Adaptive Controller to Identify Misconfigurations and Optimize the Performance of Kubernetes Clusters and IoT Edge Devices

Areeg Samir[0000−0003−4728−447X] and Håvard Dagenborg[0000−0002−1637−7262]

UiT - The Arctic University of Norway, Tromsø
areeg.s.elgazazz@uit.no

Abstract. Kubernetes default configurations do not always provide optimal security and performance for all clusters and IoT edge devices deployed, affecting the scalability of a given workload and making them vulnerable to security breaches and information leakage if misconfigured. We present an adaptive controller to identify the type of misconfiguration and its consequence threat to optimize the system behavior. Our work differs from existing approaches as it is fully automated and can diagnose various errors on the fly. The controller is evaluated in terms of quality and accuracy of identification. The results show that the controller can identify around 90% of the total number of configuration values with a reasonable average identification overhead.

Keywords: Misconfiguration · Threats · Identification · IoTs · Clusters · Markov Processes · Security · Performance.

1 Introduction

Misconfiguration is an incorrect configuration(s) within the parameters of system components (i.e., system clusters, IoT edge devices) that may lead to vulnerabilities and affect system workload and performance at different levels. At the edge level, a misconfigured edge device opens the potential for security breaches. For instance, if an edge device runs with default privileges or the same privileges as the application, vulnerabilities in any system's component can be accidental (e.g., remote SSH open) or intentional (e.g., backdoor in component). At the application level, a misconfigured container (e.g., network port open) allows an attacker to exploit the Docker API port that escalates the attack to other containers and hosts. At the cluster level, misconfigurations in core Kubernetes components (e.g., API server, Kubelet, Kube-proxy) lead to the compromise of complete clusters that cause network latency overheads, CPU throttling, or container to run out of memory. The management of configurations has been explored in the literature [8], [10], [4], [15], [2]. However, the complexity of misconfigurations does not arise only from a large number of configuration parameters, but also from their correlations and dependencies. The paper proposes a real-time misconfiguration identification controller based on the fine-grained configuration type category. The paper is organized as follows. Section II provides a background of some concepts used in the paper. Section III presents related research. Section IV provides examples of misconfigurations that motivate the paperwork. Section V presents the methodology followed to analyze misconfigurations and explains the identification of configuration errors and their threats as a consequence according to the identified configuration error cases. Section VI evaluates the controller and discusses the reported results. Section VII concludes the paper and presents the future direction of the work.

2 Background

Misconfigurations can lead to a variety of security threats and vulnerabilities. This section gives an introduction to the configuration and Hidden Markov Model that can be used to analyze the behaviors of the system components to identify potential threats.

2.1 Configurations

Configurations are a list of entries or parameters, in terms of key-value pairs, a list, and a map, that define the configurations for an object (e.g., cluster, node, pod, container, service, user) and manage its deployment. Configurations are stored in a configuration file that contains basic information about a cluster, and are written in a user-friendly YAML syntax format that is called 'manifest'. The configuration file is stored in version control before being pushed to the cluster to simplify the rollback of a configuration change, aids cluster re-creation and restoration. The configuration file has to contain four main entries, which are apiVersion (i.e., used to create the Kubernetes object), kind (e.g., Pod, Deployment, Service, Job, or DemonSets), metadata (unique properties of an object such as name, namespace, and label entries), and spec (i.e., specification, defines the operation of an object and depends upon the apiVersion). Kubernetes cluster uses configuration files to create an object based on a set of defined configurations. By concentrating on developing YAML configuration management, we can reduce configuration errors and vulnerabilities, resulting in improved cluster security and stability.

2.2 Hidden Markov Model

A Hidden Markov Model (HMM) is a statistical model that is used to describe a system that evolves over time and generates observable data sequences. It is widely applied in various fields, such as security. An HMM consists of two main components: (1) hidden states, which are the underlying, unobservable states of the system that transition from one state to another over time. The system is assumed to be in one of these hidden states at any given time. (2) Observable emissions, which are the observable outcomes associated with each hidden state. These observations are what we can measure or observe, and they provide information about the underlying hidden states. HMMs are often used for threat detection and anomaly detection to identify patterns of behavior that deviate from normal or expected behavior, which could indicate potential threats or attacks. Thus, utilizing the HMM-based detection system provides a comprehensive threat identification strategy specifically for attacks caused by configuration errors.

3 Related Work

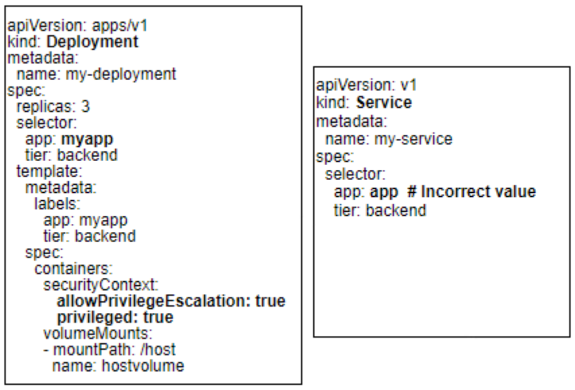
Configuration error analysis is crucial for maintaining the stability, performance, and security of a system. Current frameworks have not focused sufficiently on the essential aspect of effectively handling misconfigurations in edge devices and clusters [4], [22]. Since most tools work with predefined constraint templates, unlike our work, the following techniques lack of an adaptive misconfiguration identification that works with different types of errors, which makes configuration management a challenging task, especially when considering heterogeneous hardware and software stacks in cluster and edge environments.

To optimize and manage the configurations of containers running in a Kubernetes cluster, configuration framework solutions with a focus on performance are presented [23], [2]. These solutions focused on detecting configuration errors by analyzing the source code and generating the configuration check code. Maintenance overhead can occur with large data sets and can be time-consuming in multicomponent applications, especially in scale and load-balance environments. In such complex environments, there are often multiple layers of configuration, including their configuration parameters and interactions, leading to more complexity [23]. For example, configuring network policies involves defining how pods communicate with each other and other endpoints. An incorrect combination of policies can inadvertently block traffic or create security vulnerabilities.

In addition, misconfigurations can have detrimental effects in scenarios where load balancing and resource allocation are critical. For example, setting too high memory or CPU limits for a particular service might cause contention for resources among different services running on the same infrastructure [1]. Incorrect configurations can lead to bottlenecks in data traffic at the network level and open suspicious flows in the system [10]. Rules-based security techniques are used to detect misconfigurations and optimize system performance [5], [9], however, checking every constraint is time-consuming and can lead to more errors. An analysis of misconfiguration helps to detect which parts of the system are associated with configuration parameters. This could be achieved by deriving the specification of the configurations by designing a custom control and data flow analysis targeting the configuration-based code [24], [6], based rule [20], or based inference [25]. However, those ways are highly specialized, as some of them only focus on security, they are not simple to write and maintain, and they are geared towards a host only instead of container images and edge devices, which might result in the occurrence of false positives or false negatives.

4 Motivation Examples

Any configuration error (Misconfiguration) can lead to privilege escalation, containers running as root, and other critical vulnerability issues that have negative consequences on security, efficiency, reliability, and performance. For example, some wireless access points may have outdated or insecure wireless security services enabled (e.g., WEP or WPS) by default. Such standards could allow attackers within range of the device to gain access to the network. Since data are also often transmitted via an insecure protocol (e.g., FTP, HTTP, etc.) by default, some of it may be exposed to an attacker with such access. If credentials or encryption keys are captured, the initial access gained through these default settings could lead to further access to systems within the network or the ability to read encrypted data. For example, suppose that we have a cluster with three nodes that do not act as host control planes. Cluster nodes have some pods and a set of deployed containers with privilege and access control settings, such as privilege access (e.g., *allowPrivilegeEscalation*), as shown in Figure 1. This setting controls whether a process can gain more privileges than its parent process, and it is always true when the container is run as privileged, or has *CAP_SYS_ADMIN*. Here, a user root inside a



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-deployment
spec:
  replicas: 3
  selector:
    app: myapp
    tier: backend
  template:
    metadata:
      labels:
        app: myapp
        tier: backend
    spec:
      containers:
        securityContext:
          allowPrivilegeEscalation: true
          privileged: true
        volumeMounts:
        - mountPath: /host
          name: hostvolume
```

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: app  # Incorrect value
    tier: backend
```

Fig. 1. Setting-Value Dependency Cascading Failure in YAML

container will have the same access as the root on the host system, allowing an attacker with root access in the container to gain access to the nodes, steal the secrets of their running containers, and exploit flaws in the cluster. The severity of the attack is highly rated with a score of 7.0 according to the level of severity of the CVE.

In addition, any incorrect value might create a failure that can be cascaded if the failure in one component or resource setting impacts other dependent components or resources. For example, as shown in Figure 1, the service should reference the *Deployment* with the *selector* field, specifying the labels that correspond to the deployment pod. However, the *selector* field is given an incorrect value. The *selector* field is set to *app: app* in the *Service*, which does not match the value of the label *app: myapp* used in the *Deployment*. This wrong value enables *Service* to incorrectly target deployment pods. Such settings cause service discovery and routing failures as the service cannot correctly track the deployment pod to establish dependency between the two resources.

To identify the type of misconfiguration, we extracted the configuration error settings of the anomalous component only to reduce the complexity of identification, as shown next.

5 The Proposed Approach

This section extends the controller misconfiguration analysis phases (monitor, detection) to identify the type of misconfigurations and their threat consequence based on the output of the detection phase in [17].

5.1 Methodology of Misconfigurations Analysis

To analyze misconfigurations, the controller has 4 main phases, which are monitor, detection [17], identification, and recovery [18]. This paper aims to introduce the identification phase.

We ran the controller on a set of manifest YAML files to inspect configuration errors in edge devices and clusters and sent a warning message for any deviations from the configuration settings; more details are given in [17]. The YAML manifest files were inspected on the basis of Kind and APIVersion keys using Kubernetes utilities. We received warning messages for configuration parameters violating specified rules with approximately 36,709 configuration errors containing around 2090 unique errors from 3000 apps. Many of those errors were simple misconfigurations that could be avoided by developers, such as edge device default settings (e.g., accounts passwords) were not changed, or sensitive medical data being leaked due to the enabling of privilege escalation. We also found several errors which have serious consequences such as the port of the etcd server (Kubernetes database) not encrypted and accessible via plain HTTP, the failure to update software patches, the system network as KubePi was not configured properly or the pod was constantly crashing in an endless loop. Only 1087 apps, among the apps studied, had no configuration errors.

Based on that, we identified the categories of misconfigurations in the manifests and the sensors that were on our device by getting a reference to the sensor service to list all the sensors on the device of type. We focus on analyzing the configurations defined in the application manifest file, including components, enforced permissions, exported attribute values, and intent filters. We focus on the most common misconfigurations that negatively impact system components' workload and cause performance degradation on performance metrics (application domain metrics and system-specific metrics).

The metrics are kept within acceptable limits using dynamic thresholds to avoid degradation of system performance and maintain continuous delivery [19]. We collected data from the metrics into a two-dimensional matrix which formed our data set; the columns refer to the metrics, and the rows refer to the components. Then we derived the relationship between the performance of the components, the workload, and the misconfigurations to detect the misconfigurations for the applications deployed in the hierarchical system settings [17]. The controller detects the hierarchical path(s) that show anomalous behavior (e.g., overload) and tracks the misconfigured components under the constraints of the extracted configurations. We checked the configuration security settings and detected misconfigurations in the vulnerable path. For edge devices, the configuration check was performed on authorized devices that had legal access to the system and were assigned to authorized participants. The detection is based on statistical learning that can detect misconfigurations based on the learned configuration settings and can be used for real-time misconfiguration detection to quickly reduce the negative imprint on the system. The following represents the steps of misconfiguration detection [17] and identification by the controller:

Training phase: During the training phase, the configurations are collected in terms of labeled training sets to train the controller. The controller learns the patterns and relationships between various configuration settings and their corresponding outcomes to characterize correct and incorrect configurations. For example, if a certain setting is known to cause conflicts or performance issues, the controller learns to identify such patterns. **Feature extraction:** Relevant settings are extracted from the configuration data, such as specific settings, parameters, or dependencies. These features provide the input for the controller. **Model training:** The controller is trained using the extracted features to identify the patterns and rules associated with the correct configurations and detect deviations that can indicate errors. **Real-time detection:** Once the controller is trained, it can monitor and analyze the new configuration settings in real time. When a new configuration is encountered, the controller evaluates it against the learned rules and patterns to detect any misconfiguration; more details are given in [17]. **Prompt er-**

ror detection and mitigation: When a configuration error or anomaly is detected, an alert is generated to prompt corrective action as our proposed recovery process in [18]. **Continuous learning and improvement:** The controller continues to learn from new configurations and adapts to changing settings or emerging errors to improve its detection capabilities and accuracy over time. The result of the detection is used to identify the type of misconfiguration to locate its root cause through the configuration error-failure cases and the identification phase, as shown in the following sections. The phase aims to track the dependency between misconfigurations in edge devices and clusters of the system to show its impact on performance and workload, and to demonstrate the impact of misconfigurations on resource vulnerability.

5.2 Configuration Error-Failure Cases

To identify the type of misconfigurations, we focused on identifying common security misconfigurations (Errors) in Kubernetes, Azure, and Docker Swarm that negatively cause anomalous workload (Fault) and dramatically saturate monitoring metrics (Failure) of system components. We achieved this by focusing on misconfigurations reported in 2023 and 2022 by the CVE, NIST, OWASP, Fairwinds, ENSA, CIS Docker, and Kubernetes benchmarks. We targeted those benchmarks, as they provide a systematic analysis that addresses key architectural vulnerabilities and platform dependencies of such tools. The benchmark used artifacts reported by DevOps, Azure, Kubernetes, Google Kubernetes Engine, Docker Swarm, Amazon Elastic Kubernetes Service, Oracle, Google Cloud, Microsoft, and Alibaba practitioners. The benchmarks go through two stages of consensus review and evaluation of the results of security misconfigurations that alter the dependability, security, and cost of more than 150,000 workloads from hundreds of businesses.

We classified the types of misconfiguration into cases and we linked hidden settings of configuration errors and their faults in the system under observation to their observed failures, which are sequences of observations emitted by system resources. The type of misconfiguration falls under three main error failure cases: (1) IoT edge failure is due to a device failure that occurred during run-time or during provision and deployment. Edge device misconfigurations were classified as impaired communications, indicating limited communication between the device and the service, or non-sensor data, indicating that a device has communication with the service but only reports partial sensor data. (2) application failure is due to a pod or container failure, and (3) node or cluster failure relates to a core component failure. We used the IEEE Standard Classification for Software Anomalies [7] to analyze observed failures in multiple dimensions: Failure ID (unique identifier for the failure type and its category), Failure Description (describe an observed behavior), Failure Analysis (describe and analyze failure's root-cause), and Failure Severity (in percentage) relating to the system performance and reliability in terms of the objectives that were not met by the observed metrics and benchmarks. We used Key Performance Indicators (KPIs) to determine whether the motoring metrics met the maintenance goals and the system's performance (e.g., resource utilization, latency, response time, network congestion, throughput). The higher the value, the more severe the impact on system performance will be [21], [3].

The following cases refer to observed failures that are either emitted by an administrative operation internal to Kubernetes, Azure, and Docker Swarm or emitted by a trigger external to them as follows:

IoT Edge Cases It refers to IoT edge failure that occurs during run time or during provision and deployment such as:

Case 1: Sudden stop of the edge device. Failure ID: $\mathrm{Conf}_{\mathrm{EC1}}$. Failure Description: The edge device stopped for a specific period (e.g., minutes) after running successfully. The logs indicated that the device failed to connect to the IoT hub via AMQP or WebSocket and that the edge device existed. Failure Analysis: A misconfiguration of the host network prevented the edge agent from reaching the network. The agent attempted to connect over AMQP (port 5671) or WebSockets (port 443) as the edge device runtime set up a network for each module to communicate, either using a bridge network or NAT. Failure Severity: 70%.

Case 2: Empty Configuration File. Failure ID: $\mathrm{Conf}_{\mathrm{EC2}}$. Failure Description: The device has trouble starting the modules defined in the deployment. Only the edge agent is running, but it continually reports empty configuration files. Failure Analysis: The device may have trouble with the resolution of the DNS server name within the private network. Failure Severity: 20%$\sim$30%.

Case 3: Edge Hub Failure. Failure ID: $\mathrm{Conf}_{\mathrm{EC3}}$. Failure Description: The Edge Hub module does not start. Failure Analysis: Some process on the host machine has bound a port to which the edge hub module is trying to bind. The Edge hub maps ports 443, 5671, and 8883 for use in gateway scenarios. The module fails to start if another process has already bound one of those ports. Failure Severity: 20%$\sim$30%.

Case 4: Default Credentials. Failure ID: $\mathrm{Conf}_{\mathrm{EC4}}$. Failure Description: The default accounts/passwords of the edge device are not changed. Failure Analysis: Using vendor-supplied defaults for accounts and passwords could allow attackers to brute-force and gain unauthorized access to the system. Failure Severity: 98%$\sim$99%.

Application Cases It refers to the occurrence of failure due to the failure of a pod or container as follows:

Case 1: Privilege Escalation Flaw and Redeployment Fail. Failure ID: $\mathrm{Conf}_{\mathrm{AC1}}$. Failure Description: Sensitive medical data was leaked. Failure Analysis: An Azure function (e.g., SCM_RUN_FROM_PACKAGE) gave access to the remapped root and allowed privilege escalation to the root level. Failure Severity: 80%$\sim$90%.

Case 2: Privilege Escalation Flaw. Failure ID: $\mathrm{Conf}_{\mathrm{AC2}}$. Failure Description: Sensitive medical data was leaked. Failure Analysis: A docker engine function option (e.g., users-remap) gives access to the remapped root and allows privilege escalation to the root level. Failure Severity: 80%$\sim$90%.

Case 3: Unauthenticated Connection. Failure ID: $\mathrm{Conf}_{\mathrm{AC3}}$. Failure Description: Kubernetes labels are not validated or incorrectly typed. Failure Analysis: Privilege access to Kubelet, which allows unexpected routing from service target selectors. Failure Severity: 40%$\sim$60%.

Case 4: Outdated Package and Flow Unpatched. Failure ID: $\mathrm{Conf}_{\mathrm{AC4}}$. Failure Description: The software is outdated and flaws are unpatched. Failure Analysis: Failure to update software patches as part of the software management process, allowing attackers to inject malicious code into the application. Failure Severity: 80%$\sim$90%.

Case 5: Loop Crash. Failure ID: Conf_{AC5}. Failure Description: The pod is constantly crashing in an endless loop and cannot be started. Failure Analysis: A server cannot load the configuration file due to a typo in a configuration file system. Failure Severity: 80%∼90%.

Core Components Cases It indicates the occurrence of a failure at a node or cluster level.

Case 1: Spike Traffic Received by System. Failure ID: Conf_{CC1}. Failure Description: System services do not work properly and its resources are excessively saturated. Failure Analysis: Distributed Denial of Service (DDOS) attack prevents access to the system network, as KubePi is not configured correctly. Failure Severity: 98%∼99%.

Case 2: Data Leakage. Failure ID: Conf_{CC2}. Failure Description: Sensitive medical data was leaked. Failure Analysis: The deployment of highly sophisticated malware leads to compromise of sensitive medical data. Ingress allowed unauthorized users to access and update all secrets in the cluster. Failure Severity: 98%∼99%.

Case 3: Anonymous Authentication. Failure ID: Conf_{CC3}. Failure Description: Unauthenticated requests can be sent to Kubelet, as its configuration is not set properly, which saturated the system resources. Failure Analysis: The misconfigured Kubernetes core component gave unauthorized access to the entire cluster. Failure Severity: 80%∼99%.

Case 4: Non-Secure Cluster Transmittance. Failure ID: Conf_{CC4}. Failure description: The etcd server port (Kubernetes database) is unencrypted and accessible over plain HTTP. Failure Analysis: The etcd process on the master node exhausts all memory, as the etcd cluster is left without authentication, allowing a DDOS attack to gain unauthorized access to a system. Failure Severity: 80%∼99%.

At the end of this step, misconfiguration description profiles for the cases are created and stored to be used in the identification phase along with the output of the detection phase, which provides the path of the hierarchical anomalous misconfigured components that are affected by specific components.

5.3 Misconfiguration Identification Phase

The controller uses the output of the detection phase as input for the identification phase. For example, $AnomalousPath = \{Cluster > Node_{22} > Node_{23} > Container_{33} > Service_{43}\}$ is a hierarchy anomalous path that is affected by $Node_{23}$ with the vertical level index 2 and horizontal level index 3 in the graph, respectively. On the basis of that, we initialized a model with the configuration settings of the anomalous states and observations obtained. The model is created with a graph length of states $ConfLeng = (Conf_{ij},..,Conf_{Nj})$ and the length of observations $FLeng = \{F_1,..,F_T\}$, which are stored in a matrix $ConfMat[ConfLeng, FLeng]$. To show the dependency between misconfigurations, each $Conf_{ij}$ represents the misconfigured settings that belong to the anomalous state that has vertical i and horizontal j levels. For each $Conf_{ij}$, as shown in Figure 2, we checked the type of misconfiguration, which is hidden from the observer considering the state level in the defined failure-error cases $(Conf_{EC}, Conf_{AC}, Conf_{CC})$.

The configuration settings (key-value pairs) were extracted from the manifest of the anomalous state based on the Kind and API version objects. A state check function

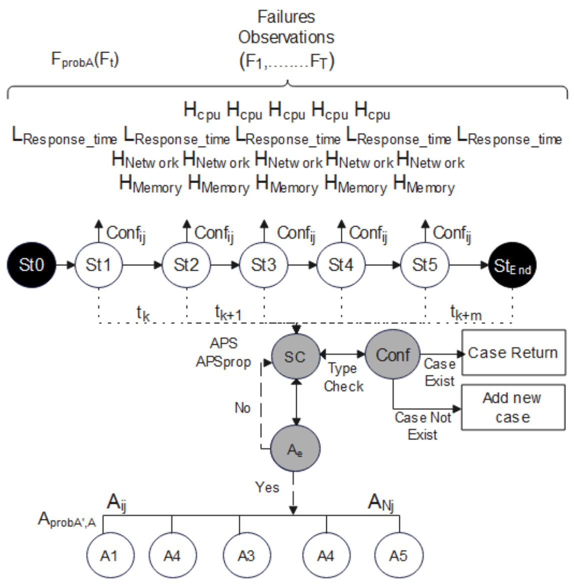


Fig. 2. Misconfigurations and Threats Identification Type

denoted *SC* checks the misconfigured settings against centrally managed correct configuration settings stored in Knowledge storage. We iterated through the manifest settings to check the key-value pairs. For each pair, we calculated the confidence score taking into account the type of key with (p-value$\leqslant$0.05) to validate our hypothesis against the difference between manifests. A low confidence score indicates a difference between the configuration settings for the anomalous state. The difference between the configuration obtained from the anomalous state (actual state) and the correct configuration (desired state) represents the incorrectly configured state that is likely to be targeted for exploitation by attackers. For the desired state, management data from the configuration settings were recorded, such as privilege, default accounts and their passwords, unnecessary ports, certificates, unpublished URLs, validation rules, default namespace, and version (e.g., deprecated API).

In case the misconfiguration is not defined within the cases, the controller records the new characteristics of the misconfiguration type and assigns unique identifiers (i.e., Failure ID, Label, component relationship information, and case type) to the selected items. The identification result is stored in the knowledge storage to enhance the identification process.

5.4 Threat Type Identification Under Misconfiguration

We are mainly concerned with threats that occur due to misconfiguration and cause data breaches and information leakages, such as botnets, ransomware, amplification, flooding, and protocol exploration. We created description profiles for each type of threat that include information about the threat (type, description, source, technique, configuration setting relation, and mitigation). We considered that the types of threats are hidden; thus, we employed the Hidden Markov Model (HMM) [16] to predict hidden threats because of its ability (1) to capture dynamic patterns by allowing the hidden states to transition between different states, reflecting changes in threat behavior. (2) to establish a baseline of normal behavior and then identify deviations from this baseline as potential threats. (3) update the model in the future to adapt to changing threat landscapes. (4) to incorporate data from multiple sources.

The controller maps the anomalous path obtained from the detection phase ($Cluster > Node_{22} > Node_{23} > Container_{33} > Service_{43}$) to a set of states ($St1 > St2 > St3 > St4 > St5$) to be fed to the model. Then, it adds a start state $St0$ and an end state St_{End} to the abnormal path ($St0 > St1 > \cdots > St5 > St_{End}$) to capture the entire information flow within that path. The controller checks the existence of a threat A_e using the SC function. If a threat exists, the model checks its type A_i according to the observations emitted (A_1: botnet, A_2: flood, A_3: amplification, A_4: protocol exploit, A_5: ransomware). For each threat A, we initialize the parameters of the state of the model A_i and the observations $F_{\{1,\cdots,T\}}$ through a graph length of the threat states APN and observation length T to track the duration of the threat states and identify them in a timely manner.

The probability of A_i is calculated assuming that a threat starts in the initial state $St1$ and might spread from one state to another. The probabilities of A_i and observations $F_{\{1,\cdots,T\}}$ are stored in matrix ATI. We calculate the probability F by summing the previous forward path probability of the previous time step $t-1$, weighted by their transition probabilities $Aprob_{A',A}$, and multiplying by the observation probability $Fprob_A(F_t)$. We sum the probabilities of all possible threats $\{A_i, \cdots, A_N\}$ that could generate the observation sequence $F_{\{1,t+1,\cdots,T\}}$. Each A represents the probability of being in A_i after seeing the first F_t observations, as shown in Algorithm 1.

Algorithm 1 Identify The Type of Threat

```
APS: Get abnormal path states $APS()$
APSprop: Get portability of abnormal path states $APSprop()$
APN: Threat states length
Store portability of $A$ at time $T$ in matrix $ATI[A,T]$
for each state $aps$ from 1 to $APS_{length}$ do
    for each portability $apsprop$ from 1 to $APSprop_{length}$ do
        if $SC$ is Threat then
            for each state $A$ from 1 to $APN$ do
                $ATI[A,1] \leftarrow \pi_A \times F_A(F_1)$
            end for
            for each time step $t$ from 2 to $T$ do
                for each state $A$ from 1 to $APN$ do
                    $ATI[A,t] \leftarrow \sum_{A'=1}^{APN} ATI[A',t-1] \times Aprob_{A',A} \times Fprob_A(F_t)$
                end for
                $ATIprob \leftarrow \sum_{A=1}^{APN} ATI[A,T]$
            end for
            return $ATIprob$
        end if
    end for
end for
```

Table 1. Misconfiguration and Threat Identification

Abnormal Flow Path	St1 > St2 > St3 > St4 > St5
Vulnerable Component	St2
Misconfigured Component	N_{23}
Misconfiguration Type	$Conf_{CC3}$
Threat Type	A_1

Transition probabilities equal to 0 are omitted since not all previous states contributed to the forward probability of the current state. Each St has a probability value reflecting the probability of a given abnormal behavior. The assumption is that a sufficiently low probability (abnormal flow) value indicates a potential threat. The decision is made by calculating a threat score $Threat_{Score}$ for each St and the whole abnormal path ($St1 > St2 > St3 > St4 > St5$). As shown in (1)–(3), the $Threat_{Score}$ for St is derived from the probability values returned by the detection model $\wp$ associated with St. The threat score value is calculated using a weighted sum $\sum_{St=1}^{w}$. The weight ω associated with the model is represented by ω_{RPV}, while M_{RPV} is the probability value returned by the model. The probability M_{RPV} is subtracted from 1 $(1-M_{RPV})$ because a value close to zero indicates a threat that should produce a high threat score. The weight ω is calculated considering the state transition probability of the type of hidden threat $Aprob_{ij}$ and the observation probability $Fprob_t$. The highest threat score is stored, and then the threshold is set to an adjustable percentage higher than the maximum score obtained, so that a user can adjust the sensitivity of the state check in terms of the number of false positives and the expected detection accuracy. The state is marked as vulnerable when $Threat_{Score}$ exceeds a predefined threshold. Once we identify the type of threat, the model transits to the next state to check the existence and type of potential threat for that state (if any); otherwise, the model returns to the state check SC to check the next state.

The process is repeated until we reach the end of the abnormal path $St5$. Then the model progresses to the end state St_{End}, ending the threat identification process. The derived hidden types of misconfiguration and threat are shown in Table 1.

$$Threat_{Score} = \sum_{St=1}^{w} \omega_{RPV} \times \wp \tag{1}$$

$$\omega = \sum_{RPV \in AI_{models}} Fprob_t \times Aprob_{ij} \tag{2}$$

$$\wp = (1 - M_{RPV}) \tag{3}$$

6 Evaluations and Results Analysis

This section evaluates the identification of the controller, focusing on measuring its performance in terms of quality and accuracy.

6.1 Environment Settings Description

The controller ran on a virtual machine equipped with Linux OS (Ubuntu 18.10 version), a VCPU, and 2GB of VRAM. The virtual platform is allocated to a physical PC equipped with Windows 11, Intel Core i7-1260P 2.10 GHz, and 32GB of RAM. A set of agents was installed to collect data on CPU, memory, network, and changes in the file system (i.e., no flow issued to the component). The agent adds a data interval function to determine the time interval to which the collected data belong. The Datadog tool is used to obtain a live data stream for the running components and to capture the request-response tuples and associated metadata. Prometheus is used to group the collected data and store them in a time series database using Timescale-DB. The size of our generated data set was approximately 10 MB with a period of 6 months. We selected a subset of the data set of around 4.3 MB mainly related to the types of misconfiguration mentioned in Misconfiguration Scenarios to train the models and provide more targeted and specialized training data sets. Data are divided into 70% training data and 30% testing data. More details about environment evaluation settings can be found in [17].

6.2 Misconfiguration Scenarios

We trained our models during the evaluation on some of the types of misconfiguration identified that allow privilege escalation at IoT edge device level [12] and at the container cluster level [13], [11], [14]. These types of errors excessively consume the usage of system resources (CPU, memory, network) as they dramatically increase the request latency and decline the request rate. The configuration files of the components are stored in GitOps version control to simplify the rollback of configuration changes.

6.3 Threat and Workload Scenarios

The threat scenario is based on the misconfiguration scenarios that lead to vulnerability in IoT edge devices and Kubernetes. The aim is to simulate the attack that could occur due to misconfiguration. The controller performance was tested in hybrid traffic, combining attack data generated by the tools at different levels. At the edge level, the IoT-Flock tool is used to generate normal and abnormal flow (threat) of IoT edge devices in a real-time network. At the container level, a Distributed Internet Traffic Generator (D-ITG) tool is used to generate normal and abnormal flow at the network, transport, and application layers with various packet sizes and a variety of probability distributions. We used OWASP-ZAP to simulate an attacker's attempt at vulnerable containers.

6.4 Identification Assessment

Assessment1: Identification Quality and Accuracy We extracted around 2090 real-world Kubernetes configuration files from version control repositories such as GitLab and GitHub, with 550 and 1540 files, respectively. The settings of the extracted files were valid in terms of format and syntax. We focus on the types of misconfiguration related to the misconfiguration cases mentioned in the paper that lead to system performance degradation, which were 279 and 1016 true positive configuration errors from GitLab and GitHub, respectively, identified by the controller. The controller reported other configuration errors; however, in this evaluation, we focused only on errors due to privilege escalation. Around 181 configuration errors in the true positives reported were due to privilege escalation, 109 from GitHub, and 72 from GitLab. The controller reported 16 false alarms, which occurred due to incorrectly skipping conditional instructions affected by the configuration value such as non-existent paths (e.g. invalid image repository path), unreachable IP addresses, or referencing a non-existent-configmap.

Hence, to measure the quality of the identification process, we split the normal behavior sequences into correct configurations using a sliding window and then further learn the controller. For testing, the first step is to split the test sequence into small segments and to calculate the probability under normal behavior. We analyzed the results with different window sizes (in hours) and state transitions by computing the Configuration Error Rate (CER), which divides the total number of unequal key-value pairs of data elements by the total number of data elements from one component to quantify the number identification error made by the controller with respect to the actual values. The total CER was approximately 10%, indicating that the model incorrectly identified 10% of the total elements.

Assessment2: Identification Overhead We measured the controller's performance baseline metrics (e.g., throughput, latency, CPU, memory, response time) under normal settings without configuration errors, workload, and without a configuration error detection mechanism enabled. The normal settings were approximately 70% and 35% for CPU and memory, respectively, the average response time was between 100 and 170 microseconds per request, 100 transactions/second of throughput, and 130 milliseconds of latency. We created the configuration error [13] at the container cluster level, which severely saturated the system resources to be 95% and 76% for CPU and memory, respectively, the average response time of 600 microseconds, the transaction per second of throughput, and the latency of 500 milliseconds. We measured the identification overhead as the time needed to identify the misconfigurations by measuring the execution time before and after the controller invocation. The average identification time taken by the controller was 301.8 milliseconds, which returned to the network and file-related checks.

6.5 Misconfiguration Identification Accuracy Under Threats

We measured the accuracy of the model identification under different misconfigurations (Test1 [13], Test2 [14], and Test3 [12]), threat and workload test scenarios, and under various window sizes during the learning process. We calculate the threat ratio (TR) to count the number of components identified as compromised due to misconfiguration during time intervals to the total number of components of the system and report the result as a percentage. As shown in Table 2, the TR under different tests represents a specific diversity. When the size of the sliding window is greater than 6, the TR improves, but the relationship between the size of the data set and the threat ratio is not always linear. Hence, to ensure accurate and unbiased identification, we measured TR under different model transitions along the test scenarios. For each transition, we applied the same data set used in the baseline transition to analyze the changes in the TR between the baseline transition and the other model's transitions. As shown in Table 3, the model transitions significantly indicate a change in identification effectiveness, which indicates an improvement in threat identification performance. To confirm the reported results, we further evaluated the unbiased performance of the model transitions based on the number of true positives, false positives, and false negatives of identification to measure precision, recall, and F1-score. The identification precision, recall, and F1-score were 0.950, 0.932, and 0.974 respectively. The recall gives an interesting insight into the performance of the controller in relation to the number of false identifications. It is important to note that the reported results were delivered according

Table 2. Threat Ratio with Different Windows Size in Hours

Windows Size	3	6	9	12
TR of Test1	8.17	5.49	3.07	0.21
TR of Test2	74.31	65.03	52.01	54.24
TR of Test3	43.09	54.29	45.24	35.09

Table 3. Threat Ratio Under Model Transitions

Transition Model	Baseline-Transition	Transition 1	Transition 2
TR of Test1	31.39	5.49	5.93
TR of Test2	74.01	54.88	55.93
TR of Test3	64.79	43.65	40.03

to the type of test scenarios for misconfiguration, threat, and workload, and the data set used to learn the model.

7 Conclusions and Future Work

The paper presented a controller for analyzing misconfigurations of container-based clusters and edge devices in a hierarchical computing environment. The aim is to propose the identification mechanism of the controller by defining a set of configuration error cases that result in the emission of failures observed through system performance metrics. The controller identifies the root cause of the configuration error and its consequence threats to optimize the system's behavior to update configurations and prevent their future occurrences. The paper evaluated the performance of the controller, focusing on accuracy and quality. The results show that the controller can deliver a performance improvement under different transitions with 0.950 precision.

In the future, we will provide technical details on the collection criteria, quality, and diversity of the data set as the scope of this paper is to present the identification phase mechanism. We plan to conduct comprehensive security assessments to evaluate the proposed controller and compare its performance against existing mechanisms under various types of configuration errors. We aim to integrate the controller into the Kubernetes workflow and CI/CD pipeline to catch invalid configurations and potential security vulnerabilities before deployment to maintain secure and reliable Kubernetes-based applications. Implement a control strategy to track configuration changes and validate configurations before applying changes to production clusters to improve cluster security and stability. Combine multiple metrics to improve the accuracy of threat identification.

References

1. Assuncao, L., Cunha, J.C.: Dynamic workflow reconfigurations for recovering from faulty cloud services. vol. 1, pp. 88–95. IEEE Computer Society (2013)
2. Chiba, T., Nakazawa, R., Horii, H., Suneja, S., Seelam, S.: Confadvisor: A performance-centric configuration tuning framework for containers on kubernetes. pp. 168–178 (2019)
3. CWE: Common weakness enumeration category: Configuration (2023), `https://cwe.mitre.org/data/definitions/16.html`

4. Fairwinds: Kubernetes benchmark report security, cost, and reliability workload results (2023), https://www.fairwinds.com/kubernetes-config-benchmark-report
5. Gantikow, H., Reich, C., Knahl, M., Clarke, N.: Rule-based security monitoring of containerized environments. vol. 1218 CCIS, pp. 66–86. Springer (2019)
6. Hu, Y., Huang, G., Huang, P.: Automated reasoning and detection of specious configuration in large systems with symbolic execution. pp. 719–734 (2020)
7. of the IEEE Computer Society, S.E.S.C.: Ieee standard classification for software anomalies (ieee 1044 - 2009) (2010)
8. Lakshmanan, R.: Microsoft confirms server misconfiguration led to 65,000+ companies' data leak (2022), https://thehackernews.com/2022/10/microsoft-confirms-server.html
9. Mahajan, V.B., Mane, S.B.: Detection, analysis and countermeasures for container based misconfiguration using docker and kubernetes. pp. 1–6. Institute of Electrical and Electronics Engineers Inc. (2022)
10. Moothedath, S., Sahabandu, D., Allen, J., Clark, A., Bushnell, L., Lee, W., Poovendran, R.: Dynamic information flow tracking for detection of advanced persistent threats: A stochastic game approach. arXiv:2006.12327 (6 2020)
11. NVD: Cve-2019-5736 (2019), https://nvd.nist.gov/vuln/detail/CVE-2019-5736
12. NVD: Cve-2019-6538 (2019), https://nvd.nist.gov/vuln/detail/CVE-2019-6538
13. NVD: Cve-2020-10749 (2020), https://nvd.nist.gov/vuln/detail/cve-2020-10749
14. NVD: Cve-2022-0811 (2022), https://nvd.nist.gov/vuln/detail/cve-2022-0811
15. Pranata, A.A., Barais, O., Bourcier, J., Noirie, L.: Misconfiguration discovery with principal component analysis for cloud-native services. pp. 269–278. Institute of Electrical and Electronics Engineers Inc. (12 2020)
16. Rabiner, L., Juang, B.H.: An introduction to hidden markov models. IEEE ASSP Magazine **3**(1), 4–16 (1986)
17. Samir, A., Dagenborg, H.: A self-configuration controller to detect, identify, and recover misconfiguration at iot edge devices and containerized cluster system. pp. 765–773 (2023)
18. Samir, A., Dagenborg, H.: Self-healing misconfiguration of cloud-based iot systems using markov decision processes. pp. 244–252 (2023)
19. Samir, A., Ioini, N.E., Fronza, I., Barzegar, H., Le, V., Pahl, C.: A controller for anomaly detection, analysis and management for self-adaptive container clusters. International Journal on Advances in Software **12**, 356–371 (2019)
20. Santolucito, M., Zhai, E., Dhodapkar, R., Shim, A., Piskac, R.: Synthesizing configuration file specifications with association rule learning. Proceedings of the ACM on Programming Languages **1** (10 2017)
21. Scarfone, K., Mell, P.: The common configuration scoring system (ccss): Metrics for software security configuration vulnerabilities. NIST interagency report **7502** (2010)
22. Wang, S., Li, C., Hoffmann, H., Lu, S., Sentosa, W., Kistijantoro, A.I.: Understanding and auto-adjusting performance-sensitive configurations. vol. 53, pp. 154–168. Association for Computing Machinery (3 2018)
23. Xu, T., Jin, X., Huang, P., Zhou, Y.: Early detection of configuration errors to reduce failure damage. pp. 619–634. USENIX Association (2016)
24. Zhang, J., Piskac, R., Zhai, E., Xu, T.: Static detection of silent misconfigurations with deep interaction analysis. Proceedings of the ACM on Programming Languages **5** (10 2021)
25. Zhang, J., Renganarayana, L., Zhang, X., Ge, N., Bala, V., Xu, T., Zhou, Y.: Encore: Exploiting system environment and correlation information for misconfiguration detection. pp. 687–700 (2014)