Received: 29 July 2019 | Revised: 22 November 2019 | Accepted: 22 November 2019

DOI: 10.1002/eng2.12080

RESEARCH ARTICLE

WILEY

KubAnomaly: Anomaly detection for the Docker orchestration platform with neural network approaches

Chin-Wei Tien[1] | **Tse-Yung Huang**[1] | **Chia-Wei Tien**[1] | **Ting-Chun Huang**[1] | **Sy-Yen Kuo**[2]

[1]Cybersecurity Technology Institute, Institute for Information Industry, Taipei, Taiwan, ROC

[2]Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan, ROC

Correspondence
Chin-Wei Tien, Cybersecurity Technology Institute, Institute for Information Industry, Taipei, Taiwan ROC.
Email: jakarence@iii.org.tw

Funding information
Ministry of Economic Affairs, 108-EC-17-A-24-1102

Kubernetes, which is the most popular orchestration platform for Docker containers, is used widely for developing microservices and automating Docker instance life cycle administration. Because of advancements in containerization technology, a single server can run multiple services and use hardware resources more efficiently. However, containerized environments also bring new challenges in terms of complete monitoring and security provision. Thus, hackers can exploit the security vulnerabilities of containers to gain remote control permissions and cause extensive damage to company assets. Therefore, in this study, we propose KubAnomaly, a system that provides security monitoring capabilities for anomaly detection on the Kubernetes orchestration platform. We develop a container monitoring module for Kubernetes and implement neural network approaches to create classification models that strengthen its ability to find abnormal behaviors such as web service attacks and common vulnerabilities and exposures attacks. We use three types of datasets to evaluate our system, including privately collected and publicly available datasets as well as real-world experiment data. Furthermore, we demonstrate the effectiveness of KubAnomaly by comparing its accuracy with that of other machine learning algorithms. KubAnomaly is shown to achieve an overall accuracy of up to 96% for anomaly detection. It successfully identifies four real attacks carried out by hackers in September 2018. Moreover, its performance overhead is only 5% greater than that of current methods. In summary, KubAnomaly significantly improves container security by avoiding anomaly attacks.

KEYWORDS
anomaly detection, cloud security, container orchestration, container security, machine learning, neural network

1 | INTRODUCTION

Kubernetes has emerged as the most popular orchestration platform for automatic deployment, expansion, and management of the Docker container life cycle. It has been used in microservice architectures, such as the Internet of Things (IoT),

cloud computing, and AI workflows. Moreover, platform-as-a-service suppliers, such as AWS and Azure, also provide Kubernetes as the main container orchestration environment for users.

The rise of the 5G communication scheme has led to the widespread use of the virtualization technology for efficient service deployment. Container virtualization is the most used widely solution for application packaging and runtime execution. The popularity of containerized applications has been driven by their improved performance relative to traditional virtual machines. In general, application containers can help pack all the system libraries, files, and code required to support the target environment. Using containers, programmers can deploy their products more efficiently and manage them easily. In particular, the overall performance of Docker containers is nine times faster than that of a virtual machine environment.[1] Because of the popularity of microservice architectures,[2] container virtualization technology is used widely in the agile software deployment environment. Docker continues to dominate the container virtualization landscape; a recent container adoption survey reported that more than 79% of companies use Docker as their primary container platform. According to DockerConf 17,18,[3,4] more than 900,000 apps have been developed by the Docker community.

From the viewpoint of cybersecurity, new technologies bring new cybersecurity challenges. In *Blackhat 2017*,[5] the researchers demonstrated the potential for abuse of the Docker API in terms of remote code execution, host rebinding, and shadow container attacks. Such attacks, called Shadow Worms, pollute the image, registry, and containers. These threats can cause severe damage to company assets. In September 2017, the National Institute of Standards and Technology (NIST) released Special Publication (SP)-800-190,[6] an application container security guide to explain the potential security concerns and recommendations associated with the use of containers, defined five significant risks for the core components of container technologies shown in Figure 1: image risks, registry risks, orchestrator risks, container risks, and host OS risks. Image risks include the use of vulnerable libraries, improper configuration settings, and embedded malware. Three registry risks are defined by the document, including insecure connections to the registry, out-of-date images, and restricted access to the registry through insufficient authentication and authorization. Administrative access, unauthorized access, and intercontainer network settings comprise significant orchestrator risks. Container risks are vulnerabilities within the runtime software itself, such as insecure container runtime configurations, app vulnerabilities, and insider attacks. Finally, host OS risks include concerns such as improper user access rights, shared kernel access, and OS component vulnerabilities.

According to the security risks defined in NIST SP-800-190, most of the security solutions are suitable for image pretest as well as for network and operating system layer detection in the host. For protection against network attacks, we can install IDS, IPS, and web application firewalls in the network layer. In the operating system layer, we can install antivirus software and host IDS for protection against malware attacks. As there are multiple services on a single host in the runtime container environment, internal activities or anomalous behavior for each container cannot be tracked by previous security solutions; only external communication attacks may be detected. This limits the ability to identify containers that are subject to hacking attacks.

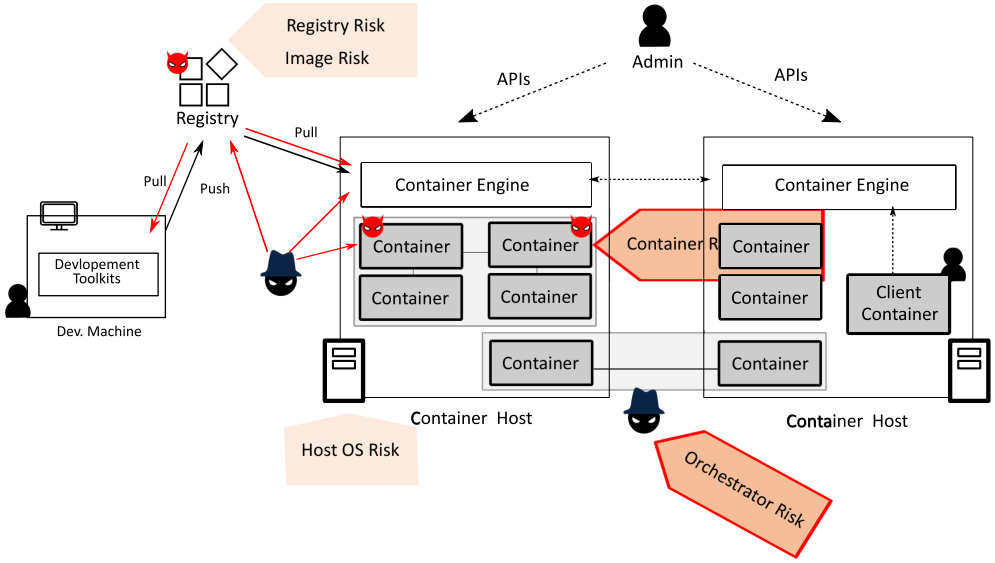


FIGURE 1 National Institute of Standards and Technology (NIST) Special Publication (SP)-800-190 lists the potential security concerns in the container environment, defining five major risks for the core components of container technologies: image risks, registry risks, orchestrator risks, container risks, and host OS risks

The above-mentioned discussion implies that two important aspects, namely orchestrator and container runtime, require security mechanisms to improve container security. Thus, to bridge this security gap in the container environment, we develop a security system called KubAnomaly, which aims to improve Docker security and is compatible with Kubernetes. KubAnomaly provides a security-monitoring module that can be installed on the Kubernetes container orchestration platform, and it observes the internal activities of containers, system calls, I/O activities, and network connections. Moreover, it performs anomaly detection using machine learning classification to identify hackers and insider intrusion events.

The major contributions of this study can be summarized as follows:

- The proposed anomaly detection model can detect the behavior of on-site containers in Kubernetes. It is positioned as the first line of protection to quickly identify anomalies. We built a monitoring module with customized rules in Sysdig[7] inside the container kernel to strengthen the security visibility. This module can be used to observe the four categories of system calls in containers, namely, file I/O, network I/O, scheduler, and memory. A neural network model was created to classify multiple types of anomalous behavior, such as injection attacks and denial-of-service (DoS) attacks.
- We used three different datasets to evaluate the proposed model and publish the repository: private data, public data (CERT), and real-world experimental data. We tested and evaluated KubAnomaly with different machine learning algorithms and public system data. The results showed that the proposed anomaly detection model can successfully detect web service attacks with an overall accuracy up to 96%.
- In the real-world experiment, we set a decoy container with some vulnerabilities. KubAnomaly discovered over 40 anomalous events since September 2018, which originated from different countries such as China and Thailand. Thus, we verified that the proposed model can detect such events efficiently and correctly with only 5% performance overhead. All the anomalous events are recorded in the KubAnomaly portal, along with the time of occurrence and source IP.

The remainder of this article is organized as follows. Section 2 reviews related studies. Section 3 describes the details of the classification model that we used to detect anomalous behaviors as well as the details of the KubAnomaly system implementation. Section 4 evaluates our defense framework from different aspects. Section 5 discusses the limitations of our work and presents a real-world case analysis. Finally, Section 6 concludes the paper.

2 | RELATED WORK

We reviewed the existing studies and focused on anomaly detection in the system and network. These two areas correspond to the orchestration and container runtime risks detailed in NIST SP-800-190. The solutions include log-based anomaly detection, virtual network monitoring, and container security. Herein, we highlight the differences between our study and previous related studies.

Log-based anomaly detection: Some studies have investigated application or system log analysis and detection. Xu et al[8] parsed console logs via source code analysis and information retrieval to create composite features for automatically detecting a system runtime problem. Lou et al[9] developed a system problem detection method that uses log message grouping and counting as well as the inherent linear characteristics of normal program workflows to detect anomalies in logs. These studies focused on detecting execution anomalies using the console log in the system. Such anomalies include service message lost and failure to close channel. However, they could not detect the anomaly behavior caused by external attacks, such as web service attacks and common vulnerabilities and exposures (CVE) attacks. According to these methods, we cannot obtain the program source code to parse the log schema in the runtime container environment. The attack is a series of operation behaviors, and the behavior patterns are not necessarily composed of the same request ID groups; hence, they cannot be effectively detected during the attacks.

Virtual network monitoring: Several studies have attempted to bridge the security gap in virtual network environments. Lin et al[10] developed a network intrusion detection system for cloud virtual networks that can monitor network intrusion events occurring between virtual machines. Gomez[11] proposed an infrastructure for container network monitoring that can be accommodated in a virtual network of containers. These studies focused only on network layer detection. In some cases, the behavior appears as normal login and install behavior through the network; however, it actually installs a malicious program that makes the virtual environment a botnet or damages the host. Such system behaviors

can avoid detection through the network monitoring mechanism, which, thus, cannot completely protect the virtual environment.

Container security: In the orchestrator and container runtime risk domains, there are open-source and commercial products that can protect the container environment. SANS[12] introduced several tools for container security. For example, AppArmor[13] is a policy-based Linux kernel security module that allows system administrators to restrict process's capabilities, such as network access or file read/write permissions, through their own security profiles. It can be used with runtime containers, and the user can use the default security profile or a customized one to optimize the container OS instance. Sysdig[7] provides system call observation for the entire system, individual applications, and containers. Falco[14] is a Sysdig project for behavior activity monitoring, that is, for detecting anomaly activity in applications. Users can design the rules to detect the runtime container and identify suspicious events immediately. Twistlock[15] also uses customized rules to detect the system behavior, and it is integrated with AppArmor to monitor anomalous behavior through policy-based mechanisms. Aqua Security[16] provides comprehensive security detection and protection solutions to enhance the security of the orchestrator and container runtime. Thus far, most commercial and open-source products support policy-based detection in the orchestrator and container runtime. Moreover, because of the rapid change in attack patterns, new attacks can easily circumvent the rules of the detection mechanism, causing the containers to become botnets.

Anomaly-based detection mechanisms have been introduced in recent studies. Gao et al[17] proposed solutions to detect the information leakage channel in the container environment. Thus, insider threats in the cloud container host can be detected; however, such threats include only attacks related to information leakage. Other attack types, such as web service attacks, cannot be detected effectively. Du et al[18] analyzed real-time performance data, including CPU, memory, and network metrics, via machine learning for each node, pod, and container in Kubernetes to detect and diagnose anomalies in container-based services. They simulated CPU fault, memory fault, network latency, and package loss in the fault injection module to validate the detection accuracy. Zou et al[19] used three categories of data, namely, container log, container information (CPU, memory, etc), and container management information to analyze the anomaly status in containers on the basis of an optimized isolation forest algorithm. They defined the anomaly status in four fields, namely CPU, memory, I/O, and network. Examples include CPU spin lock, memory overflow, and network congestion. Both studies used similar runtime container information to detect different anomalies. Such features do not fluctuate significantly when detecting cyberattacks.

The products and studies related to container security are summarized in Table 1. The products in the market are mainly for policy- and network-based detections in containers and orchestrators. They can effectively detect known attacks but not new ones. The studies on containers focus on information leakage and performance anomaly detection using different container characteristics. They are not adapted to detect diverse cyberattacks and lack practical experiments for verifying the effectiveness of their solutions.

To solve the runtime container security problem, we develop a novel monitoring module based on customized rules in Kubernetes. Our system implements neural network approaches to perform anomaly behavior classification, which strengthens the ability to find anomalous behaviors, such as web service attacks and CVE attacks. We validate the system using three different datasets, including a real-world attack, and we confirm the anomaly behavior classification

TABLE 1 Feature comparison of related studies

Risks:		**Container runtime risk**		
Related researches	**Orchestrator risk**	**Network-based**	**Policy-based**	**Anomaly-based**
AppArmor[13]	v		v	
Sysdig[7] and Falco[14]	v	v	v	
Twistlock[15]	v	v	v	
Aqua[16]	v	v	v	
Gao et al[17]			v	
Du et al[18]				v
Zou et al[19]				v
KubAnomaly	v	v		v

performance and detection accuracy. We also develop a portal to visualize the runtime container status and record the history of anomaly events to fully present the detection information.

3 | PROPOSED METHOD

We implement a system that can monitor and detect anomaly behavior at container runtime. Furthermore, we adapt our system to Kubernetes, a popular orchestration platform. There are some design issues that need to be addressed. These design issues include how to monitor and collect logs from the container runtime and how to use these logs to build an anomaly classification model with a simple user interface (UI). Herein, we discuss how we address these design issues. In Section 3.1, we introduce our agent service feature, including a description of how we design it to collect logs from the container runtime to monitor Docker-based containers. Furthermore, our agent service can be adapted to Kubernetes, that is, the user can easily deploy our agent service to a Kubernetes node. In Section 3.2, we discuss how we identify anomaly behavior in the container. We propose a container anomaly classification model using artificial intelligence technology to detect anomaly behavior of a container. This model uses supervised learning. We use a dataset that contains normal and abnormal samples to build the model. Finally, in Section 3.3, we introduce our complete container security platform, namely, KubAnomaly. Further details are provided at our website.[20] The overall relationship is shown in Figure 2. Our agent service must be installed on every host and Kubernetes node so that we can detect anomaly behavior in the containers. The service sends monitor logs to the center. When the center receives these monitor logs, the analyzer processes the data and transforms it into a feature vector. Then, our anomaly classification model is used to determine the container behavior.

3.1 | Agent Service

The agent service is used to collect monitor logs from Docker-based containers and also supports Kubernetes. There are two issues to be addressed, which are as follows.

- How to obtain the container behavior:
  To this end, we employ two third-party tools, Sysdig and Falco, to collect monitor logs. Sysdig is a popular monitoring tool that can be used to monitor all of a given container's events. Falco is a parsing tool based on Sysdig that can set customized rules to define specific events occurring in the container.
- What container behavior needs to be monitored:
  We believe that this issue is more important than the first because we have to design an event list based on our customized rules. This event list not only represents container behavior but also facilitates good performance. Monitoring too many events will result in a large overhead. We designed the event list shown in Table 2. Our event selection method is based on Reference 21. We selected four categories of system calls, namely, file I/O, network I/O, scheduler, and memory, comprising a total of 17 system calls. Furthermore, we added 14 statistical root directory access features. We designed these events such that we need not monitor all system calls, which helps minimize the overhead. We believe

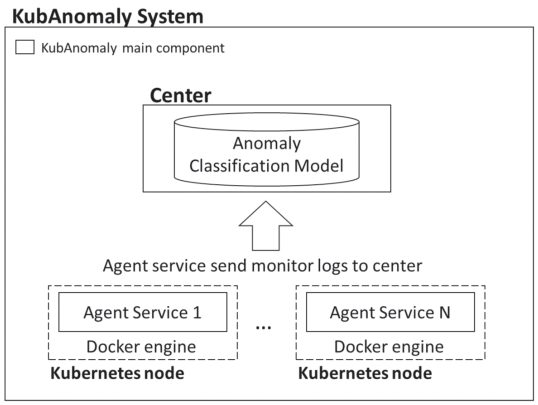


FIGURE 2 Relationship between the main components of KubAnomaly. KubAnomaly consists of a center, an anomaly classification model, and an agent service. The agent service is installed at the Kubernetes nodes and sends monitor logs to the center. Then, the analyzer processes the monitor log and detects anomalies

TABLE 2 List of all events monitored by the agent service

Event name	Description	Type
Read	Container exhibits read behavior	System call
File_IO	Container contains file operation	System call
Write	Container exhibits write behavior	System call
Accept	Container accepts connection on a socket	System call
Network_Http	Container has http flow	System call
Clone	Container creates a new program	System call
Select	Container monitors socket, waiting for data	System call
Poll	Container waits for stream event	System call
Rename	Container exhibits rename behavior	System call
Chdir	Container changes the current process working directory	System call
Kill	Container sends a kill signal	System call
Postgresql	Container exhibits postgresql flow	System call
Mkdir	Container exhibits mkdir behavior	System call
Brk	Allocate a small amount of memory	System call
Mmap	Allocate memory	System call
Munmap	Free memory	System call
Bin	Container accesses file under bin directory	Root directory access
Home	Container accesses file under home directory	Root directory access
Etc	Container accesses file under etc directory	Root directory access
Boot	Container accesses file under boot directory	Root directory access
Dev	Container accesses file under dev directory	Root directory access
Host	Container accesses file under host directory	Root directory access
Media	Container accesses file under media directory	Root directory access
Mnt	Container accesses file under mnt directory	Root directory access
Opt	Container accesses file under opt directory	Root directory access
Proc	Container accesses file under proc directory	Root directory access
Root	Container accesses file under root directory	Root directory access
Run	Container accesses file under run directory	Root directory access
Srv	Container accesses file under srv directory	Root directory access
Usr	Container accesses file under usr directory	Root directory access
Var	Container accesses file under var directory	Root directory access

Note: We use these events as features to construct our anomaly classification model. These events are of two types: system call and root directory access.

that these events are sufficient to represent container behavior with good performance. The details of our performance evaluation are presented in Section 4.4.

The agent service sends these monitor log data back to the center every 10 seconds. To prevent performance issues, we do not allow the agent to send these data every second; at the same time, we do not want to take too long for the detection of anomaly events. According to Akamai,[22] approximately 900 attacks can occur over 10 seconds; hence, we specify 10 seconds as our collection duration. We use these 10-second monitoring logs to construct our classification model. Subsequently, we can monitor the container behavior and successfully obtain the monitor logs. Next, we discuss the process of constructing the anomaly classification model.

3.2 | Anomaly Classification Model

KubAnomaly can report anomaly behavior in containers every 10 seconds. The workflow of our model is shown in Figure 3. The entire process is organized in the following four steps. First, we receive monitor log data from our agent service. Second, we extract the features that we want to use for our anomaly classification model. Third, we use a normalizing method to normalize the raw log data. Finally, we use these data as a training dataset to construct our anomaly classification model. Herein, we discuss the details of each step.

- Log collection: We use the agent service as our monitor log collector, the details of which are provided in Section 3.1.
- Feature extraction: After receiving the raw monitor logs and container hardware information, we parse them and transform them into the desired format. We use these raw monitor logs as input features and prepare them to train our classification model. Some parts of a raw monitor log are shown in Figure 4. We parse these logs and count the number of times each event occurs in a container over n seconds. Table 3 shows a sample of the feature extraction results of parsing for a 10-second event log. Then, we normalize the logs to obtain more suitable data for our classification model.
- Data normalization: Data normalization has advantages for machine learning, such as fast convergence and improved accuracy. However, not every normalization method can fit our data. To quickly apply different data normalization methods to our data, we used StandardScaler, MinMaxScaler, and Normalizer provided by sklearn, a well-known machine learning framework.[23] We found that Normalizer was the best solution for our data. Let event (E) be a set of the events shown in Equation (1). We use Equation (2) to calculate the L2 norm for (E). Once we obtain the L2 norm for one sample, we calculate the result for every element divided by the L2 norm, as shown in Equation 3, and this is the normalized result. We use this result as input data to train our classification model.
- Anomaly classification model construction: After normalizing the data, we are ready to construct our anomaly classification model using four fully connected layers. In this study, we use Keras [24] with TensorFlow[25] as the back end to build our classification model. TensorFlow is a popular open-source library created by Google, which supports machine intelligence and has good scaling and customization capabilities. Keras allows users to build models rapidly and provides complete documentation and community support. KubAnomaly consists of four layers of a fully connected layer. We use exponential linear units (ELUs) as activation functions from the first layer to the third layer. Such activation functions can accelerate learning in deep neural networks and provide higher classification accuracies. We use the original

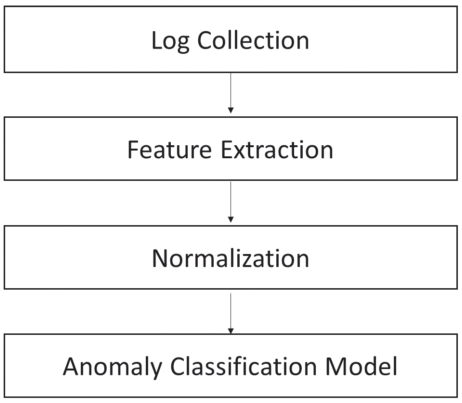


FIGURE 3 Flowchart of anomaly classification model construction. We define four stages for training our anomaly classification model from raw logs

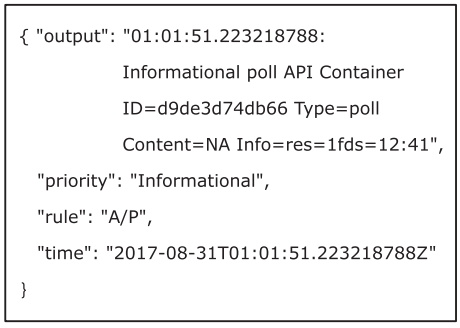


FIGURE 4 Raw log of a container from agent service. This is log collected by our agent service and contains the output of the stage. We parse this log and transform it into a feature vector in the feature extraction stage

TABLE 3 Results of feature extraction

Event name	# of times in 10 s	Event name	# of times in 10 s	Event name	# of times in 10 s
Udp	0	Run	0	Sys	0
Select	44	Poll	176	Dev	0
Bin	0	Media	0	Etc	288
Usr	0	Root	0	Host	0
Brk	0	Boot	0	Mkdir	0
Write	0	Connect	0	Tcp	17 752
clone_API	0	Proc	0	Read	0
Var	264	Network_http	16	File_IO	688
Kill_API	0	Accept	48	Home	0
Mnt	0	Opt	0	Rename	0
Srv	0	Mmap	74	Chdir_API	32
Network_Connection	1816	Execve_API	0		

Note: We transform raw logs into a feature vector, and we use this vector to train our anomaly classification model. This vector is an example of a container that has been attacked.

TABLE 4 Parameters of the anomaly classification model

Parameter name	Value
Input layer 1	N
Input layer 2	4N
Input layer 3	2N
Input layer 4	N
Layer 1 activation function	ELU
Layer 2 activation function	ELU
Layer 3 activation function	ELU
Layer 4 activation function	Softmax
Dropout layer	50%
Optimizer	Adam

Note: These parameters constitute the final version based on our experiment.
Abbreviation: ELU, exponential linear unit.

data as the input for the first layer, and we extend our feature number count to four times the number of input features. The output of the first layer is then used as the input for the second layer, where the input feature count is reduced to half that of the output. Like the second layer, the third layer uses the upper layer's output as input and reduces the feature count to half that of the output. The last layer uses softmax as the activation function to yield the classification result. We employ a dropout layer with a probability of 50%, that is, 50% of the network is useless in this iteration. Hence, we believe that we can avoid overfitting problems. Through this process, we obtain our output data. Because we use softmax as the activation function, we can obtain the probability of each output. Then, we choose the output with the highest probability as our classification result. The parameters that we use to build this model are listed in Table 4.

$$E = \{e_1, e_2 \ldots, e_n\}, \tag{1}$$

$$||E||_2 = \sqrt{\sum_{i=1}^{n} E_i^2}, \tag{2}$$

$$E_{\text{result}} = \left\{ \frac{e_1}{||E||_2}, \frac{e_2}{||E||_2}, \dots, \frac{e_n}{||E||_2} \right\} \tag{3}$$

3.3 | KubAnomaly System

To apply our anomaly classification model to the real world, we designed a system called KubAnomaly, which is published in the cloud.[20] Further usage information, such as how to install the agent service, is provided on our web portal. KubAnomaly is a system designed for Docker security management, and it can enable users to easily view all of the containers running on every Docker host. KubAnomaly can monitor all container behaviors on every user's host (Kubernetes node) in real time, and it uses our anomaly classification model to inform the user as to whether a container exhibits anomaly behavior. The detailed system architecture is shown in Figure 5. The KubAnomaly system has three major components, each of which has many subcomponents. Herein, we discuss these components and explain how they work.

- Center: The center is the main component of KubAnomaly, and it consists of several submodules. Its primary purpose is the collection of monitor logs and container hardware information from every agent service. Furthermore, it uses our pretrained anomaly classification model to detect whether a container shows anomaly behavior. In addition, it provides a RESTful API service to enable web service components to retrieve data. Herein, we discuss the subcomponents of this component.
- Analyzer: The analyzer is a virtual module in the KubAnomaly system, and its primary purposes are feature extraction and anomaly behavior detection, which are achieved by two subcomponents.

1. The feature extractor parses the raw monitor log and hardware resource information sent by the scheduler, and it extracts each container event. After extracting the features of each container and normalizing them, it sends these features to a behavior detection component.
2. The behavior detector receives the feature data from the feature extractor and loads the anomaly classification model, with which we can detect whether the container behavior is normal. Once the detection is complete, the behavior detector sends the results back to the scheduler. KubAnomaly provides two types of results. “Normal” implies that the relevant container has not exhibited any anomaly behavior in the last 10 seconds. “Anomaly” implies that the container has exhibited anomaly behavior, and the user should investigate the container. Possible anomalies include scenarios in which a container receives a large number of login requests in the last 10 seconds or those in which someone has tried to compromise a given container in the last 10 seconds. The details of the detection are discussed in Section 5.

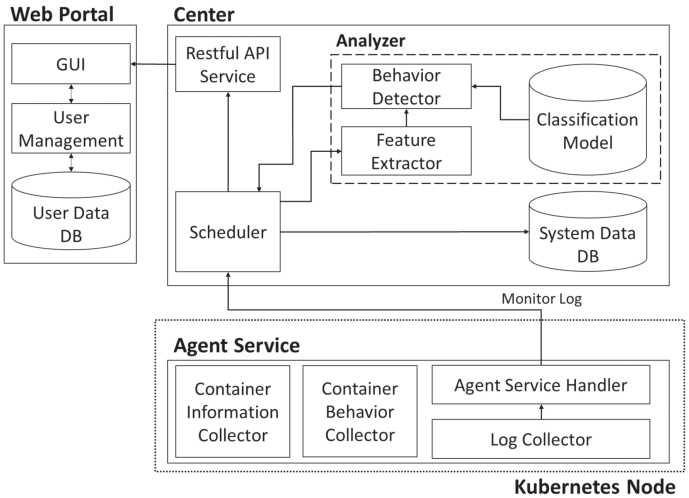
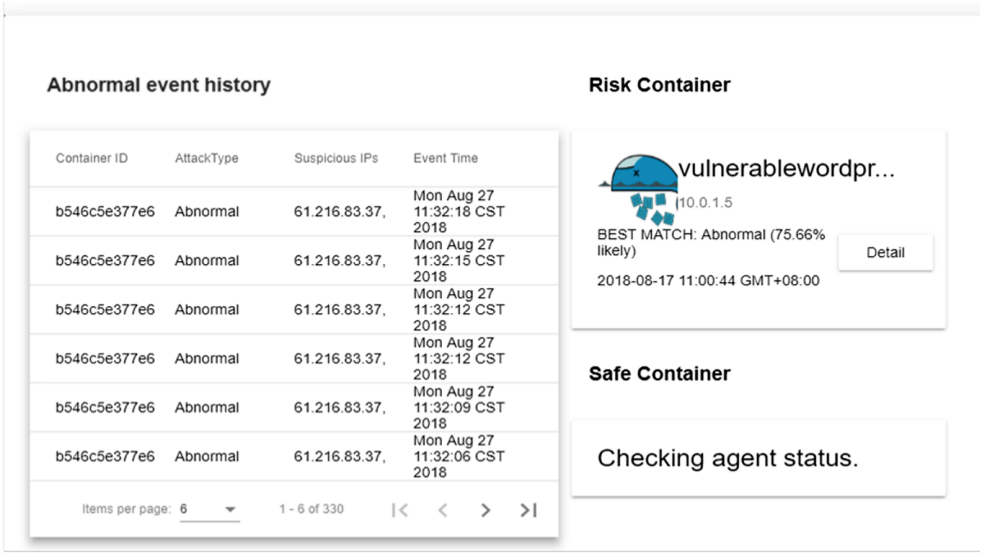


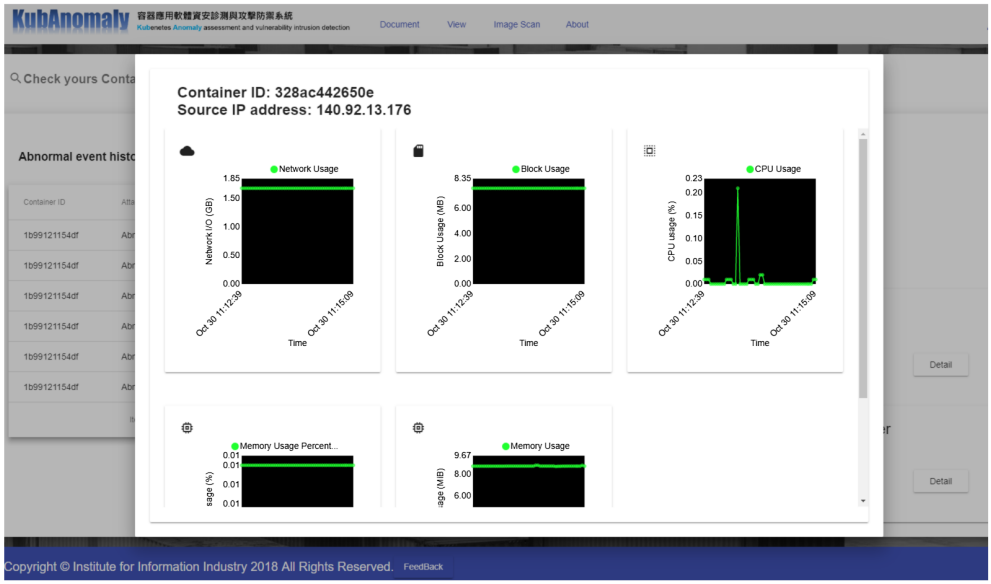
FIGURE 5 Detailed system architecture of KubAnomaly

3. Anomaly classification model: The anomaly classification model is used to detect abnormal container behavior. KubAnomaly can have multiple classification models, and if all models detect abnormal behavior, then KubAnomaly will send an alert message. We discuss the construction of our model in Section 3.2.

- Scheduler: The scheduler is used to communicate with every other module in the KubAnomaly system. It is responsible for managing samples received from the RESTful API service module, processing the detection results sent by the analyzer, and forwarding raw monitor logs and hardware resource information to the analyzer from the KubAnomaly agent. The scheduler also saves these results and the analysis status in a database, and it has the following features:
  - Construct mission: When the scheduler receives a request from the RESTful API service module or raw monitor logs from the agent service, it creates a mission flow and places the mission in a pool of processes awaiting execution. After processing, the scheduler records the data in a database.
  - Scheduler algorithm: The scheduler can perform multiple tasks simultaneously; however, it has limited capacity. When it is full, all the requests are placed in a waiting queue and processed using FIFO.



(A) KubAnomaly monitor page



(B) container status

FIGURE 6 KubAnomaly web portal. A user can register an account and view the status of all containers monitored by our agent service. A, Status of all containers. If a container exhibits anomaly behavior, we place it in a risk container. Otherwise, we place it in a save container. B, Individual container status including CPU, memory, and diskIO

- Remote procedure call (RPC) channel: In KubAnomaly, we use RPC to communicate with each module, as it provides an effective and easy means of communication. The scheduler creates one RPC channel per module in KubAnomaly.

- RESTful API service: The primary purpose of the RESTful API service module is to offer multiple RESTful APIs to provide detection reports and container information to the user. In particular, the module has a UI that can display detection results in real time and provide container hardware information to the user. In KubAnomaly, we offer two types of APIs. Users can use HTTP requests to obtain results in JSON format through these two APIs, or they can use KubAnomaly's UI to view detection results and container hardware information.
- Web portal: The web portal is used to inform users about the container status, and it refreshes the detection results every 10 seconds. In addition, KubAnomaly's UI displays hardware resource usage data so that users can quickly determine the hardware consumption of a container. We use the Java Spark framework MongoDB as well as Angular 2 to build the web service. Figure 6 shows the KubAnomaly web portal including the abnormal event history of containers and the system resources status of each container.

4 | EVALUATION

This section describes the creation of a dataset used to train the proposed anomaly classification model and provide an assessment of its performance. In Section 4.1, we introduce the dataset used to train our model, including how to create a dataset and its samples. In Section 4.2, we evaluate the performance of our model trained by different datasets and present the evaluation results. In Section 4.3, we describe how to deploy our agent on a public node running a WordPress website and then determine whether our agent can detect real hacker attacks. In Section 4.4, we evaluate the performance of our agent; in addition, we check our agent overhead and attempt to optimize it.

4.1 | Dataset

We used a simple dataset and a complex dataset, which are shown in Table 5 and have been published online.[26] Each dataset consists of two parts: 20% for testing and 80% for training. The simple dataset contains both normal and abnormal sample types. The normal samples include several types of web services run in containers. We used JMeter[27] to simulate user login behavior approximately 10 to 20 seconds after user login. The abnormal samples include two types of attacks aimed at compromising the web service. For the first type of attack, we used owasp zap[28] to simulate a hacker's attempt to attack the web service container. We also referred to Reference 17 and created a program that could autotraverse paths that may cause container information leakage. For the second type of attack, we used JMeter to generate numerous requests to simulate a DoS attack.

The complex dataset was similar to the simple dataset in that it contained both normal and abnormal sample types. However, the complex dataset differed from the simple dataset in that it exhibited a more complex behavior. For example, in the normal sample type, we define 14 user behaviors such as login and page views. Then, we randomly chose multiple behaviors for each user so that every user would behave differently and appear more like a real user. In the abnormal sample type, we have three types of abnormal samples. Compared with the simple dataset, we included additional hacker tools such as sqlmap[29] and command injection with CVE-2017-5638 vulnerability. Some of the attacks were successful, which meant that we obtained some real data from the victim containers. In this study, we focus on detecting general abnormal behavior and label all kinds of abnormal behavior as one category. Hence, Table 5 presents more than 60% of abnormal data in the datasets. We use these datasets to evaluate the accuracy of each method, although the amount of two categories in the datasets is an imbalance to affect the detection accuracy. We believe that the experiment result is credible based on the same datasets.

We used the complex and simple datasets together. Then, we used several common machine learning algorithms for performance comparison. In our evaluation, we used two categories of features for training our model and performance comparison. The first category of features includes only system calls, whereas the second category of features includes all the events that we collected. These events are listed in Table 2.

TABLE 5 Anomaly classification model training dataset

(a) Simple dataset		
Attack type	**Sample type**	**# of samples**
None	Normal	14 938
Web attack	DoS	10 455
Web attack	Zap attack	15 152
	Abnormal (total)	25 607
(b) Complex dataset		
Attack type	**Sample type**	**# of samples**
None	Normal	8850
Web attack	SQL injection	15 288
Web attack and CVE attack	Command injection	5765
Web attack	Zap attack	5607
	Abnormal (total)	26 660

Note: We use these datasets to train our model. (a) The simple dataset consists of two types of abnormal behavior. (b) The complex dataset consists of three types of abnormal behavior.

Abbreviations: CVE, common vulnerabilities and exposures; DoS, denial-of-service

TABLE 6 Anomaly classification model evaluation result using the simple dataset with only system calls as features

Machine learning algorithm	ACC (%)	Precision (%)	Recall (%)	F1 (%)	AUC (%)
SVM linear	98.03	98.03	98.03	98.03	98.46
SVM rbf	98.02	98.02	98.02	98.02	98.45
KubAnomaly	98.10	98.10	98.10	98.10	98.45
CNN	98.10	98.10	98.10	98.10	98.51

4.2 | Detection rate evaluation result

We divided our evaluation flow into two parts. First, we used the simple dataset for training the model with different machine learning algorithms, using only system call events as features. The results are summarized in Table 6, and the receiver operating characteristic (ROC) is shown in Figure 7. We found that each algorithm exhibited outstanding performance. Specifically, all algorithms achieved an accuracy of nearly 98%. Hence, we did not use all the events as features to train our model in this case. Second, we used the complex dataset to train the model with different machine learning algorithms and used only system call events as features. The results are shown in Table 7 and the ROC curve is shown in Figure 8A. The performance of all the models was degraded; however, KubAnomaly still exhibited an accuracy of 95%. To improve the performance of our model, we trained all the models again, using all the events as features. The results are summarized in Table 8 and the ROC curve is shown in Figure 8B. Herein, we found that the model accuracy improved, albeit slightly, to 96%. This experiment demonstrated that using a different number of features yields a similar result. We believe that the reason could be the absence of a proper abnormal behavior sample in our dataset. Thus, based on our experimental results summarized in Tables 7 and 8, we believe that it is sufficient to use only system calls as features. Overall, the accuracy of KubAnomaly was superior to that of the other methods.

We used our anomaly classification model with only system calls as features to evaluate the accuracy on each type of anomaly sample in the complex and simple datasets. The results are summarized in Table 9. The proposed classification model is not good at detecting the zap attack anomaly in the complex dataset. We believe that this could be because a zap attack includes multiple types of web attacks such as XSS and brute force attacks. Hence, the complexity of our model is higher than that of other models, which may result in lower performance.

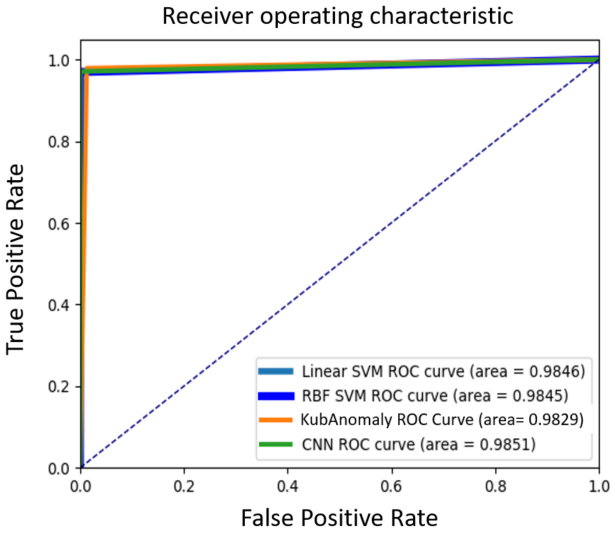


FIGURE 7 Anomaly classification model ROC curve using simple dataset with only system call events as features. In this scenario, all the methods show good performance. We can see that all the methods achieved an accuracy of 98%. ROC, receiver operating characteristic

TABLE 7 Evaluation results of anomaly classification model for the complex dataset with only system calls as features

Machine learning algorithm	ACC (%)	Precision (%)	Recall (%)	F1 (%)	AUC (%)
SVM linear	89.97	89.97	89.97	89.97	86
SVM rbf	89.77	89.77	89.77	89.77	86.39
KubAnomaly	95.33	95.33	95.33	95.33	93.57
CNN	94.65	94.65	94.65	94.65	92.7

4.3 | Real-world evaluation

To verify the ability of our modules to detect anomaly behavior in the case of real-world attacks, we set up a WordPress site containing normal content and vulnerabilities, as shown in Figure 9A. We opened this WordPress site to the public. Furthermore, we monitored it using KubAnomaly to immediately detect abnormal behavior and display alert information, as shown in Figure 6. KubAnomaly logged these alerts, allowing the user to check the alert history at any given point of time. We recorded the container ID, attack type, suspicious IPs, and event time stamp. Some of these data are presented in Table 10. Furthermore, we manually checked the WordPress access log file, as shown in Figure 12. This access log contains the HTTP request data. Hence, any attempt by an attacker will be recorded. When KubAnomaly detects an anomaly event, we can use this access log to verify whether it is a false alert.

4.4 | Performance evaluation

In this section, we discuss the performance of the agent service. Because the agent service is installed on user hosts, we aim to minimize the overhead, meanwhile allowing the agent to collect monitor logs as much as possible. To this end, we prepared a testing environment consisting of a virtual computer running on VMware with 4 CPU cores, 8 GB of RAM, and Ubuntu 16.04 as the operating system. We ran two WordPress containers on it and simulated user pageviews from a separate computer. We designed two versions of the agent and compared them to determine which version is slower. Both agents have the same monitor events, and the monitor event list is shown in Table 2. Our monitor event selection method is based on Reference 21; however, we have additional features of statistical directory access. One version attempted to have the agent parse logs on the agent side; the other version did not parse logs at the agent side, reporting to the KubAnomaly center directly. The results are shown in Figures 10 and 11. Accordingly, we found a major difference between these two versions. In the local parsing version, installing the KubAnomaly agent resulted in a high peak value between 40% and 50% CPU usage, although the average was 9%. However, in the remote parsing version, we found that the peak value varied between 20% and 30% CPU usage, with an average of 5%. Therefore, we prefer the remote parsing version, as we believe that it can provide users with a good usage test. In addition, we evaluated the per container performance impact. In this case, we used an agent without the local parsing version and JMeter to simulate user

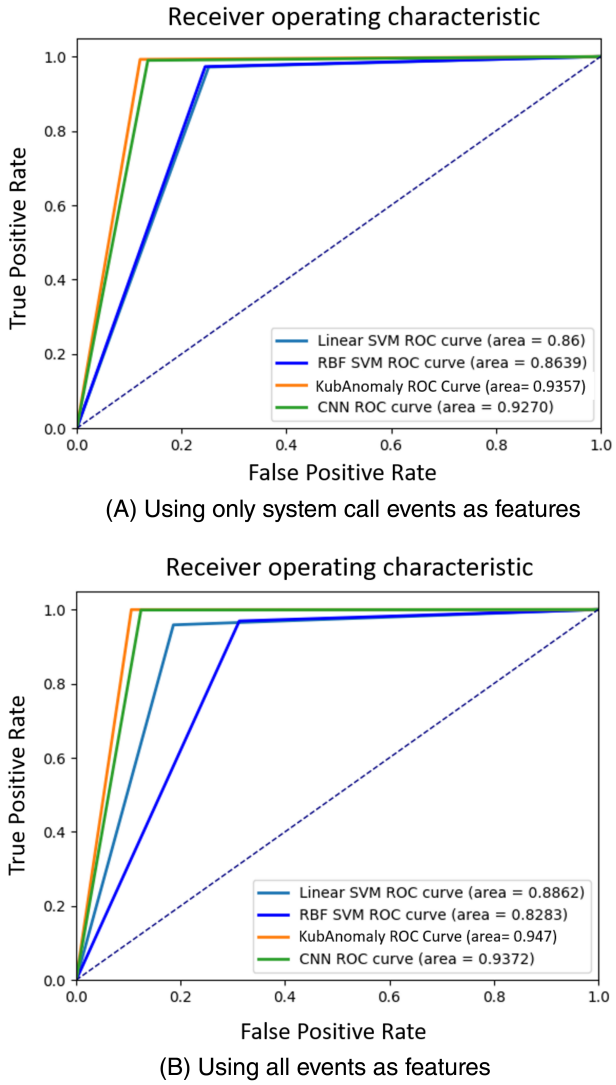


FIGURE 8 Anomaly classification model ROC curve with complex dataset. In this scenario, we found that the performance of support vector machine (SVM)-based model is much lower than that of artificial neural networks, whereas our model exhibits the best performance when compared with other models (95% accuracy). ROC, receiver operating characteristic

TABLE 8 Evaluation results of anomaly classification model for the complex dataset with all events as features

Machine learning algorithm	ACC (%)	Precision (%)	Recall (%)	F1 (%)	AUC (%)
SVM linear	90.86	90.86	90.86	90.86	88.62
SVM rbf	87.18	87.18	87.18	87.18	82.63
KubAnomaly	96.33	96.33	96.33	96.33	94.7
CNN	95.65	95.65	95.65	95.65	93.72

behavior. We have four evaluate types: wordpress1-withagent, wordpress2-withAgent, wordpress1-withoutAgent, and wordpress2-withoutAgent. The results are summarized in Table 11. Herein, we used a summary report provided by JMeter, and we found that the website performance is nearly the same, regardless of whether we install the agent. Hence, we believe that for individual containers, there is nearly no overhead because the website performance is nearly the same.

5 | DISCUSSION

In this section, we analyze a particular case that we found in our evaluation results. Moreover, we analyze a real attack detected by KubAnomaly. In particular, we detect the type and origin of the attack. Finally, we discuss the limitations of the system.

TABLE 9 Evaluation results of anomaly classification model for each anomaly sample type

(a) Complex dataset				
Attack type	**Sample type**	**Precision (%)**	**Recall (%)**	**F1**
Web attack	SQL injection	99.6	99.6	99.6
Web attack and CVE	Command injection	99.1	99.1	99.1
Web attack	Zap attack	84.5	84.5	84.5
(b) Simple dataset				
Attack type	**Sample type**	**Precision (%)**	**Recall (%)**	**F1**
Web attack	DoS	96.5	96.5	96.5
Web attack	Zap attack	97.7	97.7	97.7

Note: The results indicate that our anomaly classification model has a lower precision in the case of complex datasets.
Abbreviations: CVE, common vulnerabilities and exposures; DoS, denial-of-service.

FIGURE 9 Decoy website: we uploaded some news on it so that it appears like a real news blog

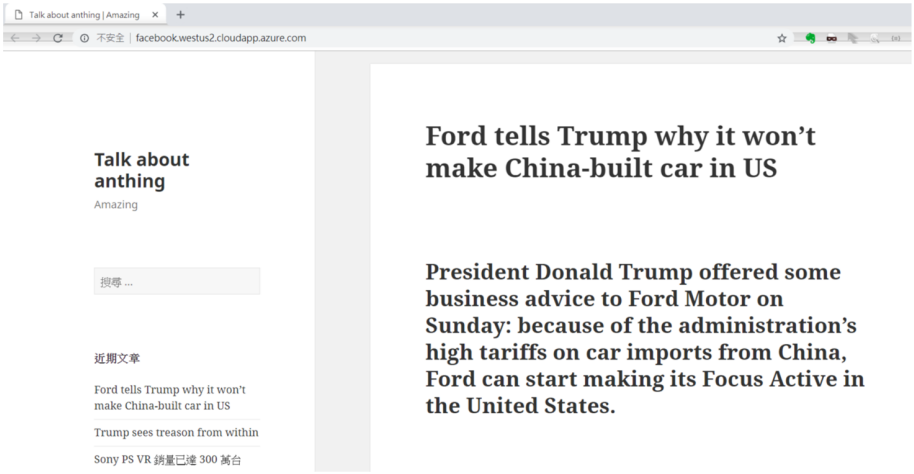


TABLE 10 Alert history classified by our anomaly classification model

Container ID	Positive	Suspicious IPs	Event time
b546c5e377e6	Anomaly	106.75.233.194	Thu Sep 09 11:18:07 CST 2018
b546c5e377e6	Anomaly	164.115.41.175	Fri Sep 08 03:53:30 CST 2018
b546c5e377e6	Anomaly	122.227.186.130	Fri Sep 07 21:46:18 CST 2018
b546c5e377e6	Anomaly	122.114.99.35	Fri Sep 07 21:45:37 CST 2018

Note: Container ID b546c5e377e6 is our decoy website.

- Evaluation case study

As mentioned in Section 4, we used two human-like datasets, namely, the simple dataset and the complex dataset. With the simple dataset, we achieved over 98% precision. We believe that this may be because the data is too simple, as it only comprises one user behavior, whereas the abnormal dataset comprises many attacker behaviors. We then used the complex dataset to evaluate our model and found that the precision was reduced to 95%. From this result, we learned that when the numbers of abnormal behaviors and normal behaviors are nearly equal, our model will have some false positives. To reduce these false positives, we added the frequency of access time for different directories in the root as a feature and retrained our model. The results showed that the precision increased to 96%. Although the improvement is not significant, we believe that these features can describe container behavior more precisely.

In addition, we wanted to use a public dataset to doubly verify our model's performance. However, such a system call log dataset is extremely rare. We found a public dataset called CERT,[30] which contains various types of log data,

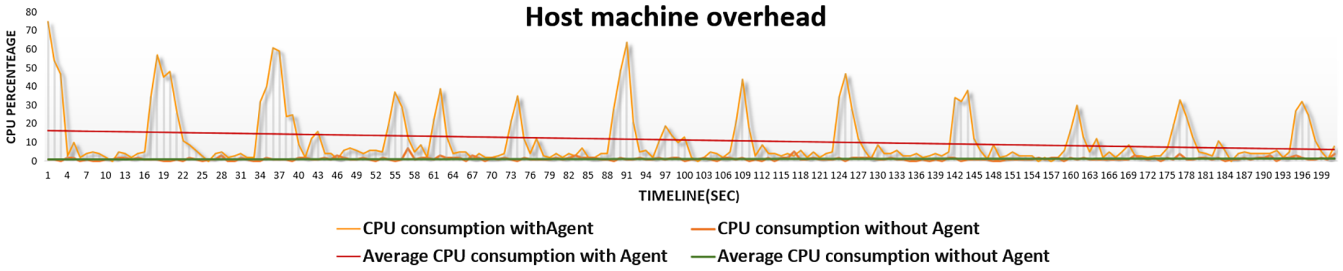


FIGURE 10 Agent service overhead before performance tuning. The peak values in the figure correspond to our agent processing log. This is a performance issue that we need to address

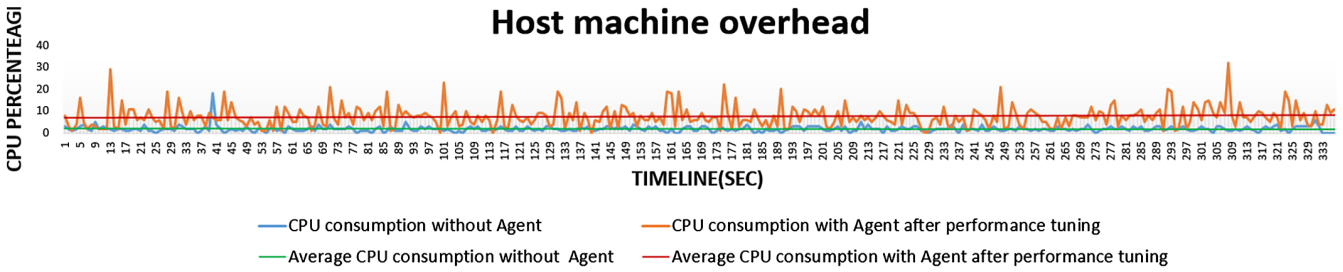


FIGURE 11 Agent service overhead after performance tuning. We can see that the peak values are reduced significantly. In this version, our agent incurs an average overhead of only 5%

TABLE 11 Performance evaluation with and without the David agent

Evaluate type	Throughput (ms)	Received (KB/s)	Sent (KB/s)	Avg. (bytes)
wordpress1_withAgent	0.18496	3.39	0.03	18759.3
wordpress2_withAgent	0.186	3.46	0.03	19039.5
wordpress1_withoutAgent	0.17952	3.33	0.03	19005.5
wordpress2_withoutAgent	0.17614	3.25	0.03	18878.4

Note: We use JMeter to generate user behavior. Then, JMeter will record the performance data. According to the data below, we found no significant overhead for individual containers.

including email and device data; however, it does not contain system call log data. We integrated all types of data from the CERT dataset into one vector, and we used the same method as that used for KubAnomaly. We defined our time window as 1 day, where KubAnomaly is executed approximately every 10 seconds. Then, we measured the frequency of every event. Our event list is presented in Table 12. We used this dataset to test our abnormal detection method. Because this dataset did not have any labeling, we used our feature extraction method with unsupervised learning to detect and categorize abnormal user behavior. First, we integrated various types of data into a single vector, as shown in Table 12. For performance evaluation, we referred to Reference 31. It was found that in the CERT dataset, user ONS0995 has one malicious behavior; hence, we used this user's log data as an example. The results are presented in Table 13. Accordingly, we found that our abnormal detection method can detect real abnormal behavior, although it returns some false positives. Because it is a security measure, we believe that detecting all true abnormalities is more important than excluding all false positives. In addition, the results showed that we achieved 100% detection, but poor accuracy. In our view, this is caused by the extremely small sample size. We believe that using fully labeled data can lead to a higher accuracy.

- Real-world attack case study

  Previously, we set up a WordPress site that contains vulnerabilities, and we installed the KubAnomaly agent service on that host. KubAnomaly detected and recorded the attack events. Then, we counted the attacks recorded by

TABLE 12 Event names of various integrated log data in the CERT dataset

Event name	Description
Remove	User removes a portable device
Insert	User inserts a portable device
Logon	User logs in to a computer
Web count	# of users visiting the website
Email count	# of users sending email
Duration	Duration between user login and logout

Note: We combine multiple types of logs in the CERT dataset and extract these events as our features.

TABLE 13 Anomaly classification model evaluation with user ONS0995 in the CERT dataset

(a) Confusion matrix for user ONS0995 with the KubAnomaly feature extraction method

	Positive	Negative
Positive	1	0
Negative	24	29

(b) Accuracy for user ONS0995 with the KubAnomaly feature extraction method

	Positive (%)
Precision	4
Recall	100

KubAnomaly over approximately 1 month. The results are presented in Table 14, and we summarize them in the following aspects:

- IP location: We found 17 different attacker IP addresses. We used Reference 32 to check the IP locations. We statistically analyzed these locations, and the results showed that most of the attacks originated in China.
- Attack time: According to the results recorded by KubAnomaly, there is no specific time preferred by attackers. Web attacks occur at nearly any time, as mentioned in Reference 22.
- Attack type: We analyzed the server log shown in Figure 12 and identified three different types of attacks. Among them, web scanning and vulnerability scanning appear to be very similar; however, we were able to identify vulnerability scanning attacks using a tool called ZmEu.[33] We do not know the other tools that the attackers used. A PROFIND attack is an attack that exploits Microsoft IIS server vulnerabilities in an attempt to crash the IIS server. In our case, the attackers seemed to carry out this attack as a broadcast to everyone on the Internet and then checked for successful compromises.

From the analysis presented above, we proved that the KubAnomaly system can detect real-world attacks. In addition, we found from the attack pattern that many attackers exist and that our web service can be attacked at nearly any time. We shall collect this data, construct a dataset, and then, use this dataset to train a web attack classification model. In future, we can use this model to know which type of anomaly it is.

- System limitation

  The implementation of KubAnomaly is complete; however, it still has some limitations, which are listed below.

  - Identifying container behavior as normal or abnormal: In this study, we focused on detecting anomaly behavior in Docker-based containers. KubAnomaly can detect which containers exhibit anomaly behavior; however, we did not analyze the type of anomaly behavior. KubAnomaly will record alert messages; then, security experts can check the anomaly containers. Our system is positioned as the first line of protection, and our goal is to quickly identify anomalies so that the overhead of other analysis tools is reduced.

TABLE 14 List of attack events detected by KubAnomaly

Attacker IP address	Country	City	Attack type	Attack time
122.114.99.35	China	Zhen	Web attack (vulnerability scanner)	2018/09/07 20:15 CST
132.232.143.187	China	Beijing	Web attack (web_scanning)	2018/09/07 21:45 CST
58.87.104.210	China	Beijing	Web attack (web_scanning)	2018/09/07 09:38 CST
122.227.186.130	China	Wangba	Web attack (web_scanning)	2018/09/07 21:53 CST
164.115.41.175	Thailand	Bangkok	Web attack (web_scanning)	2018/09/08 03:53 CST
106.75.233.194	China	Shanghai	Web attack (web_scanning)	2018/09/09 11:18 CST
123.207.77.148	China	Beijing	Web attack (web_scanning)	2018/09/12 15:53 CST
132.232.12.125	China	Beijing	Web attack (web scanning)	2018/09/13 10:45 CST
212.18.175.68	Portugal	Lisbon	Web attack (PROPFIND attack)	2018/09/17 21:05 CST
132.232.15.93	China	Beijing	Web attack (PROPFIND attack)	2018/09/18 01:20 CST
122.154.24.254	Thailand	Phra Nakhon Si Ayutthaya	Web attack (PROPFIND attack)	2018/09/18 05:36 CST
89.248.168.171	Seychelles	Victoria	Web attack(vulnerability scanner)	2018/09/18 07:32 CST
211.149.196.24	China	Chengdu	Web attack (PROPFIND attack)	2018/09/18 14:12 CST
132.232.44.24	China	Beijing	Web attack (PROPFIND attack)	2018/09/18 14:12 CST
118.24.64.15	China	Beijing	Web attack (PROPFIND attack)	2018/09/19 04:21 CST
119.29.41.87	China	Beijing	Web attack (PROPFIND attack)	2018/09/21 16:33 CST

Note: These attack events were doubly verified manually.

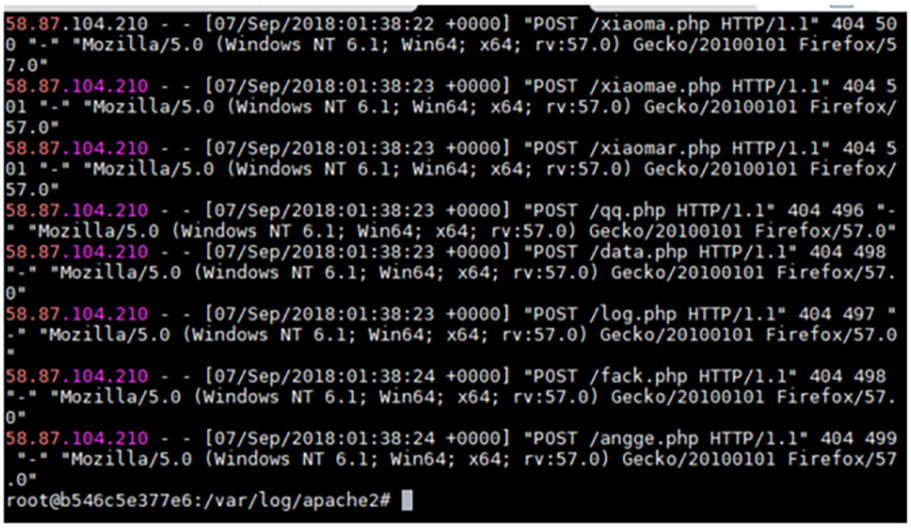


FIGURE 12 Decoy container's server logs with attack behavior. This log showed that the attacker uses a web scanning tool to scan our decoy website. We further use this log to doubly verify our classification result

- Runtime anomaly behavior detection: Our system is an anomaly behavior detection system for containers, which focuses on container security. It is another research domain focused on docker image static analysis. It scans the package inside the Docker image and then reports the vulnerability of the image.
- Risk of KubAnomaly agent: Because our agent requires privilege permission, it may have some risk to be hacked by other hackers. To address this issue, we believe that we have already done our best because our agent opens only one port to communicate with the center. We use gRPC to implement our function. Hence, our vulnerability is the same as that of gRPC.

6 | CONCLUSION

KubAnomaly is a container anomaly analysis tool that can be used by IoT platform vendors to detect security anomaly events in Kubernetes, a cloud container orchestration platform. Our experimental results showed that its accuracy is nearly 96% for detecting anomaly behavior in containers, with an F1 score of 98.1%.

Containerized applications are widely used, and the security of the container environment is becoming increasingly important. NIST has released a security guide to explain container-related security, which identifies the five main risks of using containers. On the basis of these risks, we analyzed and summarized the current security studies and tools for Docker and the Docker-based orchestration platform Kubernetes. We determined that runtime container security solutions have not discussed emerging threats such as insider attacks. Therefore, our study focused not only on network detection but also on new abnormal behavior detection for container security. We obtained the following outcomes from our security anomaly analysis of Kubernetes:

- First, we implemented and evaluated KubAnomaly, a Docker-based Kubernetes security management system that automatically monitors all container behaviors to detect attacks, such as web service attacks and CVE attacks, to solve runtime container security problems.
- Second, we proposed a feature extraction method based on system log monitoring to build an anomaly behavior data model for containers using an artificial neural network framework.
- Third, we used three different datasets, including private data, public data (CERT), and real-world experimental data, to evaluate the system accuracy and performance. The results of experiments with other machine learning algorithms showed that the accuracy of the proposed model is nearly 96% for detecting anomaly behavior in containers, with an F1 score of 98.1%. Moreover, the performance impact of the KubAnomaly system is only 5% higher than that of the original system.
- Finally, for a real-world experiment, we created an online web service containing several vulnerabilities to attract attackers. The results of this experiment showed that KubAnomaly is capable of detecting many anomalous events, such as web service attacks and CVE attacks originating from China and Thailand.

Therefore, KubAnomaly can be used to monitor threat events and improve container runtime security. Moreover, it is compatible with the Docker orchestration platform. In future, we plan to integrate KubAnomaly with 5G network function virtualization platforms to enhance its capabilities for cybersecurity.

CONFLICT OF INTEREST

The authors have no conflict of interest relevant to this article.

ORCID


Chin-Wei Tien https://orcid.org/0000-0002-5490-2953

REFERENCES

1. DataDog. *DataDog: 8 Surprising Facts About Real Docker Adoption*; 2018. https://www.datadoghq.com/dockeradoption/.
2. WIKIPEDIA. *Microservices: A Variant of the Service-Oriented Architecture (SOA) Architectural Style that Structures an Application as a Collection of Loosely Coupled Services*; 2018. https://en.wikipedia.org/wiki/Microservices.
3. Kit Colbert CTO Cloud Platform BU. *DockerCon 2017: Containers Go Mainstream*; 2017. https://blogs.vmware.com/cloudnative/2017/04/24/dockercon-2017-containers-go-mainstream/.
4. Docker Inc. *DockerCon 2018*; 2018. https://2018.dockercon.com/agenda/.
5. Aqua. *BlackHat 2017: Multi-Stage Attack Targeting Container Developers*; 2017. https://blog.aquasec.com/hostrebinding-and-shadow-containers-at-blackhat-2017.
6. Souppaya M, Morello J, Scarfone K. *NIST SP 800-190: National Institute of Standards and Technology*; 2017.
7. Sysdig. *Sysdig: Sysdig is Open-Source, System-Level Exploration*; 2018. https://www.sysdig.org/.
8. Xu W, Huang L, Fox A, Patterson D, Jordan MI. Detecting large-scale system problems by mining console logs. *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles. SOSP '09*. Big Sky, MT: ACM; 2009:117-132.
9. Lou J-G, Fu Q, Yang S, Xu Y, Li J. *Mining invariants from console logs for system problem detection*; 2010.
10. Lin CH, Tien CW, Pao HK. Efficient and effective NIDS for cloud virtualization environment. *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*. Taipei, Taiwan, ROC: IEEE; 2012:249-254.
11. Gomez ME. *Full Packet Capture Infrastructure Based on Docker Containers*. Tech. rep. SANS Institute InfoSec Reading Room; 2016.

12. Borhani A. *Anomaly Detection, Alerting, and Incident Response for Containers*; 2017. https://pentesting.sans.org/resources/papers/gcih/anomaly-detection-alerting-incident-response-containers-139309.
13. Ubuntu wiki. *AppArmor*; 2018. https://wiki.ubuntu.com/AppArmor.
14. Sysdig. *Falco: Sysdig Falco is an open-source, behavioral activity monitor powered by sysdig*; 2018. https://www.sysdig.org/falco/.
15. Twistlock. *Twisklock: The Most Complete Container Cybersecurity Platform*; 2018. https://www.twistlock.com/.
16. Aqua. *Aqua Container Security Platform*; 2018. https://www.aquasec.com/products/aqua-container-securityplatform/.
17. Gao X, Gu Z, Kayaalp M, Pendarakis D, Wang H. ContainerLeaks: emerging security threats of information leakages in container clouds. *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. Denver, CO: IEEE; 2017:237, 237-248, 248. https://ieeexplore.ieee.org/document/8023126/.
18. Du Q, Xie T, He Y. In: Vaidya J, Li J, eds. *Anomaly Detection and Diagnosis for Container-Based Microservices with Performance Monitoring. Algorithms and Architectures for Parallel Processing*. Cham, Switzerland: Springer International Publishing; 2018:560-572.
19. Zou Z., Xie Y., Huang K., Xu G., Feng D., Long D. *A Docker container anomaly monitoring system based on optimized isolation forest*. *IEEE Trans Cloud Comput*. 2019. https://doi.org/10.1109/TCC.2019.2935724.
20. Industry Institute for Information. *KubAnomaly System*; 2018. https://kubanomaly.westus2.cloudapp.azure.com/.
21. Degioanni L. *The Fascinating World of Linux System Calls*; 2014. https://sysdig.com/blog/fascinating-worldlinux-system-calls/.
22. Akamai. *Web Attack Visualization*; 2018. https://www.akamai.com/uk/en/about/our-thinking/state-of-the-internetreport/web-attack-visualization.jsp.
23. Community. *scikit-learn Machine Learning in Python*; 2018. https://scikit-learn.org/.
24. Team Keras. *Keras: The Python Deep Learning Library*; 2018. https://keras.io/.
25. Community TensorFlow. *TensorFlow: An Open-Source Software Library For Machine Intelligence*; 2018. https://www.tensorflow.org/.
26. Industry Institute for Information. *The Data Set Use for KubAnomaly Model Training*; 2018. https://github.com/a18499/KubAnomaly_DataSet.
27. Apache. *JMeter: An Open-Source Tool to Simulate User Behavior*; 2018. http://jmeter.apache.org/.
28. OWASP. *OWASP Zed Attack Proxy (ZAP): One of the World's Most Popular Free Security Tools*; 2017. https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project.
29. SQLMap. *Automatic SQL Injection and Database Takeover Tool*. 2018. http://sqlmap.org/.
30. Cert. *Insider Threat Test Dataset*; 2018. https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=508099.
31. Böse B., Avasarala B., Tirthapura S., Chung Y., Steiner D. Detecting Insider Threats Using RADISH: A System for Real-Time Anomaly Detection in Heterogeneous Data Streams, *IEEE Syst J*. 2017;11(2):471–482. https://doi.org/10.1109/JSYST.2016.2558507.
32. iplocation.net. *IP Location Finder*; 2018. https://www.iplocation.net/.
33. *ZmEu vulnerability scanner*; 2018. https://en.wikipedia.org/wiki/ZmEu_(vulnerability_scanner).

How to cite this article: Tien C-W, Huang T-Y, Tien C, Huang T-C, Kuo S-Y. KubAnomaly: Anomaly detection for the Docker orchestration platform with neural network approaches. *Engineering Reports*. 2019;1:e12080. https://doi.org/10.1002/eng2.12080