A Study on Container Vulnerability Exploit Detection

Olufogorehan Tunde-Onadele, Jingzhu He, Ting Dai, Xiaohui Gu
Department of Computer Science
North Carolina State University
Email: {oatundeo, jhe16, tdai, xgu}@ncsu.edu

***Abstract*—Containers have become increasingly popular for deploying applications in cloud computing infrastructures. However, recent studies have shown that containers are prone to various security attacks. In this paper, we conduct a study on the effectiveness of various vulnerability detection schemes for containers. Specifically, we implement and evaluate a set of static and dynamic vulnerability attack detection schemes using 28 real world vulnerability exploits that widely exist in docker images. Our results show that the static vulnerability scanning scheme only detects 3 out of 28 tested vulnerabilities and dynamic anomaly detection schemes detect 22 vulnerability exploits. Combining static and dynamic schemes can further improve the detection rate to 86% (i.e., 24 out of 28 exploits). We also observe that the dynamic anomaly detection scheme can achieve more than 20 seconds lead time (i.e., a time window before attacks succeed) for a group of commonly seen attacks in containers that try to gain a shell and execute arbitrary code.**

***Index Terms*—Container Security, Anomaly Detection, Machine Learning.**

I. Introduction

Containers have recently become a popular application deployment platform that can package an application and its dependencies (e.g., source code, system libraries) with lower overhead than virtual machines. However, due to its easy deployment nature, containers are prone to various security vulnerabilities. Previous work has shown security vulnerabilities widely exist in both official and community images [1]–[3]. Vulnerabilities in outdated packages can be exposed to various types of attacks (e.g., denial of service, gain privilege, execute code) and vulnerabilities can propagate due to dependency relationships between images [2]. Hence, security has become one of the top concerns for the user to use containers in production environments [4].

Existing container vulnerability detection schemes can be broadly classified into two groups: 1) static container image analysis and 2) dynamic runtime detection. The static schemes mainly focus on static vulnerability detection using container image scanning [5]–[8]. Static image scanners can detect known vulnerabilities by matching the packages and their versions with remote Common Vulnerabilities and Exposures (CVE) databases. However, the identified package list might not always accurately include all the packages installed, and customized code or scripts are not analyzed through static analysis. Moreover, vulnerabilities that are not included in existing CVE databases will not be detected (e.g., vulnerabilities not publicly disclosed, zero-day vulnerabilities). Dynamic runtime detection tools monitor container behaviors and detect anomalous activities during runtime [9]–[11]. However, most of these tools are policy-based, which cannot adapt to changing behaviors. For example, Sysdig Falco [11] employs pre-defined policies that describe the allowed or disallowed behaviors for a process, in terms of system calls, their arguments, and host resources accessed.

In this paper, we conduct a study over different vulnerability detection techniques and evaluate their effectiveness on detecting security vulnerabilities of the applications running inside containers. Particularly, we focus on out-of-box detection techniques which do not require any modifications to monitored applications and are more resilient to attacks than inside-box schemes. We consider both static and dynamic detection techniques and perform comparisons among them in terms of detection accuracy and overhead.

Compared to traditional host environments, containers present a set of new challenges to vulnerability exploit detection: 1) containers are often short-lived, which implies that the detection scheme needs to produce real time alerts without requiring a large amount of training data; 2) containers are often dynamic, which requires that the detection should not make any assumption about the container such as available resources or application workloads; and 3) containers are often light-weight, which requires that the detection algorithm should not impose high overhead to the container.

We first study the open source static analysis engine Clair [5] as an example for static analysis tools. Clair inspects containers layer-by-layer for known vulnerabilities, which continuously imports vulnerability data from a set of resources (e.g., Debian Security Bug Tracker, Ubuntu CVE Tracker, Red Hat Security Data). Container images are indexed into a list of features (e.g., installed packages, package versions), and Clair queries the vulnerability data to correlate the indexed features with vulnerability database to generate a list of vulnerabilities that threaten the images. We then study a set of dynamic detection schemes using unsupervised anomaly detection algorithms (e.g., clustering [12], k nearest neighbor [13], self-organizing map [14]). Compared to supervised machine learning, unsupervised anomaly detection approaches do not require labeled training data and can capture previously unseen attacks. We evaluate these different detection schemes using real-world vulnerabilities that are triggered in commonly used server applications such as Tomcat, Apache, and ElasticSearch.

Specifically, this paper makes the following contributions:

- We reproduce 28 commonly seen real world security vulnerabilities discovered in Docker Hub images and conduct a comparative study over both static and dynamic vulnerability detection schemes using those security vulnerabilities.
- We collect the detection accuracy of CoreOS Clair, an open source static Docker image vulnerability detection tool. Our results show that Clair can only detect 3 out of the 28 vulnerabilities.
- We implement a system call collection and feature extraction system and apply a set of widely used unsupervised anomaly detection schemes (i.e., k nearest neighbors, k-means clustering, k-nearest neighbors combined with principal component analysis for dimension reduction, self-organizing map) to catch triggered attacks online.

Our results show that it is promising to use dynamic anomaly detection schemes to catch vulnerability exploits in containers: self-organizing map based anomaly detection can catch 22 out of 28 tested vulnerability exploits while incurring a low false positive rate (1.7% on average). Moreover, the dynamic anomaly detection scheme can achieve more than 20 seconds lead time (e.g., a time window before attacks succeed) for a group of attacks that try to gain a shell and execute arbitrary code. We also find that it is beneficial to combine static and dynamic vulnerability detection schemes, which can further improve the detection coverage to catch 24 exploits.

The rest of the paper is organized as follows. §II presents our empirical study methodology. §III describes the experimental results. §IV compares our work with related work. Finally, the paper concludes in §V.

II. Methodology

In this section, we describe our study methodology. We first introduce the real-world vulnerabilities studied. We then describe the set of static and dynamic vulnerability detection schemes considered.

A. Real-World Vulnerabilities

Table I shows the 28 real-world vulnerabilities collected in 24 different applications from the commonly used vulnerability repository, i.e., Exploit Database [15]. We categorize all the 28 vulnerabilities into six groups based on their threat impact: 1) return a shell and execute arbitrary code, 2) execute arbitrary code, 3) disclose credential information, 4) consume excessive CPU, 5) make applications crash, and 6) perform escalation of privilege. These categories are among the top vulnerability types discovered in Docker Hub [2]. Most of these vulnerabilities are reported within the past three years and marked with “High” or “Critical” severity rankings, denoted by CVSS scores [1]. Our application set also exhibits a wide coverage, ranging from back-end database systems to front-end web servers to represent different server applications running inside containers.

TABLE I: List of Explored Real-world Vulnerabilities.

Threat Impact	CVE ID	CVSS Score	Application	Exploitation Tool
Return a shell and execute arbitrary code	CVE-2015-8103	7.5	JBoss	JexBoss
	CVE-2017-7494	10.0	Samba	Metasploit
	CVE-2016-10033	7.5	PhpMailer	Metasploit
	CVE-2015-2208	7.5	phpMoAdmin	Metasploit
	CVE-2016-9920	6.0	Webmail	PoC
	CVE-2015-1427	7.5	Elasticsearch	Metasploit
	CVE-2014-3120	6.8	Elasticsearch	Metasploit
	CVE-2012-1823	7.5	PHP	Metasploit
	CVE-2017-11610	9.0	Supervisor	Metasploit
	CVE-2017-8291	6.8	Ghostscript	PoC
	CVE-2015-3306	10.0	ProFTPd	Metasploit
	CVE-2017-12615	6.8	Apache Tomcat	PoC
	CVE-2016-3088	7.5	Activemq	Metasploit
	CVE-2017-12149	7.5	JBoss	PoC
	CVE-2015-8562	7.5	Joomla	Metasploit
Execute arbitrary code	CVE-2014-6271	10.0	Bash	Metasploit
	CVE-2017-5638	10.0	Struts	PoC
	CVE-2017-12794	4.3	Django	PoC
	CVE-2016-3714	10.0	ImageMagick	Metasploit
Disclose credential information	CVE-2017-7529	5.0	Nginx	PoC
	CVE-2015-5531	5.0	Elasticsearch	Metasploit
	CVE-2014-0160	5.0	OpenSSL	Metasploit
	CVE-2017-8917	7.5	Joomla	sqlmap
Consume excessive CPU	CVE-2016-6515	7.8	OpenSSH	PoC
	CVE-2014-0050	7.5	Apache Tomcat	PoC
Crash the application	CVE-2016-7434	5.0	NTP	PoC
	CVE-2015-5477	7.8	BIND	Metasploit
Escalation of privilege	CVE-2017-12635	10.0	Couchdb	Burp Suite

PoC: Proof of Concept code.

We exploit the vulnerabilities by either executing the Proof of Concept (PoC) code or using penetration tools (i.e., Metasploit [16], JexBoss [17], sqlmap [18], and Burp Suite [19]). To emulate dynamic applications in real world, we employ commonly used workload generator tools (e.g., Burp Suite [19], JMeter [20]) to send requests to victim containers.

For web server applications such as Apache Tomcat, Django and Nginx, we request pages from web servers with JMeter’s HTTP sampler. This sampler enables the selection of the appropriate HTTP traffic type (e.g., `GET`, `POST`, etc.) for an application. Web requests are also sent to Joomla and Couchdb front ends to induce database operations (e.g., create, update and delete documents). For FTP servers such as ProFTPd, files are downloaded from and uploaded to the FTP server using the FTP sampler. The date requests are sent to the OpenSSH application via the JMeter plugin (i.e., SSH command). The Domain Name Server (DNS) and Network Time Protocol (NTP) requests are sent to the BIND and NTP applications via the JMeter plugin (i.e., UDP request). The smbclient is used with JMeter’s OS process sampler to produce Server Message Block (SMB) network traffic for the Samba application. As for the Elasticsearch, we send search requests via Burp Suite.

[1] Common Vulnerability Scoring System (CVSS) scores are provided by National Vulnerability Database. The higher the score, the higher the severity (i.e., “None”: 0.0; “Low”: 0.1-3.9; “Medium”: 4.0-6.9; “High”: 7.0-8.9; “Critical”: 9.0-10.0).

B. Static Vulnerability Detection Scheme

We use Clair, a widely used open source tool for static analysis of vulnerabilities in docker containers as an example of static vulnerability detection schemes. Clair works by scanning docker images and matching detected packages and their versions with a remote CVE database. Vulhub [21] provides Dockerfiles for users to build vulnerable images. A Dockerfile is a script that contains all the commands that execute in succession to build container images. Dockerfiles in Vulhub use two different ways to install vulnerable applications, i.e., through the source code and by a package manager such as `apt-get install` or `dpkg install` to install a deb file. Vulnerable container images created from local Dockerfiles can be tagged and pushed to the Quay.io registry. Vulnerability scanning is automatically performed by Quay.io, and it takes about several minutes to produce the results. For each image pushed to the Quay.io registry, Clair scans the images and reports the total number of detected CVEs along with the distribution of the CVEs according to the severity rankings. For each reported CVE entry, Clair also lists a set of related information, e.g, the available CVSS score, package name, package version, and the suggestion of fixed versions of the vulnerable package. In addition, Clair also gives a hint of the specific layer where CVEs are introduced into images.

C. Dynamic Exploit Detection Approaches

Dynamic runtime detection schemes need to address two key issues: 1) what monitoring data to collect and how to extract proper features from the monitoring data? and 2) what algorithms to use for detecting vulnerability exploits?

Data Collection and Feature Extraction. The behaviors of running containers can manifest in different system metrics (e.g., CPU utilization, memory usage, and network traffic) or system calls. Although system metrics can be collected with low cost, they are heavily affected by dynamic application workloads, which makes them too noisy to be used as reliable data sources for container exploit detection. System calls are the interfaces though which applications access the services of the operating system. We observe that changes in the behaviors of containers from attempted attacks often manifest as variation in system call frequencies. For example, attempted attacks targeted at containers may introduce system calls which rarely appear during the applications' normal executions.

Our container system call logs are collected with a lightweight open source tracing tool called Sysdig [22]. Sysdig supports container monitoring with transparent instrumentation, without the agent inside each container, which enables real-time analysis of container activities.

We extract proper features from the raw system call trace within equal sampling intervals. We explored both system call frequency and system call execution time features, which are called system call frequency vectors and system call time vectors, respectively. We formulate a frequency/time vector as $V(t) = [x_1, x_2, ..., x_n]$, where x_i represents the frequency or the execution time of each type of system call in a given sample interval. Table II gives an example of the extracted frequency vectors from a processed system call list. The first line represents the number of appearances that `sys_write`, `sys_read`, `sys_futex` and `sys_epoll_wait` calls make in the time interval $[t, t+100)$ milliseconds where $t = 1516544689186$.

TABLE II: An example of frequency vectors from a processed system call list.

System call Timestamp	write	read	futex	epoll_wait
1516544689186	2	4	50	4
1516544689286	9	8	74	8
1516544689386	0	0	9	1

After extracting proper features, we need to decide what algorithms we should use to detect vulnerability exploits. As mentioned in the introduction, container vulnerability detection needs to meet a set of new challenges. First, the detection algorithm cannot assume a large amount of training data because containers are often short-lived. Second, the detection algorithm cannot assume prior knowledge about either the application behavior or the attack behavior since containers are highly dynamic. Third, the detection algorithm needs to be able to provide real time detection with low overhead. To address these unique challenges of container vulnerability detection, we chose a set of light-weight unsupervised anomaly detection schemes to evaluate.

K Nearest Neighbors (k-NN): The k-nearest neighbors algorithm (k-NN) is used to perform outlier detection. Anomalies are those samples whose average distance to its nearest neighbors fall into the top p percentile. There is a trade-off between true positive rate and false positive rate when we adjust the k and p values. If we lower p, more samples will be identified as anomalous, which might increase both true positive rate and false positive rate. The value of optimal k requires more sophisticated tuning algorithms. For container vulnerability exploit detection, it is impractical to tune the parameters on-the-fly so they can be empirically decided beforehand. In our experiments, we set k to be five and p to be 10%.

PCA + k-NN: One of the key challenges to achieve high accuracy in the k-NN algorithm lies in the presence of noise in the feature data (hundreds of different types of system calls). We choose Principal Component Analysis (PCA) as our dimension reduction strategy because PCA is fast and incurs low computation cost. In our experiments, we found that the magnitude of the top dimension is larger than that of the fifth dimension by four orders of magnitudes so we set the number of target dimensions to be five.

K-means: K-means is a traditional clustering method and easy to implement. K denotes the number of clusters of feature vectors. We consider clusters with a small number of samples, based on a cluster size threshold, as anomalous. Similar to k-NN, we can only empirically set the value of k to perform container vulnerability exploit detection.

Self-Organizing Map: Self-organizing map (SOM) [14] is a special kind of artificial neural network (ANN) which is able to reduce data dimensions and highlight similarities among data without imposing excessive learning overhead. The SOM algorithm preserves the relative distance between high dimensional data points so that points that are nearby in the input data are mapped to nearby neurons in the SOM.

We conduct training of the SOM network using the algorithm outlined by UBL [23]. A mapped neuron with a large neighborhood area value is far away from others and considered abnormal. The threshold is determined by a certain percentile value p of neighborhood area size. Intuitively, a low p value will make the detection more sensitive and raise more alerts.

III. Experimental Evaluation

In this section, we first describe our evaluation setup and then present our evaluation results in detail.

A. Experiment Setup

We set up victim containers in a virtual machine using Docker v17.05.0 in order to eliminate the interference brought by other activities in the host. The virtual machine is equipped with 2 GB memory and 40 GB disk, running 64 bit Ubuntu v16.04. Each victim container runs a vulnerable application associated with a specific CVE. The static vulnerability scanning is achieved by Clair v2.0.0. The syscall trace is collected using Sysdig v0.19.1.

To evaluate the effectiveness of each detection approach (e.g., real-time) and to restore the container practical usage scenarios (i.e. short-liveness), we collect system calls produced by the victim containers in a short period of time. Specifically, for each vulnerability, we first launch the victim container and start the vulnerable application. We then send workloads from the VM and start the Sysdig tracing module. Sysdig collects the system call traces for about six minutes, including the system calls produced by the application under normal workload and during the attack (i.e, from when the attack is triggered to when the attack succeeds). We then extract the time vectors and frequency vectors from the raw system call traces in samples of 100 milliseconds. We run different dynamic detection algorithms over those feature vectors.

B. Detection Results

We compare different vulnerability detection schemes using four metrics: 1) detection coverage: whether each approach can detect the vulnerabilities? 2) false positive rate: how accurate each approach can achieve for the detection? and 3) lead time: how quickly each approach can detect the attacks and thus prevent compromise in time?

1) Detection Coverage: Table III shows the detection coverage of different anomaly detection approaches. Overall, dynamic approaches achieve better detection coverage than the static approach. Specifically, SOM approaches achieve the highest detection coverage on average, followed by the K-means clustering approach. The k-NN and k-NN combined with PCA approaches achieve the lowest detection coverage among all dynamic approaches. The static approach (i.e., Clair) can only detect three out of 28 CVEs with the average detection coverage of 10.71%. The static approach can be utilized with a dynamic method to achieve the strengths of both techniques. Accordingly, the highest detection coverage results from combining the static and SOM frequency approaches. This pair can detect 24 out of 28 vulnerability cases, giving a detection coverage of 85.71%.

Clair achieves low detection coverage due to the lack of container image features (e.g., installed packages, package versions), or the incomplete remote vulnerability database. For example, Clair fails to detect the CVE-2017-7494 in the vulnerable docker image because vulnerable packages are installed using source code. Without using package managers to install vulnerable packages, e.g., `apt-get install`, Clair cannot extract the image features, thus it fails to detect the vulnerabilities by correlating the indexed features with remote vulnerability database. Another example is the CVE-2016-6515. Clair fails to detect this vulnerability due to the incomplete remote vulnerability database. In fact, Clair has extracted the container image feature (i.e. OpenSSH v1:7.2p2-4ubuntu2.1), but reports an incomplete list of vulnerabilities that threaten this image, e.g., CVE-2016-10009, CVE-2016-10012, CVE-2016-10010, CVE-2016-10011, CVE-2017-15906, and CVE-2016-8858.

The k-NN approach can only detect 32.14% vulnerabilities. It detects 7 out of 15 vulnerabilities that return a shell and execute arbitrary code and both the vulnerabilities that crash the applications, but fails to detect other types of vulnerabilities.

The k-NN combined with PCA approach achieves a slightly better detection coverage than the pure k-NN approach. It detects six out of 15 vulnerabilities that return a shell and execute arbitrary code, and another four vulnerabilities in different categories.

The K-means approach achieves 67.86% detection coverage by detecting 11 out of 15 vulnerabilities that return a shell and execute arbitrary code, 3 out of 4 vulnerabilities that execute arbitrary code, 3 out of 4 credential information disclosure vulnerabilities, two excessive CPU consumption vulnerabilities but it fails to detect any vulnerabilities which could crash the application or cause escalation of privilege.

The SOM approach over system call time vectors (SOM time) achieves the average detection coverage of 75% while the SOM approach over system call frequency vectors (SOM frequency) achieves the average detection coverage of 79%. In particular, they both can detect most or all of the vulnerabilities which would return an interactive shell and enable attackers to execute arbitrary code inside containers. One insight behind this is that system calls generated during the process of exploitation and the arbitrary code execution are distinct from those generated during applications' normal running process. For example, CVE-2014-3120 allows attackers to exploit a remote command execution (RCE) vulnerability in a vulnerable version of ElasticSearch (e.g., v1.1.1). We observed that certain system calls appear more frequently when the vulnerability is exploited (e.g., `sys_lseek`, `sys_mprotect`). We also found that specific system calls only appear after the attack is triggered (e.g., `sys_getuid`).

TABLE III: Detection Result of Clair and Anomaly Detection Approaches.

Threat Impact	CVE ID	CVSS Score	Clair	k-NN		PCA + k-NN		K-means		SOM time		SOM freq	
			Detected	Detected	FPR	Detected	FPR	Detected	FPR	Detected	FPR	Detected	FPR
Return a shell and execute arbitrary code	CVE-2015-8103	7.5	✗	✓	9.97%	✓	9.97%	✓	2.98%	✓	2.47%	✓	0.84%
	CVE-2017-7494	10.0	✗	✓	9.93%	✓	9.96%	✓	4.27%	✓	7.48%	✓	1.10%
	CVE-2016-10033	7.5	✗	✓	9.92%	✗	9.95%	✓	8.78%	✓	0.17%	✓	0.17%
	CVE-2015-2208	7.5	✗	✓	9.91%	✓	9.94%	✗	0.00%	✓	5.26%	✓	3.18%
	CVE-2016-9920	6.0	✗	✗	9.97%	✗	9.97%	✗	0.00%	✓	2.67%	✓	0.48%
	CVE-2015-1427	7.5	✗	✓	9.93%	✓	9.93%	✓	9.14%	✓	0.45%	✓	1.54%
	CVE-2014-3120	6.8	✗	✗	9.92%	✓	9.72%	✓	10.08%	✓	1.46%	✓	1.72%
	CVE-2012-1823	7.5	✗	✗	9.92%	✗	9.92%	✓	2.76%	✓	1.71%	✓	6.50%
	CVE-2017-11610	9.0	✗	✓	9.96%	✗	9.96%	✓	1.13%	✓	0.06%	✓	1.58%
	CVE-2017-8291	6.8	✗	✗	9.94%	✗	9.94%	✓	4.90%	✗	0.14%	✓	1.41%
	CVE-2015-3306	10.0	✗	✗	9.96%	✗	9.96%	✓	2.56%	✓	8.32%	✓	0.95%
	CVE-2017-12615	6.8	✗	✓	9.92%	✗	9.95%	✗	0.00%	✓	1.93%	✓	1.96%
	CVE-2016-3088	7.5	✗	✗	9.92%	✓	9.72%	✓	4.30%	✓	0.63%	✓	3.04%
	CVE-2017-12149	7.5	✗	✗	9.96%	✗	9.96%	✓	3.36%	✓	0.83%	✓	1.72%
	CVE-2015-8562	7.5	✗	✗	9.82%	✗	9.82%	✗	35.27%	✗	0.27%	✓	5.28%
Execute arbitrary code	CVE-2014-6271	10.0	✓	✗	9.97%	✗	9.97%	✗	1.60%	✗	4.64%	✓	0.42%
	CVE-2017-5638	10.0	✗	✗	9.95%	✓	9.65%	✓	4.09%	✓	0.84%	✓	3.17%
	CVE-2017-12794	4.3	✗	✗	9.95%	✗	9.95%	✓	8.90%	✗	0.55%	✗	3.10%
	CVE-2016-3714	10.0	✗	✗	9.97%	✗	9.97%	✓	1.06%	✓	0.36%	✓	0.26%
Disclose credential information	CVE-2017-7529	5.0	✗	✗	9.78%	✗	9.78%	✓	10.40%	✗	1.25%	✗	0.08%
	CVE-2015-5531	5.0	✗	✗	9.95%	✗	9.95%	✓	5.78%	✓	0.72%	✓	1.22%
	CVE-2014-0160	5.0	✓	✗	9.95%	✗	9.95%	✓	5.21%	✓	0.38%	✗	0.96%
	CVE-2017-8917	7.5	✗	✗	9.92%	✓	9.50%	✗	0.25%	✗	0.08%	✗	0.13%
Consume excessive CPU	CVE-2016-6515	7.8	✗	✗	9.97%	✗	9.97%	✓	1.02%	✓	6.73%	✓	3.65%
	CVE-2014-0050	7.5	✗	✗	9.92%	✓	9.72%	✓	6.30%	✓	2.01%	✓	1.97%
Crash the application	CVE-2016-7434	5.0	✗	✓	9.72%	✓	9.72%	✗	36.57%	✗	0.49%	✗	0.00%
	CVE-2015-5477	7.8	✓	✓	9.91%	✗	9.94%	✗	10.22%	✓	0.74%	✗	0.31%
Escalation of privilege	CVE-2017-12635	10.0	✗	✗	9.79%	✗	9.79%	✗	33.88%	✓	3.66%	✓	1.26%
Average Results			10.71%	32.14%	9.92%	35.71%	9.88%	67.86%	7.67%	75.00%	1.88%	78.57%	1.71%

The K-means, SOM time and SOM frequency approaches achieve 100% detection coverage for the vulnerabilities which can cause performance issues (e.g., consume excessive CPU usage). For example, in CVE-2016-6515, the `auth_password()` function in OpenSSH before version 7.3 does not limit password lengths for password authentication, which allows remote attackers to launch a DoS attack via a long string, causing infinite loops. Another example is CVE-2014-0050, where attackers send a crafted content-type header to a vulnerable version of Apache Tomcat (e.g., v7.0-v7.0.50 and v8.0-v8.0.1), causing the loop index to be always less than or equal to the upper bound, hanging Tomcat endlessly.

2) False Positive Rate: Table III also shows the false positive rate of different anomaly detection approaches. Overall, the SOM approaches achieve the lowest false positive rate (1.7% for SOM frequency and 1.9% for SOM time). followed by the K-means clustering approach (7.67%). However, K-means approach has the largest FPR range from 0% to 36.57%. The k-NN and k-NN combined with PCA approaches incur the highest false positive rate (9.92% FPR and 9.88% FPR, respectively). However, these two approaches have the smallest FPR range from 9.5% to 9.97%.

We omit the false positive rate result of Clair in our evaluation because Clair can report hundreds or thousands of CVEs for each victim container. It is extremely time-consuming to validate all of its detection results manually. It is also wrong to label all the CVEs identified by Clair but not included in our benchmark in Table I as false positives.

3) Lead Time: Table IV shows the lead time achieved by different dynamic approaches for the CVEs with the thread impact of returning a shell to the attackers for executing arbitrary code. In those type of CVEs, the attackers require time-consuming operations to exploit the vulnerability such as traversing the vulnerable container to find the path of a specific writable folder (CVE-2017-7494), or creating a backdoor file in the root folder of container-side (CVE-2016-10033).

Overall, the SOM approaches achieve the largest detection lead time (28.7 seconds for SOM frequency and 25.8 seconds for SOM time). However, the other approaches' detection lead time is very low. Specifically, the k-NN combined with PCA approach achieves the average lead time of 1 second. The k-NN approach achieves the average lead time of 0.57 second. The worst case is the K-means approach which achieves a lead time of 0.36 second.

The results show that the SOM approaches are more practical than the other machine learning methods for real-time vulnerability detection. This time window is helpful because effective emergency measures can be taken by administrators to prevent the containers from being totally compromised.

We do not conduct lead time analysis for other CVE impact types such as crash of the application, because these attacks can finish immediately after the exploitation.

IV. Related Work

Intrusion Detection using Machine Learning. Previous work has utilized supervised machine learning methods in intrusion detection [24]–[27]. Others avoid the need for ”labeled” training data with unsupervised learning [23], [28]–[32]. UBL [23] introduces an unsupervised anomaly prediction system for computing clouds using metrics such as CPU usage. Moreover,

TABLE IV: The Lead Time of Anomaly Detection Approaches for CVEs that Return a Shell and Execute Arbitrary Code. "-": the approach does not detect the vulnerability.

Threat Impact	CVE ID	CVSS Score	k-NN (seconds)	PCA + k-NN (seconds)	K-means (seconds)	SOM time (seconds)	SOM freq (seconds)
Return a shell and execute arbitrary code	CVE-2015-8103	7.5	0	0	1	1	28
	CVE-2017-7494	10.0	0	1	1	28	35
	CVE-2016-10033	7.5	0	-	0	67	124
	CVE-2015-2208	7	0	1	-	1	1
	CVE-2016-9920	6.0	-	-	-	121	118
	CVE-2015-1427	7.5	4	4	0	2	7
	CVE-2014-3120	6.8	-	0	1	7	8
	CVE-2012-1823	7.5	-	-	0	44	45
	CVE-2017-11610	9.0	0	-	0	1	1
	CVE-2017-8291	9.0	-	-	0	-	1
	CVE-2015-3306	10.0	-	-	1	1	1
	CVE-2017-12615	6.8	0	-	-	12	5
	CVE-2016-3088	7.5	-	0	0	42	48
	CVE-2017-12149	7.5	-	-	0	8	8
	CVE-2015-8562	7.5	-	-	-	-	1
Average Lead Time			0.57	1.00	0.36	25.77	28.73

deep learning techniques have been applied in intrusion detection in recent years [33], [34]. In comparison, our work focuses on studying real time container exploit detection schemes using light-weight unsupervised anomaly detection schemes.

Intrusion Detection using Static and Dynamic Analysis. Work has been done to statically analyze the application source code and identify malicious code blocks and unwanted information flows [35]–[38]. Previous work also dynamically monitor application runtime behavior to detect exploitation [39]–[42]. Sysdig Falco [11] is a rule-based checking tool. It detects vulnerabilities by a set of (27 in current release) pre-defined heuristics for each process. However, rule-based systems, specific to certain types of vulnerabilities, face challenges in detecting previously unknown vulnerabilities.

Intrusion Detection using System Calls. Numerous intrusion detection systems rely on system call information to understand malicious behaviors [24], [43]–[45]. The above approaches often require a large amount of training data and may incur high resource cost. In contrast, our study focuses on exploring practical unsupervised machine learning algorithms for detecting vulnerabilities in dynamic and short-lived containers.

V. Conclusion

Emerging container techniques speed up deployments of applications and ease the distribution and delivery of software, but securing containers still has a long way to go toward maturity. In this paper, we conduct a study to evaluate the effectiveness of different static and dynamic vulnerability exploit detection schemes for container hosted applications. Our initial experiments using 28 real world vulnerabilities discovered in 24 commonly used server applications show that static vulnerability scanning of container images alone is insufficient, which only detects 3 out of 28 vulnerabilities. Dynamic anomaly detection schemes using unsupervised machine learning methods can effectively detect 22 vulnerability exploits with low false positive rates. Combining static and dynamic schemes can further increase the detection coverage to 86% (i.e., 24 out of 28 vulnerabilities). Our experiments are still preliminary. In our future work, we plan to extend our vulnerability cases and further improve the detection accuracy by combining and augmenting our vulerability detection schemes.

VI. Acknowledgement

The authors would like to thank the anonymous reviewers for their valuable feedback. This work is supported by the NSA Science of Security Lablet: Impact through Research, Scientific Methods, and Community Development under the contract number H98230-17-D-0080. Any opinions, conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] J. Gummaraju, T. Desikan, and Y. Turner, "Over 30% of official images in docker hub contain high priority security vulnerabilities," Technical report, BanyanOps, Tech. Rep., 2015.

[2] 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*. ACM, 2017, pp. 269–280.

[3] Docker Image Vulnerability Research, https://www.federacy.com/docker_image_vulnerabilities, 2017.

[4] A. Bettini, "Vulnerability exploitation in Docker container environments," https://www.blackhat.com/docs/eu-15/materials/eu-15-Bettini-Vulnerability-Exploitation-In-Docker-Container-Environments-wp.pdf, 2015.

[5] Clair, https://github.com/coreos/clair, 2017.

[6] Dockscan, https://github.com/kost/dockscan, 2018.

[7] Banyan Collector, https://github.com/banyanops/collector, 2018.

[8] OpenSCAP Container Compliance, https://github.com/OpenSCAP/container-compliance, 2016.

[9] The Road to Twistlock 2.0: Runtime Radar for Runtime Defense, https://www.twistlock.com/2017/04/11/road-twistlock-2-0-runtime-radar/.

[10] NeuVector, https://neuvector.com/, 2018.

[11] Sysdig Falco, https://www.sysdig.org/falco/, 2018.

[12] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu, "An efficient k-means clustering algorithm: Analysis and implementation," *IEEE Transactions on Pattern Analysis & Machine Intelligence*, no. 7, pp. 881–892, 2002.

[13] N. S. Altman, "An introduction to kernel and nearest-neighbor nonparametric regression," *The American Statistician*, vol. 46, no. 3, pp. 175–185, 1992.

[14] T. Kohonen, "The self-organizing map," *Neurocomputing*, vol. 21, no. 1-3, pp. 1–6, 1998.

[15] Offensive Security's Exploit Database Archive, https://www.exploit-db.com/, 2018.

[16] Metasploit penetration testing framework, https://www.metasploit.com/, 2018.

[17] JexBoss, https://github.com/joaomatosf/jexboss, 2018.

[18] Sqlmap, sqlmap.org/, 2018.

[19] Burp Suite Scanner, https://portswigger.net/burp, 2018.

[20] "Apache JMeter," https://jmeter.apache.org/, 2018.

[21] Vulhub: Docker-Compose File for Vulnerability Environment, http://vulhub.org/, 2018.

[22] Sysdig, http://www.sysdig.org/, 2016.

[23] D. J. Dean, H. Nguyen, and X. Gu, "Ubl: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems," in *Proceedings of the 9th international conference on Autonomic computing*. ACM, 2012, pp. 191–200.

[24] S. K. Dash, G. Suarez-Tangil, S. Khan, K. Tam, M. Ahmadi, J. Kinder, and L. Cavallaro, "Droidscribe: Classifying android malware based on runtime behavior," in *Security and Privacy Workshops (SPW), 2016 IEEE*. IEEE, 2016, pp. 252–261.

[25] C. Giuffrida, S. Ortolani, and B. Crispo, "Memoirs of a browser: A cross-browser detection model for privacy-breaching extensions," in *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*. ACM, 2012, pp. 10–11.

[26] G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier, "Toward large-scale vulnerability discovery using machine learning," in *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*. ACM, 2016, pp. 85–96.

[27] Z. Gu, K. Pei, Q. Wang, L. Si, X. Zhang, and D. Xu, "Leaps: Detecting camouflaged attacks with statistical learning guided by program analysis," in *Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on*. IEEE, 2015, pp. 57–68.

[28] M. Elsabagh, D. Fleck, A. Stavrou, M. Kaplan, and T. Bowen, "Practical and accurate runtime application protection against dos attacks," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2017, pp. 450–471.

[29] T.-F. Yen, A. Oprea, K. Onarlioglu, T. Leetham, W. Robertson, A. Juels, and E. Kirda, "Beehive: Large-scale log analysis for detecting suspicious activity in enterprise networks," in *Proceedings of the 29th Annual Computer Security Applications Conference*. ACM, 2013, pp. 199–208.

[30] A. Tang, S. Sethumadhavan, and S. J. Stolfo, "Unsupervised anomaly-based malware detection using hardware features," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2014, pp. 109–129.

[31] M. Z. Rafique and J. Caballero, "Firma: Malware clustering and network signature generation with mixed network behaviors," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2013, pp. 144–163.

[32] Y. Li, J. Jang, X. Hu, and X. Ou, "Android malware clustering through malicious payload mining," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2017, pp. 192–214.

[33] Z. Yuan, Y. Lu, Z. Wang, and Y. Xue, "Droid-sec: deep learning in android malware detection," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 371–372.

[34] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 1285–1298.

[35] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, "Vulpecker: an automated vulnerability detection system based on code similarity analysis," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACM, 2016, pp. 201–213.

[36] D. Dewey and J. T. Giffin, "Static detection of c++ vtable escape vulnerabilities in binary code." in *NDSS*, 2012.

[37] J. Pewny, F. Schuster, L. Bernhard, T. Holz, and C. Rossow, "Leveraging semantic signatures for bug search in binary programs," in *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, 2014, pp. 406–415.

[38] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna, "Triggerscope: Towards detecting logic bombs in android applications," in *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016, pp. 377–396.

[39] Y. Chen, M. Khandaker, and Z. Wang, "Pinpointing vulnerabilities," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 2017, pp. 334–345.

[40] A. Kharraz, S. Arshad, C. Mulliner, W. K. Robertson, and E. Kirda, "Unveil: A large-scale, automated approach to detecting ransomware." in *USENIX Security Symposium*, 2016, pp. 757–772.

[41] P. Saxena, S. Hanna, P. Poosankam, and D. Song, "Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications." in *NDSS*, 2010.

[42] M. Nie, P. Su, Q. Li, Z. Wang, L. Ying, J. Hu, and D. Feng, "Xede: Practical exploit early detection," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2015, pp. 198–221.

[43] D. Mutz, W. Robertson, G. Vigna, and R. Kemmerer, "Exploiting execution context for the detection of anomalous system calls," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2007, pp. 1–20.

[44] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, behavior-based malware clustering," in *NDSS*, vol. 9. Citeseer, 2009, pp. 8–11.

[45] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan, "Synthesizing near-optimal malware specifications from suspicious behaviors," in *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 2010, pp. 45–60.