anomaly detection. Chapter 5 describes the memory-based version of the behavior modeling and classification module. An extension to the system to support distributed container-based cloud applications is then described in chapter 6. In chapter 7, we describe the employment of recurrent neural networks (RNN) for modeling the container behavior. Chapter 9 describes a number of approaches for handling detected incidents. Chapter 10 concludes with summary and future research directions.

Chapter 2

Linux Containers

Linux containers are computing environments apportioned and managed by a host kernel. Each container typically runs a single application that is isolated from the rest of the operating system. As shown in Figure 2.1, containers provide a runtime environment for applications and individual collections of binaries and required libraries.

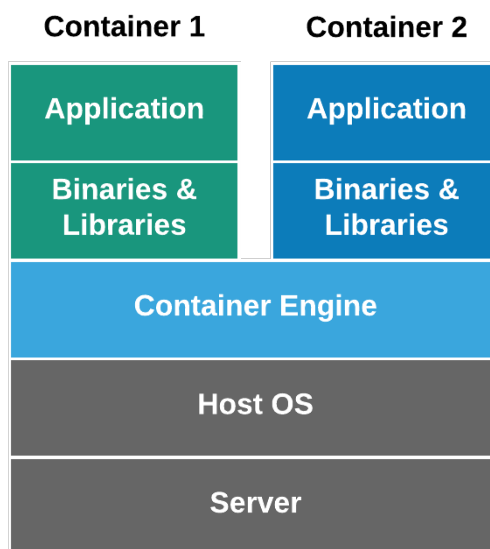

Figure 2.1: Containers provide encapsulation environment for applications and dependencies

2.1 Linux Containers versus Virtual Machines

Linux containers are not intended to replace or to be equivalent to virtual machines (VM), hence not designed to be used the same way a VM is used. For example, Docker developers encourage a microservice approach when using Docker containers, i.e. a container should only contain only one service running in one main process that spawns sub-processes [13]. To build a web application, for instance, a user needs two containers, one for the front-end web server, and one for the back-end database server.

On the other hand, a VM is usually used to provide a whole system that includes the OS and many services and applications running on the guest system. In that sense, VMs are most suitable for Infrastructure-as-a-Service (IaaS) cloud environments, while Linux container are most suitable for Software-as-a-Service (SaaS) or Platform-as-a-Service (PaaS) cloud environments.

When used for deploying a single application, Linux containers provide a significantly more efficient alternative to virtual machines, since only the application and its dependencies need to be included in the container, and not the guest OS and its processes, as shown in figure 2.2.

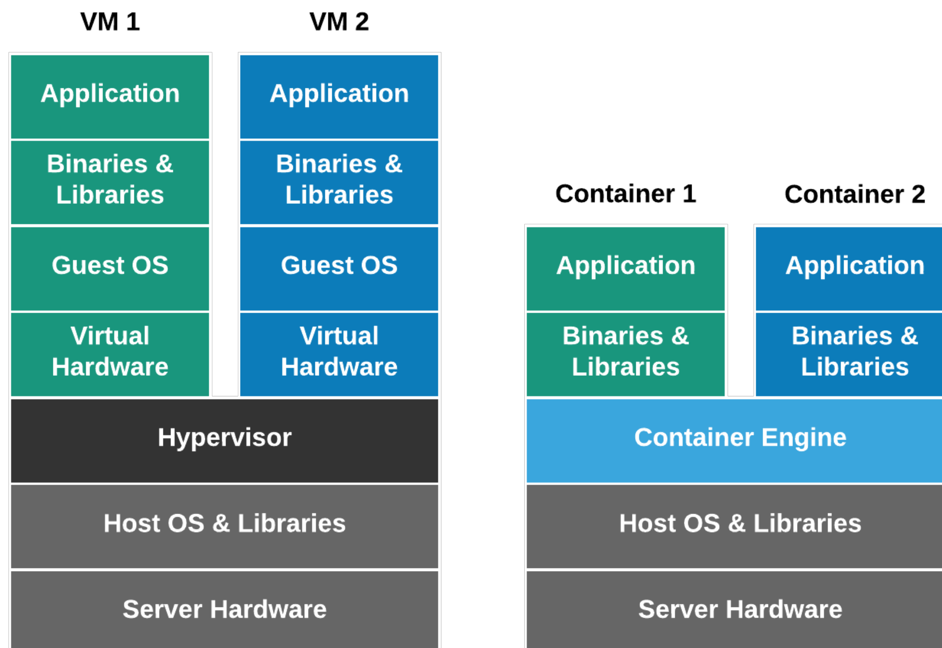

Figure 2.2: Containers provide a significantly more efficient alternative to virtual machines

Soltesz et. al. [14] compares between hypervisor-based versus container-based virtualization. They argued that container-based virtualization is more suitable than their hypervisor alternative for scenarios that require "high degrees of both isolation and efficiency". They were able to show, using a managed web hosting application, that containers provide up to double the performance of a VM-based alternative. In addition, containers also overperform VMs for I/O virtualization, since I/O devices operates in native speeds for the container case.

Pedala et. al. [15] compared Xen and OpenVZ against a base system using metrics like application performance, resource consumption, and scalability. They selected Xen and OpenVZ to be representatives of the hypervisor-based and the container-based alternatives respectively. They noticed that by increasing the application instances from one to four, the average response time increases by over 400% for Xen as compared to only 100% for OpenVZ. They ascribed that to the higher virtualization overhead in the case of Xen. They also observed a similar trend for CPU consumption.

Due to the fact that containers on the same host share the same kernel as the host, the attack surface is much wider in the case of Linux containers as discussed in more detail in section 2.3. Table 2.1 summarizes the main differences between Linux containers and virtual machines.

Table 2.1: Main differences between Linux containers and virtual machines

	Containers	Virtual Machines
Operating System	not included	included
Applications	single application	multiple applications
Cloud Environment	SaaS, PaaS	IaaS
Efficiency/Performance	higher [14, 15]	
Attack surface	wider	

2.2 History of Linux Containers

While the idea of contenarization has been around since the introduction of the `chroot` system call in 1979, it wasn't until the evolution of Docker containers in 2013 when Linux containers gained the increasing attention and popularity among users and companies. Figure 2.3 summarizes the history of Linux containers.

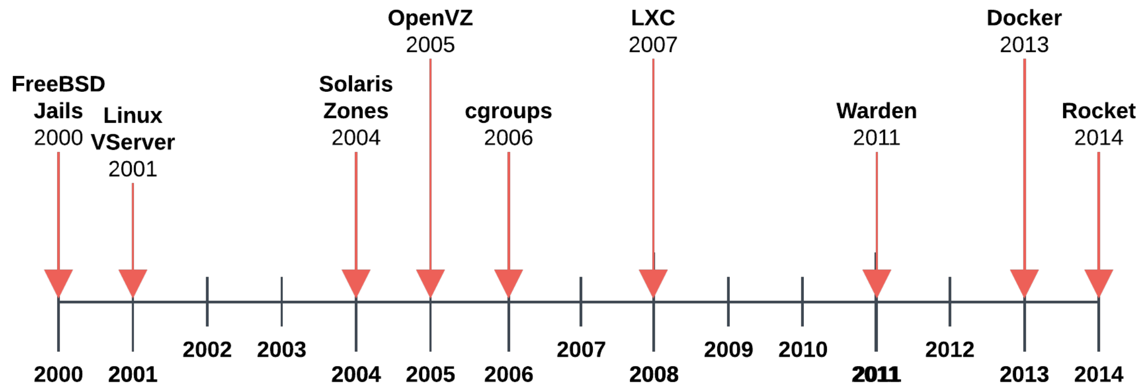

Figure 2.3: History of Linux Containers

2.2.1 The chroot system call

The concept of Linux containers was first introduced back in 1979 when the `chroot` system call was first developed as part of Unix V7. The `chroot` (change root) system call provided isolation of disk spaces between processes by changing the root directory of a process and its children to a location in the filesystem only visible to that process and sub-processes. In 1982, the `chroot` system call was added to BSD.

2.2.2 FreeBSD Jails & Linux VServer

FreeBSD Jails, introduced in 2000, provided isolation of filesystem, users, and networking between processes, and allowed system administrators to assign an IP address, custom software installations and configurations for each subsystem (jail).

Linux VServer is a another jail mechanism, introduced in 2001, that partition the resources of a computer system, such as file system, CPU time, network addresses, and memory. Each partition is called a security context, and the virtualized system within it is called a virtual private server (VPS) [16].

2.2.3 Solaris Zones

In 2004, Oracle released *Solaris Zones* for x86 and SPARC systems. While similar to Jails, Zones introduce features like snapshots and cloning from Z filesystem (ZFS).

2.2.4 OpenVZ

OpenVZ, released by Parallels in 2005, offered an operating system-level virtualization technology for Linux using a patched Linux kernel for virtualization, isolation, resource management and checkpointing. Each OpenVZ container has an isolated filesystem, users and user groups, a process tree, network, devices, and IPC objects [17].

2.2.5 Process Containers & Control Groups

Google has been using *Process Containers* since 2004 to limit, account, and isolate resource usage (CPU, memory, disk I/O, network) of a collection of processes. In 2006, they renamed it into *Control Groups* (cgroups) and merged it in the Linux Kernel in 2007 [18].

2.2.6 LXC

LXC (short for LinuX Containers) is the first most complete implementation of a Linux container manager. It was implemented in 2008 using cgroups and Linux namespaces, and it works on a single Linux kernel without requiring any patches [19].

2.2.7 Warden

Cloud Foundry started *Warden* in 2011, using LXC in early stages then replaced it with its own implementation. Warden can isolate environments on any operating system, running as a daemon and providing an API for container management.

2.2.8 LMCTFY

Let me contain that for you (LMCTFY) is the open-source version of Google's container stack. It was released in 2013 providing Linux application containers [18]. Active deployment in LMCTFY stopped in 2015 after Google started contributing core LMCTFY concepts to libcontainer, which is now part of the Open Container Foundation.

2.2.9 Docker & Rocket

It's only when *Docker* was founded in 2013 when Linux containers gained its growing popularity. Docker is now the de-facto standard, and the most widely used, Linux container management system. Unlike any other container platform, Docker introduced an entire

ecosystem for managing containers, including but not limited to, layered container image model, a global and local container registries, and a clean REST API [3].

In 2014, CoreOs released *Rocket* as a reference implementation of an open specification, standardizing the packaging of images and runtime environments for Linux containers. While not as popular as Docker, Rocket is intended to fix some of Docker drawbacks by mainly aiming to provide more rigorous security and production requirements than Docker.

2.3 Security of Linux Containers

One of the biggest concerns when using Linux containers is their security. Recent studies of Linux container security [6, 13] focus on Docker security, since Docker is becoming the de-facto standard of Linux containers. In this section, we summarize some of the security challenges when using Docker containers.

2.3.1 Overview of Docker Container Security

Docker containers are "fairly secure" even with default configurations [6]. However, it can be enhanced using custom configurations and precautions enforced by the container owner or the system administrator [7].

Isolation

Docker relies on Linux PID namespaces to isolate processes of a container from those of the host and other containers. A process within a container can only communicate with a process within the same namespace of the container, and not any process running in other containers or the host [6].

Security Hardening

While Docker supports the use of Linux Security Modules (LSM), such as SELinux and AppArmor, for security hardening, the default profiles used by Docker are quite permissive. For example, the default Apparmor profile used allows full access to the filesystem, network, and all capabilities of the containers. Also, the default SELinux profile used puts all Docker containers in one group isolating them from the host kernel but not from each other [13]. The problem can be solved using custom profiles.

Networking

One problem with Docker security is the use of a virtual Ethernet bridge as the default networking model. The bridge is vulnerable to ARP spoofing and MAC flooding attacks since it does not filter the traffic passing through it. The problem can be solved by manually adding filtering to the bridge, or changing the networking model to a more secure alternative, such as a virtual network [6].

Docker options

Many Docker options used when starting the Docker daemon or starting a container can raise the container privilege or limit its isolation from the host Kernel. Examples include the `--privileged` option and the `--net=host` [13]. If used by untrusted containers, such options raise many security concerns, such as mounting host sensitive directories in containers.

2.3.2 Current Efforts for Securing Linux Containers

LiCShield is a framework that generates a set of security rules that limits the capabilities of the created container based on the expected activities of the image used for creating the container [20]. However, LiCShield is intended to be used as a supplementary tool to complement the use of IDS and Linux Security modules (LSM) to harness the security of Linux containers [20].

DockerPolicyModules provide an extension to the Dockerfile format that allows the image owner to include SELinux mandatory access control (MAC) rules to the Dokcer image to be applied to containers created from the image for enhanced isolation [21].

DoubleGuard is an intrusion detection system that used OpenVZ Linux containers to secure web applications [22]. DoubleGuard is not intended for securing Linux containers, rather it uses Linux containers for isolating user sessions on web services. For their application, they used a typical web application with front-end web server and a back-end MySQL server. To detect malicious web requests, DoubleGuard started a separate Linux container for each user session. By monitoring the requests on both ends of the container, DoubleGuard detects when the requests arriving from the user side do not match those issued by the web server to the SQL server. We find the solution to incur too much overhead that limits its scalability as the number of user sessions increases.

Chapter 3

Intrusion Detection in the Cloud

Intrusion detection continues to be an urgent necessity for securing cloud environments [23, 24, 25, 26]. However, while a number of intrusion detection systems were introduced to secure hypervisor and virtual machine (VM) based cloud environments, none has been designed for container-based cloud environments to the best of our knowledge.

The performance gain offered by Linux containers as compared to VMs, and the continually increasing third-party adoption and support, is enabling Linux containers to become the new standard for virtualization when deploying applications in the cloud. However, current IDS options still focus on securing VM-based clouds.

3.1 Taxonomy of Intrusion Detection Systems

Intrusion detection systems in the cloud can be categorized as Host-based IDS (HIDS), Network-based IDS (NIDS), and Hypervisor-based IDS (HyIDS) [25]. HIDS is installed on host machines by cloud provider to monitor activity of the host and all guest VMs and/or containers. NIDS is deployed in underlying network to monitor network traffic to, from, and between all devices on the network. A hypervisor-based IDS is placed in the hypervisor to monitor communications between VMs, and between VM and hypervisor.

An IDS typically uses a signature-based detection technique or an anomaly-based detection technique. In a signature-based IDS, the IDS utilizes prior knowledge of the attack behavior to detect already known attacks. An anomaly-based IDS, on the other hand, learns the normal behavior of the monitored system to detect intrusion as indicated by any shift from the normal behavior model.

In our work, we introduce a real-time anomaly-based HIDS that monitors the system calls between the host kernel and all containers running on the host to learn the behavior of the containers, and detect if one or more change their behavior as an indication of intrusion.

The behavior modeling module of the IDS has two versions, one that uses a traditional memory-based learning technique, while the other relies on a RNN to learn the behavior of the monitored container.

3.2 Evaluation of Intrusion Detection Systems

This section summarizes the metrics typically used in literature when evaluating intrusion detection systems.

3.2.1 True Positives, True Negatives, and False Positives

Two of the most widely used metrics are the True Positive Rate (TPR), also known as the detection rate (DR), and the False Positive Rate (FPR). True Negative Rate (TNR) is also used sometimes.

$$TPR = \frac{N_{TP}}{N_{malicious}} \tag{3.1}$$

$$FPR = \frac{N_{FP}}{N_{normal}} \tag{3.2}$$

$$TNR = \frac{N_{TN}}{N_{normal}} = 1 - FPR \tag{3.3}$$

where N_{normal} and $N_{malicious}$ are the total number of normal and malicious data samples, respectively, and N_{TP}, N_{TN}, and N_{FP} are the number of true positives, true negatives, and false positives, respectively.

3.2.2 Accuracy, Precision, and F-score

Accuracy and precision are two other commonly used metrics.

$$Accuracy = \frac{N_{TP} + N_{TN}}{N_{TP} + N_{FP} + N_{TN} + N_{FN}} \tag{3.4}$$

$$Precision = \frac{N_{TP}}{N_{TP} + N_{FP}} \tag{3.5}$$

The F_β-score is another metric that is used when the normal and malicious subsets of the data are not balanced, e.g. when the normal data highly oversizes the malicious data. The

F_β score is the tradeoff between the precision and the true positive rate, in which the value of β is used to favor one of them over the other.

$$F_\beta = (1 + \beta^2) \frac{Precision * TPR}{\beta^2 * Precision + TPR} \tag{3.6}$$

Most commonly used variants of F_β are F_1, F_2, and $F_{0.5}$. F_1 simply combines precision and TPR into one measure, while F_2 weights TPR higher than precision, and $F_{0.5}$ favors precision over TPR.

3.2.3 Top-P accuracy

The *Top-P* anomaly detection algorithm was introduced in [27] as an efficient algorithm for detecting anomalies in large datasets. The method works by clustering the data and ordering the distance to the different dataset points, and considering the top P points to be anomalous. The Top-P accuracy measure has since been used by many researchers for evaluation of anomaly detection techniques.

3.3 Cloud-based Intrusion Detection Systems

This section summarizes the intrusion detection systems used in cloud environments categorized by the method used. Each of the current cloud-based IDSs, described in the subsections below, suffers from at least one of the following limitations that we tried to avoid in our IDS:

1. Designed for VM-based cloud environments (not deployable in a container-based Cloud)
2. Requires alteration of the host, hypervisor, and or the guest system
3. Requires prior knowledge of the OS or applications of the guest system
4. Requires prior knowledge of the attack patterns, i.e., signature-based IDS
5. Requires administrative attention
6. Imposes overhead by hosting another VM for detection purposes
7. Detection technique is complicated and requires tuning of many parameters
8. System accuracy is low or not reported

Table 3.1 summarizes the IDSs described along with their limitations.

3.3.1 Monitoring System Calls for Intrusion Detection

Alarifi and Wolthusen used the Bag of System Calls (BoSC) technique [8] for anomaly-based intrusion detection in virtual machines. In their technique, they read the input trace of system calls generated by the VM epoch by epoch. For each epoch, a sliding window of size k moves over the system calls of each epoch, adding bags of system calls to the normal behavior database. The normal behavior database holds frequencies of bags of system calls. After building the normal-behavior database, i.e. training their classifier, an epoch is declared anomalous if the change in BoSC frequencies during that epoch exceeds certain threshold. For a sliding window of size 6, their technique gave 100% detection rate, and 11.11% FPR [28].

Alarifi and Wolthusen applied HMM for learning sequences of system calls for short-lived virtual machines. They based their decision on the conclusion from [41] that "HMM almost always provides a high detection rate and a low minimum false positives but with high computational demand". They used two different scenarios for training their HMM model. In one scenario, they considered all possible system calls, while in the other, they only considered IOCTL system calls. Testing both scenarios using DoS attacks, the former provided 100% detection rate with 5.66% FPR while the latter scenario gave only 83% detection rate [29].

Arshad et al. proposed a signature-based IDS that used known attack patterns to detect suspicious system calls running in guest VMs. The system provided an average detection rate of 90.7954% [30].

Gupta and Kumar used an IDS that created a baseline database for each application running in the VM and another baseline database for the whole VM. For each VM, the VM is responsible for monitoring each application compared to the baseline of the application, and the cloud manager is responsible for monitoring the VM snapshot compared to the VM baseline database. Tested using the UNM (University of New Mexico) sendmail dataset [42], the system had an accuracy of 80% and a FPR of 3% [31].

Gupta and Kumar proposed an IDS that used a database of system calls structured in key-value pair format, where the key is the system call and the value is an immediate sequence of system calls that are expected to follow it during program execution. The system requires the cloud administrator to provide a list of programs to be monitored in a configuration file [32].

3.3.2 Monitoring Network Traffic for Intrusion Detection

Lin et al. proposed a NIDS that uses Snort [1] deployed at the privilege domain of the virtual machine manager (VMM) to detect intrusion against guest VMs. The IDS collected information about the the operating system used and the services running in the VM to update the detection rules accordingly for each VM [35].

[1] Snort - `https://www.snort.org`

Kim et al. introduced a NIDS that used support vector machines (SVM) to build a decision model to detect anomalies in the network. The system required many changes of the SVM parameter values to determine the optimal values of the parameters [33].

Li et al. proposed a NIDS that used artificial neural networks (ANN) to learn the VM profile using a large dataset of both normal and anomalous network traffic. The system provided accuracy of 99.7% for a 7-node model [34].

Pandeeswari and Kumar monitored and analyzed virtual network traffic collected at the hypervisor in both normal and anomalous scenarios. The system used Fuzzy C-mean clustering to create clusters based on membership value, then used ANN to train each cluster. The system provided an average detection rate of 97.55% with 3.77% average FPR for detecting anomalous network traffic. However, it required extensive training due to the complexity of the algorithm used [39].

Modi et al. introduced a double-layer NIDS that uses both signature and anomaly-based approach in an attempt to improve accuracy. Snort is used for signature-based detection and then a decision tree algorithm is used for anomaly-based detection. Tested using the KDD dataset [43], which exhibits different behavior than cloud traffic and hence not satisfactory, the system provided an accuracy of 84.31% with 4.81% FPR for the NSL-KDD99 dataset and 96.71% accuracy with 1.91% FPR for the KDD99 dataset [37].

3.3.3 Monitoring Program Execution for Intrusion Detection

Payne et al. proposed an IDS that placed hooks in the guest VM to pass program events to the hypervisor, which in turn passes them to a security VM. The security VM analyzes passed events and makes decisions to be passed back to the guest VM, where the decision is enforced isolating any application generating malicious code [40].

3.3.4 Monitoring System Resources for Intrusion Detection

Nikolai and Wang introduced a hypervisor-based intrusion detection system that monitored performance metrics, such as network packets, block device read/write request, and CPU utilization to build VM profile [38].

Table 3.1: Summary of Cloud-based IDSs and their limitations

IDS	**Type**	**Technique**	**Monitoring**	**Limitations**
Alarifi [28]	HIDS	anomaly	system calls	1,8
Alarifi [29]	HIDS	anomaly	system calls	1,8
Arshad [30]	HIDS	signature	system calls	1,4
Gupta [31]	HyIDS	anomaly	system calls	1,2,3,8
Gupta [32]	HIDS	anomaly	system calls	1,3,8
Kim [33]	NIDS	anomaly	network traffic	7,8
Li [34]	NIDS	anomaly	network traffic	1
Lin [35]	NIDS	anomaly	network traffic	1,2,3
Meng [36]	NIDS	signature	network traffic	4,7
Modi [37]	NIDS	hybrid	network traffic	4,8
Nikolai [38]	HyIDS	signature	system resources	1,4,5
Pandeeswari [39]	NIDS	anomaly	network traffic	1,7
Payne [40]	HyIDS	anomaly	program execution	1,2,6

Chapter 4

Deep Learning

The main challenge when addressing a machine learning problem is the *curse of dimensionality*, where the learning complexity grows exponentially when the data dimensionality increases linearly. Traditional machine learning techniques typically overcome the curse of dimensionality by preprocessing data to reduce the dimensionality of the input data, e.g. by feature extraction. However, feature extraction itself is a challenging task that mainly relies on human knowledge of the problem, which means that the machine learning accuracy is highly dependent on the accuracy of the feature extraction process.

A neuroscience discovery, that the human brain allows raw sensory data to propagate through a complex hierarchy of modules without preprocessing, inspired the evolution of deep learning[44]. Deep learning techniques allows raw input data to go through multiple levels of representation by propagating through simple but non-linear modules, each transforms the representation at one level into a representation at a higher slightly more abstract level[45].

The size of data available today mandated the shift from conventional to deep learning mechanisms, and the power of the current computing platforms made it possible.

While a number of architectures are used for deep learning, we are focusing on recurrent neural networks (RNN) and their Long Short-Term Memory (LSTM) variant, as they are the most suitable architecture for our application.

4.1 Recurrent Neural Networks

Recurrent Neural Network (RNN) is a neural network with feedback connections between units in its hidden layers. Figure 4.1 shows a sample fully-connected RNN with one hidden layer. The main difference between a traditional neural network and an RNN is that the former can only map an input to an output, while an RNN is capable of mapping a history

of inputs to an output. A traditional neural network has no way to keep track of previous input, and hence cannot build its decision on the history of inputs. On the other hand, the feedback connections in an RNN allows the network to base its decision not only on the current input but also on previous inputs.

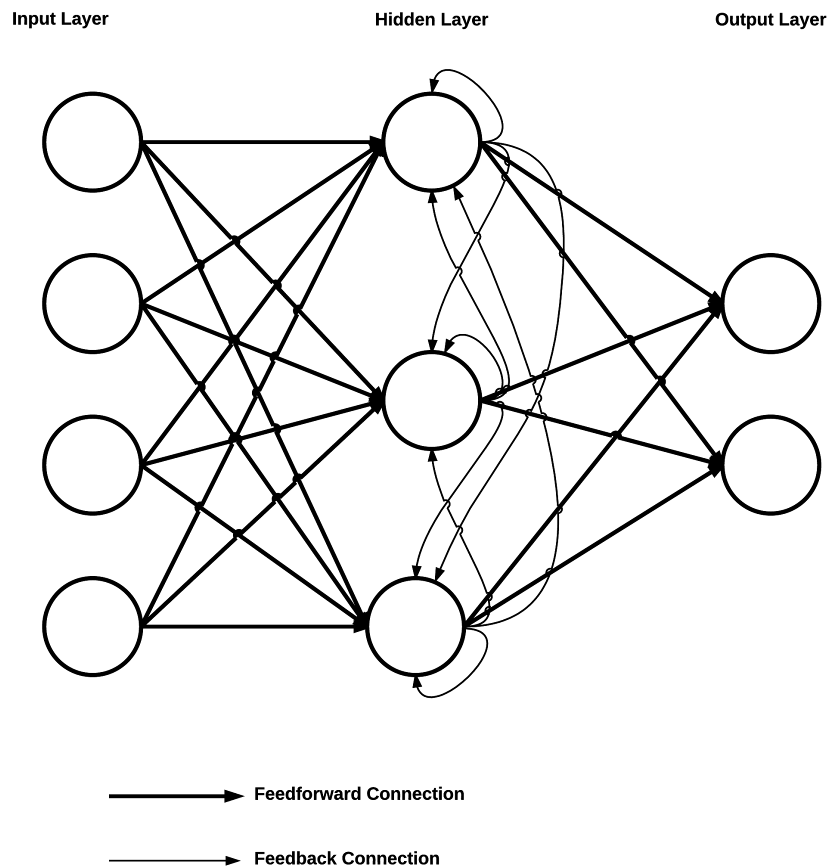

Figure 4.1: A sample fully-connected Recurrent Neural Network with one hidden layer

4.1.1 Unrolling RNN

Better visualization of RNNs is usually achieved by unrolling (or unfolding) the network as shown in figure 4.2, where the output of the hidden node at time t depends on the current and previous inputs to the node, i.e. for $t \in [0, t]$.

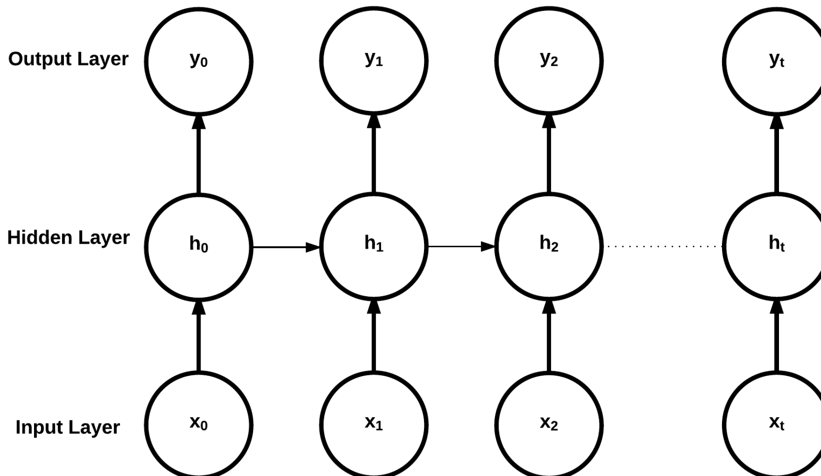

Figure 4.2: Unfolded Recurrent Neural Network for better visualization

4.2 Long Short-Term Memory

Long Short Term Memory (LSTM) networks are a special version of RNN that was first introduced by Hochreiter and Schmidhuber in [46] to address long-term dependency problem [47]. Example applications of LSTM include recognition of temporally extended patterns in noisy input sequences, recognition of the temporal order of widely separated events in noisy input streams, extraction of information conveyed by the temporal distance between events, Stable generation of precisely timed rhythms, smooth and non-smooth periodic trajectories, and robust storage of high-precision real numbers across extended time intervals.

The main difference between a standard RNN and an LSTM is that the LSTM uses a more complex unit called the memory cell. Figure 4.3 shows the architecture of a memory cell and its gates. Each gate consists of a neural net layer and a multiplicative unit to allow the LSTM to control the flow of information through memory cells. Typically, there are three types of gates, namely input gate, output gate and forget gate. The input gate protects the memory cell from irrelevant inputs, while the output gate protects other cells from irrelevant memory contents. The forget gate resets the current contents of the memory cell.

For the LSTM shown, h_t is the output of hidden layer h at time t, c_{t-1} is the current content of the memory cell, $\bar{c}_t$ is the candidate new content, and c_t is the actual new contents at time t. f_t, i_t, and o_t are the output of the forget, input, and output gates respectively. The flow of information through the cell is then given by the following equations.

$$f_t = \sigma\Bigg(W_f.[h_{t-1}, x_t]\Bigg) \tag{4.1}$$

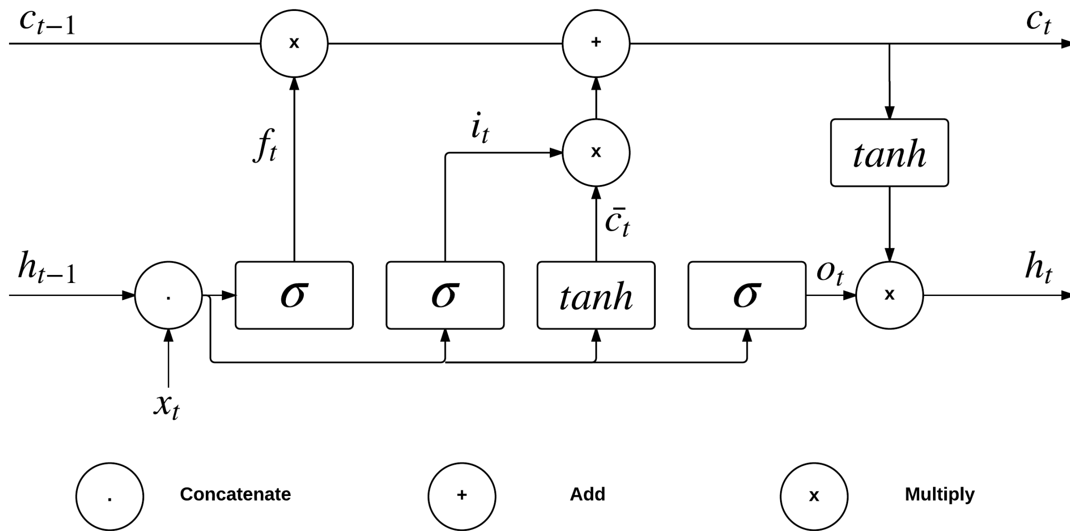

Figure 4.3: Architecture of a LSTM memory cell and gate units

$$i_t = \sigma\left(W_i.[h_{t-1}, x_t]\right) \tag{4.2}$$

$$\bar{c}_t = \tanh\left(W_c.[h_{t-1}, x_t]\right) \tag{4.3}$$

$$c_t = f_t * c_{t-1} + i_t * \bar{c}_t \tag{4.4}$$

$$o_t = \sigma\left(W_o.[h_{t-1}, x_t]\right) \tag{4.5}$$

$$h_t = o_t * tanh(c_t) \tag{4.6}$$

4.3 RNN Hyperparameter Selection

The hyperparameters of a neural network are parameters that are set prior to the actual training of the neural network, i.e. they are parameters that are not directly selected by the learning algorithm [48]. For example, the learning rate is a hyper-parameter while the neural network weights are not.

A deep learning architecture usually have more than 10 hyperparameters typically set using manual, grid, or random search. In our work, we are manually selecting values based on how they shown to perform in similar problems, such as language modeling. Examples of hyperparameters when using gradient descent for training the neural network include loss function, learning rate, and batch size.

4.3.1 Loss Function

The loss function describes the gap between the network output and the actual expected output. The learning algorithm tries to minimize the loss function over training epochs by adjusting the weights of the hidden units of the neural network.

For example, the loss function we are using for our problem is the average negative log probability of the target words as defined by:

$$loss = -\frac{1}{N}\sum_{i=1}^{N} ln(p_{target_i})$$

4.3.2 Learning Rate

The value of the learning rate λ is the most important hyper-parameter to be tuned for better accuracy of the training algorithm [48]. If the learning rate is too large, the average loss will increase. Typical values of the learning rate are between 10^{-6} and 1.

Starting from the initial value λ_0, the learning algorithm typically decreases the learning rate by a certain factor over epochs to improve accuracy. An example learning rate schedule is defined by:

$$\lambda_t = \frac{\lambda_0 \tau}{max(t, \tau)}$$

where t is the current training epoch, and τ is the number of epochs for which the initial rate λ_0 is used.

Using this schedule, the learning rate remains constant for τ epochs before being decreased by a factor of t. An adaptive and heuristic way to automatically set τ is to use constant λ_t until the training error stops decreasing significantly from epoch to epoch.

4.3.3 Batch Size

The training set is usually divided into batches with the weights updated after the calculation of loss function for each batch. The batch size B can be anywhere between 1 and the training

set size, and is typically chosen between 1 and a few hundreds. If $B = 1$, the weights are updated after each sample of the input data. On the other hand, if B equals the training set size, the update is done only once after each training epoch.

However, larger B (e.g. $B > 10$) yield faster computation, since it benefits from parallelism and efficient matrix-matrix multiplications, but requires visiting more examples in order to reach the same error, since there are less updates per epoch. In other words, the effect of B is mostly computational, i.e., it should only affect the training time rather than the test performance. In practice, $B = 32$ is usually a good default value [48].

4.3.4 Number of Training Iterations

The number of training iterations T is the number of batches used for training the network. T can be optimized by using *early stopping*, i.e. stopping the training when the validation error stops improving. Early stopping also evades overfitting [48].

4.3.5 Momentum

The momentum β is a small positive coefficient that can be used to smooth the gradient updates for faster convergence [48]:

$$\bar{g} = (1 - \beta\bar{g}) + \beta g$$

where $\bar{g}$ is the average of past gradient updates, and g is the current gradient update. The value of β is between 0 and 1, with a default value of 1 (no momentum) that works well in many cases [48].

4.3.6 Number of Hidden Units

The number of units in hidden layers n_h generally affects generalization performance, i.e. larger n_h yields better generalization performance, yet requires more computations. Using the same size for all hidden layers generally works better or the same as using a decreasing or increasing size. In addition, using a first hidden layer which is larger than the input layer tends to work better.

4.3.7 Regularization Coefficient

A regularization term is often added to the loss function to avoid overfitting, typically by pushing the network weights w_i toward 0. $L1 = \lambda \sum_i |w_i|$ and $L2 = \lambda \sum_i w_i^2$ are the most commonly used regularization terms.

4.3.8 Activation Functions

A non-linear activation function is typically used for neuron outputs. Common examples are *sigmoid*, *tanh*, and *ReLU* as defined below.

$$sigmoid(y) = \frac{1}{1 + e^{-}y}$$

$$tanh(y) = \frac{1 - e^{-2y}}{1 + e^{-2y}}$$

$$ReLU(y) = max(0, y)$$

4.3.9 Initial Weights

The initial values of the network weights impact the local minimum found by the training algorithm. Weights are often initialized using a random uniform distribution in the range of $[r, r]$ where r is the inverse of the square root of the fan-in added to the fan-out of the unit.

4.4 Using Deep Learning for Anomaly Detection

4.4.1 Anomaly Detection using Recurrent Neural Networks

Chauhan and Vig [10] utilized a recurrent neural network with LSTM cells for anomaly detection in Electrocardiography (ECG) time signals in an attempt to automate the process of detecting when a patient is suffering from Arrhythmia. They mainly used RNN for anomaly detection to avoid the need for extensive preprocessing and feature extraction. In addition, by using RNN, they alleviated the need for prior knowledge or data when detecting a new type of Arrhythmia that was not present in the original training data.

The network they used was a stacked LSTM-based RNN with one input node and l output nodes. The input node takes the signal at time t, and the output nodes gives the predictions of the next l time steps. Each hidden layer of the network was fully connected to the subsequent layer, with the final hidden layer fully connected to the output layer. They trained multiple architectures with up to three hidden layers and varying number of units in each layer, and selected the model that showed the best validation set performance. The selected architecture had two hidden layers with 20 units each.

They used the MIT-BIH Arrhythmia Database [49] to obtain ECG time series of normal periods and periods during four different types of Arrhythmia. They divided the dataset into four sets; a normal training set (s_N), a normal validation set (v_N), a normal and anomalous validation set (v_{N+A}), and a test set (t_{N+A}).

The network was trained using the normal training set (s_N) and the normal validation set (v_N) was used for early stopping. The trained network was then used to predict on v_{N+A} and the probability distribution function (PDF) of the prediction error was recorded. The recorded PDF was then used to determine a threshold value τ for discriminating between anomalous and normal data by maximizing the F_β score.

In their experiments, they chose $\beta = 0.1$ to favor precision over TPR because the normal data had no noise label, while the anomalous data had segments of normal behavior and noise. The determined threshold τ was then used to detect anomalies within the test set t_{N+A}. For $\tau = -0.94054$, they achieved a $F_{0.1}$ score of 96.45%, a precision of 97.50%, and a false positive rate (FPR) of 1.19%. However, the TPR was relatively low at 46.47%.

4.4.2 Anomaly Detection using Autoencoders

Autoencoders are auto-associative neural networks where the hidden layers have lower dimensionality then the input and output layers. The network is trained to reconstruct its own input, i.e. the output should be the same as the input, which means that the output has the same dimensionality as the input.

When using autoencoders for anomaly detection, researchers define the problem as a one-class learning problem [11, 12], in which they consider data of one class to be normal, and the data from any other class to be malicious. They use normal data for training and validation, then use a mixture of normal and malicious data for testing. In other words, the network is trained to learn normality to detect abnormality. By using only normal data for training, the network should be able to reconstruct samples from the normal dataset with minimal reconstruction error. On the other hand, the network fails to reconstruct samples from the malicious dataset, since no malicious data was introduced to it during the training or the validation phases. The reconstruction error, defined as the mean square error between the input and the generated output, is therefore used for anomaly detection. By defining a threshold value for the construction error, any sample with reconstruction error that exceeds the threshold is considered an anomaly. The detection threshold can be set to the maximum value of the reconstruction error during validation [12].

In 2002, Hawkins et. al. [11] were the first to employ Replicator Neural Network (aka Autoencoder) for anomaly detection. The reason they chose autoencoders for anomaly detection was "letting the data speak for itself", i.e. without any prior assumptions about the data. The network they used had three fully-connected hidden layers. The number of units in the hidden layers were selected such that they minimize the average reconstruction error over training data. The average reconstruction error is then used as indicator of anomaly. They applied their network to the 1999 KDD Cup network intrusion detection dataset [43], and the Wisconsin breast cancer dataset [50]. They concluded that their network was able to detect anomalies with high accuracy in both datasets.

In 2012, Dau et. al.[12] used an autoencoder network with one hidden layer for anomaly detection. They argued that using only one hidden layer can result in similar or better performance than the network used in [11] without the need for the increased number of network parameters. The number of the nodes of the hidden layer, however, depended on the dimensionality of the input data. They suggested that setting the number of the hidden units to be equal to the number of the input/output units usually gives high detection rate. They applied their network to 6 different datasets. The datasets they used are the Wisconsin breast cancer dataset [50], the Ionosphere dataset, the Musk dataset, the Biomed dataset, and two Shape Anomaly datasets. Table 4.1 shows the different parameters of the datasets used, and table 4.2 gives the network configuration, the detection threshold, and evaluation results for each dataset.

Table 4.1: Parameters of 6 datasets used for anomaly detection using autoencoders

Dataset	Size	Features	Anomalies	Training	Validation	Test
Wisconsin	683	9	239	150	50	438
Ionosphere	351	34	126	100	30	221
Musk	7074	166	1224	3000	1000	3074
Biomed	194	4	67	50	50	94
Shape1	30	162	10	7	3	20
Shape2	70	162	10	20	10	40

Table 4.2: Results for using Autoencoder network with one hidden layer on 6 datasets

Dataset	Network	Threshold	F-score	Accuracy	TPR	TNR	Top-P
Wisconsin	$9-9-9$	0.0634	94.29%	94.36%	93.30%	95.42%	94.53%
Ionosphere	$34-34-34$	0.2138	92.00%	90.74%	91.27%	90.00%	91.27%
Musk	$166-166-166$	0.4453	41.93%	67.76%	28.76%	94.28%	54.49%
Biomed	$4-7-4$	0.0238	55.67%	54.25%	40.30%	88.89%	74.63%
Shape1	$162-162-162$	0.1901	95.24%	95.00%	100.00%	90.00%	100.00%
Shape2	$162-162-162$	02.3580	51.28%	52.50%	100.00%	36.67%	70.00%

Chapter 5

Memory-based Behavior Modeling

For the memory-based behavior modeling module, the system adopts a technique that combines the bag of system calls (BoSC) technique [8] with the sliding window technique [51]. The system works in real time, i.e. it learns the container behavior and detects anomaly at runtime. It also works in opaque mode, i.e. it does not require any prior knowledge about the nature of the container nor the enclosed application. Figure 5.1 gives an overview of the system architecture and data flow as described below.

5.1 System Call Monitoring

Linux containers communicate with the host kernel, and the outer world, through system calls. (Figure 5.2). To learn the behavior of the container, our system employs a background service running on the host kernel to monitor system calls between any Docker containers and the host Kernel [1]. The system relies on the Docker command `events` to trigger the service whenever a new container is started. The service then uses the open-source `sysdig` [52] tool to trace all system calls issued by any process within the container to the host kernel.

It is worth mentioning that we started by using the more mature built-in Linux `strace` tool before deciding to switch to `sysdig` for three main reasons:

1. We noticed that `strace` slows down, and sometimes blocks, the execution of the traced application
2. Sysdig provides native support for container tracing using container name or ID
3. Sysdig allows for formatting the output for more straightforward analysis of the generated trace.

[1] Code available as open source at `https://github.com/amrabed/strace-docker`

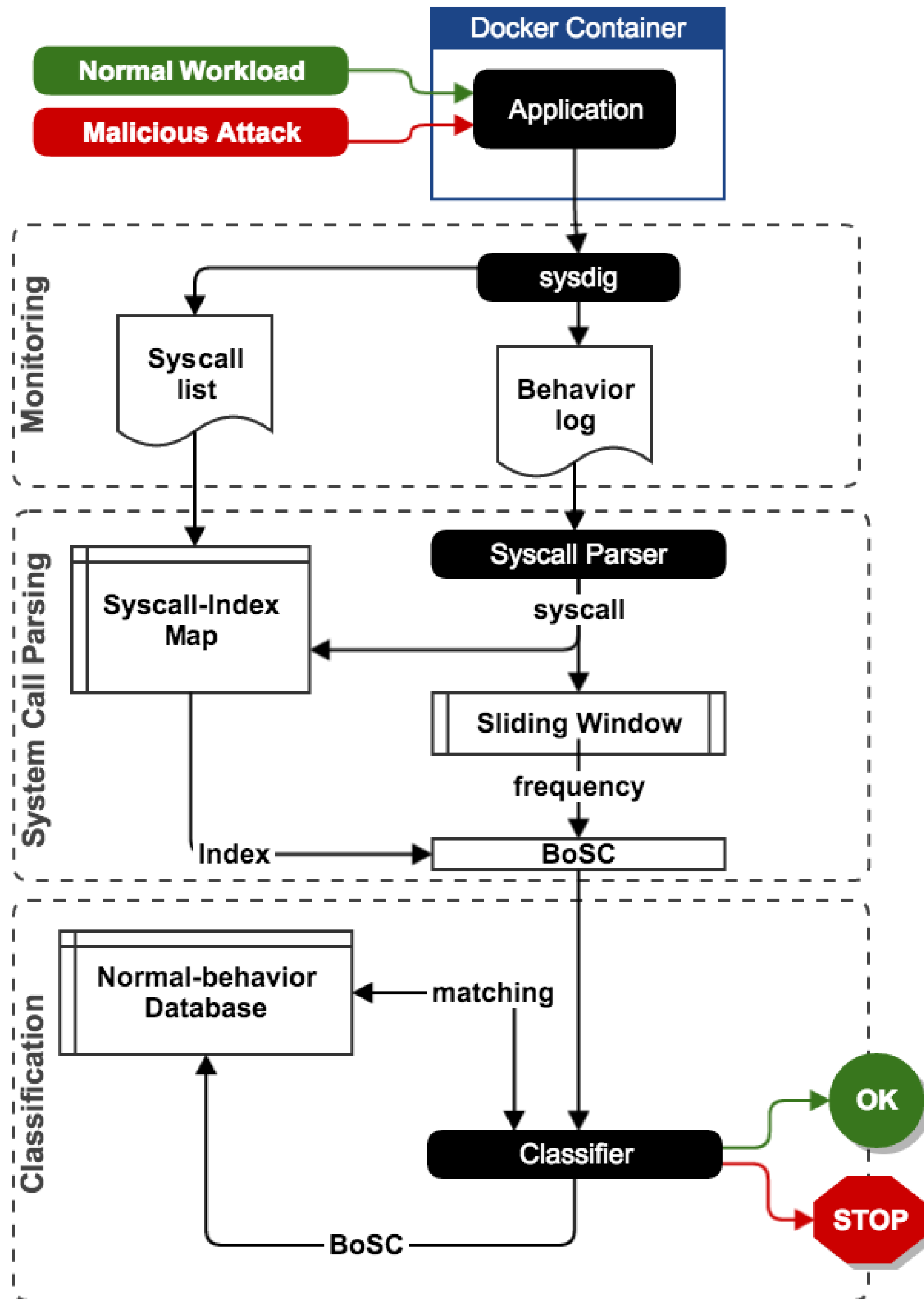

Figure 5.1: Memory-based Behavior Modeling and Classification - Logic Flow Diagram and Architecture

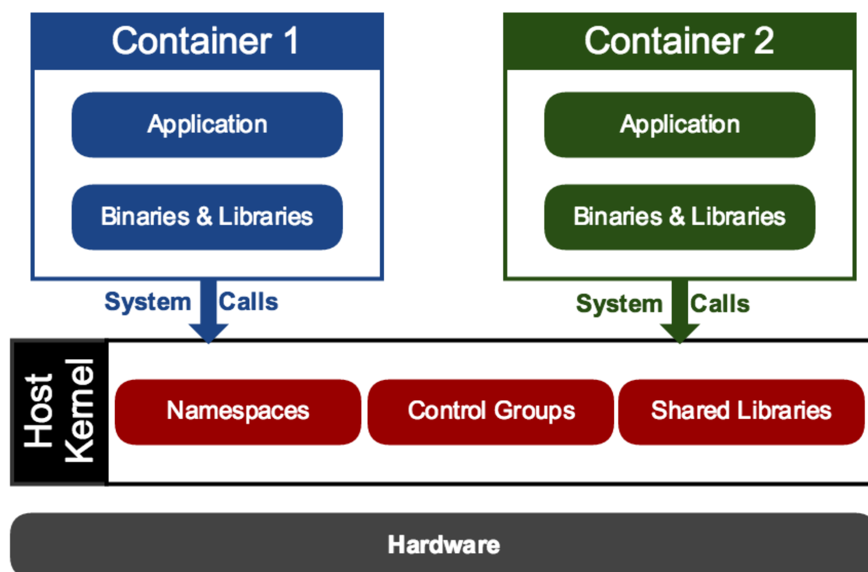

Figure 5.2: Linux containers communicate with host kernel through system calls

Table 5.1: Sample entries of the Syscall-Index Lookup Table

Syscall	Index
select	4
access	12
lseek	22
other	40

A syscall-list file, that holds a preassembled list of distinct system calls sorted by the number of occurrences, is also generated. The list is collected from a container running the same application under no attack. The syscall-list file is used to create a syscall-index lookup table. Table 5.1 shows sample entries of a typical syscall-index lookup table.

5.2 Parsing Data from Behavior Log Files

The behavior file generated by `sysdig` is then parsed in either online or offline mode. In online mode, the system-call parser reads system calls from the same file as it is being written by the `sysdig` tool for real-time classification. Offline mode, on the other hand, is only used for system evaluation as described in section 5.4. In offline mode, a copy of the original

Table 5.2: Example of system call parsing

Syscall	Index	Sliding window	BoSC
pwrite	6	[futex, futex, sendto, futex, sendto, pwrite]	[2,0,3,0,0,0,**1**,0,...,0]
sendto	0	[futex, sendto, futex, sendto, pwrite, sendto]	[**3**,0,**2**,0,0,0,1,0,...,0]
futex	2	[sendto, futex, sendto, pwrite, sendto, futex]	[3,0,**2**,0,0,0,1,0,...,0]
sendto	0	[futex, sendto, pwrite, sendto, futex, sendto]	[**3**,0,2,0,0,0,1,0,...,0]

behavior file is used as input to the system to guarantee the coherence between the collected statistics. The system call parser reads one system call at a time by trimming off arguments, return values, and process IDs.

The parsed system call is then used for updating a sliding window of size 10, and counting the number of occurrences of each distinct system call in the current window, to create a new bag of system calls. As mentioned earlier, a bag of system calls is a vector $< c_1, c_2, \ldots, c_{n_s} >$ where c_i is the number of occurrences of system call, s_i, in the current window, and n_s is the total number of distinct system calls. When a new occurrence of a system call is encountered, the application retrieves the index of the system call from the syscall-index lookup table, and updates the corresponding index of the BoSC. For a window size of 10, the sum of all entries of the array equals 10, i.e. $\sum_{i=1}^{n_s} c_i = 10$. A sequence size of 6 or 10 is usually recommended when using sliding-window techniques for better performance [51][41][53]. Here, we are using 10 since it was already shown for a similar work that size 10 gives better performance than size 6 without dramatically affecting the efficiency of the algorithm [28]. Table 5.2 shows an example of this process for $n_s = 20$ and sequence size of 6.

5.3 Real-Time Behavior Learning and Classification

The classification process starts by building a syscall-index hash map from the syscall-list file. The hash map stores distinct system calls as the key, and a corresponding index as the value. A system call that appears in the whole trace less than the total number of distinct system calls is stored in the map as "other". Using "other" for relatively rarely-used system calls saves space, memory, and computation time, as described in [28]. By using a hash map, looking up the index of a system call is an $O(1)$ operation.

The system call parser then reads one system call at a time from the behavior log file, and updates the normal-behavior database. The normal-behavior database is another hash map with the BoSC as the key and the frequency of the bag as the value. If the current bag already exists in the database, the frequency value is incremented. Otherwise, a new entry is added to the database. Again, by using a hash map for implementing the database, the time complexity for updating the database is also $O(1)$.

Table 5.3: Sample entries of the Normal Behavior Database

BoSC	Frequency
0,1,0,2,0,0,0,0,1,0,3,0,1,0,0,0,1,0,0,1	15
0,1,0,1,0,0,1,0,1,0,3,0,0,0,0,0,1,1,0,1	8
0,1,0,2,0,0,5,0,0,0,0,0,0,0,0,0,1,0,0,1	2
0,1,0,2,0,2,0,0,1,0,2,0,0,0,0,0,1,0,0,1	1

As described in section 5.2, the system uses the sliding window technique to read sequences of system calls from the trace file, with each sequence is of size 10. A bag of system calls is then created by counting the frequency of each distinct system call within the current window. The created bag of system calls is a frequency array of size n_s, where n_s is the number of distinct system calls. When a new occurrence of a system call is encountered, the application retrieves the index of the system call from the syscall-index hash map, and the corresponding index of the frequency array is updated. The new BoSC is then added to the normal-behavior database.

The created BoSC is then passed to the classifier, which works in one of two modes; training mode and detection mode. For training mode, the classifier simply adds the new BoSC to the normal-behavior database. If the current BoSC already exists in the normal-behavior database, its frequency is incremented by 1. Otherwise, the new BoSC is added to the database with initial frequency of 1. The normal-behavior database is considered stable once all expected normal-behavior patterns are applied to the container. Table 5.3 shows sample entries of a normal-behavior database.

For detection mode, the system reads the behavior file epoch by epoch. For each epoch, a sliding window is similarly used to check if the current BoSC is present in the database of normal behavior database. If a BoSC is not present in the database, a mismatch is declared. If the number of mismatches exceeds a certain threshold, T_d, within one epoch, an anomaly signal is raised.

Furthermore, a continuous training is applied during detection mode to further improve the false positive rate of the system. The bags of system calls seen during the current epoch are stored in a temporary current-epoch-change database rather than being added directly to the normal-behavior database. At the end of each epoch, if no anomaly signal was raised during the current epoch, the entries of the current-epoch-change database are committed to the normal-behavior database, to be included in classification for future epochs.

5.4 Evaluation using a single-server Application

To test the feasibility of the system, we started with a single-container test, for which we used a Docker container running on a Ubuntu Server 14.04 host operating system. The docker image we used for creating the container is the official `mysql` Docker image, which is basically a Docker image with MySQL 5.6 installed on a Debian operating system. Figure 5.3 gives an overview of the test environment used.

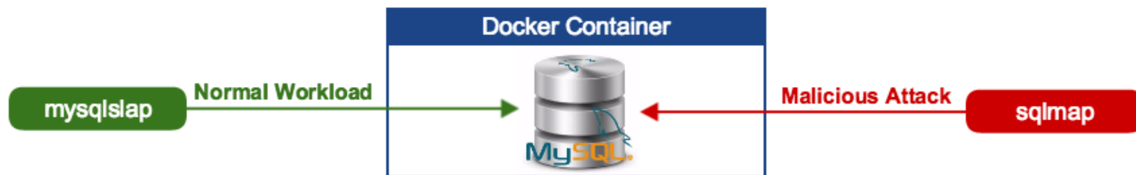

Figure 5.3: Single-server Test Environment for the Intrusion Detection System

On container start, the container automatically creates a default database, adds users defined by the environment variables passed to the container, and then starts listening for connections. Docker maps the MySQL port from the container to some custom port on the host.

Since there is no dataset available that contains system calls collected from containers, we needed to create our own datasets for both normal and anomalous behavior. For that, we created a container from the `mysql` Docker image. A normal-behavior work load was initially applied to the container, before it got "attacked" using a penetration testing tool.

5.4.1 Generating normal workload

For generating a normal-behavior dataset, we used `mysqlslap` [54]; a program that emulates client load for a MySQL server. The tool has command-line options that allow the user to select the level of concurrency, and the number of iterations to run the load test. In addition, it gives the user the option to customize the created database, e.g. by specifying the number of `varcher` and/or `int` columns to use when creating the database. Moreover, the user can select the number of insertions and queries to perform on the database.

The tool runs on the host kernel, and communicates with the MySQL server running on the container. The values we used for generating the normal-behavior workload are shown in table 5.4.

Additionally, we used the SQL dump file of a real-life database to create schemas, tables, views, and to add entries to the tables, on the MySQL server of the container.

Table 5.4: Parameters used for automatic Load generation

Parameter	Value
Number of generated `varchar` columns	4
Number of generated `int` columns	3
Number of simulated clients	50
Number of load-test iterations	5
Number of unique insertion statements	100
Total number of insertions per thread	1000
Number of unique query statements	100
Total number of queries per thread	1000

5.4.2 Simulating malicious behavior

To simulate an attack on the container, we used `sqlmap` [55]; an automatic SQL injection tool normally used for penetration testing purposes. In our experiment, we are using it to generate malicious-behavior dataset by attacking the MySQL database created on the container. Similarly, the `sqlmap` tool runs on the host kernel, and communicates with the attacked database through the Docker proxy.

We applied the following attacks on the container:

- Denial-of-Service (DoS) Attack: Using wild cards to slow down database. The attack generated an average of 37 mismatches
- Operating system takeover attempt: Attempt to run `cat /etc/passwd` shell command (failed). Generated 279 mismatches
- File-system access: Copy /etc/passwd to local machine. Generated 182 mismatches
- Brute-force attack: We used the `--all` option of `sqlmap` to retrieve all info about the database management system (DBMS), including users, roles, schemas, passwords, tables, columns, and number of entries. The attack was strong enough to generate around 42,000 mismatches.

5.4.3 Evaluation Metrics and Parameters

For evaluation purposes, the system-call parser recognizes start-of-attack and end-of-attack signals injected during the data collection phase to mark the epochs involved in the attack

as malicious. This information is used solely to accurately and automatically calculate the true positive rate (TPR) and false positive rate (FPR) metrics.

To evaluate the system accuracy with respect to different system parameters, we applied the classifier to the same input behavior file while varying the following test parameters:

- Epoch Size (S): The total number of system calls in one epoch. For our experiment, we used epoch size between 1000 and 10,000 with step of 500.
- Detection Threshold (T_d): The number of detected mismatches per epoch before raising an anomaly signal. We used values between 10 to 100 with a step of 10 for each epoch size listed above.

5.4.4 Preliminary Results

We applied the proposed system to a trace of $3,804,000$ system calls, of which the classifier used $875,000$ system calls for training. The number of distinct system calls (n_s) was 40, and the size of the normal behavior database was around $17k$ BoSCs.

The malicious data created a strong anomaly signal with an average of 695 mismatches per epoch, as compared to an average of 33 mismatches per epoch for normal data. For $S = 1000$ and $T_d = 10S$, the TPR is 100% and the FPR is 2%. Figure 5.4 shows the TPR and FPR of the system for different epoch sizes at the same detection threshold of 10. It can be seen that the lower the epoch size, the lower the FPR.

As shown in figure 5.5, the detection threshold highly affects the detection rate of the system especially when short-lived attacks are introduced to the container.

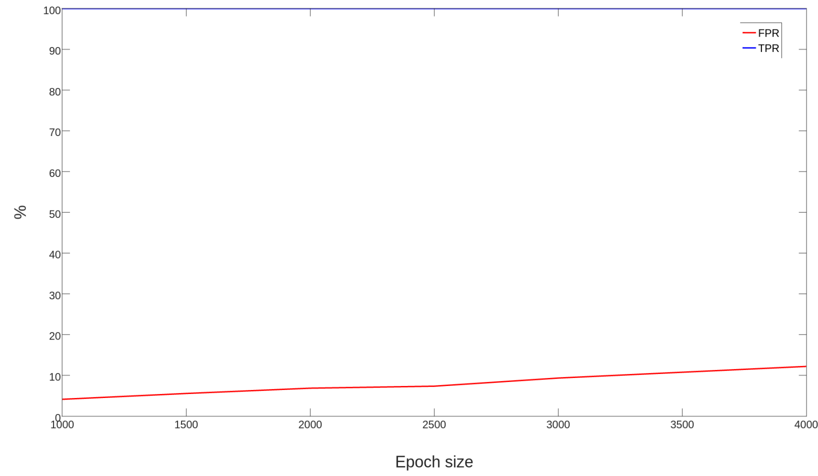

Figure 5.4: For constant detection threshold of 10 mismatches per epoch, the TPR is constant at 100%, while the FPR increases with the epoch size

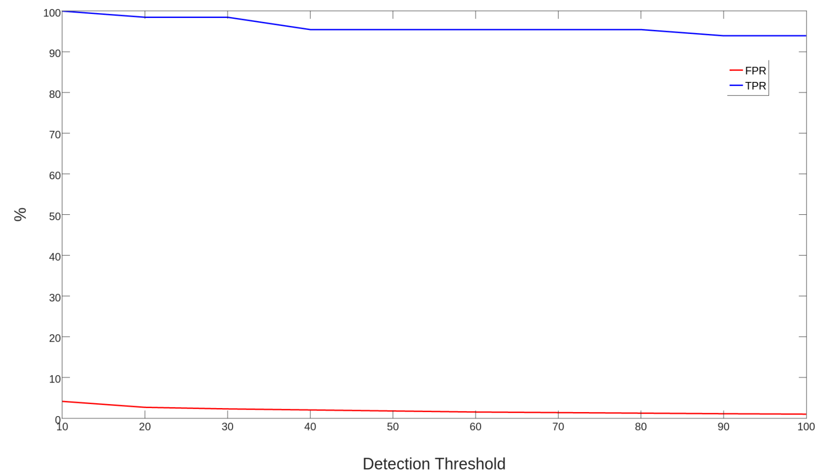

Figure 5.5: For constant epoch size of 1000 system calls, raising the detection threshold helps decrease the FPR at the expense of increasing the TPR

Chapter 6

Collaborative Behavior Modeling for Distributed Applications

For distributed applications, the system uses a hybrid mode of operation, in which each host is responsible for detecting any change in behavior of any container running on it, in addition to a centralized monitoring system that compares behavior of similar containers running on different hosts.

To test the feasibility of the proposed approach, we are using a real corporate-level big data application running in Docker containers on Amazon Web Services (AWS) cloud.

6.1 Cloud Test Environment

For our cloud environment, we are using Amazon Web Services (AWS) EC2 instances with Ubuntu operating system installed. The application we are using for cloud testing is a Yarn (MapReduce v2) application running on an Apache Hadoop cluster with each node running in a Docker container running on an EC2 instance. To automate the process of installing the cluster, we are using Apache Ambari to start the cluster. Figure 6.1 shows the details of the cloud environment we are using for testing the IDS in a corporate-level environment.

6.1.1 Environment Setup

The system works by using the AWS command line interface (AWS CLI) to remotely launch N EC2 instances from a local machine. The system then uses a pre-prepared shell script to initialize the instance using the cloud-init tool pre-installed on Ubuntu EC2 instances. The script installs and runs Docker along with the monitoring and classification systems described in chapter 5, and then starts a Docker container on each instance.

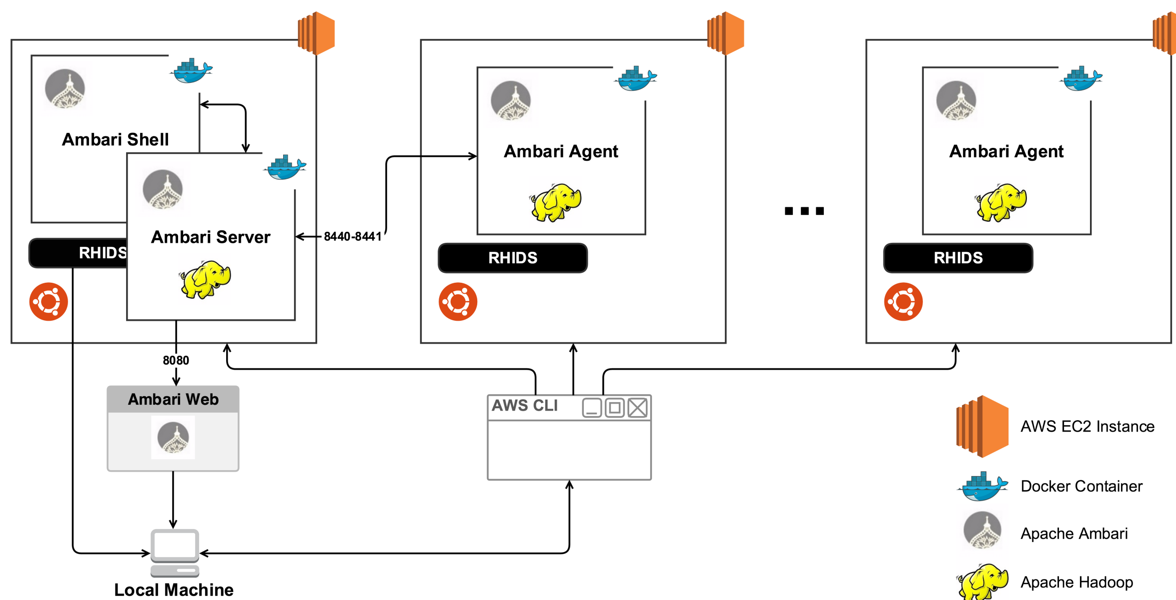

Figure 6.1: Cloud Test Environment - Ambari cluster in Docker containers running on AWS EC2 instances with RHIDS installed on Ubuuntu OS

As shown in figure 6.1, one of the instances has a Docker container with Ambari-server installed, while the other containers each has a container with an Ambari-agent running. In addition, the instance with the Ambari-server container has another container with Ambari-shell running. The Docker image we are using for starting the containers is the `sequenceiq/ambari:2.0.0` image, which is basically a Docker image that comes preconfigured with Ambari version 2.0.0 and Ambari shell.

Each of the containers takes the hostname of the Ambari-server as one of the environment variables. Once all $N-1$ agents are already connected to the server, the Ambari-shell installs the Hadoop cluster on the cluster nodes. In order to detect the connection of the $N-1$ agents to the server, the Ambari-shell container is connected to the Ambari-server container by using the Docker `--link` option.

6.1.2 Network Configuration

The security group used for the EC2 instances allows traffic to and from TCP and UDP ports expected to be used for communication between different cluster nodes and the outer world. One of the ports needed for monitoring the cluster is port 8080, where the Ambari web interface is running. The Ambari web is used to monitor the cluster from a local or a

remote machine. Moreover, an SSH port is used to communicate with the EC2 instance from the local machine. The port is used to transfer monitoring and classification data between the remote EC2 instance (with Docker container running) and the local machine.

For communication between cluster nodes, the `sequenceiq/ambari:2.0.0` Docker image internally relies on Serf [56]; an open-source cluster orchestration tool. Since the Docker containers are running on separate hosts, we needed to configure our system to pass the IP address of the server container as an environment variable to the agent containers. In addition, we passed the public IP of the EC2 instance to the container to be used as the Serf advertise IP, since the original behavior of the container was to advertise the internal local IP assigned to the container by the host OS.

6.2 Utilizing Prior Knowledge for Enhancing Accuracy

Upon start of each container, and with the monitoring service running in the background, the system traces system calls between the container and the host kernel and logs that into a log file, as described for the single-container test. The file is then read by the classifier. The classifier running on the instance detects when the container behavior changes and sends anomaly signal back to the controlling local machine. A second layer of detection is also added by sharing normal-behavior database between different containers through the centralized operator. A copy of the normal-behavior database of each container is passed to the local machine for two purposes:

1. Compare database from containers running similar jobs on different hosts, as they are expected to have similar databases.
2. Aggregate databases from similar containers into one database that is then used by each container for enhanced FPR.

6.2.1 Comparing Container Behavior

To compare the behavior of two containers, we calculate the similarity between the normal-behavior databases of both containers using the cosine similarity metric defined by the equation below. Two databases, db_A and db_B, are are identical if $Sim(db_A, db_B)$ is equal to 100%.

$$\begin{aligned} Sim(db_A, db_B) &= 100 \times \frac{db_A \cdot db_B}{\|db_A\| \|db_B\|} \\ &= 100 \times \frac{\sum_{i=1}^{\min(n_A, n_B)} db_A[i] db_B[i]}{\sqrt{\sum_{i=1}^{n_A} db_A[i]^2} \sqrt{\sum_{i=1}^{n_B} db_B[i]^2}} \end{aligned}$$

where $db[i]$ is the ith entry of the frequency vector of the database, and n is the total number of database entries.

We used the cosine similarity metric to compare the behavior of databases generated from two containers running an ambari-agent application, and to compare the database generated from the ambari-server container and both ambari-agent containers. Table 6.1 shows the similarity matrix of the three containers, namely ambari-server, ambari-agent-1, and ambari-agent-2. It can be seen that the similarity between the two agents is relatively higher than the similarity between each one of them and the server container.

Table 6.1: Similarity matrix of 1 server and 2 agent containers

	ambari-server	ambari-agnet-1	ambari-agent-2
ambari-server	100%	99.276%	99.076%
ambari-agent-1	99.276%	100%	99.968%
ambari-agent-2	99.076%	99.968%	100%

Our next step is to investigate the effect of centrally comparing databases of similar containers to each other on the TPR of the system, when only one of the containers is under attack. For example, by setting a similarity threshold of 99.9%, a centralized comparison of the behavior of similar containers is expected to further enhance the overall system accuracy. However, this is left as future work due to time constraints.

6.2.2 Aggregating databases of similar containers

We aggregated the normal behavior database from both Ambari agents into one database, and checked the similarity of the resulting database to the original database in both cases. Table 6.2 shows the similarity score between the aggregate database and the database of both agents, namely ambari-agent-1 and ambari-agent-2.

Table 6.2: Similarity between aggregate database and original databases

	Aggregate
ambari-agent-1	99.9895%
ambari-agent-2	99.9943%

While very similar, the values still show some deficiency of the original databases when compared to the aggregate one, i.e. a BoSC that may not be present in the original database, while still a normal one, would be detected as a mismatch resulting in a false positive. By using the aggregate database instead for similar containers (both agents in this case), the overall FPR of the system can be improved.

Chapter 7

LSTM-based Behavior Modeling

RNNs have been previously used for language modeling, i.e. to predict the next word in a text after being trained using a large database of text, e.g. the full work of Shakespeare. Our problem can be handled similarly. However, it is a more straightforward problem for two reasons:

1. The vocabulary used for our problem is considerably limited compared the dictionary of English words used in language modeling problems. The Linux Kernel (as of version 2.6.16) only has around 314 total system calls (for 64-bit architectures) which constitutes the full vocabulary for our system call prediction problem. In addition, the vocabulary size can further be reduced to around 300 system calls if we exclude system calls that are not yet implemented or reserved. Furthermore, if we only focus on the most commonly used system calls, and treating any other system call as a member of the 'other' class, we can go down to less than 100 system calls.

2. The network does not need to generate a meaningful piece of text as in the case of language modeling, which would require more network "intelligence". It just needs to learn the different sequence present within the current application.

By limiting the vocabulary size, and simplifying the task, we expect the network to be simpler, and the learning process to be faster. Figure 7.1 gives an overview of the logic flow and system architecture.

7.1 Network Configuration and Hyper-parameters

The hyperparameters of a neural network are parameters that are set prior to the actual training of the neural network, i.e. they are parameters that are not directly selected by

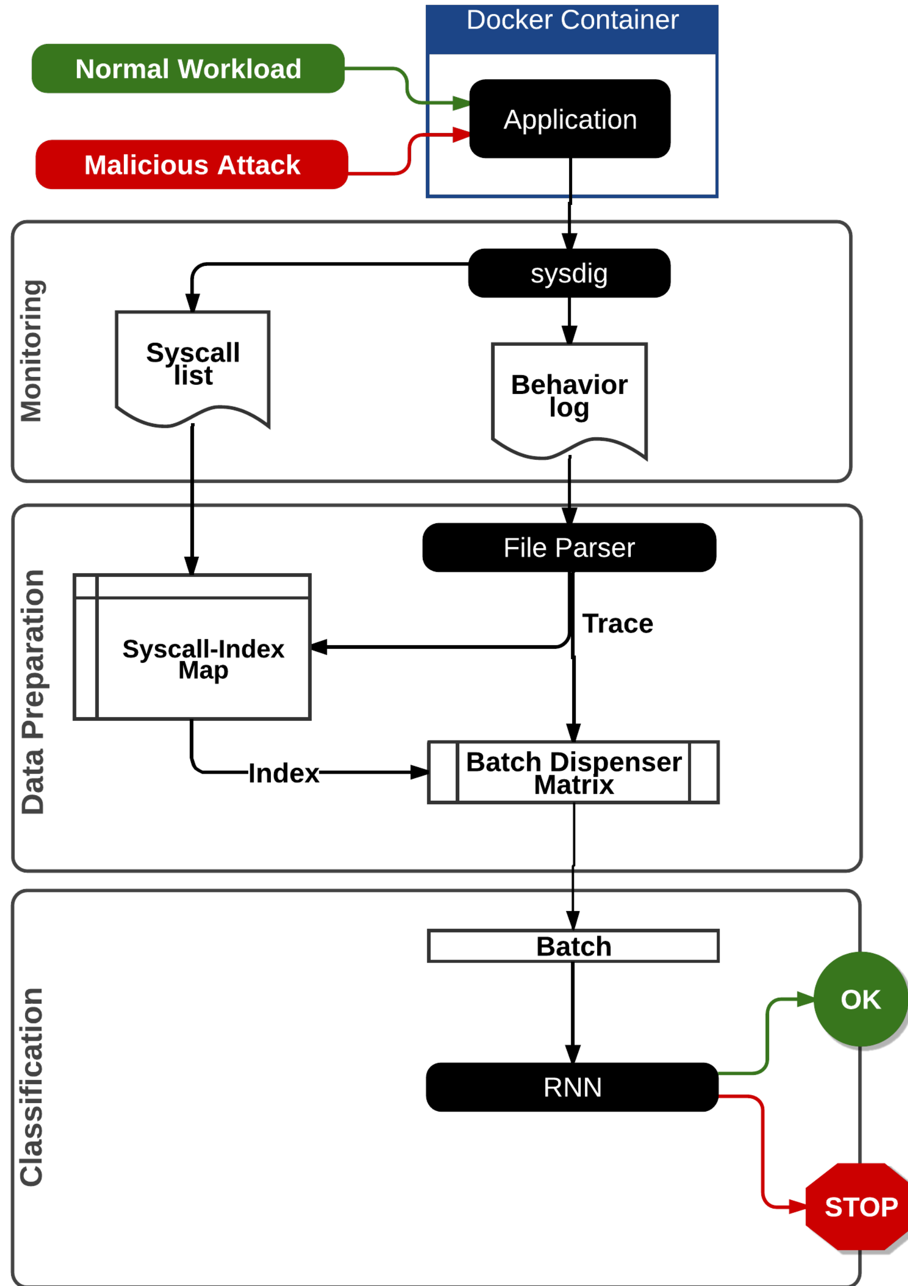

Figure 7.1: Logic Flow and System Architecture of the LSTM-based Intrusion Detection System

the learning algorithm [48]. Examples of hyperparameters when using gradient descent for training the neural network include loss function, learning rate, and batch size.

A deep learning architecture usually have more than 10 hyperparameters typically set using manual, grid, or random search. In our work, we are manually selecting values based on how they shown to perform in similar problems, such as language modeling. Table 7.1 summarizes the hyperparameters selected as described in the following subsections.

Table 7.1: Hyperparameters used for modeling container behavior using RNN

Hyperparameter	Value
Loss function	Equation 1.1
Initial learning rate (η_0)	0.01
Number of η_0 epochs (τ)	4
Learning rate (η) decay factor	0.5
Batch size (B)	32
Training epochs	13 (with early stopping)
Number of hidden layers	2
Number of hidden units (n_h)	200
Activation function	$softmax$
Initial Weights	$U(-0.1, 0.1)$

Loss Function

The loss function describes the gap between the network output and the actual expected output. The learning algorithm tries to minimize the loss function over training epochs by adjusting the weights of the hidden units of the neural network.

For example, the loss function we are using for our problem is the average negative log probability of the target words as defined by equation 1.1 (repeated below).

$$loss = -\frac{1}{N}\sum_{i=1}^{N} ln(p_{target_i})$$

Learning Rate

The value of the learning rate η is the most important hyper-parameter to be tuned for better accuracy of the training algorithm [48]. If the learning rate is too large, the average

loss will increase. Typical values of the learning rate are between 10^{-6} and 1.

Starting from the initial value η_0, the learning algorithm typically decreases the learning rate by a certain factor over epochs to improve accuracy. An example learning rate schedule is defined by:

$$\eta_t = \frac{\eta_0 \tau}{max(t, \tau)}$$

where t is the current training epoch, and τ is the number of epochs for which the initial rate λ_0 is used. Using this schedule, the learning rate remains constant for τ epochs before being decreased by a factor of t.

For our system, we are using initial learning rate $\eta_0 = 0.01$ which is constant for the first $\tau = 4$ epochs then decreases by a factor of 0.5 for each subsequent epoch as shown in figure 7.3.

Batch Size

The training set is usually divided into batches with the weights updated after the calculation of loss function for each batch. The batch size B can be anywhere between 1 and the training set size, and is typically chosen between 1 and a few hundreds. If $B = 1$, the weights are updated after each sample of the input data. On the other hand, if B equals the training set size, the update is done only once after each training epoch.

However, larger B (e.g. $B > 10$) yield faster computation, since it benefits from parallelism and efficient matrix-matrix multiplications, but requires visiting more examples in order to reach the same error, since there are less updates per epoch. In other words, the effect of B is mostly computational, i.e., it should only affect the training time rather than the test performance.

For our problem, we are using batch size $B = 32$, which is usually a good default value [48].

Number of Training Iterations

The number of training iterations T is the number of batches used for training the network. T can be optimized by using *early stopping*, i.e. stopping the training when the validation error stops improving. Early stopping also evades overfitting [48]. For our problem, we are using 13 epochs of training with early stopping.

Number of Hidden Units

The number of units in hidden layers n_h generally affects generalization performance, i.e. larger n_h yields better generalization performance, yet requires more computations. Using

the same size for all hidden layers generally works better or the same as using a decreasing or increasing size. $n_h = 200$ for all layers for our experiment.

Activation Functions

A non-linear activation function is typically used for neuron outputs. Common examples are $softmax$, $sigmoid$, $tanh$, and $ReLU$. We are using $softmax$ as defined by:

$$softmax = \frac{\exp(logits)}{\sum(exp(logits), dim)}$$

Initial Weights

The initial values of the network weights impact the local minimum found by the training algorithm. Weights are often initialized using a random uniform distribution in the range of $[r, r]$ where r is the inverse of the square root of the fan-in added to the fan-out of the unit. For our problem, we initialize the network weights uniformly in the range $[-0.1, 0.1]$

7.2 Dataset Preparation

To train the LSTM network, we are using a trace of system calls collected from a Linux container running the same application with normal behavior. The trace of system calls is divided into training data and validation data.

The raw data from the trace file is first converted into an integer array D of size S_{data} where S_{data} is the total number of system calls in the input data. To be passed to the LSTM, the data is then reshaped into a batch dispenser matrix B of size $S_{batch} \times N_{batch}$ where S_{batch} is the batch size which is equal to the size of the input layer and the output layer of the LSTM, and $N_{batches}$ is the total number of batches calculated as $mod(S_{data}, S_{batch})$.

Matrix B is then sliced into a number of input and output matrices of size $S_{batch} \times N_{steps}$ each, namely X_i and Y_i respectively, where matrix Y_i is a one-step shift of matrix X_i as follows:

$$X_i = B[1 : S_{batch}, iN_{steps} : (i+1)N_{steps}]$$
$$Y_i = B[1 : S_{batch}, iN_{steps} + 1 : (i+1)N_{steps} + 1]$$

where $0 \leq i \leq I$, N_{steps} is the number of unrolled steps, and $I = \lfloor N_{batches}/N_{steps} \rfloor$ is the total number of iterations per epoch.

The input and output slice matrices are then used for training the LSTM. An epoch is a complete run over the whole data. The number of epochs N_{epochs} depends on the selected configuration. Figure 7.2 summarizes an example of the data preparation process.

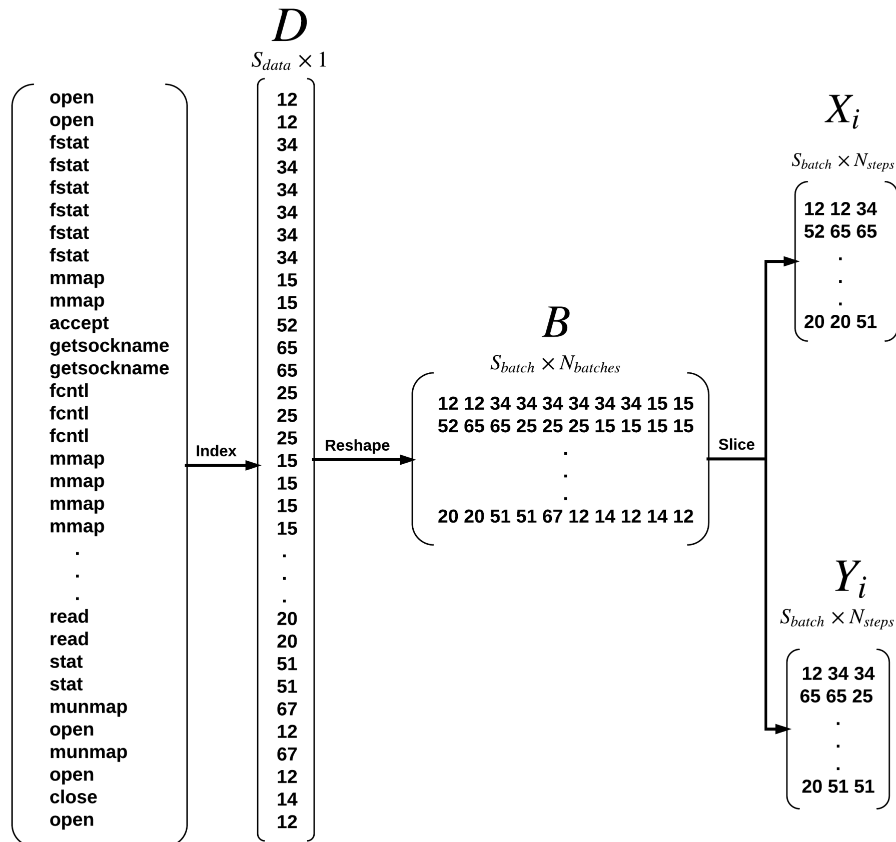

Figure 7.2: Data Preparation: raw data is indexed into an array D then reshaped into batch dispenser matrix B which then sliced into input and output matrices X_i and Y_i respectively

7.3 Container Behavior Modeling

To learn the behavior of a Linux container, we are using an RNN to predict the next system call based on the current one. The network is first trained using a portion of the trace of system calls, then another portion of the trace is used for validation. Typically, the network is trained to minimize the average per-word perplexity, or simply perplexity, as defined below. For our problem, however, we can use a threshold value of the perplexity for early stopping, since it was noticed that the minimum value is reached after a few epochs of training anyway.

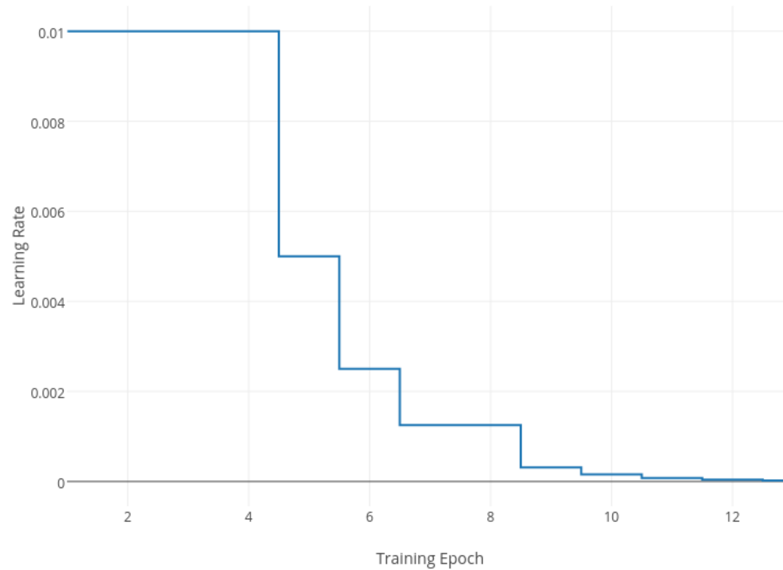

Figure 7.3: Learning rate decays by a factor of 2 every epoch after 4th epoch with initial value of 0.01

Finally, the network is used to predict following sequences while calculating the perplexity for evaluating the prediction accuracy.

$$Perplexity = e^{-\frac{1}{N}\sum_{i=1}^{N} \ln p(x_i)} \tag{7.1}$$

To train the network using the dataset, we pass the batches generated to the LSTM network. As described in section 4.3, the network is a non-regularized (0% dropout) LSTM network with 200 units in each of the two hidden layers. The number of unrolled steps of the LSTM (N_{steps}) is 20. The weights of the network are initialized uniformly in the range $[-0.1, 0.1]$. The learning rate is 0.01 for the first four training epochs then decays by a factor of 2 for each subsequent iteration as shown in figure 7.3. Figure 7.4 shows the training and validation perplexities over training epochs.

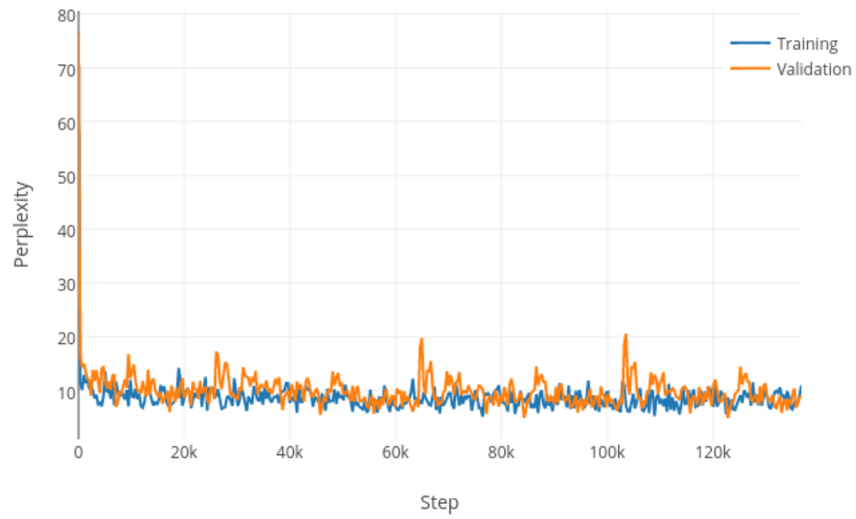

Figure 7.4: Training and validation perplexities while modeling container behavior using LSTM

Chapter 8

Test Case: Detecting Intrusion in SaaS Cloud Application

8.1 Cloud Test Environment

The cloud application we are using is a typical web application consisting of a back-end MySQL server and a front-end PHP-based Apache web server. Each server runs on a Linux container and communicate with each other through the default Docker virtual network. The MySQL and PHP containers are instantiated from the official Docker `mysql` and `php:apache` images respectively. A third container running on a remote server with Kali Linux installed is used to emulate both normal behavior and malicious attack against the web server application. Figure 8.1 shows the described test environment.

8.2 Threat Model

We are using a threat model that an attacker would typically use against a web application, starting from collecting information about the server, including but not limited to the underlying operating system, running services and applications, and system users, up to running a number of common attacks such as denial of service (DoS) and SQL injection attacks [57].

To apply those attacks, we are using the Metasploit framework running in a Kali Linux container. Metasploit is one of the most commonly used penetration testing tools and frequently used by attackers. Only successful attacks are counted toward the calculation of the detection rate.

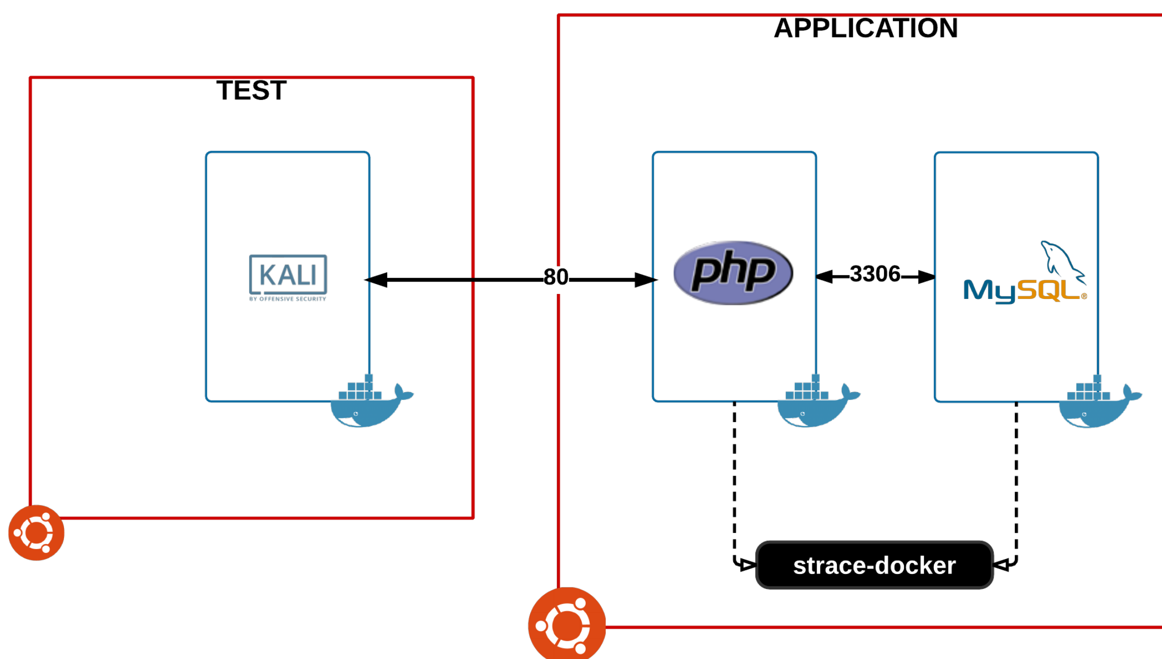

Figure 8.1: A web application consisting of MySQL server and PHP server in two containers tested using Kali Linux running in a remote container

Port Scanning

To detect running services on a remote server, an attacker typically starts by scanning for open ports on the server. Open ports reflect the nature of the services running on the server. For our application, we had ports 80 and 3306 open, i.e. HTTP and MySQL services running.

Vulnerability scanning

The next step of the information gathering phase is to scan for system vulnerabilities. In this step, the attacker collects more information about the server and applications and services running on it that may cause known vulnerabilities.

SQL Injection Attack

A SQL injection attack injects malicious code into a SQL query passed to the webserver in an attempt to reveal unauthenticated system or user information.

Denial of Service (DoS) Attack

DoS attacks are common attacks against web applications among other services. A DoS attack tries to make services unavailable to intended users by temporarily or permanently disrupting services. One of the most common DoS attacks is the flooding attack, where an attacker targets a machine with more requests than the service can handle in an attempt to cause legitimate requests to be fulfilled. In our experiment, we are using a `SYN` flood attack.

8.3 Experimental Results

We tested both machine learning techniques when applied to the behavior modeling and classification modules of the system, namely the BoSC technique and the deep learning technique.

In our experiments, we tested the effect of system parameters, e.g. detection threshold (T_d), on the accuracy of the system as determined by the accuracy metrics defined in section 3.2, namely TPR, FPR, accuracy, precision, and F_β score, defined by equations 3.1 - 3.6 (repeated below).

$$TPR = \frac{N_{TP}}{N_{malicious}}$$

$$FPR = \frac{N_{FP}}{N_{normal}}$$

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

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

$$F_\beta = (1 + \beta^2)\frac{Precision * TPR}{\beta^2 * Precision + TPR}$$

8.3.1 Behavior Modeling and Classification using BoSC

As discussed in chapter 5, the BoSC techniques learns the behavior of the container in real time. We manually signal the system to stop training after applying all the expected normal behavior. The system then switches into a hybrid training and test mode where it detects when the number of mismatches exceeds certain detection threshold (T_d) within one ranges of size S_e. A mismatch is the absence of an BoSC from the normal behavior database.

In our experiments, we tested the effect of the epoch size (S_e) and the detection threshold (T_d) on the system accuracy. We define N_{normal} and $N_{malicious}$ to be the total number of normal and malicious epochs respectively, while N_{TP} is the number of malicious epochs

correctly declared by the system as anomalous, and N_{FP} is the number of normal epochs mistakenly declared as anomalous.

Figure 8.2 shows the ROC curves for using the BoSC technique with three different epoch sizes, namely 1000, 5000, and 10000 system calls per epoch. The ROC curves are created by changing the detection threshold between 0% and 100% (with a step of 1%) of the epoch size for each case. We can see that using an epoch size of 5000 system calls, we can reach a detection rate as high as 99.3576% with a low FPR of 1.0511%. A slightly better detection rate of 99.5717% can be achieved with a FPR of 3.003%. Similar detection rates are also achievable for the 1K and 10K epoch sizes but with a higher FPR. A 100% detection rate is only achievable for an epoch size of 10,000 system calls at the expense of the unacceptably high FPR of 73.4139%.

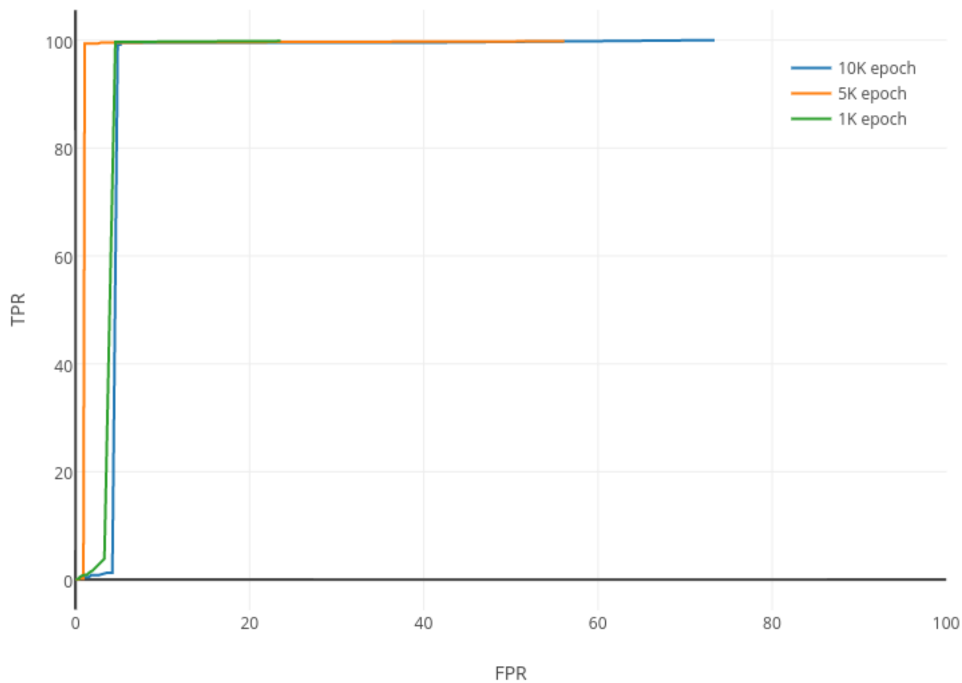

Figure 8.2: ROC curves for three different epoch sizes using BoSC

Table 8.1 shows sample value pairs of epoch size and detection threshold along with respective TPR and FPR values for each case. The highlighted values shows the best values of the TPR and FPR for each epoch size. Only the minimum values of FPR for which the TPR is greater than 99% are shown. Figure 8.3 shows the change of the accuracy measures with respect to the detection threshold.

Table 8.1: Accuracy measures for select values of epoch size and detection threshold

S_e	T_d	N_{TP}	N_{FP}	N_{TN}	N_{FN}	FPR	TPR	Precision	Accuracy	$F_{0.5}$
1000	0%	2316	925	2424	4	27.62%	99.83%	71.46%	83.61%	75.77%
1000	1%	2314	471	2878	6	14.06%	99.74%	83.09%	91.58%	85.96%
1000	2%	2312	216	3133	8	**6.45%**	**99.66%**	91.46%	96.49%	92.99%
5000	0%	466	375	291	1	56.30%	99.79%	55.41%	66.81%	60.82%
5000	4%	465	25	641	2	3.75%	99.57%	94.89%	97.62%	95.8%
5000	7%	464	6	660	3	**0.9%**	**99.36%**	98.72%	99.2%	98.85%
10000	0%	235	243	97	0	73.41%	100%	49.16%	57.07%	54.73%
10000	2%	234	33	298	1	9.97%	99.57%	87.64%	93.99%	89.79%
10000	3%	233	17	314	2	**5.14%**	**99.15%**	93.2%	96.64%	94.33%

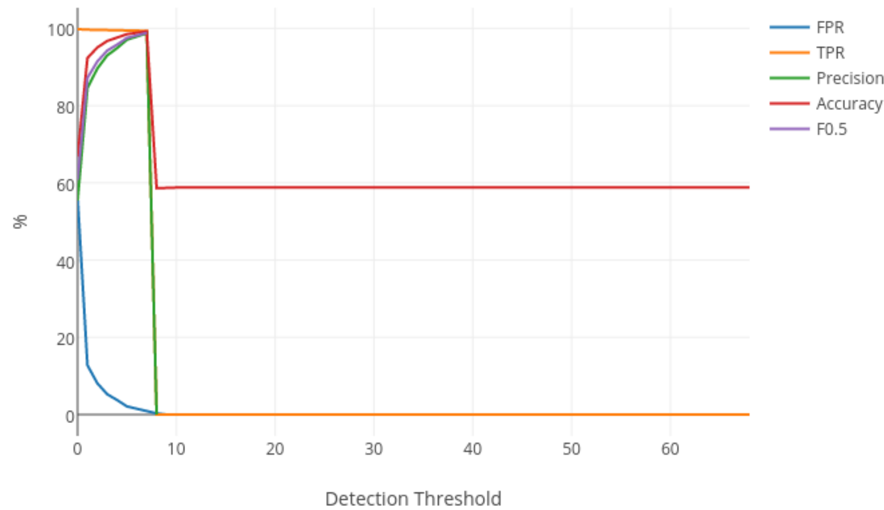

Figure 8.3: Accuracy measures for different values of detection threshold when using BoSC

In addition to varying the detection threshold for different epoch sizes, we have tested the effect of varying the epoch size for the same detection threshold, e.g. varying the epoch size between 500 and 10000 with a step of 500 for the same detection threshold of 1%. However, we noticed that the resulting ROC curve for each detection threshold value tended to be more

of a single point than a whole ROC curve indicating that adjusting the detection threshold is more effective in determining the system accuracy.

8.3.2 Behavior Modeling and Classification using LSTM

As described in chapter 7, we applied the input trace to a LSTM-based RNN to learn the container behavior in offline mode where the network iterates over the training data multiple times minimizing the average per-word perplexity loss function defined by equation 1.2. The data is applied to the network after being prepared into batches of size 32. The perplexity is calculated after applying each batch. The network was trained for 13 epochs before being applied to the test data.

Figure 8.4 shows the test perplexity of a portion of the input trace. The peaks in the figure represents malicious ranges where the test perplexity exceeded the detection threshold at least once. The vulnerability scanning attack was the strongest and longest attack where the perplexity constantly exceeded 13 (as compared to less than 2 for the normal ranges) for more than 3000 steps. Figures 8.5 and 8.6 zoom in to show the test perplexity for normal versus malicious ranges of the system call trace, respectively.

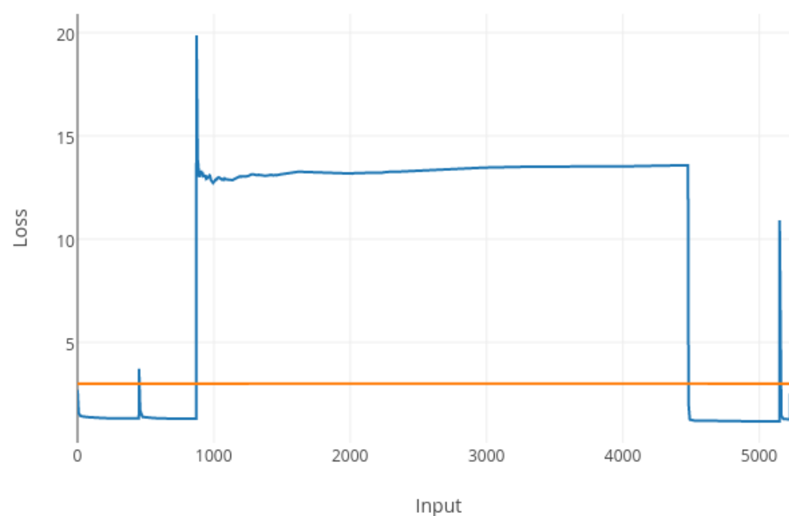

Figure 8.4: Test perplexity for portion of the input trace

Figure 8.7 shows the ROC curve of using LSTM for learning container behavior. The curve was created by changing the detection threshold between 0 and 20 with a step of 1. Again,

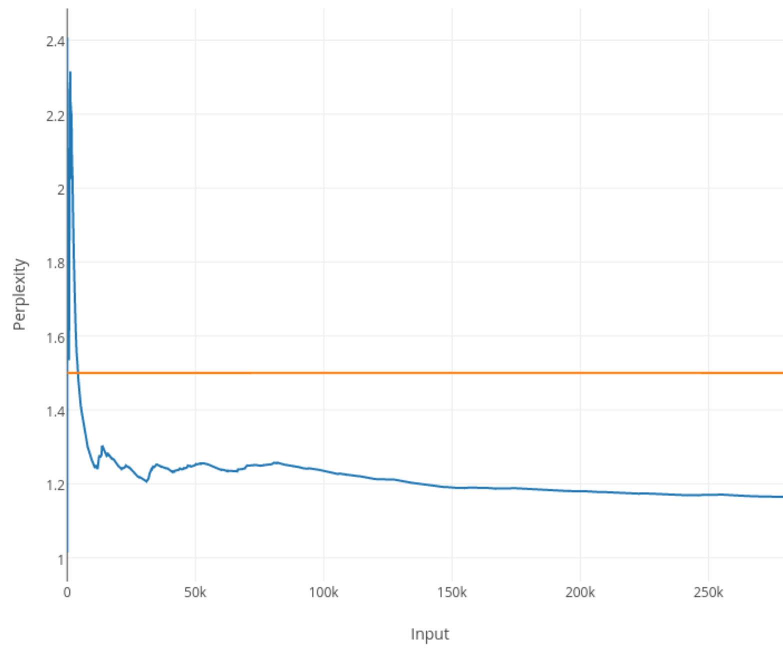

Figure 8.5: For normal ranges, the perplexity rapidly drops down to less than t_d

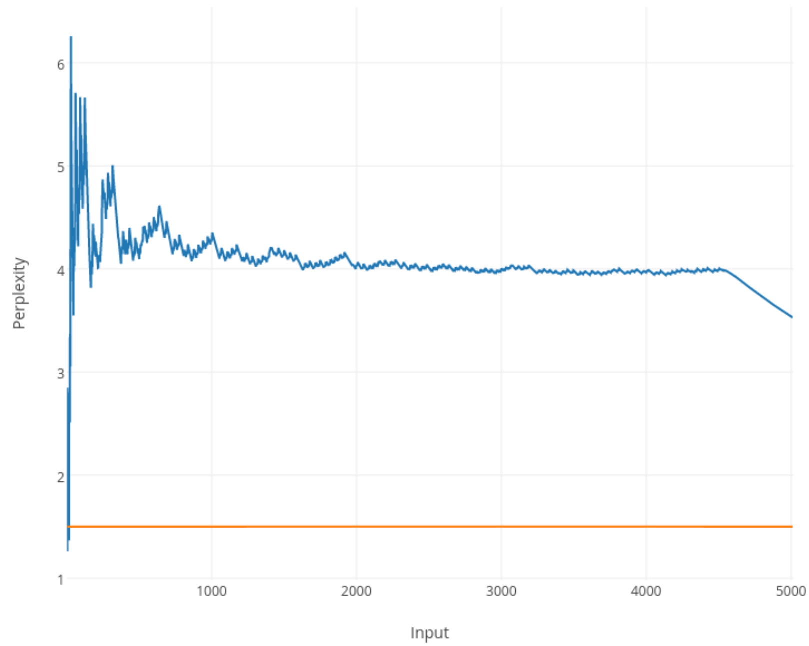

Figure 8.6: For malicious ranges, the perplexity typically exceeds t_d during the attack

the TPR and the FPR are calculated using equation 3.1 and 3.2 receptively. However, we now define N_{normal} and $N_{malicious}$ to be the total number of normal and malicious *steps* respectively. N_{TP} is the number of malicious steps correctly declared by the system as anomalous, and N_{FP} is the number of normal steps mistakenly declared as anomalous. We can see that we have a semi-perfect ROC curve with 0% FPR and 99.889% TPR achievable for a detection threshold as low as 3. A slightly better TPR is achievable using $T_d = 2$ at the expense of raising the FPR to 0.3%. Table 8.2 summarizes the accuracy measures for both values.

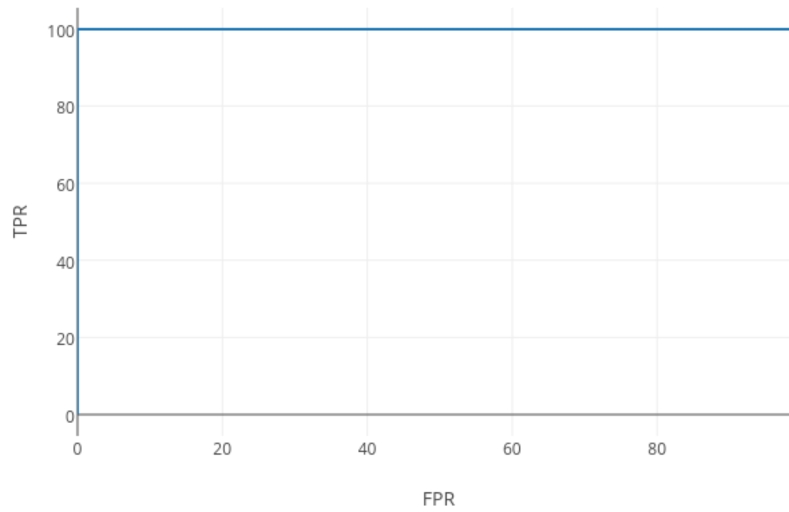

Figure 8.7: ROC curve for using LSTM for behavior modeling and classification

Table 8.2: Accuracy measures for select values of epoch size and detection threshold

T_d	N_{TP}	N_{FP}	N_{TN}	N_{FN}	FPR	TPR	Precision	Accuracy	$F_{0.5}$
2	3613	6	1672	3	0.36%	99.92%	99.83%	99.83%	99.85%
3	3612	0	1678	4	0%	99.89%	100%	99.92%	99.98%

We also calculated the accuracy measures for different values of T_d. Figure 8.8 shows how the system performs, measured using the accuracy metrics defined by equations 3.1 - 3.6, when varying the detection threshold.

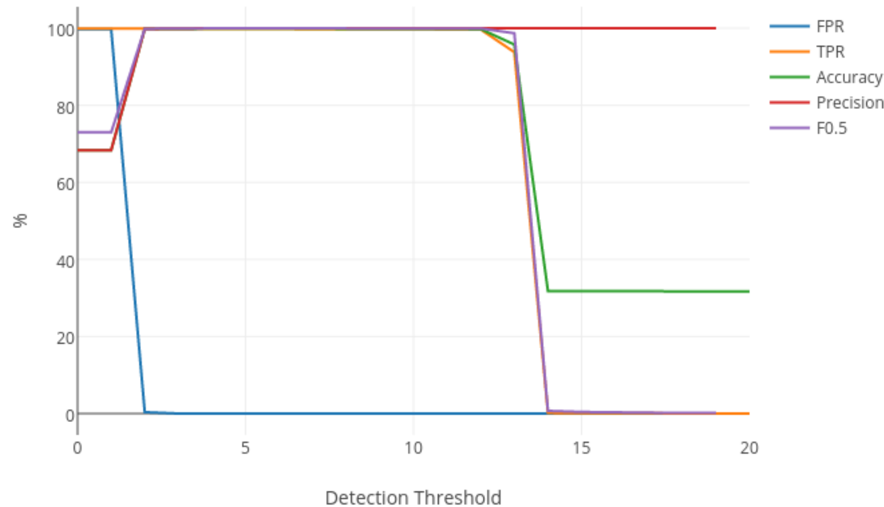

Figure 8.8: Accuracy measures for different values of detection threshold when using LSTM

8.4 Discussion

8.4.1 Memory-based versus Deep Learning Behavior Modeling

To better compare the accuracy of the LSTM approach to that of the BoSC technique, figure 8.9 shows the ROC curves of both the BoSC ($S_e = 5000$) and the LSTM versions of the behavior modeling module.

We were able to achieve a detection rate (TPR) of 99.889% with FPR of 0% using LSTM with a detection threshold as low as 3 for the test perplexity. Lowering the detection threshold to 2 would result in a slightly better TPR of 99.917% at the expense of raising the FPR to 0.3576%. In contrast, the BoSC technique resulted in a detection rate of 99.36% with FPR of 0.9% using a detection threshold of 7% of the epoch size of 5000 system calls. The accuracy of the LSTM version was 99.924% as compared to 99.2056% for the BoSC alternative. The number of false negatives (N_{FN}) was 4 steps for the LSTM and 3 epochs for the BoSC. The number of false positives (N_{FP}), however, was 0 for the LSTM as compared to 6 for the BoSC. The time-to-detection (TTD) is between 350 and 5000 system calls for the BoSC as compared to 640 system calls for the LSTM. Table 8.3 summarizes those differences between both approaches.

The slightly below perfect values of the TPR for both techniques is a result of the way

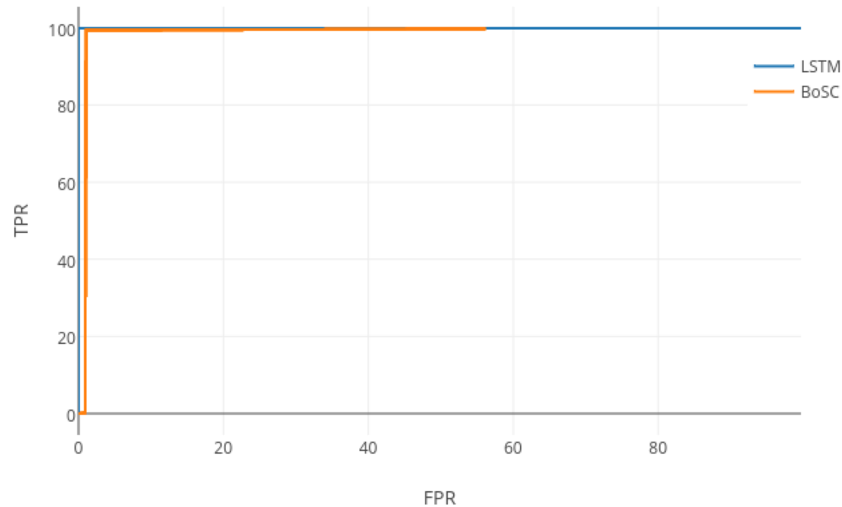

Figure 8.9: Comparing accuracy of BoSC and LSTM behavior modeling and classification

Table 8.3: Comparing performance and accuracy of using LSTM versus BoSC

	LSTM	BoSC
Step Size	640	5000
TPR	99.889%	99.36%
FPR	0%	0.9%
Accuracy	99.924%	99.2056%
Precision	100%	99.72%
N_{FN}	4	3
N_{FP}	0	6
TTD	640	35 - 5000

the TPR is calculated as the number of detected malicious steps rather than the number of detected attacks. Nevertheless, both techniques were actually able to detect all applied attacks in our case since every single attack lasted over many epochs/steps, yet maybe one of those malicious steps were not shifted enough from the normal behavior and hence was considered a normal one by the system. If we consider only the number of detected attacks

instead, the actual TPR would be 100% in our case for both techniques.

In an attempt to enhance the FPR of the BoSC technique, the system can be adjusted to only raise an anomaly signal if a few consecutive epochs are detected as anomalous. However, that may affect the TPR if a short-lived attack does not cause behavior shift that lasts over multiple epochs.

Training Time

One more major difference between the LSTM and the BoSC alternatives is the training time. The LSTM needs hours of offline training in advance before being used for behavior classification. For instance, the LSTM used in our experiment needed 14 hours and 40 minutes to complete 13 epochs of training on a Dell PowerEdge 2850 server with 3GHz dual-core processor and 8GB memory. Using a more powerful server for training may have reduced the training time. However, real-time training of the network is not possible due to the fact that the LSMT needs to go over the whole training data for multiple iterations. The BoSC technique, on the other hand, is able to learn the container behavior in real time, yet resulting in a slightly lower accuracy, i.e., there is a trade-off between the system accuracy and the time needed for training the behavior modeling module.

We conclude that the BoSC alternative would be more suitable to the case where the behavior of the container is to be learned in real time, i.e., when the cloud user provides a new container that has not been previously seen by the system. However, if the cloud service provider (CSP) is responsible for providing pre-configured containers to the users, e.g. providing containers running programming platforms in a PaaS cloud, the CSP can then use the LSTM version to train system in advance before monitoring the actual containers used by the users.

8.4.2 Comparing to other Cloud IDS

Our system is the first intrusion detection system designed specifically for container-based clouds. As discussed in chapter 3, current cloud-based intrusion detection system were designed for VM-based clouds and or suffer from one or more of the following limitations:

- Requires alteration of the host, hypervisor, and or the guest system
- Requires prior knowledge of the OS or applications of the guest system
- Requires prior knowledge of the attack patterns, i.e., signature-based IDS
- Requires administrative attention
- Imposes overhead by hosting another VM for detection purposes

- Detection technique is complicated and requires tuning of many parameters
- System accuracy is low or not reported

Table 8.4 summarizes the most-commonly used accuracy measures, namely TPR, FPR, and accuracy, reported for different cloud IDS compared to our system. It is worth mentioning, however, that the different nature of VM versus containers may introduce other factors that affect the IDS accuracy and performance.

Table 8.4: Summary of reported accuracy measures of Cloud IDS compared to our system

IDS	**TPR**	**FPR**	**Accuracy**
Abed (LSTM)	100%	0%	99.92%
Abed (BoSC)	100%	0.9%	99.2%
Alarifi [29]	100%	5.66%	97%
Alarifi [28]	100%	11.11%	98.36%
Gupta [31]	80%	3%	NR
Gupta [32]	89%	0%	98%
Modi [37]	96.71%	1.91%	NR
Pandeeswari [39]	97.55%	3.77%	NR
Arshad [30]	90.8%	NR	NR
Kim [33]	99%	NR	NR
Li [34]	NR	NR	99.7%

Chapter 9

Incident Handling and Remediation

The resiliency module of the system relies on the intrusion detection module to trigger it to protect the attacked container[1].

While a straightforward action is to kill a misbehaving container and inform the owner of the detected anomaly, such action may not be feasibly cost effective especially for long running stateful applications. For instance, if an application is running for a few days, and it gets attacked by an external attacker, and due to the behavior change, recognized by the monitoring system as a potential attack, the container is terminated. Now, the owner will need to restart the application and have it run for more few days just to get to the same point it was already at when it got terminated. The unnecessary added cost would be even more problematic if it is due to some false alarm.

A more suitable alternative is to capture a snapshot of the current status of the running application while in safe state. Upon attack detection, the system simply rolls back to the most recent safe state saved. One drawback with this approach is when the last saved safe state is a vulnerable state and/or the attack is persistent, in which case the container will go into a continuous loop of restores.

To overcome such limitation, a possible approach is to change the environment where the container is running in order to mislead a persistent attack, e.g. by moving the container to a different host in the cloud. Therefore, life migration of the attacked container can be deployed as a moving target defense (MTD) mechanism for protecting applications running in Linux containers from persistent attacks. The life migration of the container is triggered when the intrusion detection module of the system raises an anomaly signal.

Alternatively, for stateful applications, the system could be configured to start by using the checkpoint/restore mechanism before switching to the life migration solution for a more persistent attack. Delaying the use of life migration until the attack is known to be persistent

[1]Work is done in collaboration with Dr. Mohamed Azab and Dr. Bassem Mokhtar from Alexandria University and the City of Scientific Research and Technological Applications in Alexandria, Egypt

(e.g. after N rollback attempts) saves the running application the overhead associated with life migration as compared to checkpoint/restore for non-persistent attacks. For a stateless application (e.g. a webserver), however, it would be sufficient to simply restarts the server or migrates the container to a new host while rerouting the associated network connections.

9.1 Checkpoint and Restore

The application running inside the container, whether stateless or stateful, uses the host memory to store all runtime libraries, calculations, and other volatile contents. The mechanism used is intended to require minimal to no interference from the administrator or the programmer, and to avoid any container customization. Checkpointing is only necessary for containers with stateful applications. For stateless applications, the memory contents and the executed states are not important for container restoration.

To enable checkpointing of running application, we are using an experimental version of Docker that integrates a checkpointing tool named CRIU[58] to dump the running container and its enclosed applications taking a live snapshot of the memory content and any used files. The dumped images are stored into a persistent storage. CRIU is a tool to checkpoint/restore running applications in user space. CRIU momentarily freezes the running runC process (the container) and all its sub-processes and checkpoint it to a collection of image files that can be used to restore the container to the exact state later. Between these dump events, containers uses the host memory to operate to maximize the application response rate.

For a failed container to be restored, the hosting server must have access to the container image files, and the memory dump files. The container image files is usually large in terms of space. In our experiments, containers with full database server can be as large as 500MB. However, the memory dump is usually less than 10MB. The migration process for stateless type is much easier, we replicate the container on the destination server, then make a quick network switching between the source and destination. Figure 9.1 shows a timeline describing the checkpoint and restore process.

For faster instantiation, quick recovery, and easy container migration, the system uses a remote shared storage as a container repository to store runC containers. Running the container from a remote storage gives instant access to multiple remote servers to instantiate such containers. Using remote repositories to host the base image of the container, massively reduced the time needed to move all the files between hosts in case of failure or live-migration. The only files that has to be synchronized between the source and destination servers are the memory dump which are so small and synchronize momentarily.

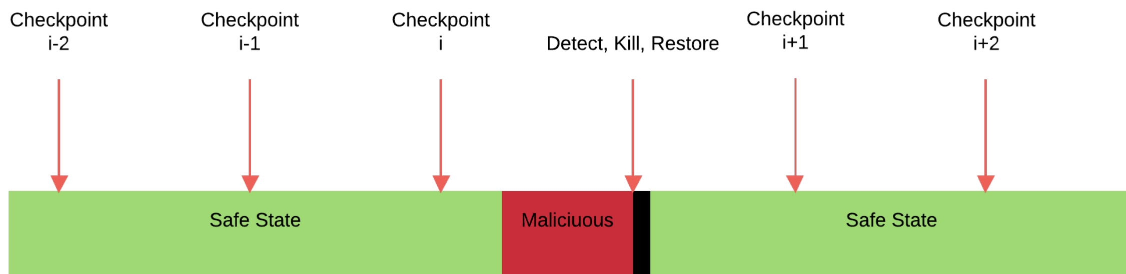

Figure 9.1: Checkpoint and Restore of a misbehaving container

9.2 Live Migration

For stateless applications, there is no need for checkpointing as the memory content is irrelevant for application re-execution. The relaxation of that constraint made it much easier for our system to recover such container in case of failure/attack.

In the case of a stateless application, the container can be instantiated from the original image and using the application files stored on a shared storage, or copied from the original host. The new container can be started on the same host or a remote host in case the attack is originating from a malicious container on the current host attacking neighbor containers.

Unlike the checkpoint and restore option, Docker does not natively support live migration yet. However, the migration process can be implemented as shown in figure 9.2 as follows:

1. Start new container on the destination host using the same image as the original container.
2. Make an ARP update to change the MAC/IP assignment of the old server network interface to match the new one while mainlining the IP value.
3. Kill the misbehaving container on the source.

Technical Considerations

To facilitate the migration process, we need to take the following considerations into account to minimize the down time of the migrated application:

- Image must be present on the destination server before starting the migration process, i.e. before killing the original container. Otherwise, the time for downloading the image from the Docker store would add extra latency to the migration process.

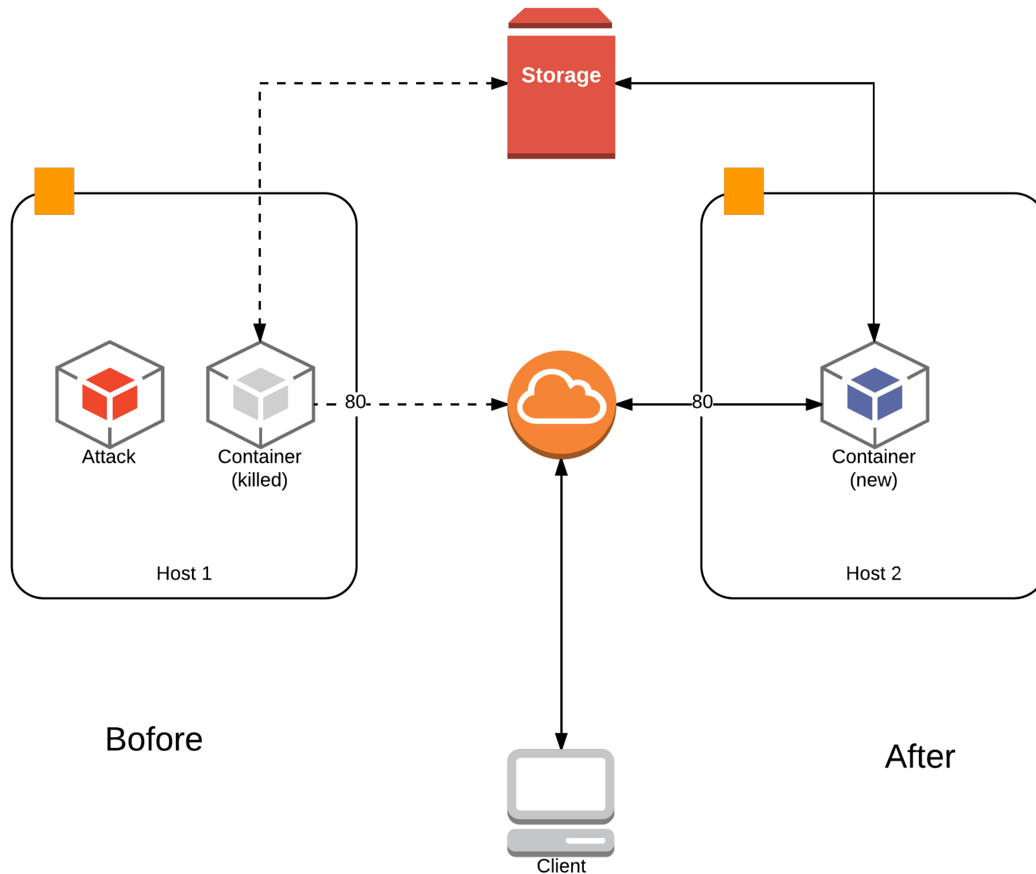

Figure 9.2: Migration process

- Images on both the source and destination servers must have the same software version, i.e. same image version must be used on both servers, to avoid any application failures that may occur after migration.
- The checkpoint/restore folder should be accessible and writable by both source and destination, e.g. by using a shared persistent storage space, to release the need to copy the dumped container files from the source to the destination server.

9.3 Replication and Isolation

In many cases, the owner of the attacked system may be interested in keeping the attack running after detection, e.g. to learn the attacker behavior or to start a counter attack. In

that case, a possible approach is to replicate the running container in a honeypot environment and forward all network traffic from the attacker to the honeypot. The original container can then be restored to a safe state where it continues to receive legitimate access.

Alternatively, an infected container can be isolated into a quarantine zone for further analysis and to prevent the attacker from attacking other guest containers or the host from the infected container. Figure 9.3 shows the replication and isolation process.

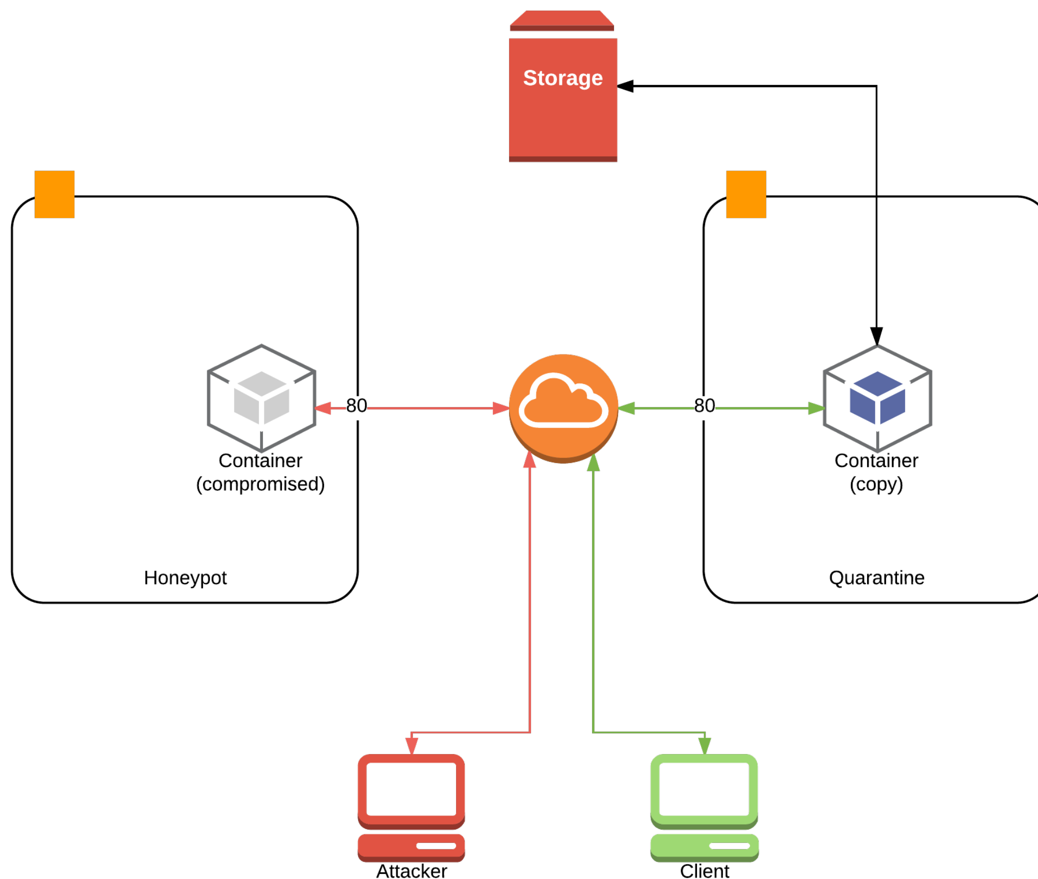

Figure 9.3: Replication and Isolation of Misbehaving containers

9.4 Prey-and-Predator Model

In our recent study [59], led by Dr. Azab, Dr. Mokhtar tested a *prey-and-predator* model applied to containers under attack. The model simulates and evaluates the effectiveness

of mobilizing targeted containers (*prey*) across a set of attack-prone hosts within a virtual network considering the case where some of the containers hosted at those hosts are used to attack neighboring containers (*predators*).

9.4.1 Mathematical Model

The model calculates the survival probability assuming that there is a capability of migrating the targeted container from one host to another. We assume that there is one targeted container that should be protected and the studied networking area is a two-dimensional square area. The following points discuss the developed model.

Survival Probability of Targeted Application Containers

The survival probability P_{static} of a static application container in a host is defined in equation 9.1

$$P_{static}(t) = e^{-\rho S(t)} \tag{9.1}$$

Where

- ρ: the hacked host density, $\rho = N/V$ that N is the number of hacked hosts and V is the number of all hosts in the studied network.
- $S(t)$: the mean number of distinct hosts visited by mobile attackers up to time t and hosts' application containers malfunction. As t increases, number of visited hosts increases (referring to the mobility of attacks and the possibility of having attacks widely spreading at many hosts). $S(t) \approx \pi t / \Big(ln(t)\Big)$ for a two-dimensional square area [60].

The survival probability $P_{mobile}(t)$ of a mobile application container is defined in equation 9.2

$$ln\Big(P_{mobile}(t)\Big) \approx \left(\frac{N}{V}\right)^2 ln\Big(P_{static}(T)\Big) \tag{9.2}$$

Impact of Attacker Success on Survival Probabilities

We assume that not all visited hosts by attackers will misbehave and only some related application containers will breakdown. In other words, we propose that a fraction of $S(t)$ will exist with exponential probability $1 - e^{-t_d}$ where t_d is the required time of detecting attacker's visits to a host.

Accordingly, as t_d increases, this means that attackers can malfunction application containers in visited hosts without detection, i.e. for $t_d = 0$, all attackers? visits are detected (no

successful attacks). t_d might depend on the operating context and hardware of related hosts. Attackers might visit many hosts, however, no fixed number of containers exist at each visited host. So, we consider average t_d for all hosts in our conducted simulation scenarios.

Equation 9.3 shows the mean number of visited hosts by mobile attackers in case that the attackers, successfully, are able to malfunction targeted application containers within the same host without detection.

$$S(t)_{success} = (1 - e^{-t_d})S(t) \tag{9.3}$$

Models of Attack Growth

We consider two different growth models of attacks which are exponential and logistic growth as discussed in [61]. Those models discuss different rates of growth which refer to the spreading rate of attack in a networking context and how ESCAPE can succeed in mitigating such growth.

- *Rapid increase (exponential) of hacked hosts*

  $$N(t) = N(0)e^{kt}$$

  where $N(0)$ is the initial number of hacked hosts at time 0 and k is a constant related to the increasing rate at any time t

- *Slow increase (logistic) of hacked hosts*

  $$N(t) = \frac{\mu N(0)e^k t}{N(0)e^k t + \mu - N(0)}$$

  where μ is the carrying capacity which controls the increasing rate of N that when N approaches μ, the increasing rate approaches zero. Also, $N(t)$ approaches $N(0)$ when $t \to 0$, and approaches μ when $t \to \infty$.

9.4.2 Simulation Scenario

We conducted a simple simulation scenario of application containers cloud established by a set of connected hosts considering the following assumptions:

- All hosts are Internet-enabled devices
- Neighboring hosts to a hacked host are susceptible to the same attack
- We have one prey (one host machine with a targeted application container) and N hacked hosts and V normal and hacked hosts (where $N \leq V$)

- The application container can be migrated from a host to another one. This will be shown in scenarios with mobility feature.

As mentioned before, the attacker's goal is to malfunction the targeted container. The N attackers are targeting the host with the valuable application container. When the attacker reaches a certain host, it spends some time on it, then, all application containers working on that host will fail. Our scenario considers the following cases

- In case of mobile application container scenario, containers can be migrated from a host to another one
- In case of static application container scenario, containers are settled in one host

We study the effect of having small and large number of hacked hosts (N) on the survival probabilities of a static and mobile targeted application container. Our simulation scenarios consider the following two cases during runs:

1. Static N
   - N is fixed during the simulation (i.e., $\frac{dN(t)}{dt} = 0$)
   - During the simulation time, the attackers migrate their related containers (attacks) from one Host to another preserving the total number of hacked Hosts (N)
2. Dynamic N
   - We repeat the scenarios when N has exponential and logistic increasing rates and how N affects the survival probabilities of targeted containers
   - N varies with the simulation time (i.e., $\frac{dN(t)}{dt} \neq 0$)
   - During the simulation, attackers migrate to other hosts and attack new hosts (i.e., number of hacked hosts increases with the simulation time)

Then, we repeat the previous scenarios when applying the concept of attacker success on survival probabilities of targeted application containers.

The mathematical model equations were built using MATLAB 2013. The simulation runs were executed on a Windows 10 operating system machine. Table 9.1 shows the simulation parameters.

Table 9.1: Prey-and-Predator Simulation Parameters

Parameter	**Value**
Number of hosts with the targeted application container (prey) (fixed)	1
Number of hacked hosts (N) (variable)	1,2,3,4,5,7
Total number of victim hosts (V) (variable)	20,30,40,50
Exponential increasing rate constant (k)	1,2,3,4
Initial number of hacked hosts at time 0	1
Log increasing rate constant (k)	1,2,3,4
Log carrying capacity constant (μ)	1,2,3,4
Average attack detection time (t_d) (variable)	0,0.2,0.5,1,10
Simulation time (time unit)	100

9.4.3 Simulation Results

Figure 9.4 shows the impact of changing the number of hacked hosts (N) on a network of 20 hosts where the targeted contained located in one host. For each studied case, the value of N does not change over the simulation time. The figure shows that we can have higher survival probabilities in case of having mobile targeted application container capability. Also, as we increase the number of hacked hosts, the survival probability gets worse.

Figure 9.5 shows that at various detection times and 7 hacked hosts, we can get better survival probabilities at smaller detection times. If we compared the obtained results in figure 9.5 with the simulated survival probability value at case $N = 7$ at figure 9.4, we can notice the improvement in the results at small detection times.

On the other hand and compared with the obtained results in figure 9.5, we got small survival probabilities at various attack detection times in case of having static application containers as depicted in figure 9.6.

Figures 9.7 and 9.8 show the impact of having logistic growth of attacks on the survival probability of a static/mobile application containers in case of having various increasing rates and attack detection times, respectively.

For figure 9.7, as we increase the growth rate of attacks and their spread in many hosts in the network, this leads to lower survival probabilities of the targeted container. In figure 9.8, we assume a certain operating parameters for the logistic growth where $N(0) = \mu = k = 4, V = 20$. As we have small attack detection times, we can mitigate the high growth rate of attacks

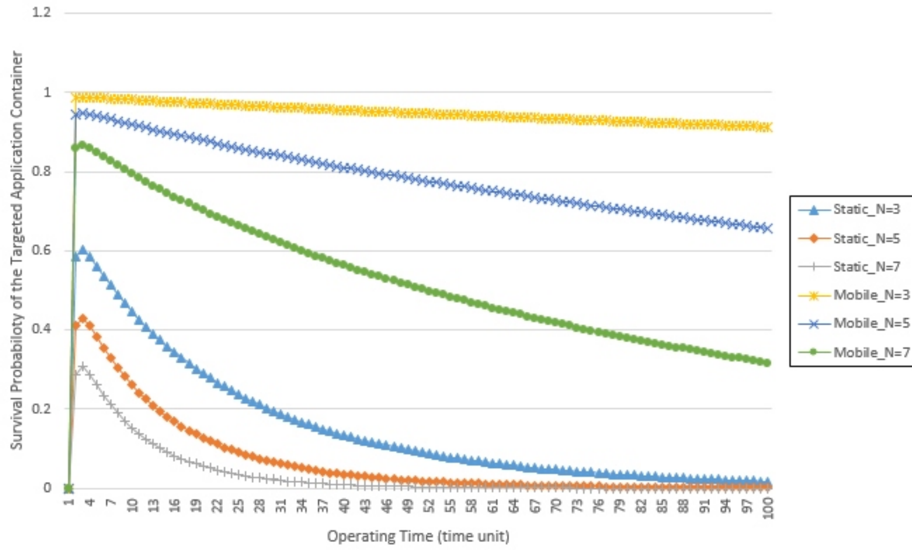

Figure 9.4: Survival probability of a static/mobile container at different number of hacked hosts (N)

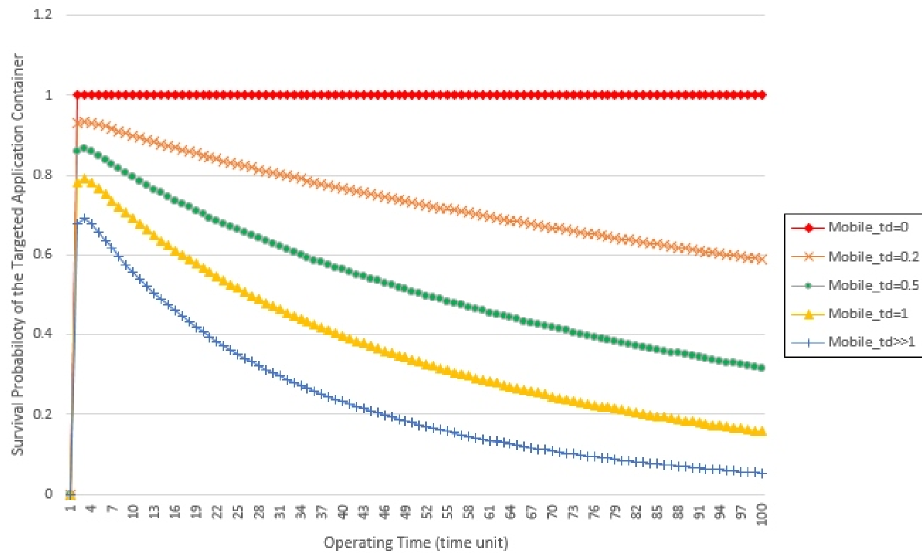

Figure 9.5: Survival probability of a targeted mobile container at 7 hacked hosts and various attack detection times

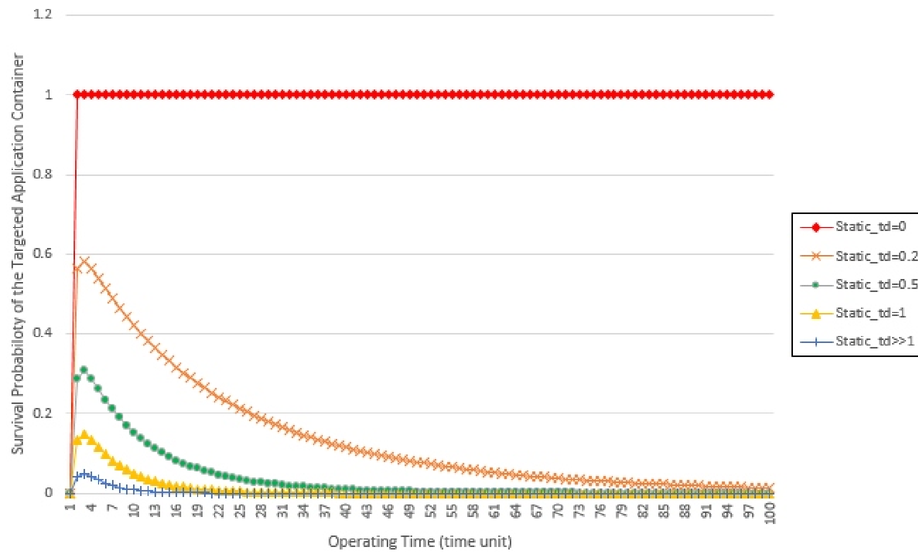

Figure 9.6: Survival probability of a targeted static container at 7 hacked hosts and various attack detection times

and their spread in many hosts. Consequently, this leads to higher survival probabilities of the targeted container compared with the same case obtained in figure 9.7.

Figure 9.9 shows that survival probabilities of a mobile targeted container improve as the number of victim hosts increases. This is because, in case of large number of victim hosts, there is a high probability of the container to move safely to a secured host.

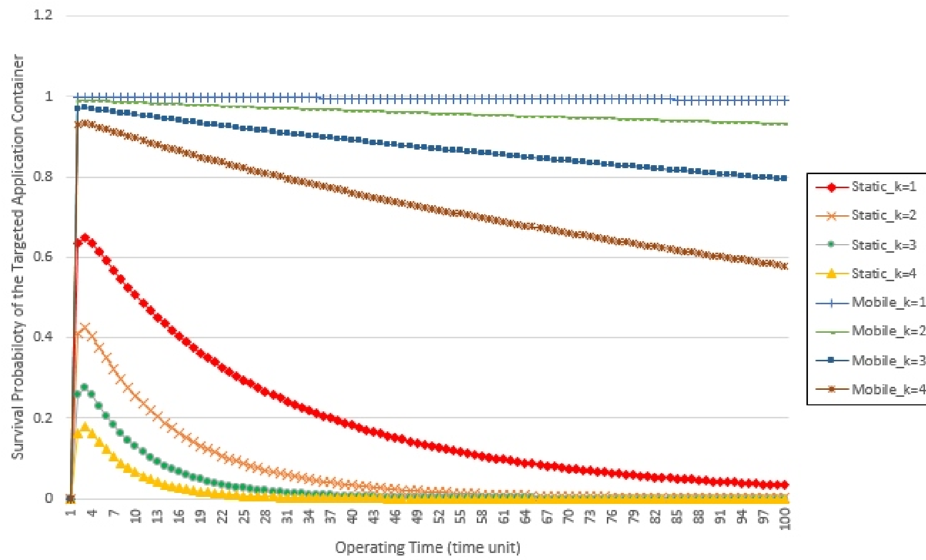

Figure 9.7: Survival probability of a static/mobile targeted container at various increasing rates of logistic attack growth

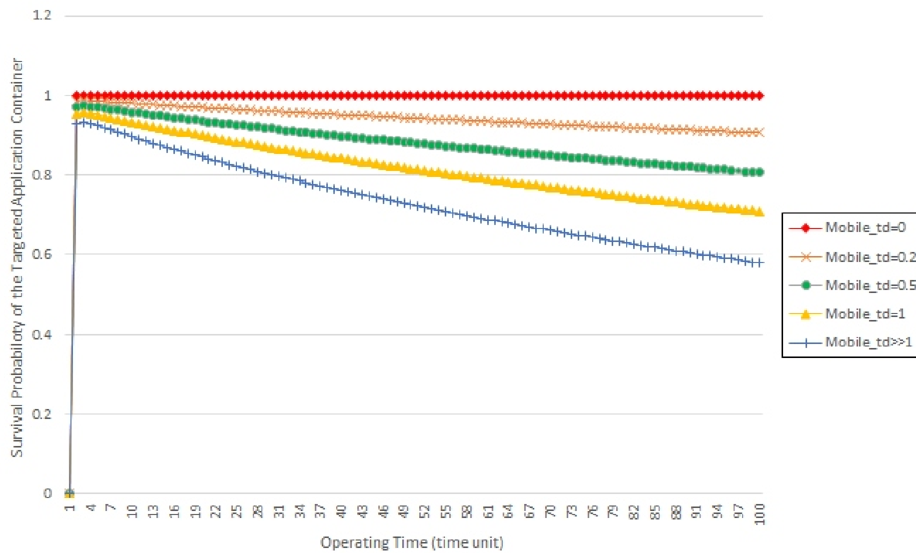

Figure 9.8: Survival probability of a mobile targeted container at logistic attack growth with various attack detection time

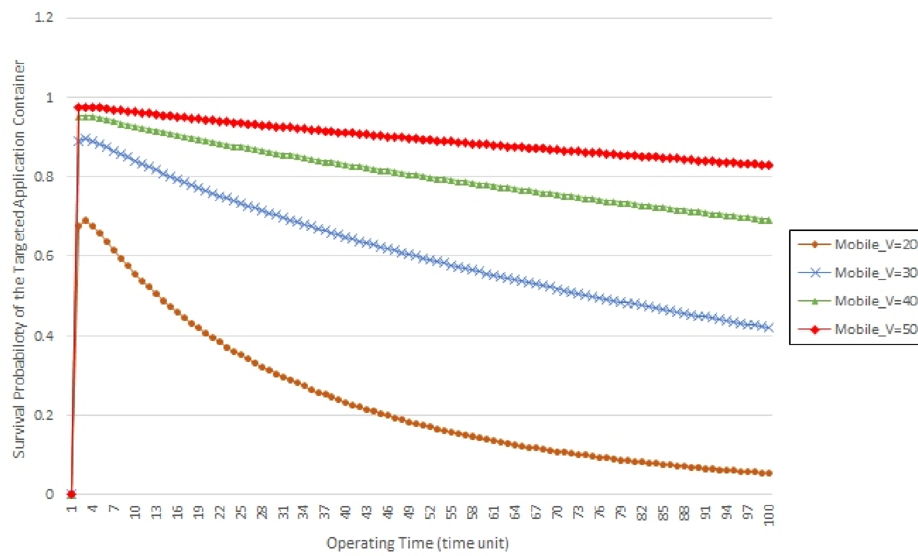

Figure 9.9: Survival probability of a mobile targeted data container with different numbers of victim hosts (V)

Chapter 10

Conclusion and Future Directions

In this dissertation, we have summarized our research efforts in securing cloud containers through intrusion detection and remediation. Considering their popularity and ease of deployment, we are focusing on securing Docker containers for this research. However, the same methods can be extended to any other Linux containers, since they all share the same underlying architecture. Furthermore, the same concepts can be extended to any Linux-based system, such as IoT, SDR, SDN, AUV, and UAS.

We introduced an Intrusion Detection as a Service (IDaaS) platform to be used in container-based SaaS and PaaS clouds. The system monitored the system calls from the guest containers to the host kernel for behavior modeling using two machine learning alternatives, a memory-based technique and a deep learning technique.

We started by using the Bag of System Calls (BoSC) technique for modeling and real-time classification of the container behavior (chapter 5). We then extended the system to be used in a distributed application where a number of the containers used are expected to have similar behavior (chapter 6).

We then designed and implemented another version of the behavior modeling and classification module that deployed a Long Short-Term Memory (LSTM) based Recurrent Neural Network (RNN) to model and classify the container behavior (chapter 7).

To evaluate both versions of the behavior modeling module of the system, we used a typical cloud-based web application hosted in two containers, one with the back-end database server, and another with the front-end webserver(chapter 8).

Both techniques have shown very promising results with slightly better results for the LSTM. We were able to achieve a TPR of 99.36% with FPR of 0.9% for the BoSC technique, and TPR of 99.889% with FPR of 0% when using LSTM. The system accuracy was 99.2056% for the BoSC and 99.924% for the LSTM.

Given that the LSTM version of the system requires hours of prior training, we conclude

that the BoSC approach would be more suitable when the behavior of the container is to be learned in real time, i.e., when the cloud user provides a new container that has not been previously seen by the system. However, if the cloud service provider (CSP) is responsible for providing pre-configured containers to the users, e.g. providing containers running programming platforms in a PaaS cloud, the CSP can then use the LSTM version to train the system in advance before monitoring the actual containers used by the users.

Finally, we discussed the three different approaches we used for handling incidents detected by the system, namely checkpoint and restore, live migration, and replication and isolation (chapter 9).

10.1 System Limitations

The IDS has the following limitations:

- Attacks that generate system calls similar to the normal application cannot be detected unless another measure is used, e.g. network connections.
- The BoSC version of the IDS can learn the container behavior in real time resulting in a 100% detection rate but the FPR is relatively high.
- The LSTM version of the IDS requires offline training that can take hours. However, once trained, a neural network can be used against any container running the same application.

10.2 Future Research Directions

The introduced intrusion detection system is the first to be introduced for container-based clouds. A lot of research is yet to be done in the area of intrusion detection in cloud containers and the area of securing cloud containers in general.

Example future direction is to test the feasibility and efficiency of using traditional machine learning techniques such as Hidden Markov Model (HMM). Another direction could be to study the effect of monitoring other aspects of the container behavior, such as network connections and resource usage. A combination of a signature-based detection and anomaly-based detection may be another promising direction.

Publications

[1] Amr S. Abed and Charles Clancy. **Intrusion detection as a Service for Container-based Clouds**. *IEEE Transactions on Services Computing*, 2017. under review.

[2] Amr S. Abed and Charles Clancy. **Intrusion Detection System for Cloud Containers**. In *2017 IEEE Global Communications Conference: Communication & Information System Security (Globecom2017 CISS)*, December 2017. under review.

[3] Amr S. Abed, Charles Clancy, and Mohamed Azab. **Resilient Intrusion Detection System for Cloud Containers**. *Computers & Electrical Engineering*, 2017. under review.

[4] Amr S. Abed, Charles Clancy, and David S. Levy. **Intrusion Detection System for Applications Using Linux Containers**. In *Security and Trust Management*, volume 9331 of *Lecture Notes in Computer Science (LNCS)*, pages 123–135, 2015.

[5] Amr S. Abed and T. Charles Clancy. **Modeling Container Behavior using Recurrent Neural Networks**. 2017. pending submission.

[6] Amr S. Abed, T. Charles Clancy, and David S. Levy. **Applying Bag of System Calls for Anomalous Behavior Detection of Applications in Linux Containers**. In *IEEE GlobeCom 2015 Workshop on Cloud Computing Systems, Networks, and Applications (CCSNA)*, December 2015.

[7] Mohamed Azab, Bassem Mokhtar, Amr S. Abed, and Mohamed Eltoweissy. **Toward Smart Moving Target Defense for Linux Container Resiliency**. In *IEEE 41st Conference on Local Computer Networks (LCN)*, 2016.

[8] Mohamed Azab, Bassem M. Mokhtar, Amr S. Abed, and Mohamed Eltoweissy. **Smart Moving Target Defense for Linux Container Resiliency**. In *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*, pages 122–130, 11 2016.

[9] Mona S. kashkoush, Mohamed Azab, Gamal Attiya, and Amr S. Abed. **Online Smart Disguise: Real-time Diversication Evading Co-Residency Based Cloud Attacks**. *Cluster Computing*, 2017. under review.

[10] Tohru Shimanaka, Amr S. Abed, and Brian Hay. **Isolation of Malware using SDN**. In *51st Hawaii International Conference on System Sciences (HICSS-51)*, 2017. under review.

Bibliography

[1] 451 Research: Enterprise IT executives expect to spend 34% of their IT budgets on hosting and cloud services in 2017.
URL https://451research.com/images/Marketing/press_releases/451_press_release_11_2016.pdf

[2] D. A. B. Fernandes, L. F. B. Soares, J. V. Gomes, M. M. Freire, P. R. M. In??cio, Security issues in cloud environments: A survey, International Journal of Information Security 13 (2) (2014) 113–170. doi:10.1007/s10207-013-0208-7.

[3] D. Merkel, Docker: lightweight linux containers for consistent development and deployment, Linux Journal 2014 (239) (2014) 2.

[4] Linux Containers : Why They re in Your Future and What Has to Happen First, Tech. rep. (2014).

[5] S. Julian, M. Shuey, S. Cook, Containers in Research: Initial Experiences with Lightweight Infrastructure, Proceedings of the XSEDE16 Conference on Diversity, Big Data, and Science at Scaledoi:10.1145/2949550.2949562.

[6] T. Bui, Analysis of Docker Security, Computing Research Repository abs/1501.0 (2015) 7.

[7] J. Petazzoni, Containers & Docker: How Secure Are They? (2013).
URL http://blog.docker.com/2013/08/containers-docker-how-secure-are-they

[8] D. Fuller, V. Honavar, Learning classifiers for misuse and anomaly detection using a bag of system calls representation, in: Proceedings from the Sixth Annual IEEE Systems, Man and Cybernetics (SMC) Information Assurance Workshop, 2005., Ieee, 2005, pp. 118–125. doi:10.1109/IAW.2005.1495942.

[9] A. Eriksson, Anomaly Detection in Machine-Generated Data: A Structured Approach, Ph.D. thesis, Royal Institute of Technology (2013).

[10] S. Chauhan, Anomaly Detection in ECG Time signals via Deep Long Short-Term Memory Networks, in: IEEE International Conference on Data Science and Advanced Analytics (DSAA), 2015. doi:10.1109/DSAA.2015.7344872.

[11] S. Hawkins, H. He, G. Williams, R. Baxter, Outlier Detection Using Replicator Neural Networks, Data Warehousing and Knowledge Discovery (2002) 170–180doi:10.1007/3-540-46145-0{_}17.

[12] V. Ciesielski, V. P. Ha, Texture Detection Using Replicator Neural Networks Trained on Examples of One Class, Lecture Notes in Computer Science 5866 LNAI (2009) 140–149. doi:10.1007/978-3-642-10439-8{_}15.

[13] T. Combe, A. Martin, R. Di Pietro, To Docker or Not to Docker: A Security Perspective, IEEE Cloud Computingdoi:10.1109/MCC.2016.100.

[14] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, L. Peterson, Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors, ACM SIGOPS Operating Systems Reviewdoi:10.1145/1272998.1273025.

[15] P. Padala, X. Zhu, Z. Wang, S. Singhal, K. G. Shin, Performance Evaluation of Virtualization Technologies for Server Consolidation (2007). doi:10.1.1.70.4605.

[16] Linux VServer (2011).
URL http://linux-vserver.org/Paper

[17] A. Mirkin, A. Kuznetsov, K. Kolyshkin, Containers checkpointing and live migration, 2008 Linux Symposium 2 (2008) 1–8. doi:10.1093/annonc/mdr161.
URL http://www.kernel.org/doc/ols/2008/ols2008v2-pages-85-90.pdf

[18] Everything at Google Runs in Containers (2014).
URL https://www.infoq.com/news/2014/06/everything-google-containers

[19] M. Helsley, LXC: Linux container tools, IBM devloperWorks Technical Library.

[20] M. Mattetti, A. Shulman-Peleg, Y. Allouche, A. Corradi, S. Dolev, L. Foschini, Securing the infrastructure and the workloads of linux containers, 2015 IEEE Conference on Communications and NetworkSecurity, CNS 2015 (Spc) (2015) 559–567. doi:10.1109/CNS.2015.7346869.

[21] E. Bacis, S. Mutti, S. Capelli, S. Paraboschi, DockerPolicyModules: Mandatory Access Control for Docker containers, in: 2015 IEEE Conference on Communications and NetworkSecurity, CNS 2015, 2015. doi:10.1109/CNS.2015.7346917.

[22] M. Le, A. Stavrou, B. B. Kang, DoubleGuard: Detecting intrusions in multitier web applications, IEEE Transactions on Dependable and Secure Computing 9 (4) (2012) 512–525. doi:10.1109/TDSC.2011.59.

[23] P. Mishra, E. S. Pilli, V. Varadharajan, U. Tupakula, Intrusion detection techniques in cloud environment: A survey (2017). doi:10.1016/j.jnca.2016.10.015.

[24] C. Modi, D. Patel, B. Borisaniya, H. Patel, A. Patel, M. Rajarajan, A survey of intrusion detection techniques in Cloud (2013). doi:10.1016/j.jnca.2012.05.003.

[25] S. Iqbal, M. L. Mat Kiah, B. Dhaghighi, M. Hussain, S. Khan, M. K. Khan, K. K. Raymond Choo, On cloud security attacks: A taxonomy and intrusion detection and prevention as a service (2016). doi:10.1016/j.jnca.2016.08.016.

[26] Y. Mehmood, M. A. Shibli, U. Habiba, R. Masood, Intrusion detection system in cloud computing: Challenges and opportunities, Conference Proceedings - 2013 2nd National Conference on Information Assurance, NCIA 2013doi:10.1109/NCIA.2013.6725325.

[27] S. Ramaswamy, R. Rastogi, K. Shim, Efficient algorithms for mining outliers from large data sets, ACM SIGMOD Record (2000) 427–438doi:10.1145/342009.335437.

[28] S. S. Alarifi, S. D. Wolthusen, Detecting Anomalies in IaaS Environments Through Virtual Machine Host System Call Analysis, in: International Conference for Internet Technology And Secured Transactions, 2012, pp. 211–218.

[29] S. Alarifi, S. Wolthusen, Anomaly detection for ephemeral cloud IaaS virtual machines, in: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Vol. 7873 LNCS, 2013, pp. 321–335.

[30] J. Arshad, P. Townend, J. Xu, A novel intrusion severity analysis approach for Clouds, Future Generation Computer Systems 29 (1) (2013) 416–428. doi:10.1016/j.future.2011.08.009.

[31] S. Gupta, P. Kumar, System cum Program-Wide Lightweight Malicious Program Execution Detection Scheme for Cloud, Information Security Journal: A Global Perspective 23 (3) (2014) 86–99. doi:10.1080/19393555.2014.942017.

[32] S. Gupta, P. Kumar, An Immediate System Call Sequence Based Approach for Detecting Malicious Program Executions in Cloud Environment, Wireless Personal Communications 81 (1) (2015) 405–425. doi:10.1007/s11277-014-2136-x.

[33] G. Kim, S. Lee, S. Kim, A novel hybrid intrusion detection method integrating anomaly detection with misuse detection, Expert Systems with Applications 41 (4 PART 2) (2014) 1690–1700. doi:10.1016/j.eswa.2013.08.066.

[34] Z. Li, W. Sun, L. Wang, A neural network based distributed intrusion detection system on cloud platform, in: 2012 IEEE 2nd International Conference on Cloud Computing and Intelligence Systems, Vol. 1, 2012, pp. 75–79. doi:10.1109/CCIS.2012.6664371.

[35] C. H. Lin, C. W. Tien, H. K. Pao, Efficient and effective NIDS for cloud virtualization environment, in: CloudCom 2012 - Proceedings: 2012 4th IEEE International Conference on Cloud Computing Technology and Science, 2012, pp. 249–254. doi:10.1109/CloudCom.2012.6427583.

[36] Y. Meng, W. Li, L. F. Kwok, Design of cloud-based parallel exclusive signature matching model in intrusion detection, in: Proceedings - 2013 IEEE International Conference on High Performance Computing and Communications, HPCC 2013 and 2013 IEEE International Conference on Embedded and Ubiquitous Computing, EUC 2013, 2014, pp. 175–182. doi:10.1109/HPCC.and.EUC.2013.34.

[37] C. Modi, D. Patel, B. Borisanya, A. Patel, M. Rajarajan, A novel framework for intrusion detection in cloud, in: Proceedings of the Fifth International Conference on Security of Information and Networks - SIN '12, 2012, pp. 67–74. doi:10.1145/2388576.2388585.

[38] J. Nikolai, Y. Wang, Hypervisor-based cloud intrusion detection system, 2014 International Conference on Computing, Networking and Communications, ICNC 2014doi:10.1109/ICCNC.2014.6785472.

[39] N. Pandeeswari, G. Kumar, Anomaly Detection System in Cloud Environment Using Fuzzy Clustering Based ANN, Mobile Networks and Applicationsdoi:10.1007/s11036-015-0644-x.

[40] B. D. Payne, M. Carbone, M. Sharif, W. Lee, Lares: An architecture for secure active monitoring using virtualization, in: Proceedings - IEEE Symposium on Security and Privacy, 2008, pp. 233–247. doi:10.1109/SP.2008.24.

[41] C. Warrender, S. Forrest, B. Pearlmutter, Detecting intrusions using system calls: alternative data models, in: Proceedings of the 1999 IEEE Symposium on Security and Privacy (Cat. No.99CB36344), IEEE Comput. Soc, 1999, pp. 133–145. doi:10.1109/SECPRI.1999.766910.

[42] University of New Mexico system call datasets (2015).
URL http://www.cs.unm.edu/~immsec/systemcalls.htm

[43] KDD Cup 1999 Data (1999).
URL http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html

[44] I. Arel, D. C. Rose, T. P. Karnowski, Deep Machine Learning - A New Frontier in Artificial Intelligence Research, IEEE Computational Intelligence Magazine 5 (4) (2010) 13–18. doi:10.1109/MCI.2010.938364.

[45] Y. Lecun, Y. Bengio, G. Hinton, Deep learning, Nature 521 (7553) (2015) 436–444. doi:10.1038/nature14539.

[46] S. Hochreiter, S. Hochreiter, J. Schmidhuber, J. Schmidhuber, Long short-term memory., Neural computation 9 (8) (1997) 1735–80. doi:10.1162/neco.1997.9.8.1735.
URL http://www.ncbi.nlm.nih.gov/pubmed/9377276

[47] Y. Bengio, P. Simard, P. Frasconi, Learning Long-Term Dependencies with Gradient Descent is Difficult, IEEE Transactions on Neural Networks 5 (2).

[48] Y. Bengio, Practical recommendations for gradient-based training of deep architectures, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 7700 LECTU (2012) 437–478. doi:10.1007/978-3-642-35289-8-26.

[49] R. G. Mark, P. S. Schluter, G. Moody, P. Devlin, D. Chernoff, An annotated ECG database for evaluating arrhythmia detectors, in: IEEE Transactions on Biomedical Engineering, Vol. 29, 1982, p. 600.

[50] The Breast Cancer Wisconsin Dataset (1990).
URL http://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin

[51] S. Forrest, S. A. Hofmeyr, A. Somayaji, T. A. Longstaff, A sense of self for Unix processes, in: Proceedings of the 1996 IEEE Symposium on Security and Privacy, 1996, pp. 120–128.

[52] Sysdig (2017).
URL http://www.sysdig.org

[53] S. Hofmeyr, S. Forrest, A. Somayaji, Intrusion Detection using Sequences of System Calls, Journal of computer security 6 (3) (1998) 151–180.

[54] mysqlslap - Load Emulation Client (2017).
URL http://dev.mysql.com/doc/refman/5.6/en/mysqlslap.html

[55] B. Damele, M. Stampar, sqlmap: Automatic SQL injection and database takeover tool (2015).
URL http://sqlmap.org

[56] Serf (2015).
URL https://www.serfdom.io

[57] D. Kennedy, J. OGorman, D. Kearns, M. Aharoni, Metasploit - The Penetration Tester's Guide, 2011.

[58] CRIU - Checkpoint/Restore In Userspace (2015).
URL http://criu.org

[59] M. Azab, B. M. Mokhtar, A. S. Abed, M. Eltoweissy, Smart Moving Target Defense for Linux Container Resiliency, 2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC) (2016) 122–130doi:10.1109/CIC.2016.028.

[60] G. Oshanin, O. Vasilyev, P. L. Krapivsky, J. Klafter, Survival of an evasive prey, Proceedings of the National Academy of Sciences 106 (33) (2009) 13696–13701.

[61] S. Alpern, S. Gal, The theory of search games and rendezvous, Vol. 55, Springer Science & Business Media, 2006.