An Empirical Analysis of IDS Approaches in Container Security

Yigit Sever
Department of Computer Engineering
Middle East Technical University
Ankara, Turkey
yigit@ceng.metu.edu.tr

Goktug Ekinci
Department of Computer Engineering
Middle East Technical University
Ankara, Turkey
goktug.ekinci@metu.edu.tr

Adnan Harun Dogan
Department of Computer Engineering
Middle East Technical University
Ankara, Turkey
adnan.dogan@metu.edu.tr

Bugra Alparslan
Department of Computer Engineering
Middle East Technical University
Ankara, Turkey
bugra.alparslan@metu.edu.tr

Abdurrahman Said Gurbuz
Department of Computer Engineering
Middle East Technical University
Ankara, Turkey
said.gurbuz@metu.edu.tr

Vahab Jabrayilov
Department of Computer Engineering
Middle East Technical University
Ankara, Turkey
vahab.jabrayilov@metu.edu.tr

Pelin Angin
Department of Computer Engineering
Middle East Technical University
Ankara, Turkey
pangin@ceng.metu.edu.tr

***Abstract*—Microservices architecture has been praised as a lightweight, modular and robust alternative to monolithic software in recent years with software containerization bringing parallel ideas to the table against bare metal and even virtual machine based software deployment solutions. While containers provide support for agile software development in the cloud, they suffer from security issues due to their lightweight structure not providing isolation as strong as that of virtual machines. This calls for the development of robust intrusion detection systems (IDS) for containers, taking into account their specific vulnerabilities. Existing IDS for containerized software deployments have mainly used host-based `syscall` monitoring, with only a few utilizing network-based monitoring without justification for the particular sensor used. In this paper, we aim to close this research gap by empirically evaluating the performances of system call and network flow based features in machine learning-based intrusion detection for containers when subjected to the same attacks. Our results show that basing the IDS on the network layer exhibits better performance than the host-based IDS for the investigated vulnerabilities, demonstrating the need for network monitoring for enhanced container security.**

***Index Terms*—Containers, IDS, Cloud Computing, Microservices, Network Security**

This research has been supported by the TÜBİTAK 3501 Career Development Program under grant number 120E537. However, the entire responsibility of the publication belongs to the owners of the research. The financial support received from TÜBİTAK does not mean that the content of the publication is approved in a scientific sense by TÜBİTAK.

I. Introduction

Advances in cloud computing systems in recent years have led to the development of applications based on the microservices architecture to meet the high performance requirements of the systems on the cloud. This architecture includes a complex application structure that consists of services that exist as independent entities and interact with each other through specific APIs. Microservices architecture frequently uses a structure called container, which is more lightweight than virtual machines in the cloud [1]. The rapid adoption of cloud technologies experienced in recent years has led to the widespread use of container-based application structures. Furthermore, the proliferation of container networks has revealed the potential of exposing these networks to many cyberattacks caused by the malicious capture of endpoints. In addition to attacks that affect legacy application systems, containers are vulnerable to attacks peculiar to them due to the differences in their working mechanism, such as the dynamic building of images. Aware of the fact that containers will form the backbone of many systems including increasingly virtualized networks such as 5G and beyond, it is of utmost importance to ensure robust security practices for these systems.

Despite the plethora of work in intrusion detection systems (IDS) for legacy systems, automated attack detection and prevention for containers and container networks have not been sufficiently explored in the research literature or real-world applications. As in the case for general IDSs, we can use host-based or network-based approaches or a combination of them for detecting intrusions on container-based application systems.

Unix-based operating systems separate user threads and kernel threads, where only the kernel has direct access to the hardware. User processes use system calls (`syscalls`) with a well-defined interface to use kernel level privileged instructions. As these instructions are present whenever a process reads or writes data from and to the disk or sends or receives data through the network, it offers a valuable source for inspection. A majority of the existing approaches for container security have favored monitoring `syscalls`, often with the use of the Sysdig [2] tool over monitoring the network packets from and to the container networks. However, to the best of our knowledge, no study has advocated for their choice of monitoring `syscalls` over network packets for their intrusion detection frameworks or vice versa.

In this paper, we perform an analysis of approaches utilizing `syscall` data and network flow data for detecting attacks on containers using machine learning (ML) algorithms with proven success in intrusion detection tasks. This paper makes the following contributions to the literature on container security:

- We present reproducible steps towards generating a dataset consisting of benign and malicious traffic generated through interactions with real-world vulnerable container images discovered using the Common Vulnerabilities and Exposures (CVE) database [3], which includes both system call and network flow data.
- We provide an empirical comparison of ML-based intrusion detection approaches for containers using network-based monitoring and host-based monitoring.

The remainder of this paper is structured as follows: In Section II, we provide a brief overview of related work in the field of container security. Section III provides necessary background information on IDS and virtualization. Section IV describes our container IDS approach, providing details on data collection and processing using ML algorithms. Section V provides details of our experimental setup for evaluating the performance of the container IDS. Section VI provides the results and discussion of the experiments performed and Section VII concludes the paper with future work directions.

II. Related Work

The past decade has seen an increasing use of ML techniques for cyberattack detection in various systems. This is because attacks are getting more sophisticated and classical attack signature-based solutions cannot effectively detect previously unobserved attacks. Learning algorithms such as Support Vector Machines (SVM) [4], which is a classical machine learning model, deep learning models such as convolutional neural networks (CNNs) [5] and autoencoders [6] achieved quite high (over 90% in most cases) intrusion detection rates on datasets containing traffic for legacy network and application settings.

Effective attack detection and prevention in the cloud faces many challenges, even in the presence of fast streaming data analytics. Therefore, machine learning algorithms and tools have been developed specifically for cloud environments, mostly for virtual machine-based deployments, and achieved successful results in detecting anomalies for certain scenarios including classical enterprise networks [7], industrial IoT (IIoT) systems [8], and sensor data publish-subscribe systems [9].

Despite the existence of many approaches in the field of intrusion detection including both host-based (HIDS) and network-based (NIDS) solutions (e.g. [10]–[14]), little attention has been paid to IDSs for containerized environments. Table I provides a summary of previous studies on intrusion detection for containerized environments. We present the surveyed works according to the target applications they used to perform attacks on, the choice of the benign traffic source they used to train their models, how they performed the attacks and finally the monitor they used.

Tien et al. [20] introduced an HIDS aimed for a Kubernetes cluster, the leading container orchestration framework. Using supervised learning methods, they developed an anomaly classification model which they named KubAnomaly. KubAnomaly is a neural network model with four fully connected layers; the first three layers utilize exponential linear units (ELU) and the last layer utilizes softmax as the activation function. To create a dataset for training their models, they used system calls and root directory access features, which takes place when the container accesses a file under the root directory (e.g. `bin, var`).

Flora et al. [19] proposed using attack injection on a containerized MariaDB database and the workload of the TPC-C database transaction benchmark [23] to evaluate their HIDS's performance. They used 3 algorithms and representations; sequence time delaying embedding (STIDE), Bags of System Calls (BoSC) and Hidden Markov Models (HMM) and performed their evaluations on Docker and LXC containers. To compare the performance of the HIDS on the containerized environment, they also performed evaluations on a pseudo bare metal installation, using a KVM Virtual Machine.

Tunde-Onadele et al. [21] combined signature-based and anomaly-based approaches in their HIDS. They evaluated common intrusion detection methods for containerized environments. Srinivasan et al. [15] proposed a real-time HIDS using system calls. They fed n-grams of system calls to Maximum Likelihood Estimator (MLE) and Simple Good Turing (SMG) to do real-time classification. Cavalcanti et al. [18] evaluated the effectiveness of 8 machine learning algorithms for IDS on containerized environments. They made use of the BoSC approach to create their dataset and evaluated these algorithms on this dataset. Chen et al. [24] proposed a framework named Informer. This framework is used to detect chains of anomalous Remote Procedure Calls (RPC), which are used for communication between agents in a microservices architecture.

All the approaches mentioned above have either focused on

TABLE I
CONTAINER SECURITY APPROACHES

Author	Target Application(s)	Benign Traffic Source	Malicious Traffic Source	Monitor
Srinivasan et al. [15]	DVWA [16]	*unknown*	`sqlmap`	`strace`
Abed et al. [17]	MySQL	`mysqlslap`	`sqlmap`	`strace`
Cavalcanti et al. [18]	MySQL	TPC-C Benchmark	TPC-C Benchmark	Sysdig
Flora et al. [19]	MariaDB	TPC-C Benchmark	PoC code from exploit-db.com	Sysdig
Tien et al. [20]	*unknown*	JMeter	JMeter, `sqlmap`, OWASP ZAP	Sysdig, Falco
Tunde-Onadele et al. [21]	*various programs*	Burp Suite, JMeter	JexBoss, Metasploit, PoC code, `sqlmap`, Burp Suite	Clair, Sysdig
Röhling et al. [22]	MariaDB	*various tools*	*various tools*	Sysdig

host-level features or network-level features, with the majority using host-level features. Our work in this paper differs from the surveyed works in that we analyze both system call-based and network flow-based approaches for intrusion detection on the same attack simulations that we performed on vulnerable containers.

III. Preliminaries

A. Intrusion Detection Systems (IDS)

In computer network security, an intrusion is an unauthorized access or an attempt to access confidential or sensitive data or resources of a system. These resources can include data, network or hardware elements. Intrusion detection is the collective process which includes monitoring, detection and reporting of intrusions on a target system or network of systems [25]. An IDS uses sensors to gather and collate relevant data such as network flows and system calls.

IDSs are categorized according to the set of sensors they use; a Host-based IDS (HIDS) monitors the computer infrastructure with parameters like CPU usage, RAM usage, and `syscalls` whereas a Network-based IDS (NIDS) tracks the network traffic in order to detect intrusions [26]. In a computing system, there are many resources that can be used to build an HIDS [26]. Some examples are `syscalls`, log files, file integrity checksums etc.

IDSs can also be classified as signature-based IDSs, anomaly-based IDSs and hybrid IDSs. In a signature-based IDS, system behavior is monitored, and this behavior is matched against known attack signatures. However, a signature-based IDS is not capable of detecting previously unseen (i.e. zero-day) attacks. Furthermore, when a signature-based IDS is used in a containerized environment and proper care is not taken, dependencies between Docker images may result in vulnerability propagation from the base image to the child image [27]. Anomaly-based IDSs monitor the system and detect any abnormal behavior. While these IDSs can detect zero-day attacks, they have the risk of tagging an unseen benign behavior as malicious. Hybrid IDSs, on the other hand, incorporate ideas from both of these approaches to tackle the discussed drawbacks.

B. Virtualization

Virtualization is an intrinsic component of modern data centers and cloud environments to decouple applications from the hardware they are hosted on. There are two main ways of achieving virtualization; hardware virtualization and operating system (OS) level virtualization. In this paper, we are particularly interested in containers, which use OS-level virtualization. OS-level virtualization leverages the OS kernel rather than the physical hardware itself. The OS kernel undertakes the responsibility to implement container abstraction by allocating CPU shares, memory, network I/O, and managing file system isolation.

Linux containers benefit from two main kernel features: Control groups (*cgroups*) and *namespaces* [28]. *cgroups* manage resource allocation among processes which can belong to different groups. All major resource types (CPU, memory, network, block I/O) have their corresponding `cgroups` to manage resource limits. *Namespaces* enable resource isolation among several container instances. The Linux kernel provides process ID, user ID, file system mount points, networking, inter process communication (IPC) and host name namespaces. Since the kernel is shared among native containers, an exposed container can further exploit kernel vulnerabilities and get access to other containers, which means shared kernel design cannot isolate kernel vulnerabilities [29].

IV. Container IDS Approach

The core of any intrusion detection system is the tools and techniques to efficiently monitor and analyze system behavior. For our container IDS approach, we continuously monitor applications running on containers and process the data resulting from user/service interactions. We use an ML approach for detecting intrusions rather than a rule-based approach due to the high generalizability ML algorithms provide. In order to train our ML models, we need both the normal behavior of the system over a period of regular user activity and the anomalous behavior that arises during an attack. We collect the behavior traces during both benign and malicious traffic in a sanitized environment to label the data trivially and to ensure that the intrusion detection system is trained on noise-free data. Data is pre-processed at the beginning of the ML pipeline since, depending on the monitoring tool, the output is in different formats and not readily suitable for ML algorithms.

In this study, we use features extracted from `syscalls` and network traffic data. `syscalls` are present for every meaningful interaction between the user and the software inside the container, whereas network traffic results from the interactions between the cloud services and users. With our `syscall` monitor of choice, every `syscall` from the userland to the kernel is logged with timestamps. We can process these logs to get frequency features over a window of time. On the other hand, by using network traffic monitoring, we can examine every packet "on the wire" and then collate that information into "flow" features. Flow features contain aggregate information such as the duration of network communication for a given request. We further explain the collection of raw network packets and `syscalls` and the extraction of network flows and `syscall` frequency in the following subsections.

A. Training Data Generation

IDS literature has two options to evaluate their proposed methods: using publicly available datasets of real-world traffic to replay them on the target network or generating malicious and benign traffic using automated tools [30]. Considering that IDSs are deployed in the real world and used to detect real-world attacks, artificial datasets that the literature uses to build these IDSs should be as realistic as possible. Choosing the traffic generation approach entails certain characteristics in order to make the resulting dataset and the resulting evaluation reliable and reproducible. In this study, we have decided to follow the requirements put forth by Sharafaldin et al. [30], which includes points such as having the complete network traffic available in the dataset (packets originating from both source and the destination). We have also relied on the insights given by Viegas et al. [31] on traffic generation for IDS research. Authors in the said work advocate for using well known CVEs and attack tools to keep the work reproducible. They also advise future researchers to craft realistic user traffic by making sure that the generated user traffic is variable enough to not follow any statistical distribution. We created our benign user behavior and performed our malicious attacks according to the works mentioned above. To create valid benign network traffic, requests to the server had to be varied in both content and frequency. Also, only valid request-response pairs had to be generated. While evaluating the dataset, environment-specific variables were not considered as features, e.g. no detection was made using the IP address of the attacker [31].

B. Machine Learning Algorithms

In this study, we used varieties of Decision Trees and SMO, a variant of Support Vector Machines (SVM), for the intrusion detection problem.

A decision tree (DT) is a common machine learning approach with no hyperparameters for both classification and regression problems. The decision trees' structure resembles a flowchart, with each internal node standing in for a test condition, each branch for the result of the test condition, and each leaf node for a class label, which indeed is a decision taken after computing all attributes. Classification rules are represented by the routes from root to leaf. A well-known decision tree algorithm is C4.5 [32].

For the remainder of this section, we explain the machine learning algorithms we used in detail and note the configurations with which we ran the algorithms on the WEKA ML framework, which is discussed in Section VI-A.

1) REPTree: A Fast Decision Tree learner [33] (REPTree) constructs either a classification or a regression tree with information gain (IG) (Eq. 1) as the splitting criterion and reduced error pruning is used to prune the tree with backfitting.

$$\text{IG}(\mathbf{S}, X) = \text{Entropy}(\mathbf{S}) - \sum_{x \in X} \frac{|\mathbf{S}_x|}{|\mathbf{S}|} \text{Entropy}(\mathbf{S}_x) \qquad (1)$$

Where $\mathbf{S}$ is the training set (or the dataset), X is a column in the training set (i.e. an attribute), and x is an element in the attribute (column) vector X. $\mathbf{S}_x$ is a subset of $\mathbf{S}$ such that all training points therein satisfy the following condition $X = x$. In other words, the value of the attribute X in the set $\mathbf{S}_x$ is x.

The Entropy, on the other hand, is defined in Eq. 2 as;

$$\text{Entropy}(\mathbf{S}) = -\sum_{c=1}^{|C|} P_{\mathbf{S}}(c_i) \log P_{\mathbf{S}}(c_i) \qquad (2)$$

Where $P_{\mathbf{S}}(c_i)$ is the estimated percentage of labels of training points that belong to $s_i < |C|$ in the training set $\mathbf{S}$, and $|C|$ is the number of classes.

In this study, we used the following parameter configuration for REPTree in WEKA: `batchSize` = 100, `initialCount` = 0.0, `maxDepth` = -1.0, `minNum` = 2.0, `minVarianceProp` = 0.001, `numDecimalPlaces` = 2, `numFolds` = 3, `seed` = 1. We have not used pruning.

2) Random Tree: Random Tree is a supervised classifier based on a decision tree. The algorithm is called "random" because of the bagging paradigm; it chooses a random set of data during the construction of the decision tree. The algorithm is used extensively in machine learning for both classification and regression problems. Random trees can be generated efficiently and the combination of large sets of random trees generally leads to accurate models.

For this model, we used the following configuration in WEKA: `batchSize` = 100, `maxDepth` = 0.0, `minNum` = 1.0, `minVarianceProp` = 0.001, `numDecimalPlaces` = 2, `numFolds` = 0, `seed` = 1.

3) Random Forest: Random Forests [34] have shown to improve the performance of single decision trees considerably, with tree diversity created by two ways of randomization: first the training data is sampled with replacement for each single tree like in bagging. Secondly, when growing a tree, instead of always computing the best possible split for each node, only a random subset of all attributes is considered at every node, and the best split for that subset is computed.

For this model, we used the following configuration in WEKA: `bagSizePercent` = 100, `batchSize` = 100, `maxDepth` = 0.0, `numDecimalPlaces` = 1, `numFeatures` = 0, `numIterations` = 100, `numExecutionSlots` = 0.0, `seed` = 1. We did not use impurity decrease.

4) SMO: Support Vector Machines (SVM) [35] is a supervised classification and regression algorithm with proven success in many tasks including intrusion detection. It maps training examples to a coordinate space to maximize the width of the gap between the two categories using the following quadratic and convex objective function given in Eq. 3, where $\forall i, y_i(W^T X_i + b) \geq 1$.

$$\min_{W,b} \frac{1}{2} W^T W \tag{3}$$

Therefore, the training process of a SVM includes solving a quadratic programming (QP) optimization problem. Sequential Minimal Optimization (SMO) [36] divides the large QP problem into a sequence of simple smallest possible QP problems to be solved analytically. Therefore, SMO abstains from time-consuming numerical QP optimization loops.

In a binary classification problem like intrusion detection, SMO trains a soft-margin SVM by solving the QP problem given in Eq 4;

$$\max_{a} \sum_{i=1}^{n} a_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} y_i y_j K(x_i, x_j) \alpha_i \alpha_j \tag{4}$$

Where (x_i, y_i) is a training point in the dataset of size n, α_i is a Lagrange multiplier on an interval $[0, C]$ such that $\sum_{i=1}^{n} y_i \alpha_i = 0$. C is an SVM hyperparameter and $K(x_i, x_j)$ is a (positive-definitive) kernel function.

C. Bag-of-System-Calls

Bag-of-system-calls (BoSC) [37] is a method for representing the `syscall` traces of a computer or process. The representation involves creating a frequency list $S = \{s_1, s_2, \ldots, s_n$ where s_i is the number of times the `syscall` at that window is observed [38]. A sliding window technique is used to sample at time t with the sampling window defined by the window size.

By default, the Sysdig tool captures system call traces with every event information available and saves them to the disk. This information can be filtered from the disk again or in a pipeline to get suitable output for BoSC processing.

To create a frequency list, the number of available `syscalls` must be known to determine the list length. The available `syscall` list is constructed by first using the appropriate `syscall` table with respect to the architecture of the system from the Linux kernel. We then extend this table further by comparing the list above with the list of `syscalls` captured by Sysdig. Hence, our BoSC representation vectors are of length $n = 332$. We opted to use 6 seconds for our window size and L1 normalized the BoSC vectors.

V. Experiment Design

This section describes the setup procedure of the simulation environment and our data collection process. The generation steps of the training data set covers both benign and malicious traces with the simulation environment under normal traffic and traffic during attack injection, respectively.

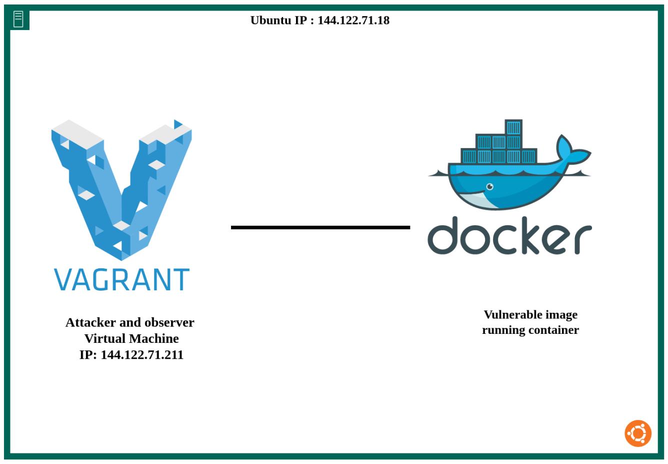
Ubuntu IP : 144.122.71.18
VAGRANT
Attacker and observer Virtual Machine IP: 144.122.71.211
docker
Vulnerable image running container

Fig. 1. Lab environment topology.

We used a closed-loop client with independent threads during our evaluation. Each thread sends a logically coupled set of HTTP requests and the next request set is not sent until a response is received for the preceding request set.

Apache JMeter [39] is an open source software designed to load-test functional behavior and measure web application performance. We used JMeter to achieve the realistic workload we defined above. Furthermore, we used the BlazeMeter[1] browser plugin to record regular user behavior including timing information, further increasing the realistic nature of the load test recipes.

Vulnerable images were run as containers on the host machine powered by Docker [40]. Docker is an open platform for developing, shipping and running applications. Moreover, it is the most widespread tool for container technology.

To find vulnerabilities and develop exploit scenarios, Exploit Database (ExploitDB)[2] was used to perform attacks and generate malicious activity. It is a CVE compliant archive of exploits for the purpose of public security, developed for use by penetration testers and vulnerability researchers. We also benefited from the Metasploit Framework [41] to execute exploit code against the deployed containers on the host machine. Metasploit Framework enables us to compare the identified vulnerabilities to its database for accurate exploitation.

We used Sysdig to collect the system calls for the container as a whole, due to its native support for containers. Sysdig monitors containers at the operating system level and captures `syscalls`.

We used `tcpdump`[3] to intercept network packets both for benign and malicious traffic. Network packets were recorded from the beginning of the simulation until the end. Normal user activity and attack scenarios were recorded separately.

The machine that hosted the simulation environment has an Intel Core i7-11700 CPU at 2.50GHz with 16GBs of

[1]https://www.blazemeter.com/
[2]https://www.exploit-db.com/
[3]https://www.tcpdump.org

TABLE II
VULNERABILITIES USED

Application (version)	**CVE**	**CWE**
rConfig (v3.9.2)	CVE-2019-16662	CWE-78
rConfig (v3.9.2)	CVE-2019-19509	CWE-78
rConfig (v3.9.2)	CVE-2020-10220	CWE-89

RAM, running Ubuntu 20.04. We used the same machine for our machine learning algorithm evaluations as well. The overall topology is presented in Figure 1. The simulation environment machine hosts the Vagrant box (Ubuntu 20.04) and the vulnerable docker image. JMeter and Metasploit are run from the Vagrant box whereas `tcpdump` and Sysdig are ran on the host machine.

A. Vulnerable Image Selection

Common Weakness Enumeration (CWE)[4] is a common standard for classifying possible causes of exploits in software. CVEs are the resulting vulnerabilities, which stem from a particular CWE. To have a diverse set of attacks, we looked up container images that contain multiple CVEs, each belonging to a distinct CWE. We also leveraged the ExploitDB and Metasploit modules to access the exploit scenarios for those CVEs. We then filtered further for compatible images that can be accurately monitored in a realistic workload with JMeter. All in all, we decided on using rConfig [42] version `3.9.2` with the following CVEs.

B. Exploited Weakness Types

1) CWE-78 (OS Command Injection): To exploit CVE-2019-16662[5], since `rootUname` parameter is passed to the `exec` function without filtering, we send a `GET` request to `ajaxServerSettingsChk.php` by setting the `rootUname` parameter as a system command that we want to execute in the system.

To exploit CVE-2019-19509[6], similar to the first exploitation, we sent a `GET` request to `ajaxArchiveFiles.php` by setting the path parameter system command that we want to execute in the system.

2) CWE-89 (SQL command Injection): The web interface of this image is vulnerable to SQL command injection through the `commands.inc.php` `searchColumn` parameter. Thus, for exploiting CVE-2020-10220[7], we inject a SQL command to the system via the `searchColumn` parameter.

C. Data Collection

During our experiments, we monitored both the network traffic and system calls on the host, then evaluated our models using both kinds of data. We explain the generation process of these datasets in the subsections below.

[4]https://cwe.mitre.org
[5]https://nvd.nist.gov/vuln/detail/CVE-2019-16662
[6]https://nvd.nist.gov/vuln/detail/CVE-2019-19509
[7]https://nvd.nist.gov/vuln/detail/CVE-2020-10220

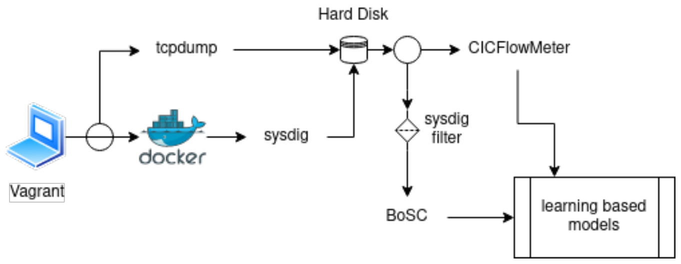

Fig. 2. Evaluation pipeline.

1) Network Data Collection: Alongside network traffic, payload is also present in the packet-based network traffic whereas flow information on the network traffic consists of only the cumulative information about requests and responses [43]. Our work utilizes the latter one.

2) System Call Collection: Our literature search presented us two widely used tools to monitor system calls. The first is `strace`[8], a Linux tool that is used to trace system calls and signals. The other is Sysdig [2], an open-source system monitoring tool. Our work used Sysdig due to the insights presented by Röhling et al. [22]; efficiency and being non-blocking compared to `strace`.

D. Flow Extraction

A set features collected from a series of network packets are said to belong in a *flow* [44]. To extract the features from the intercepted packages which are in `pcap` file format and get flow information, we used the up-to-date fork of the open source CICIDSFlowMeter tool [45]. The tool extracts more than 80 network traffic flow features.

E. Feature Preprocessing

Since our malicious traffic and benign traffic are captured in separate instances, we labeled the data by hand, 1 for malicious, 0 for benign. Some flow features were taken out of the dataset, as they have minimum entropy, i.e. all training points gave almost the same values. Furthermore, some features were taken out since they do not carry generalizable information such as source IP address and destination IP address. All in all, we removed: Flow ID, Src IP, Dst IP, Timestamp, Flow Bytes/s and Flow Packets/s, Fwd URG Flags, Bwd URG Flags, URG Flag Count, CWR Flag Count, ECE Flag Count, Subflow Fwd Packets, and Subflow Bwd Packets.

We transformed the label attribute from numeric to nominal so that we can have a detailed and elaborated summary in WEKA.

VI. EVALUATION AND RESULTS

We present our evaluation pipeline that we used in the experiments in Figure 2. We trained our ML models using WEKA described in Section VI-A and performed experiments with multiple models as discussed in Section IV.

[8]https://man7.org/linux/man-pages/man1/strace.1.html

TABLE III
RESULTS OF THE EVALUATION ON NETWORK FLOW DATA

Model	TP Rate	FP Rate	Precision	Recall	F-Measure	Label
REPTree	1.000	0.002	1.000	1.000	1.000	Benign
	0.998	0.000	0.999	0.998	0.999	Malicious
Random Tree	1.000	0.003	1.000	1.000	1.000	Benign
	0.997	0.000	0.999	0.997	0.998	Malicious
Random Forest	1.000	0.002	1.000	1.000	1.000	Benign
	0.998	0.000	1.000	0.998	0.999	Malicious
SMO	1.000	0.013	1.000	1.000	1.000	Benign
	0.987	0.000	0.998	0.987	0.993	Malicious

TABLE IV
RESULTS OF THE EVALUATION ON BOSC DATA

Model	TP Rate	FP Rate	Precision	Recall	F-Measure	Label
REPTree	0.998	0.007	1.000	0.998	0.999	Benign
	0.993	0.002	0.937	0.993	0.964	Malicious
Random Tree	0.998	0.030	0.999	0.998	0.999	Benign
	0.970	0.002	0.942	0.970	0.956	Malicious
Random Forest	0.999	0.007	1.000	0.999	0.999	Benign
	0.993	0.001	0.964	0.993	0.978	Malicious
SMO	0.998	0.000	1.000	0.998	0.999	Benign
	1.000	0.002	0.944	1.000	0.971	Malicious

TABLE V
CONFUSION MATRIX OF EVALUATION ON NETWORK FLOW DATA

	a	b	Actual
REPTree	279336	4	a = 0
	8	4524	b = 1
Random Tree	279332	8	a = 0
	13	4519	b = 1
Random Forest	279338	2	a = 0
	9	4523	b = 1
SMO	279333	7	a = 0
	60	4472	b = 1

TABLE VI
CONFUSION MATRIX OF EVALUATION ON BOSC DATA

	a	b	Actual
REPTree	4956	9	a = 0
	1	133	b = 1
Random Tree	4957	8	a = 0
	4	130	b = 1
Random Forest	4960	5	a = 0
	1	133	b = 1
SMO	4957	8	a = 0
	0	134	b = 1

A. WEKA

WEKA [46] is an open-source Machine Learning/Data Processing application with tools for Data Preprocessing, Data Manipulation, and a Library for data classification/regression. WEKA has tools for attribute selection, experiment creation, and clustering. It has been observed that many studies with or without intrusion detection used WEKA for classifying and data processing [47], [48]. All the data processing and anomaly detection models we have used were implemented and tested using WEKA on the machine described in Section V.

B. Overview of the Results

In our experiments, we considered three decision tree approaches and the SMO Algorithm. The flow dataset includes 279340 benign flows and 4532 malicious flows. The BoSC dataset includes 4965 benign BoSC vectors and 134 malicious BoSC vectors. We can see that the benign data points are in much higher quantity than the malicious ones. This is not unlike real life setting in which regular users will always be the overwhelming majority compared to malicious actors. The reason for the disparity for our study is that benign traffic generation takes upwards of 2 hours while exploit code including exploit discovery included in the Metasploit modules takes around a minute. We also opted out of using neural network based classification due to this imbalance.

We evaluated the True Positive Rate, False Positive Rate, Precision Score, Recall, and F-Measure of the models on the created dataset. Precision is the metric which evaluates the ratio of desired instances in the whole set of instances that the model predicted the and recall is the performance of the model regarding how many of those desired instances it classified correctly compared to how many it had missed. The F-Measure metric combines recall and precision [49]. The overall results for the evaluation on network flow feature can be seen in Table III and Table IV shows the evaluation results for system call data.

We present the predictions and actual values in confusion matrix form for the test data of network flow evaluation in Table V and BoSC in Table VI.

We observe that all algorithms have achieved high performance. Regarding network flow evaluation, REPTree and Random Forest achieved the best F-Measure scores. This is reflected in the confusion matrix as well, with REPTree having the highest true positive and true negative predictions. On the other hand, looking at the BoSC based evaluation, SMO has the best performance regarding true negative predictions while Random Forest has the best performance in true positive predictions. The interesting result which is at the core of the present study arises when we consider the difference in performances between network flow based evaluation and BoSC based evaluation. Across the board, network flow based evaluation has resulted in better performance than BoSC based evaluation. However, we recognize that the magnitude of data points between the representation differ greatly even though they have been extracted simultaneously. The feature extraction methods and the data they use to get those features differ.

Overall, we can surmise that for the current study, network flow data has provided better insights for intrusion detection on containers than BoSC representation.

VII. Conclusion

In this study, we set up a simulation environment with software running in a containerized setting. We then subjected vulnerable containers running in the lab setting to known attacks and simulated legitimate activity to create two datasets for detecting intrusion activity on containers, one based on system calls and the other based on network flows. Through our experiments, we found that opting for network flow based detection results in a better detection rate than the BoSC representation gathered from `syscalls`.

We recognize that the results we have gotten are lacking in generalizability due to the fact that we used only one vulnerable application with 3 CVEs distributed over 2 CWEs. A straightforward future work would be to increase the number of vulnerable applications with a diverse set of attacks to subject them to. This would result in a more generalizable result. On a related note, the question of which of those CVEs or CWEs yield better information for the intrusion detection system is one that is worth answering, using the ideas we discussed. Providing insight on types of CWEs that are better recognizable with different IDS approaches can pave the way for IDS that are more prepared for different kinds of weaknesses and exploits.

References

[1] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2015, pp. 171–172.

[2] S. Inc., "Sysdig," https://github.com/draios/sysdig, last accessed: 24 June, 2022.

[3] MITRE, "Common vulnerabilities and exposures (cve)," https://cve.mitre.org/, last accessed: 24 June, 2022.

[4] W. An and M. Liang, "A new intrusion detection method based on SVM with minimum within-class scatter," *Security and Communication Networks*, vol. 6, no. 9, pp. 1064–1074, 2013.

[5] Y. Xiao, C. Xing, T. Zhang, and Z. Zhao, "An Intrusion Detection Model Based on Feature Reduction and Convolutional Neural Networks," *IEEE Access*, vol. 7, pp. 42 210–42 219, 2019.

[6] N. Shone, T. N. Ngoc, V. D. Phai, and Q. Shi, "A deep learning approach to network intrusion detection," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 2, no. 1, pp. 41–50, 2018.

[7] S. Xu, Y. Qian, and R. Q. Hu, "Data-driven network intelligence for anomaly detection," *IEEE Network*, vol. 33, no. 3, pp. 88–95, 2019.

[8] M. AL-Hawawreh, N. Moustafa, and E. Sitnikova, "Identification of malicious activities in industrial internet of things based on deep learning models," *Journal of Information Security and Applications*, vol. 41, pp. 1–11, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2214212617306002

[9] E. Ciklabakkal, A. Donmez, M. Erdemir, E. Süren, M. K. Yilmaz, and P. Angin, "ARTEMIS: an intrusion detection system for MQTT attacks in internet of things," in *38th Symposium on Reliable Distributed Systems, SRDS 2019, Lyon, France, October 1-4, 2019*. IEEE, 2019, pp. 369–371. [Online]. Available: https://doi.org/10.1109/SRDS47363.2019.00053

[10] Y. Guan and N. Ezzati-Jivan, "Malware system calls detection using hybrid system," in *2021 IEEE International Systems Conference (SysCon)*, 2021, pp. 1–8.

[11] P. Deshpande, S. C. Sharma, S. K. Peddoju, and S. Junaid, "Hids: A host based intrusion detection system for cloud computing environment," *International Journal of System Assurance Engineering and Management*, vol. 9, pp. 567–576, 2018.

[12] G. Karatas and O. K. Sahingoz, "Neural network based intrusion detection systems with different training functions," *2018 6th International Symposium on Digital Forensic and Security (ISDFS)*, pp. 1–6, 2018.

[13] K. Singh, S. C. Guntuku, A. Thakur, and C. Hota, "Big Data Analytics framework for Peer-to-Peer Botnet detection using Random Forests," *Information Sciences*, vol. 278, pp. 488–497, Sep. 2014.

[14] F. J. Mora-Gimeno, H. Mora-Mora, B. Volckaert, and A. Atrey, "Intrusion Detection System Based on Integrated System Calls Graph and Neural Networks," *IEEE Access*, vol. 9, pp. 9822–9833, 2021.

[15] S. Srinivasan, A. Kumar, M. Mahajan, D. Sitaram, and S. Gupta, "Probabilistic real-time intrusion detection system for docker containers," in *SSCC*, 2018.

[16] R. Wood, "Damn Vulnerable Web Application (DVWA)," Jul. 2022, last accessed: 30 July, 2022.

[17] A. S. Abed, T. Clancy, and D. S. Levy, "Intrusion Detection System for Applications Using Linux Containers," *STM*, 2015.

[18] M. Cavalcanti, P. Inacio, and M. Freire, "Performance Evaluation of Container-Level Anomaly-Based Intrusion Detection Systems for Multi-Tenant Applications Using Machine Learning Algorithms," in *The 16th International Conference on Availability, Reliability and Security*, ser. ARES 2021. New York, NY, USA: Association for Computing Machinery, Aug. 2021, pp. 1–9.

[19] J. Flora, P. Gonçalves, and N. Antunes, "Using Attack Injection to Evaluate Intrusion Detection Effectiveness in Container-based Systems," in *2020 IEEE 25th Pacific Rim International Symposium on Dependable Computing (PRDC)*, Dec. 2020, pp. 60–69.

[20] C.-W. Tien, T.-Y. Huang, C.-W. Tien, T.-C. Huang, and S.-Y. Kuo, "KubAnomaly: Anomaly detection for the Docker orchestration platform with neural network approaches," *Engineering Reports*, vol. 1, no. 5, p. e12080, 2019.

[21] O. Tunde-Onadele, J. He, T. Dai, and X. Gu, "A Study on Container Vulnerability Exploit Detection," in *2019 IEEE International Conference on Cloud Engineering (IC2E)*, Jun. 2019, pp. 121–127.

[22] M. M. Röhling, M. Grimmer, D. Kreubel, J. Hoffmann, and B. Franczyk, "Standardized container virtualization approach for collecting host intrusion detection data," in *2019 Federated Conference on Computer Science and Information Systems (FedCSIS)*, Sep. 2019, pp. 459–463.

[23] "Tpc-c," https://www.tpc.org/tpcc/, last accessed: 24 June, 2022.

[24] J. Chen, H. Huang, and H. Chen, "Informer: Irregular traffic detection for containerized microservices RPC in the real world," *High-Confidence Computing*, vol. 2, no. 2, p. 100050, Jun. 2022.

[25] R. W. Shirey, "Internet Security Glossary, Version 2," Internet Engineering Task Force, Request for Comments RFC 4949, Aug. 2007.

[26] W. Stallings and L. Brown, *Computer Security: Principles and Practice*, 3rd ed. Pearson, 2014.

[27] R. Shu, X. Gu, and W. Enck, "A study of security vulnerabilities on docker hub," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, ser. CODASPY '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 269–280. [Online]. Available: https://doi.org/10.1145/3029806.3029832

[28] P. Sharma, L. Chaufournier, P. Shenoy, and Y. C. Tay, "Containers and Virtual Machines at Scale: A Comparative Study," in *Proceedings of the 17th International Middleware Conference*. Trento Italy: ACM, Nov. 2016, pp. 1–13.

[29] S. Sultan, I. Ahmad, and T. Dimitriou, "Container Security: Issues, Challenges, and the Road Ahead," *IEEE Access*, vol. 7, pp. 52 976–52 996, 2019.

[30] I. Sharafaldin, Canadian Institute for Cybersecurity (CIC), UNB, Fredericton, Canada, A. Gharib, Canadian Institute for Cybersecurity (CIC), UNB, Fredericton, Canada, A. H. Lashkari, Canadian Institute for Cybersecurity (CIC), UNB, Fredericton, Canada, A. A. Ghorbani, and Canadian Institute for Cybersecurity (CIC), UNB, Fredericton, Canada, "Towards a Reliable Intrusion Detection Benchmark Dataset," *Software Networking*, vol. 2017, no. 1, pp. 177–200, 2017.

[31] E. K. Viegas, A. O. Santin, and L. S. Oliveira, "Toward a reliable anomaly-based intrusion detection in real-world environments," *Computer Networks*, vol. 127, pp. 200–216, Nov. 2017.

[32] S. L. Salzberg, "C4.5: Programs for machine learning by j. ross quinlan. morgan kaufmann publishers, inc., 1993," *Machine Learning*, vol. 16, no. 3, p. 235–240, 1994.

[33] J. Su and H. Zhang, "A fast decision tree learning algorithm," in *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, July 16-20, 2006, Boston, Massachusetts, USA*. AAAI Press, 2006, pp. 500–505. [Online]. Available: http://www.aaai.org/Library/AAAI/2006/aaai06-080.php

[34] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001. [Online]. Available: https://doi.org/10.1023/A:1010933404324

[35] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, no. 3, p. 273–297, 1995.

[36] J. Platt, "Sequential minimal optimization: A fast algorithm for training support vector machines," *Advances in Kernel Methods-Support Vector Learning*, vol. 208, 07 1998.

[37] D.-K. Kang, D. Fuller, and V. Honavar, "Learning classifiers for misuse and anomaly detection using a bag of system calls representation," in *Proceedings from the Sixth Annual IEEE SMC Information Assurance Workshop*, Jun. 2005, pp. 118–125.

[38] A. S. Abed, T. C. Clancy, and D. S. Levy, "Applying Bag of System Calls for Anomalous Behavior Detection of Applications in Linux Containers," in *2015 IEEE Globecom Workshops (GC Wkshps)*, Dec. 2015, pp. 1–5.

[39] The Apache Software Foundation, "Jmeter," The Apache Software Foundation, Jul. 2022, last accessed: 30 July, 2022.

[40] "Docker," https://www.docker.com/, last accessed: 24 June, 2022.

[41] Rapid7, "Metasploit," Rapid7, Jul. 2022, last accessed: 30 July, 2022.

[42] S. Stack, "rConfig," https://github.com/rconfig/rconfig, Jun. 2022, last accessed: 30 July, 2022.

[43] M. Ring, S. Wunderlich, D. Scheuring, D. Landes, and A. Hotho, "A Survey of Network-based Intrusion Detection Data Sets," *Computers & Security*, vol. 86, pp. 147–167, Sep. 2019.

[44] T. T. Nguyen and G. Armitage, "A survey of techniques for internet traffic classification using machine learning," *IEEE Communications Surveys & Tutorials*, vol. 10, no. 4, pp. 56–76, 2008.

[45] G. Engelen, V. Rimmer, and W. Joosen, "Troubleshooting an Intrusion Detection Dataset: The CICIDS2017 Case Study," in *2021 IEEE Security and Privacy Workshops (SPW)*. San Francisco, CA, USA: IEEE, May 2021, pp. 7–12.

[46] I. H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques*, 2nd ed. San Francisco: Morgan Kaufmann, 2005.

[47] F. Alam and S. Pachauri, "Comparative Study of J48, Naive Bayes and One-R Classification Technique for Credit Card Fraud Detection using WEKA," *Advances in Computational Sciences and Technology*, vol. 10, p. 14, 2017.

[48] T. Garg and S. S. Khurana, "Comparison of classification techniques for intrusion detection dataset using WEKA," in *International Conference on Recent Advances and Innovations in Engineering (ICRAIE-2014)*, May 2014, pp. 1–5.

[49] J. Platt, "Sequential minimal optimization : A fast algorithm for training support vector machines," *Microsoft Research Technical Report*, 1998.