Learning State Machines to Monitor and Detect Anomalies on a Kubernetes Cluster

Cao, Clinton; Blaise, Agathe; Verwer, Sicco; Rebecchi, Filippo

DOI
10.1145/3538969.3543810

Publication date
2022

Document Version
Final published version

Published in
Proceedings of the 17th International Conference on Availability, Reliability and Security, ARES 2022

Learning State Machines to Monitor and Detect Anomalies on a Kubernetes Cluster

Clinton Cao
Delft University of Technology
Netherlands
c.s.cao@tudelft.nl

Agathe Blaise
Thales SIX GTS France
France
agathe.blaise@thalesgroup.com

Sicco Verwer
Delft University of Technology
Netherlands
s.e.verwer@tudelft.nl

Filippo Rebecchi
Thales SIX GTS France
France
filippo.rebecchi@thalesgroup.com

ABSTRACT

These days more companies are shifting towards using cloud environments to provide their services to their client. While it is easy to set up a cloud environment, it is equally important to monitor the system's runtime behaviour and identify anomalous behaviours that occur during its operation. In recent years, the utilisation of Recurrent Neural Networks (RNNs) and Deep Neural Networks (DNNs) to detect anomalies that might occur during runtime has been a trending approach. However, it is unclear how to explain the decisions made by these networks and how these networks should be interpreted to understand the runtime behaviour that they model. On the contrary, state machine models provide an easier manner to interpret and understand the behaviour that they model. In this work, we propose an approach that learns state machine models to model the runtime behaviour of a cloud environment that runs multiple microservice applications. To the best of our knowledge, this is the first work that tries to apply state machine models to microservice architectures. The state machine model is used to detect the different types of attacks that we launch on the cloud environment. From our experiment results, our approach can detect the attacks very well, achieving a balanced accuracy of 99.2% and a F_1 score of 0.982.

CCS CONCEPTS

• **Computing methodologies** → **Anomaly detection**.

KEYWORDS

Kubernetes, Runtime Monitoring, State Machine, Anomaly Detection, Microservice Architecture

ACM Reference Format:
Clinton Cao, Agathe Blaise, Sicco Verwer, and Filippo Rebecchi. 2022. Learning State Machines to Monitor and Detect Anomalies on a Kubernetes Cluster. In *The 17th International Conference on Availability, Reliability and Security (ARES 2022), August 23–26, 2022, Vienna, Austria.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3538969.3543810

ARES 2022, August 23–26, 2022, Vienna, Austria

ACM ISBN 978-1-4503-9670-7/22/08.
https://doi.org/10.1145/3538969.3543810

1 INTRODUCTION

These days more devices are being connected to the Internet. The Internet enables users to access data and services with high availability. There are a plethora of cloud services that allow users to store their data online on a cloud server. Dropbox[1] and Google Drive[2] are two well-known examples of such cloud services. More companies are shifting towards the usage of a cloud environment to provide their services to their customers. Setting up such an environment these days is also not difficult as there exist tools that can be used to quickly set up a cloud environment. A set of detailed instructions is provided by the tools to guide developers during the setup. Kubernetes (K8s)[3] is an example of such a tool that provides instruction on how to set up a cluster and use it to deploy different services.

Though the tools make it easier for a company to deploy a cloud environment and use it to provide their services to their customer, it is equally important to keep this environment secure from any malicious actor that might want to gather private (confidential) data from the customers or take down the whole environment. In this respect, microservice-based applications must be monitored at runtime to identify anomalous behaviours that occur during their operations. This requires having a constantly updated view of the actions and events happening and then comparing them to an expected model of operation. Logs and network traffic information such as NetFlow can be collected and monitored by the services running within the environment. The challenge here is to show how to deal with the large volume of log/NetFlow data that are generated by the services.

In recent years, the utilisation of RNNs or DNNs to detect anomalies that occur during the runtime of a software system has been a trending approach investigated by many researchers [7, 14, 19, 24]. A neural network can be trained with the log data or NetFlow data produced by the applications to model the behaviour of the applications. While it has been shown that using this approach can detect anomalies with high accuracy, it is not clear how to explain the classification decisions made by the neural network. Furthermore, it

[1] https://www.dropbox.com/?landing=dbv2
[2] https://www.google.com/drive/
[3] https://kubernetes.io/

is also not clear how the neural network should be interpreted (and visualized) to understand the behaviours that are modelled. Not being able to understand or explain the decisions that are made by a model could lead a company to not adopt these models in practice as there might be legal/regulatory compliance requirements that they need to follow.

On the contrary, is it easier to understand the behaviour that is modelled using a state machine similar to what is done for the design and analysis of software [11]. A state machine is a mathematical model used to model sequential behaviour and can be learned automatically from the traces produced by any software system. Based on a given input sequence, one can understand in which state a complex system will end up. Hence, state machines provide insights into the inner working of a software system and the models can be used to verify whether the software system is behaving as it should be [5, 11]. Extending such a state machine approach to microservice architectures is currently an open challenge.

When it comes to the learning of state machines, there are two approaches: the active-learning approach and the passive-learning approach. In the former, the learning algorithm continuously queries the system to generate training data for the learning process. LearnLib [8] is a well-known framework that utilizes active learning to learn state machines. In the latter, the algorithm does not query the system to generate training data, but it simply observes the data that is produced by the system. The observations are then used for the learning of a state machine.

To detect the unusual behaviour using state machines, a probabilistic state machine can be employed. The probabilistic state machine can compute the probabilities for the traces that were produced by the system. Traces with low probabilities are considered to be unusual and an alarm should be raised. One effective method for learning probabilistic state machines in the passive-learning approach is the state-merging heuristic method.

In this work, we provide a novel anomaly detection approach that uses a passive-learning approach to learn probabilistic state machines from the NetFlow data gathered from a K8s cluster. The state machines are used to model the normal network behaviour of the services running with the cluster. Large deviations that are detected within the models are considered to be suspicious and an alarm should be raised. We show that we can use state machines to model the runtime behaviour of a K8s cluster and detect attacks that were launched against the cluster.

The remainder of this paper is structured as follows. Section 2 addresses related work and positions our work. Section 3 introduces different approaches to model attacks in K8s. Section 4 details the approach using the learning of probabilistic state machine models to detect anomalies in the cluster. Section 5 provides details on the microservice orchestration testbed that we designed, the multi-step attack scenarios deployed, and the creation of a labelled dataset. Section 6 presents the evaluation of the methodology, including the detection results that we obtained. Finally, Section 7 draws our conclusions and points out future work.

2 RELATED WORK

In this section, we focus our attention on the relevant literature in the field of K8s security management and the general approaches, including state-merging algorithms.

2.1 Kubernetes security management

While simple applications may be composed of only a few tens of components, the largest ones, such as those operated by Netflix and Uber, can run hundreds or thousands of containers. Their composition leads to complex interaction patterns that can be a source of security issues. This complexity is well understood by professionals, who require a thorough security assessment of both K8s clusters and microservice behaviours.

By limiting the scope of our analysis to K8s, different approaches exist to monitor the run phase of a microservice lifecycle, with the objective to detect anomalous behaviours at runtime. Application security in runtime is very different from the build and deployment stages. Dynamic analysis is in charge of assessing the security level in a K8s cluster running in production. Compared to the analysis of monolithic applications, a microservice analysis must monitor a large number of services, each one of them running on different clusters possibly hosted on heterogeneous application platforms. Tools in this category also allow the user to define security policies, and prevent specific vulnerabilities to be exploited, providing networking segmentation and access control. Some techniques only rely on a set of signatures, while hybrid techniques use both signatures and Machine learning (ML) applied to logs, traffic, or both.

As a summary, Table 1 lists the main tools for performing dynamic analysis in a K8s cluster. Falco [23] detects unexpected application behaviour and alerts on threats at runtime as defined by a customizable set of rules. Aqua Container Security Platform [3] monitors orchestration at runtime and includes a configurable firewall. kAudit [2] and kArt perform log analysis to spot unusual or suspicious network behaviour. A firewall is embedded to allow the collection of traffic for policy compliance and anomalous behaviours detection based on ML. Cilium [9] observes network security, to enforce network policies, networking segmentation, load balancing or security policies at the kernel level. Calico cloud [25] provides intrusion and malware detection. Qualys Cloud Platform [21] provides visibility over running services through the collection of communication data. It detects anomalous container behaviours, file access, or process activity.

Such tools have a number of shortcomings that reduce their effectiveness. First, the effectiveness of signature-based tools depends on the completeness of their definition and they may not detect zero-day attacks whose signature is not already known. Also, the commercial tools using ML are not open-source and the methods they use are quite opaque.

2.2 General approaches

Several approaches exist in the literature to detect malicious behaviours in microservice architectures, including signature-based approaches, ML approaches, and state-merging algorithms.

2.2.1 Signature-based approaches. Dynamic analysis in the microservice architecture can be performed by monitoring systems,

Tool	Company	Open-source	Type	Use eBPF
Falco [23]	Sysdig	Yes	Signatures	Yes
Container Security Platform [3]	Aqua	No	Signatures and ML	No
kAudit kArt [2]	Rapid7 (Alcide)	No	Signatures and ML	No
Cilium [9]	Isovalent	Yes	L7 policy enforcement	Yes
Calico cloud [25]	Tigera	Yes	Signatures and ML	Yes
Qualys Cloud Platform [21]	Qualys	No	Signatures and ML	No

Table 1: Classification of state-of-the art tools for dynamic analysis.

similar to intrusion detection systems (IDS) that monitor all traffic exchanged by microservices for signs of malicious activity or security policy violations. The signature-based analysis is performed for identifying known threats, employing a set of signatures for known threats, and their connected Indicators of Compromise (IOCs). However, these tools are not specific to the microservice world, and usually, they do not consider the specifics of a K8s deployment, e.g., network policies, and pod security policies, which can be beneficial to provide a context to the analysis.

2.2.2 ML approaches. On the other hand, ML-based approaches provide anomaly detection either via neural network or state machines approaches that can be effectively applied to learn base characteristics and identify risky deviations in behaviour. In recent years, the utilisation of neural networks has been used to detect anomalies occurring during runtime. However, it is not clear how to explain the classification decisions made by the neural network.

2.2.3 State-merging algorithms. State-merging algorithms first construct a Prefix Tree Acceptor (PTA) and then iteratively go over the states of the PTA to check which states can be merged to form a smaller state machine model. Different heuristics can be used to compute the similarities between states and to decide which similar states should be merged. There exist several tools that use state-merging algorithms to learn state machine models.

CSight is a tool that uses state-merging algorithms to a learn state machine model from partial log data gathered from concurrent systems. The state machine model is used to model the behaviour of a distributed system [4].

Flexfringe is another tool that uses state-merging algorithms to learn state machine models from input data. It provides a list of heuristics that can be used to decide which states should be merged. The tool allows the user to use their own heuristic so that they can compare the performance of their own heuristic to the one that is already provided [26].

MINT uses state-merging algorithms to learn extended state machine models from input data. The difference between an extended state machine model and the definition that we provided in this paper is that a transition would only occur on an extended state machine model if several conditions are satisfied instead of one single condition. MINT also uses Genetic Programming to reduce the number of transition computations between the states during the learning process [29].

Several works proposed different state-merging algorithms; Recently, Mediouni et al. provided an improvement on the RTI algorithm that was proposed in [27] and with this improvement, the algorithm can learn more accurate models [13]. Pastore et al. proposed the TkT algorithm that computes k number of transitions for each state s from the initial state machine model. The k transitions are considered a future transition for a state s and if two states have the same k transitions, these states are then merged together [20]. Vodenčarević et al. proposed an algorithm that learns a hybrid state machine model by taking different aspects of a system into account during the learning process [28].

2.3 Our contribution

With respect to related works, our methodology can be assimilated to using state-merging algorithms to learn behavioural models from log data gathered from the runtime of microservice applications. Our models can be used to gather insights, context and explanations on why a particular behaviour has occurred during the runtime of a system. This can be crucial for a system administrator that needs to understand whether their system is being attacked or that there is a bug in their system. Additionally, we show that we can learn a state machine model that models the runtime behaviour of a system with a microservice architecture. Furthermore, our approach uses open-source tools, which gives transparency to its inner working. The overall objective is to be able to reconstruct a baseline interaction model at runtime for the microservice applications, and then interpret deviations from the model as anomalies and potential attacks. Finally, we provide a new labelled dataset, containing both malicious and normal network data, that can be used by researchers to evaluate their methods on data that is collected from a K8s setup.

3 THREAT MODELLING IN KUBERNETES

Microservice architectures including K8s are composed of hundreds of containers with complex interaction patterns that can be a source of security issues. Different models have been designed to provide a taxonomy of the threats that may target such platforms.

3.1 Microservices architectures

In a microservices architecture, the starting point of an attack may be achieved by exploiting a misconfiguration or a vulnerability on the orchestration platform (e.g., K8s), or compromising a vulnerability within an application (e.g., in a container). In K8s, platform-related vulnerabilities include a vulnerable cluster configuration, a kubelet's unauthenticated, an attacker joining as a fake worker node, Role-Based Access Control (RBAC) issues, vulnerable network endpoints, or vulnerable service tokens. The attacker got access to the network and can then move within the cluster. Application-related vulnerabilities include misconfigured Docker, malicious Docker images, and vulnerable software applications within a pod. An application that has been compromised then leads to a foothold into the container.

With the functioning of traditional networks in mind, intuitively, such attacks are supposed to remain confined to the initially compromised pod as network-security measures are in place. Yet, exploiting the connectivity of K8s components and the software isolation of resources, the cluster may still be compromised.

3.2 Kubernetes threat matrices

These sophisticated attacks usually consist of several steps that may require the detection of different indicators (e.g., malicious network traffic, CPU overload, mounting sensitive directories) to identify specific attack patterns. As of today, the correlation of data from different sources remains complex. The MITRE Adversarial Tactics, Techniques and Common Knowledge (ATT&CK) (adversarial tactics, techniques, and common knowledge) framework is an up-to-date database of attack techniques grouped by objectives known as *tactics*. The ATT&CK framework, originally generic, has been specialised for K8s, highlighting multi-stage cyber-attack patterns and techniques that attackers can use to infiltrate and perform damage on K8s clusters [15]. The threat matrix is made up of ten distinct families of tactics that an attacker can combine to reconstruct and exploit possible attack paths, e.g., lateral movement and privilege escalation.

Up to our knowledge, the K8s threat matrix has been very rarely used in the literature as a means to efficiently design systems considering real-world attack scenarios. Authors in [16] and [12] introduce several attack scenarios derived from the attack life-cycle introduced in the K8s threat matrix.

The Financial Services User Group (FSUG) from the Cloud Native Computing Foundation (CNCF) proposed a threat modelling in a K8s cluster with attack trees [6]. From their analysis, different end goals from an attacker have been identified:

- *Establishing foothold and persistence in the cluster.* The goal is to maintain persistence in a K8s system, by compromising the platform and ensuring an ability to persist without detection. Different levels of resilience exist, ranging from none, resilience to container restart, pod deletion, node restart, and entire cluster recycling;
- *Causing a denial of service towards the cluster.* The attacker aims to exhaust computing resources in the cluster, disrupt networking, or blackhole application traffic;
- *Running a malicious code in workload.* The attacker aims to inject malicious code into a container, deploy poisoned container images, or write new workloads into the etcd server;
- *Reading sensitive data.* The goal is to read or delete sensitive data, e.g., dumping the host memory, reading in the pod cache, or reading from the file system on the K8s host.

4 METHODOLOGY

The novel state anomaly detection approach presented in this work learns a state machine model from the NetFlow data collected from the K8s cluster. As it is an open challenge for using this approach on microservice architectures, we formulate the following research questions:

- *RQ1: How can state machine models be used for anomaly detection on a K8s cluster?*
- *RQ2: How effective are the state machine models for detecting anomalies on a K8s cluster?*

We first answer *RQ1* in the following subsections.

4.1 State-Machine Learning

For this work, we use Flexfringe to learn Probabilistic Deterministic Finite Automaton (PDFA)s from the NetFlow data gathered from a K8s cluster as this framework provides us with the ability to use different state-merging heuristics for the learning of PDFAs. The goal is to use Flexfringe to learn a PDFA that models the normal behaviour of the K8s cluster. Any large deviation in the behaviour is considered as an anomaly and an alarm is raised.

4.1.1 Probabilistic State Machines. State machines or automata are mathematical models that are used to model sequential behaviour and they can be either deterministic or non-deterministic. In the former one, the automaton can only switch to the next state if a symbol a_i has been read from the input; thus an automaton cannot change state if no symbol has been read from the input. Furthermore given the current state q_i of the automaton and the next input symbol a_{i+1}, there is a unique next state q_{i+q} that the automaton will switch to. On the contrary, a non-deterministic automaton can switch to another state if no symbol has been read from the input and the automaton can switch to multiple different states when the next input symbol has been read. Determinism can be important when we are modelling sequential behaviour as deterministic models are easier to interpret and it is much more efficient to compute the transitions and probabilities on these models.

In this work, we use a specific variant of a state machine, namely the probabilistic state machine which is also known as a PDFA. A PDFA is a tuple $\mathcal{A} = \{\Sigma, Q, q_0, \delta, S, F\}$, where:

- Σ is a finite alphabet
- Q is a finite set of states
- $q_0 \in Q$ is a unique start state
- $\delta : Q \times \Sigma \rightarrow Q \cup \{0\}$ is the transition function
- $S : Q \times \Sigma \rightarrow [0, 1]$ is the symbol probability function
- $F : Q \rightarrow [0, 1]$ is the final probability function, such that $F(q) + \sum_{a \in \Sigma} S(q, a) = 1$ for all $q \in Q$

Given a sequence of symbols $s = a_1, \ldots, a_n$, a PDFA can be used to compute the probability for the given sequence: $\mathcal{A}(s) = S(q_0, a_1) \cdot S(q_1, a_2) \ldots S(q_{n-1}, a_n)F(q_n)$, where $\delta(q_i, a_{i+1}) = q_{i+1}$ for all $0 \leq i \leq n - 1$. As an example, take the PDFA that is shown in Figure 1. The probability for the sequence $\mathcal{A}(aaab) = 0.75 \cdot 0.80 \cdot 0.80 \cdot 0.20 = 0.096$. A PDFA is called probabilistic because it assigns probabilities based on the symbols S and final F probability functions. It is called deterministic because the transition function δ (and hence its structure) is deterministic. The transition function is extended with a null symbol, representing that the transition does not exist, thus a PDFA model can be incomplete. A PDFA computes a probability distribution over Σ^*, i.e., $\sum_{s \in \Sigma^*} \mathcal{A}(s) = 1$. A PDFA can also be defined without final probability function F, in that case it computes probability functions over Σ^n, i.e., $\sum_{s \in \Sigma^n} \mathcal{A}(s) = 1$ for all $1 \leq n$.

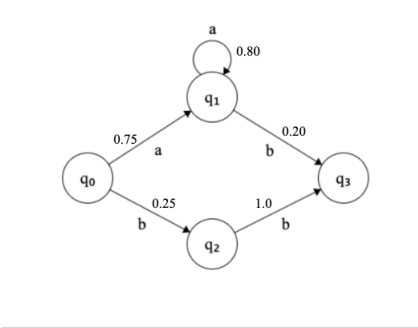

Figure 1: An example of a probabilistic state machine. Notice that each transition has a probability for it to occur and the sum of all probabilities of all outgoing transitions for a given state q_i is equal to one.

4.1.2 Learning State Machine Models with State-merging Algorithms. One effective method for learning probabilistic state machines in the passive-learning approach is the utilisation of the state merging heuristic method [10]. This method essentially starts with a state machine that has a shape of a tree, also called a PTA. The algorithm then iteratively goes over the states of the PTA and does several comparisons between the states. A score is given for each comparison and this score is used to merge similar states, forming a smaller state machine. This process repeats until the algorithm cannot find similar states that can be merged. Figure 2 shows an example of a merge that could occur in a PTA.

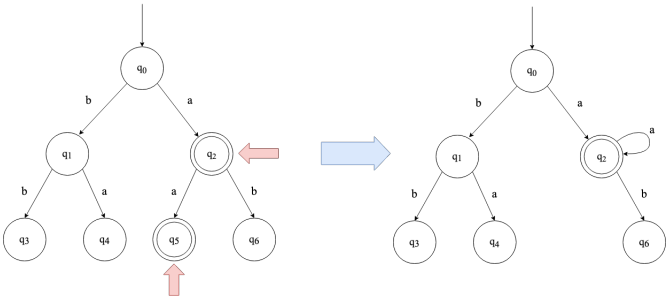

Figure 2: An example of merge performed a PTA.

4.2 Feature Selection & NetFlow data Encoding (Step 1)

For the learning of a PDFA, we have selected three features that we used to create the traces, namely: the protocol that was used in the flow, the total number of bytes sent within the flow and the duration of the flow. We have selected these three features as we consider these three features to best describe the communication behaviour between the microservice applications and our approach does not use any service-dependent information (e.g. source/destination ports used by a service) to learn a model, making our approach more generic. Furthermore, our approach is less privacy intrusive as we try to use as few as possible features for the learning of our models.

As we are using Flexfringe as our framework for learning of PDFA, we need to transform the raw NetFlow data into the corresponding input format of Flexfringe i.e. traces. Each flow also needs to be transformed into a single symbol. To create traces from the NetFlow data, a fixed-length sliding window can be used to create multiple traces. To transform a flow into a single symbol, it is possible to concatenate the raw values of three selected features together to form a single symbol that has the following form: *PROTOCOL_BYTES_DURATION*. However, using the raw values would lead to a very large PTA as the alphabet would be very large and this also increases the learning time. To reduce the learning time, we use three encoding methods: percentile encoding, frequency encoding and contextual frequency encoding. The encoding methods work as follows:

4.2.1 Percentile Encoding. For the percentile encoding, we take each non-categorical feature (number of bytes sent within a flow and the duration of a flow) and divide its space into bins. The encoding for a given non-categorical feature of a given flow would be the bin number that the feature value falls into.

4.2.2 Frequency Encoding. For the frequency encoding, we use a threshold to decide whether a feature value occurs very frequently or not. If it is above the threshold, it is considered a frequently occurring value and we give the value a unique encoding label. If it is below the threshold, it would then be put together with other infrequent feature values and then the space is then further divided using percentiles. The drawback of using percentiles is that frequency information of the feature values is lost when the space is divided into bins. Infrequent values could be put together with frequent values and this causes the model to treat these values the same and it could cause the model to treat these infrequent values as frequently occurring values. This could lead the model to make a wrong prediction. Thus we have decided to use this encoding that takes frequency information into account when we are encoding the feature values.

4.2.3 Contextual Frequency Encoding. The drawback of the frequency encoding is that it would create a very large alphabet as there could be many feature values that have a frequency that exceeds the threshold that we set. Consequently, this would create a very large PTA and thus also increases the learning time. To resolve this issue, we use an encoding that computes the counts of the next and previous feature values for each of the non-categorical feature values. The counts are then used to create a matrix and the rows are then clustered using KMeans. The clustering labels are then the encoding for the given feature. The intuition of this method is that we can use the past and future information of a flow to compute the contextual information for which a feature value occurs. Similar feature values would occur in a similar context and would therefore be clustered together, and thus also have the same encoding. With this method, we use the frequency information of the feature values and we have control over the size of the alphabet.

4.3 Anomaly Detection using PDFA (Step 2)

Once a PDFA is learned from the NetFlow data, we can use it to compute likelihood probabilities for new traces that have been gathered during the runtime of the K8s cluster. Whenever a trace

has a very low likelihood probability, it means that the behaviour of the model deviates a lot from what is seen during training and an alarm would be raised. This trace will be then considered an anomaly.

5 CLUSTER ENVIRONMENT SETUP

The state-of-the-art lacks representative datasets and relevant runtime examples, mostly due to the privacy and operational specificity of deployed systems in operation. Therefore, we designed an open-source microservice orchestration testbed to collect real-world traffic and evaluate our methodology. Multi-step attack scenarios were then deployed, exploiting vulnerabilities of the platform or the applications. We generated a labelled dataset with mixed background traffic generated by benign users and malicious traffic collected from the generation of the attack scenarios.

5.1 Kubernetes cluster setup

We deploy a K8s testbed that acts as a playground on which experiments can be performed, which is easy to replicate on different platforms and adapt to different hardware requirements. The practical testbed is built on Vagrant and Ansible for reproducibility reasons, where the above scenarios can be replicated through containers and Virtual Machines (VMs). VMs are created and deployed created from a host machine in a private network, not accessible from the internet. VMs can, however, reach the Internet via a Network Address Translator (NAT).

The testbed is a single-cluster setup, composed of one coordinator and at least one worker node (three in our setting), with resource isolation performed by K8s namespaces. The applications aimed at the users are deployed on two "development" and "production" namespaces: they contain the same set of applications but with different levels of security, typically more restricted for the production namespace. Figure 3 provides an overview of the multi-cluster setup and its main components.

The *kube-system* namespace hosts core K8s services, such as the DNS and the Application Programming Interface (API) server. The *default* namespaces host the objects created without specifying the namespace. By default, resources hosted in these two namespaces are accessible from any other namespace. Four additional K8s namespaces were deployed in the cluster:

- *Storage*: providing distributed storage through Longhorn and persistent storage through the NFS server;
- *Development*: three applications – a Joomla blog (with MySQL database), an OpenSSH server, and a guestbook (with a Redis leader, Redis follower, and front end);
- *Production*: containing the same applications as the development namespace;
- *Monitoring*: Elastic stack composed of Elasticsearch, Kibana and Beats agents (Packetbeat, Metricbeat, Auditbeat, and Filebeat).

5.2 Description of the attack scenarios

We propose a scenario-based view that illustrates different goals that an attacker could achieve in a K8s cluster. Each attack scenario is fragmented into several steps to follow the K8s threat matrix

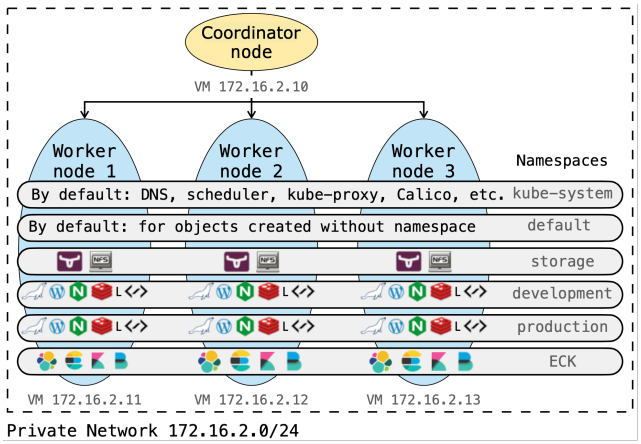

Figure 3: Application deployment on top of the Kubernetes testbed.

pattern. Figure 4 introduces three sample attack scenarios related to each category in the K8s threat matrix.

***Scenario 1:* Establishing foothold and persistence.** This scenario illustrates how an attacker can establish sophisticated pod persistence by exploiting a Remote Code Execution (RCE) vulnerability in a web application and getting a reverse shell. The guestbook application, composed of the front end and the Redis leader and followers, is exploited.

(1) *Initial access*: discovering the pages of the web application and potential vulnerabilities (`dirb`);
(2) *Execution*: exploiting an RCE to open a reverse shell on the guestbook frontend;
(3) *Discovery*: fingerprinting the type of system and environment, discovering secrets mounted in the container, and getting the list of actions and permissions of the attacker based on its service account, e.g., get/list/exec pods located under the same namespace of the currently compromised pod.
(4) *Lateral Movement*: the attacker uses this information to move to another pod that has root privileges.
(5) *Privilege Escalation*: deploying a pod to own the node, e.g., enabling the "`hostPID`" parameter to remove the isolation and the "`nsenter`" command to give low-level access to handle the Linux kernel namespace subsystem. The attacker then breaks out of the container and gains root in the host.
(6) *Persistence*: two variants corresponding to different persistent levels are deployed. First, a daemon set resilient to container restart but not to deployment deletion is created; then a Docker container is deployed outside the cluster, thus resilient to the entire cluster recycling.

***Scenario 2:* Denial of Service (DoS) towards the cluster.** This scenario illustrates a CPU DoS via the use of a vulnerable OpenSSH server. The front end of the guestbook has been built from a Docker image based on Ubuntu 16.04. This Ubuntu version includes OpenSSH v7.2, affected by *CVE-2016-6515* [18] that causes the CPU to overload as it does not limit password lengths for password authentication.

Initial Access	Execution	Persistence	Privilege Escalation	Defense Evasion	Credential Access	Discovery	Lateral Mouvement	Collection	Impact
Using Cloud credentials	Exec into container	Backdoor container	Privileged container	Clear container logs	List K8s secrets	Access the K8s API server	Access cloud resources	Image from a private registry	Data destruction
Compromised images in registry	Bash/cmd inside container	Writable hostPath mount	Cluster-admin binding	Delete K8s events	Mount service principal	Access Kubelet API	Container service account		Resource Hijacking
Kubeconfig file	New container	Kubernetes Cronjob	hostPath mount	Pod/container name similarity	Access container service account	Network Mapping	Cluster internal networking		Denial of Service
Application vulnerability	Application exploit (RCE)	Malicious admission controller	Access cloud resources	Connect from proxy server	Application credentials in config files	Access Kubernetes Dashboard	Applications credentials in config files		
Exposed Dashboard	SSH server running inside container				Access managed identity credential	Instance Metadata API	Writable volume mounts on the host		
Exposed sensitive interfaces	Sidecar injection				Malicious admission controller		Access Kubernetes Dashboard		
							Access tiller endpoint		
							CoreDNS poisoning		
							ARP poisoning and IP spoofing		

Figure 4: Threat matrix for K8s including the three multi-step attack scenarios.

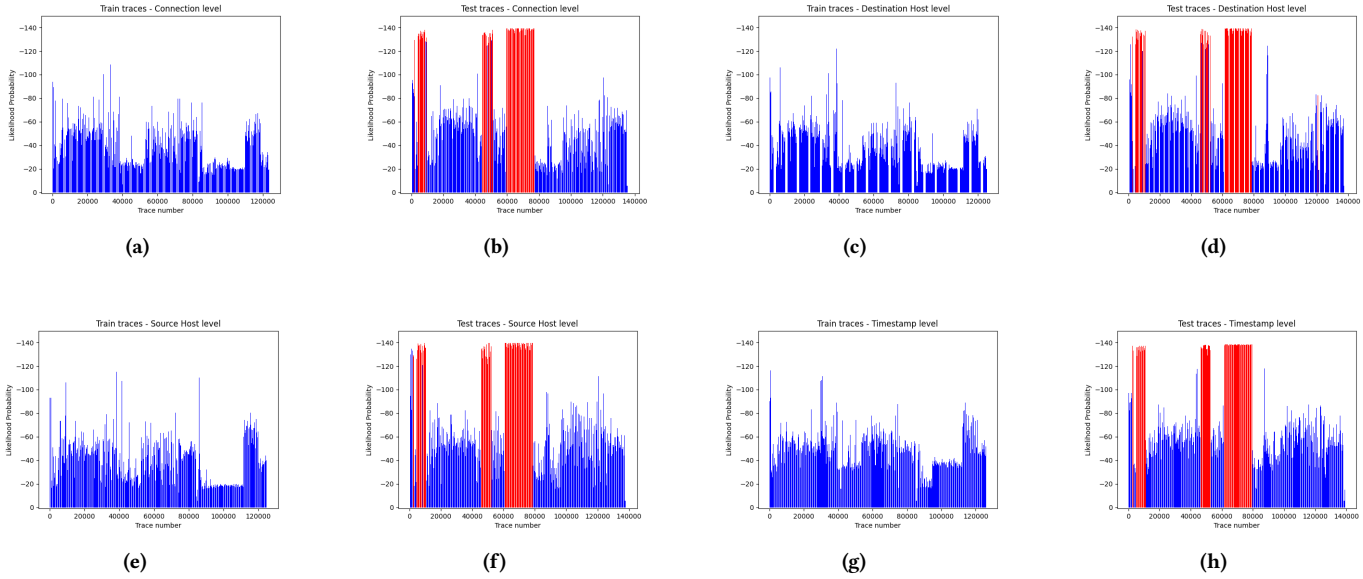

Figure 5: Likelihood probability plots of the traces created using the contextual frequency encoding. The higher the peak, the lower the likelihood probability of a given trace. Subfigures *a* and *b* present the probabilities of the train and test traces sorted by the connections, *c* and *d* present the probabilities of the train and test traces sorted by destination hosts, *e* and *f* present the probabilities of the train and test traces sorted by source hosts, and remaining two subfigures (*g* and *h*) present the probabilities of the train and test traces sorted by timestamp.

(1) *Initial access*: attacker able to communicate remotely with the pod hosting the vulnerable OpenSSH server.
(2) *Execution*: the remote attacker simultaneously sends 1000 requests with very long passwords to the front end pod, to cause a CPU overload.
(3) *Impact*: the CPU is overloading as it receives many simultaneous requests. With the K8s Metrics Server [22], we identify that the CPU overload comes from the front end pod running the OpenSSH server. The memory of this pod goes from 10 to

150 MiBytes and from 5 to 550m CPU cores, only 5 seconds after launching the attack.

Scenario 3: **Malicious code in workload.** This scenario illustrates how an attacker can open a container shell through an RCE vulnerability in a Joomla application. The attacker then deploys malicious code in workload, e.g., a bitcoin miner in the cluster.

(1) *Initial access*: exploiting the *CVE-2015-8562* vulnerability [17] on Joomla, which allows remote attackers to conduct PHP object injection attacks and execute arbitrary PHP code via the HTTP User-Agent header;
(2) *Execution*: RCE attack deployed by unauthenticated remote attackers to gain instant access to the target server on which the vulnerable Joomla core version is installed.
(3) *Discovery*: fingerprinting the type of system and environment and listing the running pods in the cluster.
(4) *Lateral Movement*: accessing a privileged pod.
(5) *Execution*: the attacker can deploy a bitcoin miner in the cluster once it gained access to a root pod.

5.3 Data collection

To compensate for the lack of representative datasets, we generated our own labelled dataset collected on the microservice orchestration testbed. The objective is to collect both background traffic, produced by benign users with realistic behaviours, and malicious traffic simultaneously. A data collection session has been organised for two hours: during the first hour, thirty benign users were using the applications on the K8s testbed, generating background traffic, and during the second hour, the attack scenarios were deployed on the cluster while benign users were still using the applications. Then, the training set is composed of traces collected during the first hour, and the test set is composed of traces collected afterwards. Traces were manually labelled depending on the IP addresses of users and the timestamps of attacks. The dataset is publically available at [1].

6 EXPERIMENT RESULTS

In this section, we present the performance results of our novel anomaly detection method that uses a state machine model. The performance results that are presented in this section form the answer to *RQ2*.

6.1 Performance Results

The state machine model was trained on the NetFlow data that were gathered before the attacks were launched against the K8s cluster. When creating the traces, we have sorted the NetFlow data in four different manners: connection, source host, destination host and timestamp of the flows. The intuition for sorting the NetFlow data in different manners is that some behaviour might only be seen when a particular sorting level is used. We have computed the performance results of our approach using each of the different encodings that we have mentioned in Section 4. For comparison, we have trained an isolation forest with 100 estimators on the same training data that was used to learn our state machine model. We have used the isolation forest that is provided by the scikit-learn machine-learning package[4].

[4]https://scikit-learn.org/stable/modules/generated/ sklearn.ensemble.IsolationForest.html

Table 2 presents the performance results of our approach using the percentile encoding, frequency encoding and contextual frequency encoding, and the performance results of the isolation forest that we have trained. Based on the results that are shown in the table, our approach performs much better than an isolation forest, achieving a balanced accuracy of 99.2% and a F_1 score of 0.982.

6.2 Likelihood Probability Plots

In our approach, we use the state machine model to compute likelihood probabilities for the network traces and use the probabilities to assess whether there is a large deviation in the behaviour that was learned during training. Figure 5 presents the plots of the likelihood probabilities of train and test traces for each sorting level. We only present the plots that were generated using the state machine model learned with the contextual frequency encoding as this encoding method gave the best results. For the sake of simplicity, we have coloured all malicious traces in red and all benign traces in blue. From the probabilities plots, we can see that the is a large deviation in the network traces whenever an attack has been launched against the K8s cluster. We also see that the normal network traces produce similar probabilities as what was seen during training. From these plots, we can see why our model can detect attacks very well; visually, it is very obvious that some strange behaviour is occurring in the network of the cluster.

Table 2: Performance results of our approach

Method	Encoding	Sorting Level	Balanced Accuracy	F_1 Score
Our Approach	Percentile	Connection	0.888	0.714
		Source Host	0.883	0.702
		Destination Host	0.884	0.704
		Timestamp	0.885	0.715
	Frequency	Connection	0.828	0.689
		Source Host	0.817	0.675
		Destination Host	0.815	0.673
		Timestamp	0.796	0.655
	Contextual Frequency	Connection	0.991	0.975
		Source Host	0.988	0.967
		Destination Host	0.989	0.972
		Timestamp	0.992	0.982
Isolation Forest	N/A	Connection	0.374	0.0006
		Source Host	0.375	0.0006
		Destination Host	0.373	0.0006
		Timestamp	0.373	0.0006

7 CONCLUSION & FUTURE WORK

In this work, we presented a novel anomaly detection method that uses a state machine model to detect anomalies within a K8s cluster. To the best of our knowledge, this work is the first that tries to apply state machine models to microservice architectures. The state machine model was learned from NetFlow data gathered from the K8s cluster when it was used normally. We have set up an experiment in which we launched three different attack scenarios

on the cluster. Using our state machine approach, we managed to detect all three attack scenarios, achieving a balanced accuracy of 99.2% and a F_1 score of 0.982.

For our future research directions, we would like to use other types of log data gathered from the cluster to learn our models as currently we only use the NetFlow data. Different types of log data contain different types of information that are useful to learn a more accurate model. We can assess whether using another type of log data or a combination of different log data can help us learn a more accurate model.

Furthermore, we also want to use other datasets to evaluate whether our approach is generalisable and can be applied to other types of systems as for this work we have only used data that were collected from a K8s cluster to run our experiments. This future research direction also allows us to compare our approach to other existing approaches.

ACKNOWLEDGMENTS

This work is funded under the Assurance and certification in secure Multi-party Open Software and Services (AssureMOSS) Project, (https://assuremoss.eu/en/), with the support of the European Commission and H2020 Program, under Grant Agreement No. 952647.

REFERENCES

[1] [n.d.]. AssureMOSS NetFlow Dataset for Anomaly Detection. https://surfdrive.surf.nl/files/index.php/s/CV2fOFlbtsADX9z
[2] AlcideIO. [n.d.]. KAudit. https://github.com/alcideio/kaudit
[3] aqua. [n.d.]. Your Cloud Native Applications Secured From the Start. https://www.aquasec.com/
[4] Ivan Beschastnikh, Yuriy Brun, Michael D. Ernst, and A. Krishnamurthy. 2014. Inferring models of concurrent systems from logs of their behavior with CSight. *Proceedings of the 36th International Conference on Software Engineering* (2014).
[5] Edmund M. Clarke, O. Grumberg, and D. Peled. 1999. *Model checking*. MIT Press.
[6] CNCF. [n.d.]. K8s Attack Tree - Summary. https://github.com/cncf/financial-user-group/tree/master/projects/k8s-threat-model (Accessed on: 30/06/2021).
[7] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. DeepLog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the ACM Conference on Computer and Communications Security*. Association for Computing Machinery, 1285–1298. https://doi.org/10.1145/3133956.3134015
[8] Malte Isberner, Falk Howar, and Bernhard Steffen. 2015. The Open-Source LearnLib: A Framework for Active Automata Learning. In *Computer Aided Verification*. https://doi.org/10.1007/978-3-319-21690-4_32
[9] Isovalent. [n.d.]. Cilium - eBPF-based Networking, Observability, Security. https://cilium.io/
[10] Kevin J. Lang, Barak A. Pearlmutter, and Rodney A. Price. 1998. Results of the abbadingo one DFA learning competition and a new evidence-driven state merging algorithm. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 1433. Springer Verlag, 1–12. https://doi.org/10.1007/bfb0054059
[11] David Lee and Mihalis Yannakakis. 1996. Principles and methods of testing finite state machines - A survey. *Proc. IEEE* 84, 8 (1996), 1090–1123. https://doi.org/10.1109/5.533956
[12] Ahmed Massoud. 2021. *Threat Simulations of Cloud-Native Telecom Applications*. Master's thesis. Aalto University.
[13] Braham Lotfi Mediouni, Ayoub Nouri, Marius Bozga, and Saddek Bensalem. 2017. Improved Learning for Stochastic Timed Models by State-Merging Algorithms. In *NASA Formal Methods*, Clark Barrett, Misty Davies, and Temesghen Kahsai (Eds.). Springer International Publishing, Cham, 178–193.
[14] Weibin Meng, Ying Liu, Yichen Zhu, Shenglin Zhang, Dan Pei, Yuqing Liu, Yihao Chen, Ruizhi Zhang, Shimin Tao, Pei Sun, and Rong Zhou. 2019. Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs. In *IJCAI International Joint Conference on Artificial Intelligence*, Vol. 2019-Augus. 4739–4745. https://doi.org/10.24963/ijcai.2019/658
[15] Microsoft. 2020. Threat matrix for Kubernetes. https://www.microsoft.com/security/blog/2020/04/02/attack-matrix-kubernetes/
[16] Francesco Minna, Agathe Blaise, Filippo Rebecchi, Balakrishnan Chandrasekaran, and Fabio Massacci. 2021. Understanding the Security Implications of Kubernetes Networking. *IEEE Security & Privacy* 19, 5 (2021), 46–56. https://doi.org/10.1109/msec.2021.3094726
[17] MITRE. [n.d.]. CVE-2015-8562. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-8562 (Accessed on: 14/09/2021).
[18] MITRE. [n.d.]. CVE-2016-6515. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-6515 (Accessed on: 14/09/2021).
[19] Sheraz Naseer, Yasir Saleem, Shehzad Khalid, Muhammad Khawar Bashir, Jihun Han, Muhammad Munwar Iqbal, and Kijun Han. 2018. Enhanced network anomaly detection based on deep neural networks. *IEEE Access* 6 (2018), 48231–48246. https://doi.org/10.1109/ACCESS.2018.2863036
[20] Fabrizio Pastore, Daniela Micucci, and Leonardo Mariani. 2017. Timed k-Tail: Automatic Inference of Timed Automata. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 401–411. https://doi.org/10.1109/ICST.2017.43
[21] Qualys. [n.d.]. Qualys Cloud Platform. https://www.qualys.com/cloud-platform/
[22] Kubernetes SIGs. [n.d.]. Kubernetes Metrics Server. https://github.com/kubernetes-sigs/metrics-server (Accessed on: 14/09/2021).
[23] sysdig. [n.d.]. The Falco Project - Cloud-Native runtime security. https://falco.org/
[24] Tuan A. Tang, Lotfi Mhamdi, Des McLernon, Syed Ali Raza Zaidi, and Mounir Ghogho. 2016. Deep learning approach for Network Intrusion Detection in Software Defined Networking. In *Proceedings - 2016 International Conference on Wireless Networks and Mobile Communications, WINCOM 2016: Green Communications and Networking*. 258–263. https://doi.org/10.1109/WINCOM.2016.7777224
[25] Tigera. [n.d.]. Get Calico up and running in your Kubernetes cluster. https://projectcalico.docs.tigera.io/getting-started/kubernetes/
[26] Sicco Verwer and Christian A. Hammerschmidt. 2017. flexfringe: A Passive Automaton Learning Package. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 638–642. https://doi.org/10.1109/ICSME.2017.58
[27] Sicco Verwer, Mathijs Weerdt, and Cees Witteveen. 2012. Efficiently identifying deterministic real-time automata from labeled data. *Machine Learning* 86 (03 2012), 295–333. https://doi.org/10.1007/s10994-011-5265-4
[28] Asmir Vodenčarević, Hans Kleine Bürring, Oliver Niggemann, and Alexander Maier. 2011. Identifying behavior models for process plants. In *ETFA2011*. 1–8. https://doi.org/10.1109/ETFA.2011.6059080
[29] Neil Walkinshaw, Ramsay Taylor, and John Derrick. 2016. Inferring extended finite state machine models from software executions. *Empirical Software Engineering* 21, 3 (2016), 811–853. https://doi.org/10.1007/s10664-015-9367-7