A Multi-pronged Self-Adaptive Controller for Analyzing Misconfigurations for Kubernetes Clusters and IoT Edge Devices

Areeg Samir$^{[0000-0003-4728-447X]}$, Abdo Al-Wosabi$^{[0000-0002-3655-8140]}$, Mohsin Khan$^{[0000-0003-1815-8642]}$, and Håvard Dagenborg$^{[0000-0002-1637-7262]}$

Universitetet i Tromsø - Norges arktiske universitet, Hansine Hansens veg 18, 9019 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, making them vulnerable to security breaches and information leakage if misconfigured. Misconfiguration leads to a compromised system that disrupts the workload, allows access to system resources, and degrades the system's performance. To provide optimal security for deployed clusters and IoT edge devices, the system should detect misconfigurations to secure and optimize its performance. We consider that configurations are hidden, as they are some sort of secret key or access token for an external service. We aim to link the clusters and IoT edge devices' undesirable observed performance to their hidden configurations by providing a multi-pronged self-adaptive controller to monitor and detect misconfigurations in such settings. Furthermore, the controller implements standardized enforcement policies, demonstrating the controls required for regulatory compliance and providing users with appropriate access to the system resources. The aim of this paper is to introduce the controller mechanism by providing its main processes. Initial evaluations are done to assess the reliability and performance of the controller under different misconfiguration scenarios.

Keywords: Misconfiguration · Monitor · Detection · RBAC · IoTs Edge Devices · Clusters · Markov Processes

1 Introduction

Misconfiguration is unsecured default configurations or incorrect configuration(s) within the parameters of the system components (i.e., system clusters, IoT edge devices) that violate a configuration policy and may lead to vulnerabilities that affect the system's workload and performance at different system 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, severely impacting system performance. To optimize a system's performance, system resources (e.g., CPU, memory) should be maximized for a workload; however, knowing the right limits to set for smooth application performance with different resource settings can be tricky. Large cloud service providers (e.g., Google, Microsoft, Amazon, Netflix) experienced misconfigurations that resulted in a vulnerable system [11], [25]. The management of configurations has been explored in the literature [4], [3], [8], however, the complexity of misconfigurations arose from a large number of configuration parameters, their correlations, and dependencies makes the reasoning about the misconfigurations difficult.

This paper extends the work in [20], [21] by proposing more details about a self-adaptive controller that detects misconfigurations of edge devices and clusters. The proposed controller is based on Hierarchical Hidden Markov Models (HHMMs), which we chose to (1) map the observed failure in metrics variations (e.g., CPU, Network, Memory, Workflow, Response Time) to the hidden misconfigurations in edge devices and system clusters. (2) Track the path of misconfiguration to show its impact on performance and workload. Furthermore, the controller extension implements standardized enforcement policies, demonstrating the controls required for regulatory compliance, and providing users with appropriate access to the system resources by extending the HHMMs to restrict access to our system and prevent security policy violations. The objective of this paper is to introduce the controller in terms of its architecture and processing activities, focusing on performance and reliability concerns. The remainder of this paper is organized as follows. Section II presents the research challenges. Section III discusses a use-case. Section IV introduces the self-adaptive controller phases. Section V evaluates the controller. Sections VI and VII conclude the article and present the future direction of the work.

2 Research Challenges

Managing the misconfiguration of Kubernetes clusters and edge devices offers several challenges, such as: **Workload Misconfiguration:** containers have built-in configuration settings to determine the amount of CPU and memory resources they use (via resource requests and limits). These settings in essence override some auto-scaling capabilities of the underlying platform and can lead to under-provision of the workloads, which causes performance issues, or over-provision, which can lead to dramatic inefficiency and cost overruns. For example, a container may run with more security permissions than required and escalate its own privileges, e.g., root-level access, which consumes considerable resources to fix and cause system downtime. A single workload may require significant configuration to ensure a more secure and scalable application [18], [26], [23].

Resource-Limit: misconfigurations of Kubernetes workloads often involve inefficient provisioning of compute resources, leading to an over-sized bill for cloud computing. To maximize CPU efficiency and memory utilization for a workload, teams need to set resource limits and requests. But knowing the right limits to set for smooth application performance can be tricky [2], [13], [10].

Dependency between System Components: while many tools are available for configuration scanning, there are some challenges ahead of them: (a) some configuration processes are done manually, which could lead to a risk of user error [19], [22], [27] (b) Configuration dependencies between different system components are passed manually as configuration parameters, which could lead to a complex set of CI/CD pipelines that is difficult to maintain [12], [7], [6].

Shared-Configurations: A configuration can be used by multiple applications that are themselves managed by different teams. While a configuration's name (key) stays the same across environments, a configuration's value varies across environments, which makes configuration changes hard to test, as changing a shared configuration requires coordination across teams, coordinated testing, and coordinated deployment [19], [12], [13], [29].

Configuration Change Late Check: While some configuration parameters may be checked when used in specific tasks at startup time, other parameters may not be checked or used. These parameters might have errors that wouldn't be detected until they showed up later (e.g., error handling). Before deploying, the configuration parameters must be validated to optimize system performance, which is a time-consuming task, and failing to validate a change could lead to undesirable downtime.

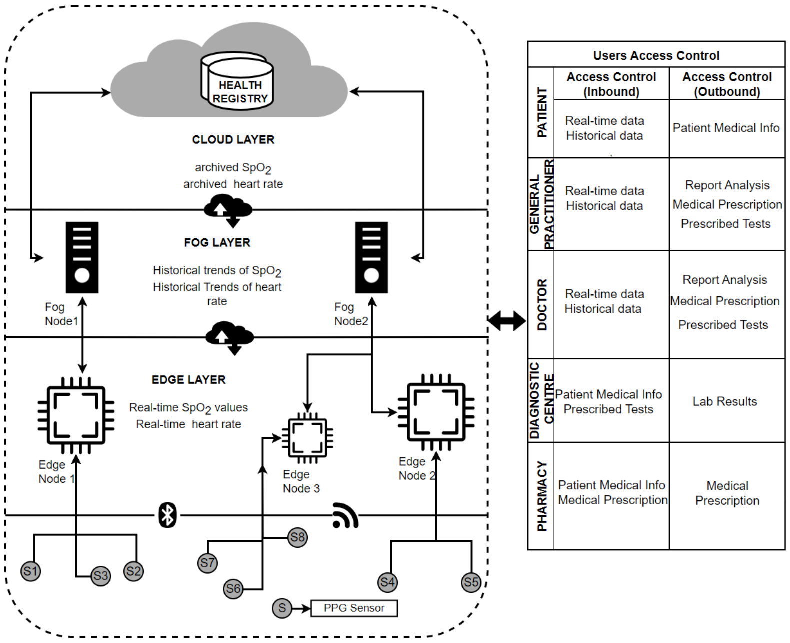
3 The System Under Observation - A Healthcare Use-case

Our system comprises hierarchical components with different configurations, resources, and policies. Components include gateways, sensors, services (e.g., monitor heart rate), edge devices, clusters, nodes, containers, and system users (e.g., healthcare participants), including their roles and access control to manage and control sensors and actuators attached to the system, as shown in Figure 1.

At the edge layer, misconfiguration (e.g., lack of authentication and authorization [15]) could affect device monitoring and allow an attacker to inject or modify data to reprogram the device. At the fog layer, misconfiguration at the cluster level (e.g., vulnerable product version [14], [16], no parameter validation [17]) could allow an attacker to gain root-level access to the host and exploit system processes. At the cloud layer, misconfiguration, such as enabling anonymous access to blob containers in cloud storage, might result in the leakage of sensitive information. In such settings, participants linked to the system may experience anomalous behavior or threats that stress the system and its performance. Hence, we differentiated between the types of observation concerning misconfiguration and performance degradation: **error** that refer to a misconfigured system component, which is unknown and hidden from the participants and could lead to **threats** such as distributed denial of service attacks that target the configuration of the component to impact the trust established between the IoT devices and the system. Error and its consequences of threats can lead to anomalous or faulty behavior **anomaly/fault**, which is hidden from the participants (i.e., overload and abnormal flow of information characterizing stealthy threat strategies conditioned on the system model and the control policy). Such settings are observed by the occurrence of an observed **failure** (e.g., saturated resources) emitted from the settings of hidden components.

4 Self-Adaptive Controller

This section presented the main phases of the controller. The controller adopted the Monitor, Analysis, Plan, Execute, and Knowledge (MAPE-K) architecture for self-adaptive systems and consists of (1) *Monitoring* that collects performance data; (2)



Users Access Control		
	Access Control (Inbound)	Access Control (Outbound)
PATIENT	Real-time data Historical data	Patient Medical Info
GENERAL PRACTITIONER	Real-time data Historical data	Report Analysis Medical Prescription Prescribed Tests
DOCTOR	Real-time data Historical data	Report Analysis Medical Prescription Prescribed Tests
DIAGNOSTIC CENTRE	Patient Medical Info Prescribed Tests	Lab Results
PHARMACY	Patient Medical Info Medical Prescription	Medical Prescription

Fig. 1. The Hierarchical Interaction between System Components, Participants, and The Access Control

Detection and Identification that analyzes detected misconfigurations and vulnerabilities in edge device(s) and container-based cluster and identifies its type. To control access from edge devices to system components, we extended HHMM to manage constraints in role-based access control. Models are implemented at the gateway to collect and transmit measurements from the edge device to the fog.

4.1 Phase 1: System Components Monitor

We checked the normality of the workload of the components under observation using the Spearman rank correlation coefficient to estimate the dissociation between the observations emitted (failures) and the amount of flow (hidden workload). To achieve that, we wrote an algorithm that is used as a general threshold to highlight the occurrence of faults in managed components (for more details, see [21]). The controller checked the configuration settings against the benchmarks of Azure Security, CIS Docker, and Kubernetes to detect any mismatch between the settings and the requirements of secure deployment in components.

4.2 Phase 2: Access Control Policy Management

We controlled the information flow from/to the system by managing the interaction of participants with the system. Each participant has allowed actions and roles to access nodes and services of the system (see Figure 1). We identified a list of the roles and actions of the participants, which has a set of access variables for each participant, such as the roles, actions, access to the API, the authorizations they have, the permissions, the boundaries of the permissions, and the conditions. The permission limit defines the maximum permissions granted to participants and roles using an enumeration-type action with two values (true and false). If the permission action is true, then the permission is allowed; otherwise, it is rejected. Moreover, we assumed that if no information flow policy is specified in the domain, the inbound and outbound flow will be set if the policy has any outbound rules. The policies in the observed system do not conflict as they are addictive. We extended our HHMM model with a set of controlling labels made up of tags, each of which stands for a specific integrity issue (private data) and outlines the information flow allowed. We define a role-based access space and a set of policies for each participant to allow specific participants access to specific system services. The access control policies are specified in the form of YAML format by writing a script that defines a template for generating YAML definitions based on the external policies. The script iterates over each policy, fills in the template with policy details, and accumulates the generated YAML. The script writes the accumulated YAML to an output file that is applied to the Kubernetes cluster and ensures that the translated policies are properly enforced.

System Component and Participants Role we specified a set of labels made up of tags to represent certain integrity (private data) and secrecy issues (sanitized data) to manage system components and access of participants from medical devices. Tags outline information flows by regulating the sensitive flow of information, such as patient personal information and related medical reports/outcomes. Tags correlate objects, such as patient and data items, with the secrecy and integrity flow constraints required to formulate a policy. Each tag is decomposed into a pair $\langle c, s \rangle$ of concern and a specifier. For example, the pair $\langle medical, Patient432 \rangle$ symbolizes Patient432's health information. We defined all data records of a particular type without listing all potential tags, as shown in Figure 2. Each tag has one or more subtag connections defined for any concern and specifier. For example, a tag $T_0 = \langle c, s \rangle$ is a subtag of tags $T_{\{1,0\}} = \langle c, * \rangle$ and $T_{\{1,1\}} = \langle *, s \rangle$, which are in turn subtags of the tag $T_2 = \langle *, * \rangle$ as shown in Figure 3. In addition to the tags, every participant has an access variable Λ that expresses (1) the access role AR: read R, write W, update U or combination of them (see Figure 2), (2) access status AS successful $Approval$ or $Fail$, (3) device id DID (see Figure 4), (4) device type DTY, (5) component label *Cabel* (i.e., node label), (6) component type *Comty* either (*node*, *container*, or *services*), (7) component id *ComID* to access node, container or service, (8) user id $UserID$, and (9) data access type *DAT* either *PatientMedicalInfo, ReportAnalysis, PrescribedTests, MedicalPrescription,* and/or *LabResults*. The state space is a set of $\Lambda = \{AR, AS, DID, DTY, Cabel, ComID, UserID, DAT\}$. To define the maximum permissions granted to participants, a set of access boundaries is defined based on conditions and actions. The access boundaries take effect only if all conditions are satisfied. The access boundaries are accompanied by an enumeration type that takes true or false as a value to permit or reject access to the system. For each component and participant, we specify access control permissions as follows.

Access Control Permission for each type of component *Comty*, each participant $Participant\{p\}$ *Doctor, Patient, GP, DigCen, or Pharmacy* has two label constraints (secrecy and integrity). The secrecy label restricts the read operation (i.e., incoming data flow), for example, $Sec(Participant\{Patient\}) = \{DAT, UserID\}$. The

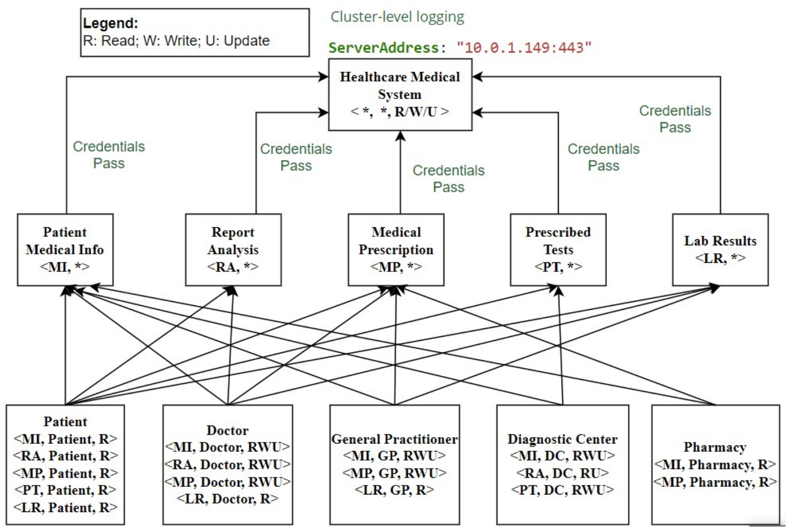

Legend:
R: Read; W: Write; U: Update
Cluster-level logging
ServerAddress: "10.0.1.149:443"
Healthcare Medical System < *, *, R/W/U >
Credentials Pass
Credentials Pass
Credentials Pass
Credentials Pass
Credentials Pass
Patient Medical Info <MI, *>
Report Analysis <RA, *>
Medical Prescription <MP, *>
Prescribed Tests <PT, *>
Lab Results <LR, *>
Patient <MI, Patient, R> <RA, Patient, R> <MP, Patient, R> <PT, Patient, R> <LR, Patient, R>
Doctor <MI, Doctor, RWU> <RA, Doctor, RWU> <MP, Doctor, RWU> <LR, Doctor, R>
General Practitioner <MI, GP, RWU> <MP, GP, RWU> <LR, GP, R>
Diagnostic Center <MI, DC, RWU> <RA, DC, RU> <PT, DC, RWU>
Pharmacy <MI, Pharmacy, R> <MP, Pharmacy, R>

Fig. 2. Role-Based Access Control

$$\begin{tikzcd}
 & \langle *, * \rangle & \\
\langle c, * \rangle \arrow[ur, "\succeq"] & & \langle *, s \rangle \arrow[ul, "\succeq"'] \\
 & \langle c, s \rangle \arrow[ul, "\succeq"] \arrow[ur, "\succeq"'] &
\end{tikzcd}$$

Fig. 3. Tag and Sub-Tags Relations

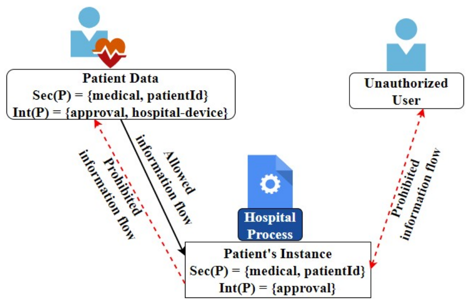

Patient Data
Sec(P) = {medical, patientId}
Int(P) = {approval, hospital-device}
Unauthorized User
Allowed information flow
Prohibited information flow
Prohibited information flow
Hospital Process
Patient's Instance
Sec(P) = {medical, patientId}
Int(P) = {approval}

Fig. 4. Information Flow Access Constraints

integrity label constrains the write operation (i.e., outgoing data flow), for example, $Int(Participant\{Patient\}) = \{Approval, DID, DTY\}$. The status of these two labels specifies the security context of accessing a specific component by a specific participant. For example, $Patient$ and $PatientMedicalInfo$ tags with the type of data accessed $'DAT'$ are presented in the hospital process to obtain patient data $Sec(Participant\{p\})$ according to (1).

$$\begin{aligned} &\forall p \in Participant\,p\ \exists!\,(Sec(Participant\,p) \wedge Int(Participant\,p)),\\ &where\ (Sec()\ and\ Int()) \supset \Lambda, \exists!\,(DID \wedge DTY)\ ForEvery\ Patient, \wedge \\ &\iff DID \subset Comty \end{aligned} \tag{1}$$

To ensure integrity consistency between groups, our system only accepts data from authorized medical devices based on the confirmation of approval of a patient and gives access to a specific node $Int(Participant\{Patient\}) =$

$\{Approval, HeartMon24329, N21\}$ as shown in Figure 4, and according to the rule in (2).

$$E1 \rightarrow E2, if\, Sec(E1) \precsim Sec(E2) \wedge Int(E2) \precsim Int(E1) \tag{2}$$

A decision rule $\Omega = \{AccessGranted, PerformManagement\}$ is added to allow certain actions in the access role (read, write, or update information) and to provide system-wide enforcement of the information flow policy, as shown in Figure 2. The decision rule represented a generated probability distribution $\mu T_{ins}(Comty)$ from a type of component *Comty*, which is labeled *Cabel*, with specific actions at a time instant T_{ins} as shown in (3).

$$\omega = \begin{cases} AccessGranted, PerformManagement\ If \\ Comty \in \{AR, AS, DID, DTY, ComID, UserID, DAT\} \\ AccessDenied,\ Otherwise \end{cases} \tag{3}$$

Addition of New Edge Device to adapt the controller to accept new data from a new device, we introduced entities E with actions and events. Each entity has participants and sensors that interact with system nodes through mobile applications/interfaces and medical devices. Here, $E1$ (e.g., patient's device) has access to entity $E2$ (e.g., specific node '$N21$') with any preorder relationship $\precsim$. For example, $E2$ can read data from entity $E1$ only if the secrecy of $E1$ is preorder (i.e., subset) of the secrecy of $E2$, while entity $E1$ can write to entity $E2$ only if the integrity of $E2$ is preorder of the integrity of $E1$. Entities are registered in a domain with credentials to be validated upon the authorized participant's request.

Domain Management we define a *domain* as a named grouping structure with a particular function. The *domain* represents an organizational system cluster (s) with participants associated with devices. Each domain maintains its policies to control interactions with entities E and other domains. The *domain* indicates that if either of its policies returns $\Omega = AccessGranted$, then the information flow is granted for specific entities E with an annotation of the sequence, denoted X. To map the domain of participants to the components of the system without requiring them to store these mappings and to manage the workload of the system under observation, each component with a component label *Cabel* is registered in a domain. Depending on the policy, entities might perform actions in other domains for which they are not registered. Thus, to allow domains controlling the circumstances in which data is released and in which information can be accessed, we considered that entities could communicate with the system through a combination of the following three identifiers: (a) other nodes that are allowed (exception: a node cannot block access to itself), (b) namespaces that are allowed, and (c) IP blocks (exception: traffic to and from the node where a running container is always allowed, regardless of the IP address of the node). Hence, for each entity, we computed the most probable extended annotation considering (Len, SN) at time tim. We defined *Len* as the length of the graph sequence and *SN* as the number of states of the HHMM. We constructed a directed cyclic graph in which every path has a start vertex device id *DID* and an end vertex component id *ComID* corresponding to an annotation of the sequence X of that path (e.g., $path1:\ DID_1 > Cl_1 > N21$; $path2:\ DID_2 > Cl_1 > N22$; $path3:\ DID_1 > Cl_1 > N23$). On the contrary, for every annotation of X, there is a path with specific properties. For example, $X_{i=1} = \{path1, tim, \Omega, Participant\{p\}, \Lambda\}$. We considered that the only allowed connections in the graph are those from the containers and nodes in our domains under some containers and policies, which do not conflict, as they are addictive. Hence, for a connection from a source node to a destination node, both the inbound and outbound flow policy on the source node and the destination node must allow the connection. If either side does not allow the connection, the connection will be rejected. If no information flow policy is specified in the domain, then by default, the inbound flow will always be set, and the outbound flow will be set if the policy has any outbound rules. Each information flow policy permits participants to access the system's components in a Namespace. Each policy includes a type list *PolicyType*, which may include inbound, outbound, or both in a namespace. The *PolicyType* indicates whether the given policy applies to the inbound flow to one or more selected nodes (s), the outbound flow from one or more selected nodes (s), or both. If no *PolicyType* are specified in a Namespace, then by default, inbound and outbound will be set if the policy has any outbound rules, and all inbound and outbound flows are not allowed to and from nodes in the Namespace. Participants access the system according to a request that includes the username, the requested action, and the object affected by the action. The request is authorized if the existing policy for a Namespace declares that the participants have permission to complete the requested action (write, read, update, or combination of them).

Our intention is not to prescribe action sequences for participants. Instead, we provide mechanisms to control the system's access actively, according to the flow of information from the participants, through adaptation and conditional access to the system components. Once access to the system is secured, the controller moves on to the next phase.

4.3 Phase 3: Misconfiguration Detection

We use HHMM [5] to model the hierarchical structure of our system and map the hidden misconfiguration settings from the observer to the performance metric. We choose HHMM because every component, along with its dependence on configuration settings, can be represented as a set of hierarchically interlinked HMMs, as shown in Figure 5.

The components of our system under observation have a hierarchical structure. The system consists of one or more clusters Cl (root state) that are composed of a set of nodes N (internal states) that host containers C (substates) with one or more deployed services S (production state) as a component of the application. Each component emits observations, which are emissions of failures from a component resource. Each component has configuration settings. The node assigns requests to its containers, communicating at the same node or externally. A service could be deployed in several containers simultaneously, and a container is defined as a group of one or more containers that constitute one service. The system has more than one cluster $Cl^{j=1}$ and has internal states $N_i^{j=2}$, which represent our virtual machines (nodes) with horizontal i and vertical j. Each node has a substate C_i^{j+1} that represents our containers (e.g., C_1^3 at vertical level 3 and horizontal level 1). Each container has deployed services S_i^{j+2} that emit Observations Space OS_n, which reflects a sequence of workload fluctuations for CPU, Memory, Network, and Response time. The fluctuation is associated with the saturation

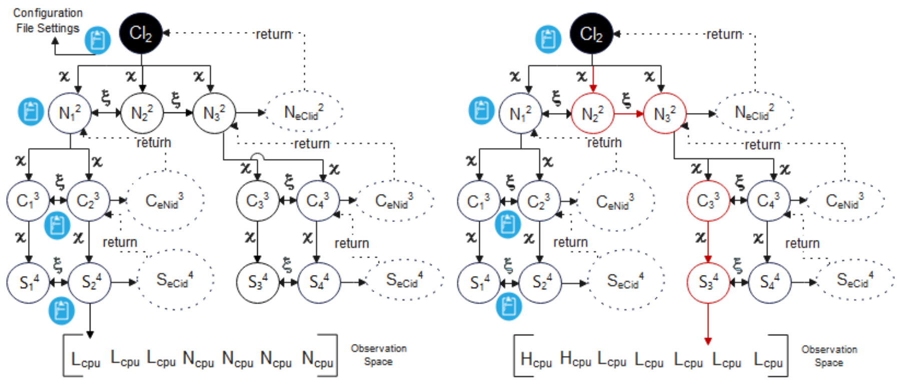

Configuration File Settings
Cl2
return
N1 2
N2 2
N3 2
NeCId 2
C1 3
C2 3
CeNId 3
C3 3
C4 3
S1 4
S2 4
SeCId 4
S3 4
S4 4
Lcpu Lcpu Lcpu Ncpu Ncpu Ncpu Ncpu
Hcpu Hcpu Lcpu Lcpu Lcpu Lcpu Lcpu
Observation Space

Fig. 5. Misconfiguration Detection in Multi-Clusters System(s) Using HHMM

of observed computing resources to be either H: High, L: sLow, or N: Normal, more details in [21]. This fluctuation is associated with a probability that reflects the state transition status from AF (Abnormal Flow) to NL (Normal Flow) at a failure rate $\Re$, which indicates the number of failures emitted from our Cluster Space (ClS) over a period of time. ClS consists of a set of Ns, containers C, and services S. The edge direction indicates the information flow and the dependency between states. For example, $N_1^2 = \{C_1^3, C_2^3\}$, N_2^2, $N_3^2 = \{C_3^3, C_4^3\}$, $C_1^3 = \{S_1^4\}$, $C_2^3 = \{S_2^4\}$, $C_3^3 = \{S_3^4\}$, $C_4^3 = \{S_4^4\}$.

For each participant in a specific *domain* with $X_i = \{path_i, tim, \Omega, Participantp, \Lambda\}$ where $\Omega = \{Access\ Granted\}$, our system calls its nodes to enter their containers and services $N_1^2 = \{C_1^3, C_2^3\}$,N_2^2, $N_3^2 = \{C_3^3, C_4^3\}$, $C_4^3 = \{S_1^4, S_2^4, S_3^4, S_4^4\}$. Here, each service emits observations and transits to its final state S_{eCid}^4 to end the transition for the services and to return the control to its calling parent C_2^3, as shown in (4) and (5). The same process is followed, however, to make a horizontal transition to C_2^3 to obtain the observations at the container level. Once the horizontal transition is completed, the transition goes to the end state C_{eNid}^3 to make a vertical transition to the state N_1^2. Once all transitions are achieved under this node, the control returns to N_3^2, as shown in Figure 5. The model is trained by calculating the probabilities of the parameters to obtain a hierarchy of abnormal flow path $AF_{seq} = \{Cl, N_2^2, N_3^2, C_3^3, S_3^4\}$, which is affected by the misconfigured component (N_2^2) and might cause a threat, as shown in (6). Here, we recursively calculate $\Im$, which is ψ for a time set $(\bar{t} = \psi(t, t+k, S_i^j, S^{j-1}))$, where ψ is a state list, which is the index of the most probable production state to be activated by S^{j-1} before activating S_i^j. $\bar{t}$ is the time when S_i^j was activated by S^{j-1}. The δ is the likelihood that the most probable state sequence generates $(O_t, \cdots, O_{(t+k)})$ by recursive activation. τ is the transition time at which S_i^j was called by S^{j-1}. Once all recursive transitions are returned to Cl_w, we get the most probable hierarchies starting from Cl_w the production states in the T period by scanning the state list ψ, the likelihood of states δ and the transition time τ. The same previous steps are taken for each cluster where w refers to the cluster number $w = \{1, 2, \cdots, m\}$. We correlated each state with time to know its activation time, its activated substates, and the time at which the control returns to the calling state to have more information about the occurrence time and dependency of the misconfiguration. In the end, a sequence of anomalous hierarchical states is obtained. We compared the detected hierarchies with the observed ones to detect the misconfigured state and its impact on the flow of information. The detected path with the lowest probabilities will be considered the vulnerable path with misconfigured states and abnormal flow. For example, the observed failure $L_{Network}$ is associated with a vulnerable abnormal flow path $AF_{seq} = \{Cl, N_2^2, N_3^2, C_3^3, S_3^4\}$ that is affected by N_2^2.

$$\Upsilon_S = \max_{(1 \leq r \leq S_i^j)} \left\{ \delta(\bar{t}, t+k, S_r^j, S_i^j)\, a_{S_{eC_i}^j}^{S_i^j} \right\} \tag{4}$$

$$\Im_S = \max_{(1 \leq y \leq S^{j-1})} \left\{ \delta(t, \bar{t}-1, S_y^j, S^{j-1}) a_{S_{eC_y}}^{S^{j-1}} \Upsilon_S \right\} \tag{5}$$

$$AF_{seq} = \max_{Cl_{(1 \leq w \leq m)}} \left\{ \delta(T, Cl_w), \tau(T, Cl_w), \psi(T, Cl_w) \right\} \tag{6}$$

5 Evaluations

This section assesses the detection and role-based access control, focusing on the measurement of performance and reliability.

5.1 Detection Evaluation

Assessment1: Simulation Environment

Environment Settings: The testing environment is built with Python. It runs on VMware and consists of three nodes (i.e., VM instances), which are for (1) VM1: the Healthcare application that handles patient data. The VM1 is connected to an edge gateway device that acts as a local hub for data aggregation and processing. The edge gateway collects data from the IoT device and transmits the collected data to VM1 for further processing. The device communicates with the edge gateway through Wi-Fi. Edge devices with similar functionality are grouped and allocated to a respective group. For VM1, we created a patient application on the edge device that generates patient information using the Python Faker and paho-mqtt libraries and transmits the information to VM1. The edge gateway is presented by the MQTT client, which establishes a connection to the gateway's IP address and port. (2) VM2: correctly configured container-based cluster node, and (3) VM3: controller. For each node, we implemented a set of containers and services. Each node is 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. Around 30 namespaces are created, each with 4 microservices (pods) used for performance measurements and assigned the same number of network policies. The number of policies created was 900, which were ordered, managed, and evaluated using Calico, Open Policy Agent, and Styra DAS, respectively. A set of agents was installed to collect data on CPU, memory, network, changes in the file system (i.e., no flow issued to the component), patient health information, device operation status, device id, and system components service status. The agent adds a data interval function to determine the time interval to which the collected data belong. The agent is configured to connect to the system automatically with the valid credentials of the system users for authentication. 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 to store them in a time series database using Timescale-DB.

Table 1. Detection Evaluation

Models	RMSE	PFD	Recall	Accuracy
HHMM	0.3299	0.4050	95.01%	94%
CRFs	0.4023	0.4208	92.86%	92%

Edge Device Configuration Errors: We installed K3s on a Raspberry Pi 3 Model B+ and set up a single node Kubernetes cluster. We created three configuration errors. The first one is an 'empty configuration file' that makes the device have trouble starting modules. The second one is 'enabling unnecessary port', in which the YAML configuration defines a pod with a single container running the Nginx image. The container is configured to expose three ports: 80, 443, and 8080. The third one deployed a container image 'Simulated Hospital[1]' that generates patient data. The YAML image file is configured to allow privilege escalation.

Container-based Cluster Configuration Error Scenarios: We write our configuration files using YAML. Privilege escalation configuration errors were deployed, such as Privilege Escalation Flaw and Privilege Escalation Flaw and Redeployment Fail [20]. The configuration files of the components are stored in GitOps version control to simplify the rollback of configuration changes. We use Kube-Applier to fetch our declarative configuration files for our clusters from the Git repository.

Performance Evaluation: The model was trained on the collected data and on the configuration error scenarios all at once. The performance of the detection model is evaluated by the root mean square error (RMSE) and the probability of false detection (PFD), which are the metrics commonly used to assess the accuracy of the detection. The RMSE measures the differences between the detected value and the one observed by the model. A lower RMSE value indicates a more effective detection scheme. The PFD measures the number of components normally detected that have been misdetected as anomalies by the model. A lower PFD value indicates a more effective detection scheme. The efficiency of the model is compared to Conditional Random Fields (CRFs); see Table 1. We noticed that the computation of CRFs is harder than that of the HHMM. The results show that the performance of the proposed detection is better than that of the CRF, as it correctly detected misconfigurations with around 95% recall and 94% accuracy with few false positives, which occur due to some ports being identified as insecure (e.g., insecure docker registry 5000). A false positive occurred at the edge level due to a condition identified in the YAML rule that is based on a fixed threshold without taking into account personal variations that could affect normal ranges of vital signs (e.g., $condition : heart\_rate > 100$).

Assessment2: Real-World Scenario

Learning Settings: The controller is further trained in some of the misconfigurations that allow an escalation of privileges to the host [15], [16]. We evaluated the detection performance by comparing the HHMM with the HMM and measuring their log-likelihood. We ran each model for a maximum of 10 iterations with a random start and

[1] https://github.com/google/simhospital

an approximate training period ranging from 164 seconds to 9 minutes with two layers of HHMM. 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 above to train the models and provide more targeted and specialized training data sets. We trained the models on different hidden state numbers (8, 16, 32, 64) and evaluated their performance. The data are divided into 70% training data and 30% testing data, more details on the environment settings in [20].

Performance Evaluation: In the training data, the log-likelihood of the HHMM was around -63 with 8 states, which increased with increasing number of states to -50 with 10 states and -20 with 30 states. The HMM was -30 with eight states, -20 with 10 states, and reached -15 with 30 states. On the test data, the log-likelihood of HHMM gradually increased from -50 with 8 states to a peak of -5 with 30 states. Then the model increased slightly with increasing number of states, while the HMM fluctuated to -65 with 8 states, -50 with 10 states, and -60 with 30 states. After that, the HMM decayed to -70 with increasing numbers of states. We observed that HHMM outperforms HMM in different states. With a decreasing number of states, both models show good performance. However, with an increasing number of states, HHMM performance shows better results, whereas HMM performance gradually decays, showing a symptom of overfitting as its likelihood drops from training data to testing data. This returns to the larger capacity of the HHMM, which allows the model to adapt to new changes and to be less prone to overfitting.

5.2 Rule-based Access Control Performance Evaluation

Test Settings: We created access control roles for system participants with different roles and different access levels, as shown in Figure 2. We evaluated the performance of rule-based access control under misconfiguration by giving the Pharmacy the Doctor role. This misconfiguration violates the principle of privileged access and leads to security breaches or unauthorized access. We use NetPerf and iPerf to measure network latency. The NetPerf is configured with a 200-second test duration and a goal of 99% confidence that the measured mean values are within +/- 2.5% of the real mean values. The iPerf is configured with a 30-second duration of the test, a 5-second reporting interval, and 3 numbers of parallel user threads to use, in which each thread will initiate a separate connection to the server. To constitute unauthorized access to the system, we define the rule that a pharmacy can write and update patient data. We implemented a trace file with 5000 requests that correspond to 15 minutes of workload. The trace file is generated using OpenTelemetry and Jaeger in Kubernetes. We generated several unauthorized access attempts from the Pharmacy due to the misconfiguration. We focussed on measuring the overhead latency of network performance during the creation of several network access policies.

Performance Evaluation: We created a cluster with two namespaces: ”users-namespace” and ”healthcare-namespace” with 4 microservices (pods) for each. We created an access control policy that assigns to the pharmacy the ”edit” ClusterRole, which allows for write and update access to resources within the ”healthcare-namespace”. We measured unauthorized access to the application. The average access was about 3100 access attempts per 2 minutes with CPU and memory loads 80% and 75%, respectively. We measured the performance overhead for network latency by increasing the number of policies from 100 to 900 policies by 100 policies at a time. We created some policies, one that allows traffic communication between pods and gave it the highest order, and the other one, a policy that disallows traffic between pods, and we gave it the lowest order. We increase the number of policies to measure network performance in terms of latency. Our goal is to enforce a network policy that restricts communication between these namespaces. Hence, the pods in the ”users-namespace” namespace can only communicate with pods in the ”healthcare-namespace” namespace on a specific port while denying all other traffic and on specific roles. We created a misconfiguration that violates the principle of privileged access in the ”users-namespace” namespace by assigning the user ”pharmacy” the ”cluster-admin” ClusterRole, which grants the pharmacy full access including the ability to modify resources and update other patient’s data at least. During the evaluation, the network performance shows an unremarkable impact on latency while increasing the number of policies. The latency was almost stable, from 70 microseconds with 100 policies to 85 microseconds with 900 policies. Due to the misconfiguration of the network policy, the pods within the nodes were able to communicate with resources that should be restricted. We created another VM (VM4) with 4 pods to communicate with VM1 to measure the latency of the pods between the nodes. We stress the resources of VM4 with Locust with a waiting time between requests of 5 to 15 milliseconds. The latency for the pods’ communications between the nodes was higher than that for the pods’ communication within the node. It increased from 250 microseconds with 100 policies to 280 microseconds with 900 policies. This leads to shared resources and direct communications between the pods on the same network, resulting in lower latency.

From the results obtained, we conclude that system performance is directly affected by configuration errors. The higher the number of configuration errors, the more likely the system will experience performance degradation.

6 Related Work

Existing frameworks have paid limited attention to the critical role of efficient management of misconfiguration in edge devices and clusters [24], [4], [1]. The work in [19] conducted an empirical study with 2,039 Kubernetes manifests to categorize the security misconfiguration and quantify it. Another work [3] presented a performance-centric configuration framework for containers on Kubernetes that gives unified key-value data, including configurations and metrics, to analysis plugins by providing a built engine for processing defined rules in analysis plugins. However, those techniques are time-consuming to come up with good result quality and are unmanageable with large datasets. The techniques suffer from catching every code defect and are limited when it comes to addressing issues in complex, multi-component applications, especially in scale and load balance environments. The work in [27] focused on detecting configuration errors at the startup time by analyzing the source code and generating the configuration checking code. However, this technique cannot handle the interaction between the configuration parameters. An analysis of misconfigurations and their associated code blocks helps in detecting which parts of the system code 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 [28], [9], based rule [22], or based inference. However, those ways are highly specialized as some of them only focus on security, are not simple to write and maintain, are geared towards a host only instead of container images and edge devices, and can result in false positives or false negatives. Unlike our work, previous techniques are (1) limited in the types of configuration errors that can be detected. (2) Focus on detecting misconfiguration based on the type inference of the source code. (3) lack of adaptable detection that works on configurations inherited from different systems or incorrect settings that fall into normal ranges.

7 Conclusions and Future Work

The paper presented a controller that detects misconfigurations of container-based clusters and edge devices in hierarchical computing environments. The controller mapped observable quality concerns onto hidden settings to track misconfiguration paths and enforced access to informational constraints derived from healthcare legislation. The controller used the Hierarchical HMM mechanism and extended its mechanism to propose an access control policy model to increase the flexibility of role-based access control so that users can gain access to resources with regard to the model constraints, and the permissions could be adjusted based on user and environment conditions. Compared with other techniques, the evaluation presented the ability of the controller to detect misconfigurations with few false positive instances and promised log-likelihood. The purpose of the paper is to introduce the controller mechanism by providing its main processes and evaluations to assess its reliability and performance.

In the future, our objective is to carry out more experiments to confirm the results concluded, highlight the difference between the controller detection and other misconfiguration tools, improve the security of the access control model to handle system failure, and expand the evaluation of access control and policy rules.

References

1. Alspach, K.: Major vulnerability found in open source dev tool for kubernetes (2022), https://venturebeat.com/security/major-vulnerability-found-in-open-source-dev-tool-for-kubernetes/
2. Assuncao, L., Cunha, J.C.: Dynamic workflow reconfigurations for recovering from faulty cloud services. vol. 1, pp. 88–95. IEEE Computer Society (2013)
3. 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)
4. Fairwinds: Kubernetes benchmark report security, cost, and reliability workload results (2023), https://www.fairwinds.com/kubernetes-config-benchmark-report
5. Fine, S., Singer, Y., Tishby, N.: The hierarchical hidden markov model: Analysis and applications. Machine Learning **32**, 41–62 (1998)
6. Gantikow, H., Reich, C., Knahl, M., Clarke, N.: Rule-based security monitoring of containerized environments. vol. 1218 CCIS, pp. 66–86. Springer (2019)
7. Haque, M.U., Kholoosi, M.M., Babar, M.A.: Kgsecconfig: A knowledge graph based approach for secured container orchestrator configuration. pp. 420–431. Institute of Electrical and Electronics Engineers Inc. (2022)
8. Hicks, M., Tse, S., Hicks, B., Zdancewic, S.: Dynamic updating of information-flow policies. pp. 7–18 (2005)

9. Hu, Y., Huang, G., Huang, P.: Automated reasoning and detection of specious configuration in large systems with symbolic execution. pp. 719–734 (2020)
10. Kermabon-Bobinnec, H., Gholipourchoubeh, M., Bagheri, S., Majumdar, S., Jarraya, Y., Pourzandi, M., Wang, L.: Prospec: Proactive security policy enforcement for containers. pp. 155–166. Association for Computing Machinery, Inc (4 2022)
11. Lakshmanan, R.: Microsoft confirms server misconfiguration led to 65,000+ companies' data leak (2022), `https://thehackernews.com/2022/10/microsoft-confirms-server.html`
12. 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)
13. 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 (2020)
14. NVD: Cve-2019-5736 (2019), `https://nvd.nist.gov/vuln/detail/CVE-2019-5736`
15. NVD: Cve-2019-6538 (2019), `https://nvd.nist.gov/vuln/detail/CVE-2019-6538`
16. NVD: Cve-2020-10749 (2020), `https://nvd.nist.gov/vuln/detail/cve-2020-10749`
17. NVD: Cve-2022-0811 (2022), `https://nvd.nist.gov/vuln/detail/cve-2022-0811`
18. 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)
19. Rahman, A., Shamim, S.I., Bose, D.B., Pandita, R.: Security misconfigurations in open source kubernetes manifests: An empirical study. ACM Transactions on Software Engineering and Methodology pp. 1–37 (2023)
20. 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)
21. 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)
22. 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** (2017)
23. Sorkunlu, N., Chandola, V., Patra, A.: Tracking system behavior from resource usage data. vol. 2017-Sept, pp. 410–418 (2017)
24. Taft, D.K.: Armo: Misconfiguration is number 1 kubernetes security risk (2022), `https://thenewstack.io/armo-misconfiguration-is-number-1-kubernetes-security-risk/`
25. Venkat, A.: Misconfiguration and vulnerabilities biggest risks in cloud security: Report (2023), `https://www.csoonline.com/article/3686579/misconfiguration-and-vulnerabilities.html`
26. Wang, T., Xu, J., Zhang, W., Gu, Z., Zhong, H.: Self-adaptive cloud monitoring with online anomaly detection. Future Generation Computer Systems **80**, 89–101 (2018)
27. Xu, T., Jin, X., Huang, P., Zhou, Y.: Early detection of configuration errors to reduce failure damage. pp. 619–634. USENIX Association (2016)
28. 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**, 1–30 (2021)
29. 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)