CDL: Classified Distributed Learning for Detecting Security Attacks in Containerized Applications

Yuhang Lin
ylin34@ncsu.edu
North Carolina State University
Raleigh, North Carolina

Olufogorehan Tunde-Onadele
oatundeo@ncsu.edu
North Carolina State University
Raleigh, North Carolina

Xiaohui Gu
xgu@ncsu.edu
North Carolina State University
Raleigh, North Carolina

ABSTRACT

Containers have been widely adopted in production computing environments for their efficiency and low overhead of isolation. However, recent studies have shown that containerized applications are prone to various security attacks. Moreover, containerized applications are often highly dynamic and short-lived, which further exacerbates the problem. In this paper, we present CDL, a classified distributed learning framework to achieve efficient security attack detection for containerized applications. CDL integrates online application classification and anomaly detection to overcome the challenge of lacking sufficient training data for dynamic short-lived containers while considering diversified normal behaviors in different applications. We have implemented a prototype of CDL and evaluated it over 33 real world vulnerability attacks in 24 commonly used server applications. Our experimental results show that CDL can reduce the false positive rate from over 12% to 0.24% compared to traditional anomaly detection schemes without aggregating training data. By introducing application classification into container behavior learning, CDL can improve the detection rate from catching 20 attacks to 31 attacks before those attacks succeed. CDL is light-weight, which can complete application classification and anomaly detection for each data sample within a few milliseconds.

CCS CONCEPTS

• **Security and privacy → Virtualization and security**.

KEYWORDS

Container Security, Anomaly Detection, Machine Learning

ACM Reference Format:
Yuhang Lin, Olufogorehan Tunde-Onadele, and Xiaohui Gu. 2020. CDL: Classified Distributed Learning for Detecting Security Attacks in Containerized Applications. In *Annual Computer Security Applications Conference (ACSAC 2020), December 7–11, 2020, Austin, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3427228.3427236

ACSAC 2020, December 7–11, 2020, Austin, USA

ACM ISBN 978-1-4503-8858-0/20/12…$15.00
https://doi.org/10.1145/3427228.3427236

1 INTRODUCTION

Container technology is widely adopted in today's distributed computing environments for its efficiency and low overhead of isolation. However, recent studies [2, 24] have shown that containers are prone to various security attacks, which has become one of the top concerns for users to fully adopt container technology [6]. Indeed, previous study [24] reveals an alarming degree of vulnerability exposure and spread in the official Docker Hub container repository.

Security attack detection can be achieved by either a signature-driven approach [16, 17] or an anomaly detection approach [10, 11]. In this work, we focus on studying the latter approach for dynamically detecting both known and unknown attacks. Container-based distributed systems bring both new challenges and opportunities for security attack detection. On one hand, containerized applications are often ephemeral, which typically run for a short period of time before the container is stopped for saving resources because restarting a container is typically fast and incurs low cost [3, 7]. As a result, it is challenging for the anomaly detection system to collect sufficient training data to build a reliable normal behavior model. On the other hand, containerized applications are highly replicated both spatially and temporally. The user often spawns a large number of containers from the same container image to achieve concurrent processing of a large workload. The same container might be restarted at different times to process periodically repeating requests. Thus, the inherent redundancy of the container environment presents new opportunities for the anomaly detection model to leverage the power of distributed learning which can aggregate related training data from a large number of distributed containers to create a robust normal behavior model.

In this paper, we present a new *classified distributed learning* (CDL) framework for achieving efficient security attack detection in containerized applications. CDL implements a distributed learning framework to overcome the challenge of insufficient training data in ephemeral container environments. Different from the traditional learning scheme which derives an independent normal behavior model for each container from a limited set of training data obtained during its short lifetime, CDL aims at building robust normal behavior models by performing distributed learning over aggregated training data from a group of distributed containers. However, different applications can have very distinct normal behaviors. So it is insufficient to simply aggregate all training data from all containers without considering the specific applications running inside different containers. To address the challenge, we propose to incorporate application-based classification into the distributed learning framework to create more precise normal behavior model for different applications.

CDL adopts a system call driven anomaly detection approach using out-of-box container monitoring tools [1] to achieve non-intrusive, low-cost security attack detection. Specifically, we continuously collect system call traces produced by each container and extract feature vectors such as the invocation frequencies of different system call types within each sampling interval (e.g., 0.1 second). We then feed the feature vectors into anomaly detection models to detect different attacks. In this work, we choose autoencoder neural network [28] as our anomaly detection model because of its computation efficiency and high accuracy.

Specifically, CDL consists of three integrated components: 1) *application classifiers* which categorize different system call vectors into their corresponding application groups for creating precise normal behavior model for each application; 2) *data assemblers* which collect system call feature vectors from different containers and group system call feature vectors based on the application classification results (e.g., Apache Tomcat versus OpenSSL), and 3) *classified learning* which build normal behavior models for different applications and perform attack detection using application-specific models. This paper makes the following contributions:

- We propose a new classified distributed learning framework to achieve efficient security attack detection for containerized applications.
- We present efficient application classification and anomaly detection schemes using light-weight, black-box online learning methods.
- We have implemented a prototype of CDL and evaluated it over 33 recent real critical vulnerabilities with high CVSS scores in 24 commonly used server applications.

Our results show that CDL can successfully detect 31 out of 33 attacks with a low false positive rate (0.24% on average). In contrast, the traditional learning scheme without aggregating any training data incurs orders of magnitude higher false positive rate (12.74% on average) because of lacking sufficient training data. We also compare CDL with conventional distributed learning methods which aggregate all training data without considering behavior variations among different applications. The resulting model can only detect 20 out of 33 tested attacks due to unsorted training data. Moreover, CDL can detect two critical attack types (i.e., return a shell and execute arbitrary code, consume excessive CPU) 15-18 seconds *before* attacks succeed. CDL is lightweight and efficient, which can finish online anomaly detection for each extracted system call feature vector sampled at 100 millisecond intervals within a few milliseconds.

The rest of the paper is structured as follows. Section 2 presents CDL design in detail. Section 3 describes our experimental evaluation methodology and our experimental results. Section 4 compares CDL with related work. Section 5 concludes this paper.

2 SYSTEM DESIGN

In this section, we present the design of the CDL system. We first provide an overview about the CDL system. We then describe each CDL component in detail.

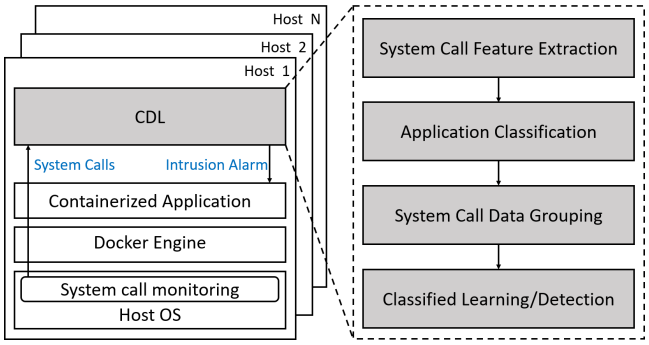


Figure 1: System overview of CDL.

2.1 System Overview

CDL implements a classified distributed learning framework shown by Figure 1. CDL leverages an open-source container monitoring tool called Sysdig [1] to collect system call traces from the outside of the containers. We leverage system calls for their ability to reveal security attacks with low cost. CDL performs continuous analysis over collected system call traces to achieve online attack detection, which consists of four major steps: 1) *system call feature extraction*, 2) *application classification*, 3) *system call data grouping*, and 4) *classified learning/detection*. Specifically, the system call feature extraction component performs continuous system call data pre-processing to extract useful features for attack detection. For example, we compute the frequency count of each system call type within a certain sampling interval to categorize the interactions between the application and the kernel. Next, the application classifier identifies groupings of containerized applications and versions (e.g., ActiveMQ v1.0, Bash v3.0) based on their system call composition and frequency. We currently leverage the random forest model to perform application classification. Such an ensemble model offers good accuracy by minimizing the over-fitting errors [13]. Next, the system call grouping module assembles the feature data from different containers into groups based on their application tags. Finally, the grouped data are fed into an unsupervised neural network to perform classified learning or detection. During classified training, we create separate models for each application class. Those models are then used to perform anomaly detection over the application containers within their respective categories.

2.2 System Call Feature Extraction

To monitor containerized applications, we trace the system calls they produce using the open source tool Sysdig [1]. The tracing tool accesses the host kernel to provide helpful information about operating system (OS) level events such as file read/write and synchronization operations. Sysdig is also able to filter the events by containers. Thus, the result is a detailed capture of the system calls made by a specified container.

To achieve anomaly detection, we process the raw system call trace into a stream of frequency vectors, that is, for each sampling point, we calculate the occurrences of each system call type and record it as a frequency vector feature for detecting security attacks. To handle the diversity of system call types produced by different

Table 1: A frequency vector sample for the *Elasticsearch* application (CVE-2015-1427). An attack is triggered at t = 1586738324176. CDL raises alarms from t = 1586738324476.

Timestamp	System Call Frequency			
	futex	lseek	read	stat
1586738323776	8	0	361	0
1586738323876	6	0	349	0
1586738323976	8	0	344	0
1586738324076	6	0	309	0
1586738324176 (attack starts)	*12*	*0*	*297*	*0*
1586738324276	6	0	344	0
1586738324376	10	0	383	0
1586738324476 (attack detected by CDL)	**8**	**0**	**451**	**0**
1586738324576	**8**	**6**	**375**	**3**
1586738324676	**370**	**64**	**434**	**32**
1586738324776	**118**	**193**	**625**	**94**
1586738325876 (attack completes)	**24**	**76**	**378**	**35**

applications, we expand the frequency vector to the same dimension by including all existing Linux system call types.

Table 1 shows an example of a partial frequency vector for an attack exploiting the vulnerability CVE-2015-1427 in the Elasticsearch application. Each timestamp t is associated with a vector $V = [f_1, f_2, f_3, ..., f_k]$ where $f_i (i \in [1, k])$ is the occurrence count for system call type s_i during the last sampling period (100 milliseconds) starting at time t. For example, during the time period $[t, t + 100)$ milliseconds with t = 1586738323776 (shown by the first row in Table 1), the *futex* system call occurs 8 times, whereas *read* is called 361 times. The attack starts at t =1586738324176. We highlight those system calls with abnormal frequency changes after the attack starts. We notice significant frequency increases in `futex`, `lseek`, `read` and `stat` system calls. CDL starts to raise an alarm at t = 1586738324476 during an initial increase in the number of *read* calls. In the next sample, the containerized application starts to invoke the *stat* call. However, by time t = 1586738324676, the frequency of the *futex* call is over an order of magnitude higher than that before the attack starts.

It is also noteworthy that our extracted frequency vector trace is orders of magnitude smaller than the original system call trace in data size. For example, in our experiments, the average raw system call trace has an average size of 1.2 GiB while the extracted feature vector trace is only 4.8 MiB on average. During the following application classification and classified training/learning steps, we only need to process feature vectors without transmitting large raw system call traces over networks.

2.3 Application Classification

In order to create precise normal behavior models for different applications, it is important to distinguish different applications. However, we need to tackle a set of challenges to achieve the goal in production container environments. First, we cannot rely on human inputs to manually label each container since containers can be highly dynamic and a production system often consists of tens of thousands of containers. Second, we have to avoid intrusive monitoring tools which can bring large overhead to light-weight containers. Third, our application classification schemes need to be workload insensitive since the containers of the same application

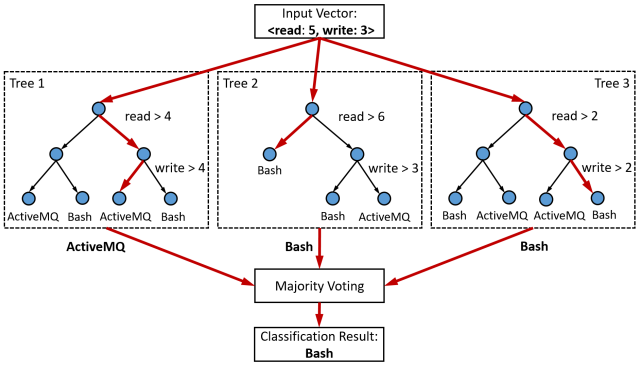


Figure 2: An example of random forest operation.

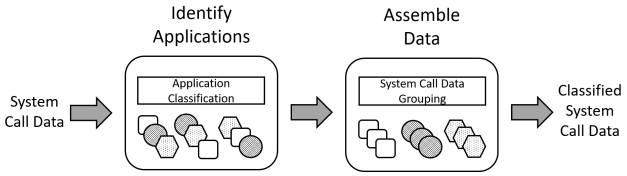


Figure 3: System call data grouping in the CDL system.

might process different workloads at different time. To this end, we leverage the *random forest* learning scheme [13] to achieve the goal shown by Figure 2. During our experiments, we observe that system call feature vectors extracted by our system call preprocessing scheme provide sufficient and workload insensitive patterns for us to distinguish different applications.

The random forest classifier uses a number of decision trees to improve accuracy while combating over-fitting. Each decision tree receives only a portion of the training data. Each decision tree makes local optimal splitting decisions among a random subset of the features when splitting nodes. As a result, the trained decision trees are usually quite different from one another. The output of the random forest classifier then uses the majority voting result among individual decision trees. For example, in Figure 2, the random forest model consists of three decision trees. Given a system call frequency vector, the first decision tree classifies the input data is from the `ActiveMQ` application; the second and the third decision trees classify the input data is from the `Bash` application. The random forest model will output `Bash` as the final application classification result.

2.4 System Call Data Grouping

The system call data grouping component handles the assembly of data from multiple containers into different application classes assigned by the application classifier. System call data grouping operates in conjunction with the application classifier as shown by Figure 3. Once the containers of the same application have been identified by the application classifier, their data are aggregated by concatenation, that is, the frequency vector traces of different containers described in section 2.2 are appended to one another for model training or attack detection. As all feature vectors are aligned

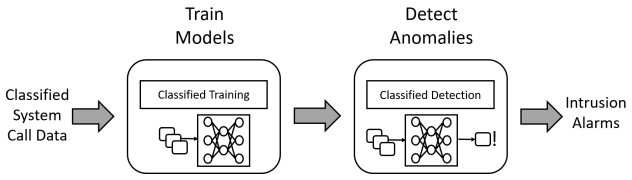


Figure 4: Classified learning and anomaly detection of CDL.

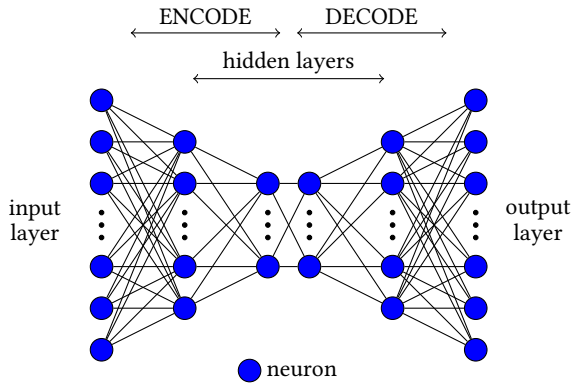


Figure 5: Architecture of the autoencoder.

with the same dimension, it is easy to concatenate the classified data from distributed containers.

Specifically, each application class has its own file with the feature vector data obtained from multiple containers of the same class. Each feature vector is appended to the file corresponding to the group predicted by the application classifier. The data grouping module performs periodical data segmentation and send the newest data segment to different application models for classified training or attack detection, which will be described next.

2.5 Classified Learning and Anomaly Detection

Our classified learning scheme creates and maintains a model ensemble consisting of different models for different application classes shown by Figure 4. In this work, we chose autoencoder neural networks for unsupervised model training and anomaly detection to meet our goal of achieving online security attack detection for containerized applications.

The autoencoder network consists of *encode* and *decode* regions shown by Figure 5. Input data are compressed in the *encode* region and reconstructed in the *decode region*. The model replicates its input data after compressing the input data through intermediate neural network layers. During the training phase, a model is built to minimize the difference between its input and output. When the autoencoder model is sufficiently trained, the model is able to produce an output data with small reconstruction error compared with the normal input data. We can then use the reconstruction error to implement anomaly detection. Specifically, when the autoencoder model produces an output with high reconstruction error, we can infer an abnormal input data is detected.

Our neural network consists of four layers of neurons with sigmoid activation in addition to the input and output layers. Each input data trains the autoencoder model for 10 iterations. We choose to use a small neural network with just four layers and a small number of training iterations in order to achieve online training for the container environment. We implement the autoencoder with Tensorflow.

During training, our objective is to minimize mean squared error (MSE). We perform backpropagation with the Tensorflow implementation of the root mean square propagation (*RMSProp*) optimizer, whose steps are executed with the following equations. Weight w is updated according to the formula:

$$w_t = w_{t-1} - \alpha * \frac{g_t}{\sqrt{rms_t + \epsilon}} \tag{1}$$

where α corresponds to learning rate, L is MSE loss, g is the gradient of the loss with respect to the weight ($\frac{dL}{dw}$), ρ and ϵ are small Tensorflow default constants, and the root mean square of the gradient rms is given by:

$$rms_t = \rho * rms_{t-1} + (1 - \rho) * g_t^2 \tag{2}$$

To determine proper error threshold for anomaly detection, we adopt a statistical approach. Assuming majority of input data are normal data during the training phase, we collect all the reconstruction errors produced by the training data and compute a high percentile value (e.g., 99.9 percentile) as the threshold. We refer to this percentile value as the value of the *training reconstruction error*. This percentile value selection represents the tradeoff between detection rate and false alarm rate. Intuitively, smaller threshold yields more detections but also more false alarms. We conduct experiments to illustrate such tradeoffs in Section 3.

3 EXPERIMENTAL EVALUATION

In this section, we present our experimental evaluation. We implement our prototype system and evaluate it on Amazon EC2 t3.large instances with two 2.5 GHz vCPUs and 8 GB memory running Ubuntu 16.04 64-bit.

3.1 Evaluation Methodology

In this subsection, we give details about our evaluation methodology as well as the alternative approaches that we compare our results against.

3.1.1 Real-world vulnerabilities. We investigate 33 recent real-world vulnerabilities documented in the Common Vulnerabilities and Exposures (CVE) database. These selected vulnerabilities appear in 24 open source software, varying from back-end to front-end applications, which covers different types of containerized applications commonly used in practice [3, 7]. All the studied vulnerabilities and their attributes are summarized in Table 2.

3.1.2 Experiment setup. As mentioned in Section 2, we collect system call data for each vulnerability in multiple containers for training and testing. We design different workload intensity levels that represent the normal operation of each application. To emulate real world workload variations in dynamic containers, we change the request rate by mutiplying a scale factor for each new container. Each container receives a multiple (i.e. 1x, 2x, 4x and 8x.)

Table 2: List of explored real-world vulnerabilities.

Threat Impact	CVE ID	CVSS Score	Application	Version
Return a shell and execute arbitrary code	CVE-2012-1823	7.5	PHP	5.4.1
	CVE-2014-3120	6.8	Elasticsearch	1.1.1
	CVE-2015-1427	7.5	Elasticsearch	1.4.2
	CVE-2015-2208	7.5	phpMoAdmin	1.1.2
	CVE-2015-3306	10.0	ProFTPd	1.3.5
	CVE-2015-8103	7.5	JBoss	6.1.0
	CVE-2016-3088	7.5	Apache ActiveMQ	5.11.1
	CVE-2016-9920	6.0	Roundcube	1.2.2
	CVE-2016-10033	7.5	PHPMailer	5.2.16
	CVE-2017-7494	10.0	Samba	4.5.9
	CVE-2017-8291	6.8	Ghostscript	9.2.1
	CVE-2017-11610	9.0	Supervisor	3.3.2
	CVE-2017-12149	7.5	JBoss	6.1.0
	CVE-2017-12615	6.8	Apache Tomcat	8.5.19
Execute arbitrary code	CVE-2014-6271	10.0	Bash	4.2.37
	CVE-2015-8562	7.5	Joomla	3.4.2
	CVE-2016-3714	10.0	ImageMagick	6.7.9
	CVE-2017-5638	10.0	Apache Struts 2	2.5
	CVE-2017-12794	4.3	Django	1.11.4
	CVE-2018-16509	9.3	Ghostscript	9.23
	CVE-2018-19475	6.8	Ghostscript	9.25
	CVE-2019-6116	6.8	Ghostscript	9.26
Disclose credential information	CVE-2014-0160	5.0	OpenSSL	1.0.1e
	CVE-2015-5531	5.0	Elasticsearch	1.6.0
	CVE-2017-7529	5.0	Nginx	1.13.2-1
	CVE-2017-8917	7.5	Joomla	3.7.0
	CVE-2018-15473	5.0	OpenSSH	7.7p1
	CVE-2020-1938	7.5	Apache Tomcat	9.0.30
Consume excessive CPU	CVE-2014-0050	7.5	Apache Commons FileUpload	1.3.1
	CVE-2016-6515	7.8	OpenSSH	7.2p2
Crash the application	CVE-2015-5477	7.8	BIND	9
	CVE-2016-7434	5.0	NTP	1.4.2.8
Escalate privilege level	CVE-2017-12635	10.0	CouchDB	2.1.0

of the workload. We use Apache Jmeter to deliver different kinds of workload. For example, we send HTTP GET requests to the web application containers with the PHP vulnerability (CVE-2012-1823). Whereas, application containers providing Network Time Protocol (NTP) services are sent current time requests. While containers are under the appropriate workload, we exploit their security vulnerabilities to investigate our attack detection model. We test four containers for each vulnerability. For each container, the experiments last a total time of seven minutes except in cases where the attack crashes the application. The experiment procedure is as follows. First, we run the application under normal workload conditions for four minutes. Thereafter, we trigger the attack and allow it to run until it succeeds. Finally, we exit the attack after it completes where applicable. Thus, with a sample time of 0.1 seconds, each container contributes about 2400 normal samples. The first minute of each container data is used for training the application classifier, the next two minutes are used for training autoencoder models, and the last four minutes are used for testing the trained models.

3.1.3 Application classification setup. We consider each unique application and version combination in Table 2 as a distinct application class. There are two instances, corresponding to CVE-2015-8103 and CVE-2017-12149 vulnerabilities, that share the same application and version number so we consider them to be the same application. We compile a list of 555 system calls that are used by current Linux kernels (as of version 4.19) as the frequency vector. This is done to cover all system calls encountered both in our experiments and in unobserved application environments CDL would operate on. For any new application, we first expand its frequency vector dimension to 555 by inserting zero values at the positions of system calls that are not used during the application run-time.

The expanded data are then fed into a random forest classifier.

We use the scikit-learn implementation of random forest [4]. In our experiment, every random forest classifier uses 200 decision tree classifiers with no specified maximum depth. To produce a stable output, we use a random state of zero. For each application, we train a random forest classifier using the first minute of each container instance of that application.

3.1.4 Autoencoder neural network setup. CDL adopts a small four layer neural network for each of its individual models. The first two layers form the encoder part of autoencoder while the third and fourth layers form the decoder as depicted in Figure 5. Our autoencoder prototype, implemented with Tensorflow, consists of 278 neurons in the first and fourth hidden layers and 70 in the second and third layers. We set learning rate to 0.001 and utilize the root mean square propagation (RMSProp) optimizer to minimize the mean squared error (MSE) loss function. Once we train an autoencoder model, we run anomaly detection on its training data to analyze its reconstruction errors for choosing a threshold. We select the values corresponding to the 99.9 percentile of those errors as the default *training reconstruction error* for detecting anomalies.

3.1.5 Alternative approaches. We evaluate our classified learning approach against two commonly used existing training methods.

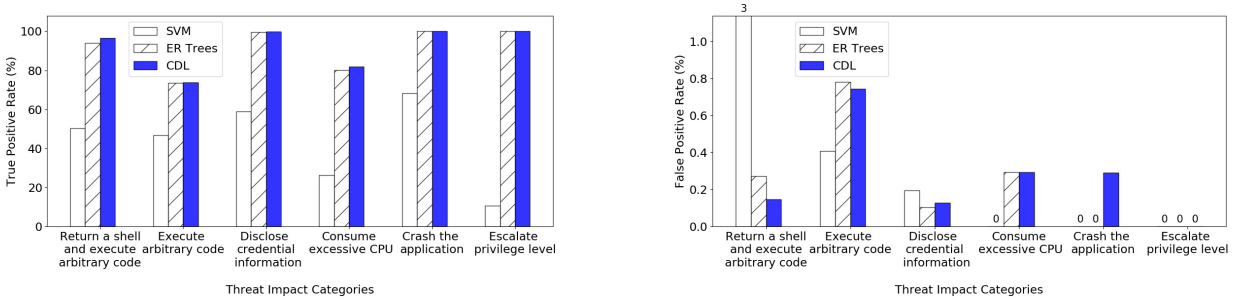
The sampling method: Without aggregating training data from multiple containers, each model is trained with a subset of data samples. We call this traditional learning method the sampling method. The drawback of this approach is that the captured portions of historical training data may not adequately represent all normal application behaviors.

The monolithic method: The monolithic approach combines data from all containers without distinguishing different applications. This approach assumes that all data would improve the model which may not necessarily be true. For instance, distinct applications that experience differing trends may interfere with one another during the training. This monolithic scenario represents the other extreme where an excessive amount of data is utilized blindly.

3.1.6 Evaluation metrics. We use true positive rate (TPR) and false positive rate (FPR) metrics to compare the detection results among different approaches. The calculation for these metrics are given by the following equations where TP denotes the number of samples that the detector correctly identifies while FP denotes the number of samples that the detector falsely identifies and TN denotes the number of the samples the detector correctly rejects while FN denotes the number of the samples the detector falsely rejects.

$$TPR = \frac{TP}{TP + FN} \tag{3}$$

$$FPR = \frac{FP}{FP + TN} \tag{4}$$



(a) True positive rate.

(b) False positive rate.

Figure 6: Classification results among different application classifiers.

Table 3: Classification results among different application classifiers.

Application Classifier	TPR	FPR
SVM	49.44%	1.60%
Extremely Randomized Trees	89.84%	0.33%
CDL	91.03%	0.29%

We also measure the time from when the attack is first detected to the time when the attack succeeds. We use call this time duration the *lead time*. The longer the lead time is, the more likely the attack can be stopped before it compromises the application.

3.2 Results Analysis

3.2.1 Application classification. We compare the random forest classifier used by CDL with extremely randomized trees (ER Trees) and support vector machine (SVM) classifiers. Similar to the random forest case, we use scikit-learn implementations of both extremely randomized trees and SVM with a random state of zero. Table 3 gives the overall true positive rate and false positive rate results of each classifier. In addition, we present the classifier performance across the different threat impact categories in Figure 6. Each classifier has to accurately classify system call data from various applications collected under different workload conditions.

SVM, which has only one classifier instance, finds this task difficult. It also generates the lowest true positive rate among all attack categories. In contrast, both random forest and extremely randomized trees are ensemble models which make use of the voting classification results of multiple decision trees. Thus, both CDL and extremely randomized trees achieve much higher true positive rate and much lower false positive rate than SVM. CDL achieves higher true positive rate and lower false positive rate than extremely randomized trees on average. CDL attains the highest true positive rate among all threat impact categories.

3.2.2 Classified detection. Table 4 shows the detection results over all vulnerability attacks used in our experiments. The results show that CDL can successfully detect 31 out of 33 attacks while the monolithic method can only detect 20 out of 33 attacks due to conflicting training data. Although the sampling method can detect

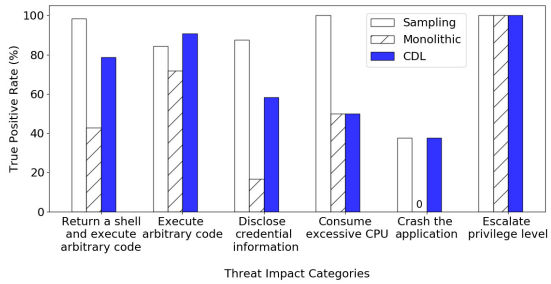
Table 4: Detection results of all CVE examined among different models.

Threat Impact	CVE ID	Detected		
		Sampling	Monolitic	CDL
Return a shell and execute arbitrary code	CVE-2012-1823	✓	✗	✓
	CVE-2014-3120	✓	✓	✓
	CVE-2015-1427	✓	✓	✓
	CVE-2015-2208	✓	✓	✓
	CVE-2015-3306	✓	✗	✓
	CVE-2015-8103	✓	✓	✓
	CVE-2016-3088	✓	✗	✓
	CVE-2016-9920	✓	✓	✓
	CVE-2016-10033	✓	✓	✓
	CVE-2017-7494	✓	✓	✗
	CVE-2017-8291	✓	✗	✓
	CVE-2017-11610	✓	✗	✓
	CVE-2017-12149	✓	✓	✓
	CVE-2017-12615	✓	✓	✓
Execute arbitrary code	CVE-2014-6271	✓	✗	✓
	CVE-2015-8562	✗	✓	✓
	CVE-2016-3714	✓	✓	✓
	CVE-2017-5638	✓	✓	✓
	CVE-2017-12794	✓	✓	✓
	CVE-2018-16509	✓	✓	✓
	CVE-2018-19475	✓	✓	✓
	CVE-2019-6116	✓	✓	✓
Disclose credential information	CVE-2014-0160	✓	✗	✓
	CVE-2015-5531	✓	✓	✓
	CVE-2017-7529	✓	✗	✓
	CVE-2017-8917	✓	✗	✓
	CVE-2018-15473	✓	✗	✗
	CVE-2020-1938	✓	✗	✓
Consume excessive CPU	CVE-2014-0050	✓	✓	✓
	CVE-2016-6515	✓	✓	✓
Crash the application	CVE-2015-5477	✓	✗	✓
	CVE-2016-7434	✓	✗	✓
Escalate privilege level	CVE-2017-12635	✓	✓	✓
Total successful detection		32	20	31

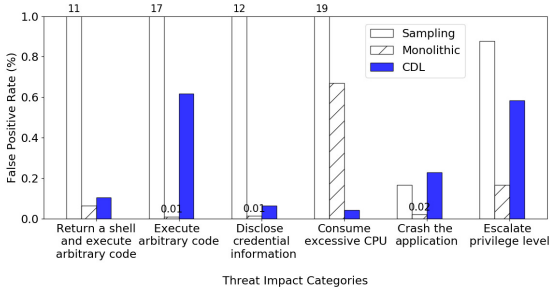
Table 5: Summary of detection results among different models.

Model	TPR	FPR	Lead time
Sampling	89.39%	12.74%	12.16s
Monolithic	44.70%	0.08%	5.77s
CDL	74.24%	0.24%	9.23s

32 out of 33 attacks, it produces orders of magnitude higher false positives than CDL, shown by Table 5. We also varied the percentile threshold in the autoencoder model and show the results. The results using 99 percentile of training reconstruction error are listed

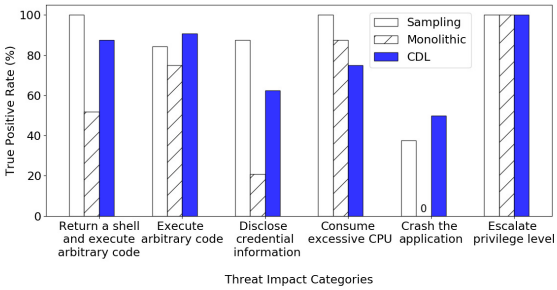


(a) True positive rate.

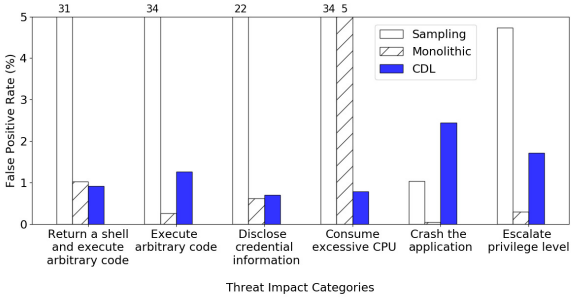


(b) False position rate.

Figure 7: Detection results among different models with 99.9 percentile of training reconstruction error.

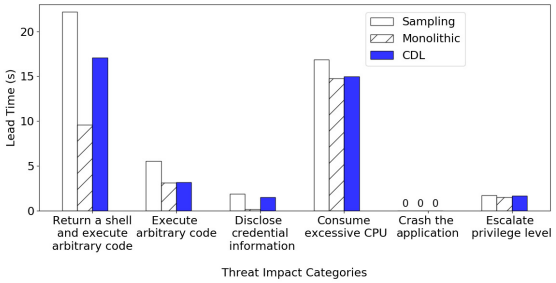


(a) True positive rate.

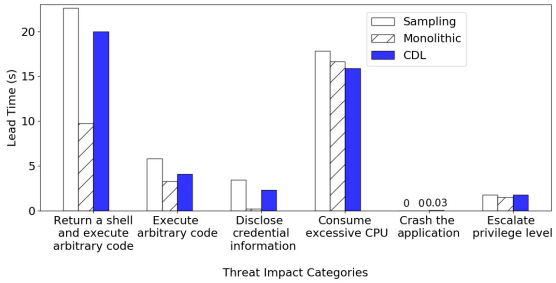


(b) False positive rate.

Figure 8: Detection results among different models with 99 percentile of training reconstruction error.



(a) Lead time with 99.9 percentile of training reconstruction error.



(b) Lead time with 99 percentile of training reconstruction error.

Figure 9: Comparisons of lead time among different models.

Table 6: Detection results among different models with 99 percentile of training reconstruction error.

Model	TPR	FPR	Lead time
Sampling	90.15%	27.88%	12.77s
Monolithic	52.27%	0.93%	6.01s
CDL	81.06%	1.07%	10.91s

in Table 6. We observe that CDL can achieve both high true positive rate and low false positive rate.

Figure 7 and Figure 8 illustrate the detection results across all attack threat impact categories. CDL models achieve the highest true positive rate in detecting “execute arbitrary code” attacks and the lowest false positive rate in detecting “disclose credential information” and “consume excessive CPU" attacks. Due to insufficient training, sampling models usually do not have enough knowledge of normal behavior. Thus, they under-fit the data and have high

Table 7: CDL system run time measurements.

System Modules	Time per sample
Classifier training	135.35±0.15 ms
Autoencoder training	2.53±0.01 ms
Application classification	3.80±0.04 ms
Attack detection	3.50±0.06 ms

true positive rate and high false positive rate at the same time. The monolithic model is trained using all the training data from all the different applications. Thus, it over-fits the data and has low true positive rate and low false positive rate. CDL achieves good trade-off between true positive rate and false positive rate. In terms of true positive rate, CDL performs well in three categories: "return a shell and execute arbitary code", "execute arbitrary code" and "escalate privilege level", but less accurately in the remaining three categories: "disclose credential information", "consume excessive CPU" and "crash the application". We discuss in the next section, Section 3.3, our observation that attacks in those three categories usually result in significant changes to the top frequent system calls, while the rest three categories usually happen suddenly with little deviation in system calls.

The lead time comparison between two different percentile of training reconstruction error values can be found in Figure 9 as well as in Table 5 and Table 6. Higher percentile of training reconstruction error generally leads to lower lead time. However, the lead time result of the "escalate privilege level" category is not sensitive to the percentile of training reconstruction error used. CDL models have much larger lead time than the monolithic models in almost all cases except in the "execute arbitrary code" attacks. Sampling models generally have the largest lead time among all models. Note, however, that sampling models have much higher false positive rate which makes them unsuitable to apply in practice.

3.2.3 System Run Time Analysis. Lastly, we evaluate the run time for different operations of the CDL system. Table 7 shows the time per sample for key steps in our system. Classifier training is the most time consuming step and it needs about 135.35 ms to train each sample. Autoencoder training takes about 2.53 ms for training each sample. Application classification takes about 3.8 ms to complete and attack detection by autoencoder model takes about 3.5 ms for each sample. Recall that each sample covers a sampling period of 100 ms, so CDL can detect attacks in real time. Overall, the run time analysis show that CDL is lightweight and applicable for detecting attacks in real time under real world settings.

3.3 Case Study

In this subsection, we discuss an example in detail to show how CDL's identification results help to better detect attacks. Thereafter, we highlight an attack from each of the other categories.

3.3.1 Execute arbitrary code. We investigate an attack to the Joomla CVE-2015-8562 exposure. The attack takes advantage of a vulnerability in the HTTP User-Agent field of a request that allows PHP object injection. In particular, the attack we trigger causes a deserialization error. We study the alarms in the containers detected by CDL but not by sampling and monolithic models by examining the average system call frequencies in the normal period to compare with that of the attack period. In all containers, the top frequent system calls during the alarm period are the following: *fstat*, *lstat*, *access*, *close* and *open*. These are also the system calls with the most-changed average counts between the normal and the attack period. *Fstat* and *lstat* are responsible for retrieving file status information like file owner ID, file size or time of last file access. *Access* checks whether the calling process can access a specified file. *Open* readies a requested file, while *close* discontinues the use of a file descriptor. These calls are very relevant to this Joomla attack that accesses cookies secured in a MySQL database. Attempts are also made to convert the file content from its serial format into data structures in memory.

3.3.2 Return a shell and execute arbitrary code. Here, we highlight the CVE-2016-9920 vulnerability of the Roundcube mail application. This exposure allows custom parameters to be accepted in the mail fields of the application. This allows an attacker to insert commands that execute and can return a shell. The top system calls made by the container while under attack are *read*, *stat*, *close*, *open*, *write* and *mmap*. The attack triggers Roundcube to save a PHP shell command into a file that will be run. Thus, *stat* obtains file information that *write* can use to record the crafted command. When the file contents are invoked to run, they are mapped into memory with *mmap*. Finally, *close* and *open* calls are made to use file descriptors as needed.

3.3.3 Disclose credential information. Next, we discuss the Heartbleed bug (CVE-2014-0160) of the OpenSSL library. The bug allows a malicious user to access protected information outside an assigned memory buffer when receiving SSL Heartbeat responses. We record the following top system calls during the attack: *gettimeofday*, *stat*, *poll*, *writev*, *close*, and *open*. The attack incorporates a timeout mechanism while waiting for a server response to confirm whether the server is vulnerable. This is responsible for the *gettimeofday* calls, which significantly outnumber the other system calls. *Stat* and *writev* calls retrieve file information and write from multiple buffers respectively, to prepare Heartbeat response messages. Meanwhile, *poll* waits for the above files to ready their I/O data. *Open* and *close* calls are also notably involved in managing the life cycles of the files used.

3.3.4 Consume excessive CPU. In CVE-2016-6515, OpenSSH before version 7.3 does not limit the length for passwords, which can result in a denial-of-service attack (DoS attack) by a long string. The top frequent system calls during the alarm period are: *close*, *read*, *mmap*, *open*, *mprotect* and *fstat*. Those system calls are also the top frequent system calls before triggering the attack, but we observe the frequency of each system calls increase by at most 5 times. Processing password information is sensitive. Thus, we deduce that *mmap* maps related disk contents into memory while *mprotect* ensures proper access restrictions. The remaining system calls get related file information, read the data and then close the file descriptor when done. The infinite loop triggered by the attack would likely cause these tasks to occur repeatedly.

3.3.5 Crash the application. According to CVE-2016-7434, NTP suffers from a null pointer reference which could lead to crashing. Because of the nature of crashing, the attack period usually lasts

less than 0.3 second. The top frequent system calls during the alarm period are: *gettid*, *rt_sigprocmask*, *read*, *write*, *clock_gettime* and *recvmsg*. We believe that the attack happens in such a short time, that there is not a significant change in system call composition. Nevertheless, the NTP attack is triggered upon receiving a malicious most recently used list (mrulist) query. *Gettid* may be used to obtain the thread ID of the request delivered by *recvmsg* over the socket connection. The mrulist then needs to be read and processed by *read* and *write* calls. Meanwhile, *clock_gettime* would correspond to expected NTP workload to acquire time information.

3.3.6 Escalate privilege level. We analyze the CouchDB CVE-2017-12635 vulnerability. Because of different ways of parsing JSON objects by JavaScript and Erlang, this vulnerability can be used to gain access to create an administrator account. The top frequent system calls during the alarm period are: *close*, *epoll_wait*, *sched_yield*, *futex*, and *switch*. We notice that the appearance of system call *close* is significantly larger than that of any other system calls. The reason is that the attack changes the behavior of the application, so that it deletes file descriptors more often. The attack is administered with multiple HTTP requests with the close option set in the *Connection* field. The *close* system call will be useful for closing unneeded files related to those packets.

4 RELATED WORK

In this section, we compare our work with closely related work.

Container Vulnerability. The security of containers has attracted the attention of a lot of researchers in recent years. Zerouali et al. [33] find that among 7,380 studied official and community Docker images, every release is vulnerable. Docker image vulnerability analysis (DIVA) [24] is a scalable framework for discovering, downloading and analyzing both official and community Docker images. DIVA shows that both official and community Docker images contain more than 180 vulnerabilities on average. Tunde-Onadele et al. [29] compare multiple detection schemes and suggest that dynamic detection outperforms static vulnerability scanning for containers. By combing static and dynamic detection schemes, the detection rate can be further improved. Martin et al. [22] identify, in the different components of the Docker ecosystem, several vulnerabilities and detailed real world exploitation scenarios. They also propose possible fixes and discuss the adoption of Docker by platform-as-a-service (PaaS) providers. Lin et al. [18] study 11 exploits that can successfully bypass the isolation provided by the container to achieve privilege escalation. The authors then propose a defense mechanism to defeat those identified privilege escalation exploits. These studies emphasize the current vulnerable state of the container environment. Thus, there is a strong need for effective methods of detecting security attacks with a system like CDL.

Anomaly Detection. Abed et al. [5] apply the bag of system calls technique to detect anomalies in containers. This processes a system call trace into vectors in intervals of the same total count. Yolacan et al. [32] propose a process trace clustering approach using multi-hidden Markov models (HMM) to detect system call anomalies. Maggi et al. [20] combine clustering and a behavioral Markov model to build an unsupervised host-based intrusion detection system based on system call arguments and sequences analysis. Geng et al. [11] improve the efficiency of the sequence time delay embedding (STIDE) algorithm by only considering system call sequences that contain axis system calls. These axis system calls could more effectively represent the characteristics of normal behaviors with low overhead. Deep learning models have been recently explored for anomaly detection. Greenhouse [15] is designed as a zero-positive machine learning system which does not require any anomalous sample using long short-term memory (LSTM) method. Malhotra et al. [21] present stacked LSTM networks for detecting anomalies in several time series datasets. Taylor et al. [27] apply LSTM to detect anomalies in a car's controller area network (CAN) with low false alarm rate for catching possible intrusion to CAN. Sakurada et al. [23] propose to use autoencoders with nonlinear dimensionality reduction for general anomaly detection.

In comparison to existing anomaly detection schemes, CDL focuses on addressing special challenges of insufficient training data in container environments. CDL proposes a new classified distributed learning framework which is orthogonal to specific machine learning algorithms used for anomaly detection. Although CDL currently employs the autoencoder anomaly detection algorithm, it can be easily applied to other anomaly detection algorithms.

Federated Learning. Konečnỳ et al. [14] propose a decentralized approach to learning a shared centralized model, called *federated learning*. This is executed by aggregating local-computed updates from a large number of clients over an unreliable network. Later, Lin et al. [19] design deep gradient compression (DGC) to reduce the communication bandwidth of federated learning. Yao et al. [31] further improve federated learning by aggregating features from both the local and global models to achieve higher accuracy with less communication cost. Sozinov et al. [26] compare centralized training with federated learning and show that federated learning can achieve acceptable accuracy similar to centralized learning. CDL implements distributed learning using aggregated training data in a similar way as federated learning. However, CDL incorporates application classification into distributed learning to overcome the challenge of detecting security attacks in dynamic container systems.

Distributed Machine Learning. Hashdoop [9] improves the detection accuracy of network traffic anomaly detectors on Hadoop. They achieve this by carefully splitting network traffic such that the sampled traffic maintains its original structure. Song et al. [25] provide a parallel k-medoids clustering algorithm for high accuracy and efficiency. Chen et al. [8] provide a robust model training system which is orders of magnitude faster than alternate median-based approaches. Gopal et al. [12] achieve an order of magnitude decrease in training time through a parallel calculation of the likelihood function in logistic models. Petuum [30] provides a unified parallel optimization framework to help machine learning (ML) programs run faster. Similar to the above approaches, CDL promotes the distributed learning approach. However, CDL differs from the above work by focusing on improving the accuracy of security attack detection using input data from similar applications to create lightweight application-specific models with low training cost.

5 CONCLUSION

In this paper, we have presented CDL, a new classified distributed learning framework that aims at achieving practical and efficient security attack detection for containerized applications. CDL integrates online application classification and application-specific anomaly detection models to overcome the challenges of lacking sufficient training data for individual short-lived containers. We have implemented a prototype of CDL and conducted experiments over 33 real world vulnerability exploits in 24 commonly used applications. Our results show that CDL can reduce false positive rates from over 12% to 0.24% compared to traditional learning methods without aggregating training data from different containers and increase the true positive rate from 45% to 74% compared to simple training data aggregation without performing application classifications. CDL supports real time security attack detection, which makes it practical for production computing environments.

ACKNOWLEDGMENTS

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] 2016. Secure DevOps platform for cloud native. https://www.sysdig.com
[2] 2017. Docker image vulnerability research. https://www.federacy.com/docker_image_vulnerabilities
[3] 2018. 8 surprising facts about real Docker adoption. https://datadoghq.com/docker-adoption
[4] 2019. Random forest classifier. https://scikit-learn.org
[5] Amr S Abed, T Charles Clancy, and David S Levy. 2015. Applying bag of system calls for anomalous behavior detection of applications in Linux containers. In *2015 IEEE Globecom Workshops (GC Wkshps)*. IEEE, 1–5.
[6] Anthony Bettini. 2015. Vulnerability exploitation in Docker container environments. *FlawCheck, Black Hat Europe* (2015).
[7] Eric Carter. 2018. 2018 Docker usage report. https://sysdig.com/blog/2018-docker-usage-report
[8] Lingjiao Chen, Hongyi Wang, Zachary Charles, and Dimitris Papailiopoulos. 2018. DRACO: Byzantine-resilient Distributed Training via Redundant Gradients. In *International Conference on Machine Learning*. 902–911.
[9] Romain Fontugne, Johan Mazel, and Kensuke Fukuda. 2014. Hashdoop: A MapReduce framework for network anomaly detection. In *Computer Communications Workshops (INFOCOM WKSHPS), 2014 IEEE Conference on*. IEEE, 494–499.
[10] Stephanie Forrest, Steven A Hofmeyr, Anil Somayaji, and Thomas A Longstaff. 1996. A sense of self for UNIX processes. In *Symposium on Security and Privacy*. IEEE, 120–128.
[11] Li-zhong Geng and Hui-bo Jia. 2009. A low-cost method to intrusion detection system using sequences of system calls. In *2009 Second International Conference on Information and Computing Science*, Vol. 1. IEEE, 143–146.
[12] Siddharth Gopal and Yiming Yang. 2013. Distributed training of large-scale logistic models. In *International Conference on Machine Learning*. 289–297.
[13] Tin Kam Ho. 1995. Random decision forests. In *Proceedings of 3rd international conference on document analysis and recognition*, Vol. 1. IEEE, 278–282.
[14] Jakub Konečnỳ, H Brendan McMahan, Felix X Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. 2016. Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492* (2016).
[15] Tae Jun Lee, Justin Gottschlich, Nesime Tatbul, Eric Metcalf, and Stan Zdonik. 2018. Greenhouse: a zero-positive machine learning system for time-series anomaly detection. *arXiv preprint arXiv:1801.03168* (2018).
[16] Wei Li. 2004. Using genetic algorithm for network intrusion detection. *Proceedings of the United States department of energy cyber security group* 1 (2004), 1–8.
[17] Hung-Jen Liao, Chun-Hung Richard Lin, Ying-Chih Lin, and Kuang-Yuan Tung. 2013. Intrusion detection system: A comprehensive review. *Journal of Network and Computer Applications* 36, 1 (2013), 16–24.
[18] Xin Lin, Lingguang Lei, Yuewu Wang, Jiwu Jing, Kun Sun, and Quan Zhou. 2018. A measurement study on Linux container security: Attacks and countermeasures. In *Proceedings of the 34th Annual Computer Security Applications Conference*. 418–429.
[19] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. 2017. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887* (2017).
[20] Federico Maggi, Matteo Matteucci, and Stefano Zanero. 2008. Detecting intrusions through system call sequence and argument analysis. *IEEE Transactions on Dependable and Secure Computing* 7, 4 (2008), 381–395.
[21] Pankaj Malhotra, Lovekesh Vig, Gautam Shroff, and Puneet Agarwal. 2015. Long short term memory networks for anomaly detection in time series. In *Proceedings*, Vol. 89. Presses universitaires de Louvain.
[22] Antony Martin, Simone Raponi, Théo Combe, and Roberto Di Pietro. 2018. Docker ecosystem–vulnerability analysis. *Computer Communications* 122 (2018), 30–43.
[23] Mayu Sakurada and Takehisa Yairi. 2014. Anomaly detection using autoencoders with nonlinear dimensionality reduction. In *Proceedings of the MLSDA 2014 2nd Workshop on Machine Learning for Sensory Data Analysis*. ACM, 4.
[24] Rui Shu, Xiaohui Gu, and William Enck. 2017. A study of security vulnerabilities on Docker hub. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. 269–280.
[25] Hwanjun Song, Jae-Gil Lee, and Wook-Shin Han. 2017. PAMAE: parallel k-medoids clustering with high accuracy and efficiency. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 1087–1096.
[26] Konstantin Sozinov, Vladimir Vlassov, and Sarunas Girdzijauskas. 2018. Human activity recognition using federated learning. In *2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*. IEEE, 1103–1111.
[27] Adrian Taylor, Sylvain Leblanc, and Nathalie Japkowicz. 2016. Anomaly detection in automobile control network data with long short-term memory networks. In *2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*. IEEE, 130–139.
[28] Benjamin Berry Thompson, Robert J Marks, Jai J Choi, Mohamed A El-Sharkawi, Ming-Yuh Huang, and Carl Bunje. 2002. Implicit learning in autoencoder novelty assessment. In *Proceedings of the 2002 International Joint Conference on Neural Networks. IJCNN'02 (Cat. No. 02CH37290)*, Vol. 3. IEEE, 2878–2883.
[29] Olufogorehan Tunde-Onadele, Jingzhu He, Ting Dai, and Xiaohui Gu. 2019. A Study on Container Vulnerability Exploit Detection. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 121–127.
[30] Eric P Xing, Qirong Ho, Wei Dai, Jin Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. 2015. Petuum: A new platform for distributed machine learning on big data. *IEEE Transactions on Big Data* 1, 2 (2015), 49–67.
[31] Xin Yao, Tianchi Huang, Chenglei Wu, Ruixiao Zhang, and Lifeng Sun. 2019. Towards faster and better federated learning: A feature fusion approach. In *2019 IEEE International Conference on Image Processing (ICIP)*. IEEE, 175–179.
[32] Esra N Yolacan, Jennifer G Dy, and David R Kaeli. 2014. System call anomaly detection using multi-hmms. In *2014 IEEE Eighth International Conference on Software Security and Reliability-Companion*. IEEE, 25–30.
[33] Ahmed Zerouali, Tom Mens, Gregorio Robles, and Jesus M Gonzalez-Barahona. 2019. On the relation between outdated Docker containers, severity vulnerabilities, and bugs. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 491–501.