Enhancing IoT Device Security in Kubernetes: An Approach with Network Policies and the SARIK Framework

Jonathan G. P. dos Santos[a], Geraldo P. Rocha Filho[b], Rodolfo I. Meneguette[c], Rodrigo Bonacin[d], Gustavo Pessin[e] and Vinícius P. Gonçalves[a]

[a]*University of Brasilia, Electrical Engineering Department, Brasilia, Distrito Federal, Brazil*
[b]*State University of Southwest Bahia, Department of Exact and Technological Sciences, Candeias, Bahia, Brazil*
[c]*University of São Paulo, Department of Computing System, Sao Carlos, Sao Paulo, Brazil*
[d]*CTI Renato Archer, Campinas, Sao Paulo, Brazil*
[e]*Instituto Tecnológico Vale, Ouro Preto, Minas Gerais, Brazil*

ARTICLE INFO

Keywords:
Kubernetes
Cluster
Network policy
Kube-proxy
SARIK
IoT
Edge computing

ABSTRACT

The Internet of Things (IoT) has ushered in an era of connected devices that, while facilitating real-time data collection and sharing, it also exposes these devices to significant security risks. This study addresses security risks challenges by employing the Network Policy feature in Kubernetes, focusing on the SARIK framework. SARIK is designed to automate the creation and implementation of network policies, aiming to enhance both the efficiency and protection of IoT devices. In experiments conducted in a controlled environment with Minikube in Kubernetes, the implementation of SARIK notably improved the security of IoT devices. Key observations included a measurable decrease in vulnerability to cyberattacks and a significant increase in the overall system resilience. Notably, the study revealed improvements in the performance metrics analyzed, providing evidence of SARIK's effectiveness in real-world scenarios. Compared to existing solutions - e.g., Sysdig -, SARIK stands out in its integration with Kubernetes network policies and its emphasis on automated security management. Although automation is a common feature in related works, SARIK's unique approach to leveraging the native capabilities of Kubernetes offers a distinct advantage in securing IoT environments. This aspect, along with its performance benefits, marks SARIK as a notable contribution to IoT security. The application of SARIK in securing IoT devices in Kubernetes environments underscores the need for automated and cohesive strategies to tackle contemporary security challenges. This study not only highlights the efficiency of SARIK but also emphasizes the need for continuous evolution in security strategies, adapting to the dynamic threats in complex and interconnected IT environments.

1. Introduction

The security of the Internet of Things (IoT) is a significant challenge, especially in smart urban environments where many devices are not initially designed with security in mind, making them vulnerable to cyberattacks. As highlighted by Bardoutsos et al. [1], it is crucial to implement effective measures to protect these devices and the sensitive data they manage. One such essential measure is the implementation of security policies, such as Network Policy in Kubernetes, which has become a vital tool in managing IoT devices.

According to Alawneh et al. [2], a container is a software process that runs a microservices image in a predefined execution environment with allocated resources. In the IoT scenario, the use of containers and Kubernetes is becoming common, in which various technologies are applied to solve specific problems faced by cities. Kubernetes, in particular, is often used in IoT applications that require device management on a large scale [3], being capable of handling the deployment, update, and monitoring of devices deployed as containers.

The literature presents several research works focused on open-source tools and projects related to the protection of IoT systems and Kubernetes clusters. For example, kub-Sec [4] generates AppArmor [5] profiles for Kubernetes clusters based on container behavior at runtime, while Sysdig [6] acts as an admission controller applying security policies during the deployment process. Another work [7] proposes an integrity protection solution for Kubernetes resources based on digital signatures. However, these existing works, including [4], [5], [6], and [7], did not explore scenarios with network policies and the application of rules to kube-proxy.

This paper presents the SARIK framework [8], which stands for "Automatic Security of Iptables Rules in Kubernetes." SARIK was developed to enhance container security in Kubernetes by applying blocking rules directly to the Pods. The previous work presented a preliminary approach that is limited to implementing rules only at the application layer. In this work, we propose improvements in the SARIK approach by suggesting that blocking rules be handled at the network and transport layers through kube-proxy. This innovation eliminates the need to create Pods with the privileged flag, providing a safer and more comprehensive approach to configuring network policies in the Kubernetes cluster.

Aiming to advance this research theme, this paper also describes and evaluates a voting system in a Kubernetes cluster utilizing the SARIK framework. Nine experiments

jonathanti@unb.br (J.G.P.d. Santos); geraldo.rocha@uesb.edu.br (G.P.R. Filho); meneguette@icmc.usp.br (R.I. Meneguette); rodrigo.bonacin@gmail.com (R. Bonacin); gustavo.pessin@itv.org (G. Pessin); vpgvinicius@unb.br (V.P. Gonçalves)
ORCID(s):

were conducted to evaluate several metrics, including latency, response rate, and transfer rate, and to quantify the amount of data generated. The CPU and memory usage of the cluster's Pods were analyzed using Kubernetes resources like Prometheus and Grafana. The main objective of this study is to verify whether the selected metrics are capable of evaluating port and protocol blocking in accordance with network policies in Kubernetes using the SARIK framework. Thus, this study contributes by providing an approach to minimally block the communication between Pods using the cluster's output interface as defined by the network policies.

This paper is organized as follows. Section 2 presents the theoretical exploration and literature review. Section 3 presents the proposed solution. Section 4 describes the experiment and illustrates the application of the proposed tool in a use case. Section 5 presents performance evaluation results, Section 6 presents discussions and implications and Section 7 presents conclusions and future work.

2. Theoretical Exploration and Literature Review

In this section, we will present an in-depth theoretical exploration, covering the fundamental concepts that underpin our study. Initially, in the subsection of 'Theoretical Framework', we will discuss the key works highlighted in the literature that are related to this research. Then, in the subsection of 'Related Works', we will conduct a comprehensive review of the existing literature, analyzing previous studies that address similar or complementary topics to our research.

2.1. Theoretical Framework

The work by Borgwardt et al. [9] compares the container orchestration systems Borg, Omega, and Kubernetes, highlighting that each has distinct strengths and limitations. Borg is scalable and mature but complex; Omega is simpler and easier to use but less scalable; and Kubernetes is an intermediate option, combining scalability with usability complexity. The authors argue that the choice of the most suitable system depends on the specific needs of an organization: Borg is recommended for scalability and maturity, Omega for simplicity and ease of use, and Kubernetes for a combination of scalability, maturity, and ease of use. This comparative analysis helps organizations select the container orchestration system that best suits their needs.

In Medel et al. [10], the authors present a methodology for predicting application performance in Kubernetes environments. The methodology is based on the analysis of application execution data, such as creation time, completion time, memory usage, and CPU usage. Using this data, a model is constructed that can predict the performance of new applications. This approach provides a valuable tool for developers and system administrators who wish to improve the performance of their applications in Kubernetes environments.

In the work of Chang et al. [11], the authors present a Kubernetes-based monitoring platform for dynamic cloud resource provisioning. The platform utilizes a data collector and a data analyzer to gather performance information from applications running on the Kubernetes cluster. Based on this data, the platform is capable of predicting and provisioning additional resources for the cluster as needed. Evaluation of the platform demonstrated its effectiveness in resource prediction and provisioning, contributing to improved performance and scalability of applications hosted on the Kubernetes cluster. The platform offers a comprehensive monitoring mechanism, deployment flexibility, and automatic operation for continuous optimization of resource provisioning.

In another work [12], the authors Vayghan et al. explore the high availability (HA) architecture in microservices applications, highlighting Kubernetes as a suitable platform for its implementation. The author emphasizes the importance of designing fault-tolerant applications when using Kubernetes for HA. A case study is presented to exemplify the challenges and lessons learned during the implementation of a microservices application on Kubernetes. The article is well-structured, providing a clear and concise explanation of HA concepts, along with valuable recommendations for resilient application design. Overall, this work is a useful read for developers interested in deepening their understanding of HA in microservices applications, providing practical insights through the presented case study.

Muralidharan et al. [13] address the implementation of a secure, distributed, and reliable cloud monitoring system for IoT applications in smart cities. The proposal presents a container-based system with low latency and reliable communication between IoT devices, focusing on horizontal interoperability between different applications. The contribution of the work is to provide an efficient solution for managing IoT applications in smart cities. The system was evaluated through experimentation with containerization techniques using Docker and the Kubernetes orchestration platform.

In this work [14] new resource scheduling algorithm for Kubernetes is proposed based on ant colony and particle swarm optimization, aiming to enhance scheduling efficiency. The algorithm consists of two phases: host filtering and host scoring. Host filtering uses ant colonies to identify the most suitable hosts for specific Pods, while the host scoring phase utilizes optimization techniques to assign weights based on available resources and current load. Compared to the standard Kubernetes algorithm, experiments demonstrated that the new algorithm scheduled Pods more efficiently, reducing resource wastage and showing higher capacity in handling dynamic workloads, minimizing the risk of network congestion.

2.2. Related works

This subsection presents an analysis of related works about security in IoT and Kubernetes cluster environments. Based on a literature review and authors' previous experience, this subsection provides an overview of the most recent works that investigate and propose solutions in this

Reference	Consolidated Framework	Dependency-free Framework	Egress interface	Ingress interface	Conducting Experiments
BALABANIAN et al., 2019 [15]	✓		✓	✓	✓
NAM et al., 2020 [16]			✓	✓	✓
KULATHUNGA, 2021 [17]					✓
ROCHA et al., 2023 [18]	✓				✓
SYSDIG Secure, 2023 [6]	✓		✓	✓	
KUDO et al., 2021 [7]					
ZHU et al., 2022 [4]	✓				✓
BRINGHENTI et al., 2023 [19]	✓				
Kano et al., 2022 [20]			✓	✓	
LEE et al., 2023 [21]	✓		✓	✓	✓
Our work	✓	✓	✓	✓	✓

Table 1
Related works characteristics

field, as well as it provides a summary of these studies and a comparison between the solutions found for Kubernetes (Table 1).

Balabanian et al. [15] developed the Tocker framework with the specific goal of restricting communication between containers to the bare minimum. Tocker achieves this by blocking unnecessary ports and introducing firewalls as an additional layer of security. This approach not only minimizes attack surfaces but also offers an automated approach to security management in container environments. This innovative focus of Tocker boosted subsequent research in the area, including the development of the SARIK framework for Kubernetes environments. SARIK was designed to address similar challenges in container security, but within a Kubernetes context, thus filling a gap left by the work on Tocker.

In the BASTION proposal by Nam et al. [16], the authors identified a significant gap in container network security and introduced the innovative BASTION security network stack. This network stack not only provides effective isolation between containers but also implements more advanced and detailed network security policies. The relevance of the BASTION proposal is evidenced by its ability to mitigate a variety of adversarial attacks, as well as improve network performance by up to 25.4%. The efficacy of BASTION was proven through rigorous testing, standing out as a valuable contribution to the field of container network security, especially in terms of innovation in security management and isolation.

Kulathunga's work [17] represents a significant contribution to security in container orchestration platforms. The author proposes a dynamic security model that incorporates an Intrusion Detection System (IDS) designed to efficiently monitor network traffic in Kubernetes environments. This IDS is configured to categorize traffic based on applications in relevant namespaces, allowing for precise identification of suspicious or anomalous patterns. A new component, named operator-IDS, is introduced to extend the Kubernetes API, facilitating an advanced level of monitoring. This innovative component enables the system to detect and respond to threats more effectively, providing a more granular and adaptable security approach.

The paper by Rocha et al. [18] addresses the growing need for more efficient Intrusion Detection Systems (IDS) in cloud container environments. The study proposes an innovative framework that employs machine learning techniques for anomaly detection in system calls, offering a robust and adaptable solution for security in these environments. This framework stands out for its ability to integrate with other security tools, promoting a more collaborative and comprehensive approach to security. The system's validation was carried out in an emulation environment with GNS3, where it demonstrated superior effectiveness in intrusion detection compared to traditional methods.

The Sysdig Secure platform [6] addresses security in container environments, integrating monitoring, compliance, and incident response. Its admission controller generates security policies before image deployment, contributing to more effective security management. Sysdig Secure collects real-time data from sources such as the Linux kernel Docker and Kubernetes APIs, analyzing them with machine learning and AI techniques for efficient threat detection. The solution stands out for its complete integration, enabling real-time threat detection, compliance with security policies, and agile responses to incidents. Being a commercial solution, access may be limited for some users.

Another relevant work in the field, presented by Kudo et al. [7], proposes an approach to reinforce the integrity of resources managed by Kubernetes. The research focuses on signature verification in the admission controller to address the recurring issue of inconsistencies between signed resources in the admission request and signature messages automatically generated by Kubernetes. These discrepancies can lead to failures in signature verification, compromising the system's integrity. The main contribution of this study is the creation of an effective solution to ensure the integrity of resources in the Kubernetes environment, implementing a more reliable and integrated signature verification mechanism. However, an important limitation identified is the potential increase in overhead during the admission process due to signature verification.

In the study conducted by Zhu et al. [4], an automated solution named Kub-Sec was proposed for creating AppArmor profiles in Kubernetes clusters. The identification of gaps in manual profile generation is a time-consuming and error-prone process, thus Kub-Sec introduces an automated approach. The system analyzes network traffic to identify the resources used by running containers, and automatically generates corresponding AppArmor profiles. This methodology reduces the time and efforts necessary for implementing security policies, as well as it ensures that only authorized resources are accessed, significantly elevating the security level of Kubernetes clusters. The effectiveness of Kub-Sec was proven in empirical evaluations, demonstrating its ability to create accurate and reliable AppArmor profiles for operational containers.

The paper by Bringhenti et al. [19] addresses the challenging task of manually configuring security policies in multi-cluster Kubernetes architectures. The proposed solution, the "Multi-Cluster Orchestrator", automates the generation and implementation of network security policies in each cluster. This approach significantly reduces the possibility of human errors and facilitates efficient communication between clusters, thus contributing to the improvement of operational efficiency and security in multi-cluster environments. Although the paper indicates that the solution was validated in realistic use scenarios, a more detailed description of the validation methodology could enrich the reader's understanding of the applicability and effectiveness of the "Multi-Cluster Orchestrator".

In the study conducted by Kano et al. [20], a tool is proposed for the verification of network policies in native cloud environments. This tool utilizes a formal modeling approach, creating a formal model based on automata networks to accurately represent network traffic behavior in Kubernetes environments. The main contribution of this research is the development of an efficient solution for network policy verification, capable of identifying and resolving conflicts and inconsistencies in real-time. This mitigates the risk of network disruptions, improving the reliability and efficiency of the system. The study also identified some limitations of the proposal, including the lack of support for all Kubernetes network features and the need for manual intervention in certain situations.

Lee et al. [21] address the security challenges in container environments, with a particular focus on human errors and inadequate configurations that can compromise security. They introduce KUNERVA, an automated solution that analyzes network logs to generate an optimized and effective set of network security policies. The tool stands out for its integration with the Gatekeeper policy application system, which significantly increases the reliability of the generated policies. This work represents a significant advancement in container security automation, offering a practical solution to mitigate common risks. The authors detail the methodology used for evaluating and validating the effectiveness of KUNERVA, including tests in realistic scenarios to ensure the applicability and efficacy of the solution in dynamic container environments.

The SARIK framework stands out from other works analyzed in several aspects. While many frameworks and solutions focus on specific aspects of container security, such as network isolation [15], [16], intrusion detection [17], [18], or compliance [6], SARIK offers a more comprehensive and modular approach to network policy management in Kubernetes environments. The framework allows not only manual and automatic configuration but also the exclusion of network policies for both inbound (ingress) and outbound (egress) traffic, as well as it offers advanced functionalities such as real-time monitoring, policy validation, and backup. This flexibility and comprehensiveness make SARIK a unique solution, as it addresses multiple facets of network security within a single framework. Additionally, SARIK incorporates an integrated help module and environment checks, making it more accessible to users with varying levels of expertise. This combination of features makes SARIK a robust and flexible solution, filling gaps left by other works that focus on more limited areas of security in container and Kubernetes environments.

3. Proposed solution

This section presents the SARIK framework, which provides the ability to configure Pods through network policies, which enables the control of network traffic. Our goal is to allow blocking of protocols and ports in the cluster, thereby adding a layer of security. In this way, it is expected to prevent security attacks on misconfigured or vulnerable Pods, strengthening the overall protection of the cluster. By using SARIK, it is possible to establish effective network policies that contribute to the mitigation of potential risks and enhance security in the Kubernetes environment.

Figure 1 presents the SARIK execution diagram, divided into seven steps, which are described below in a summarized manner, steps (1) and (2) involve performing various tests to determine if the framework is capable of executing the script. In step (3), the framework maps the cluster configurations and stores this data in variables or arrays. In step (4), after storing the data, the framework handles with line breaks by using the readarray function, which stores each line in a distinct index of the array. In step (5), the framework generates several YAML manifests for each Pod, which include network policy rules based on the extracted mapping data. This includes changing IPs, protocols and ports, as well as blocking other protocols. In step (6), the framework generate network policies in each Pod, and then these rules are converted into iptables rules, which are executed in kube-proxy, allowing or denying packet traffic between different hosts or networks. Finally, in step (7), the script completes the configuration of network policies in the Pods.

3.1. Concepts, Techniques and Tools Employed in SARIK Framework

SARIK utilizes Kubernetes network policies, which contribute to provide the necessary protections to restrict traffic

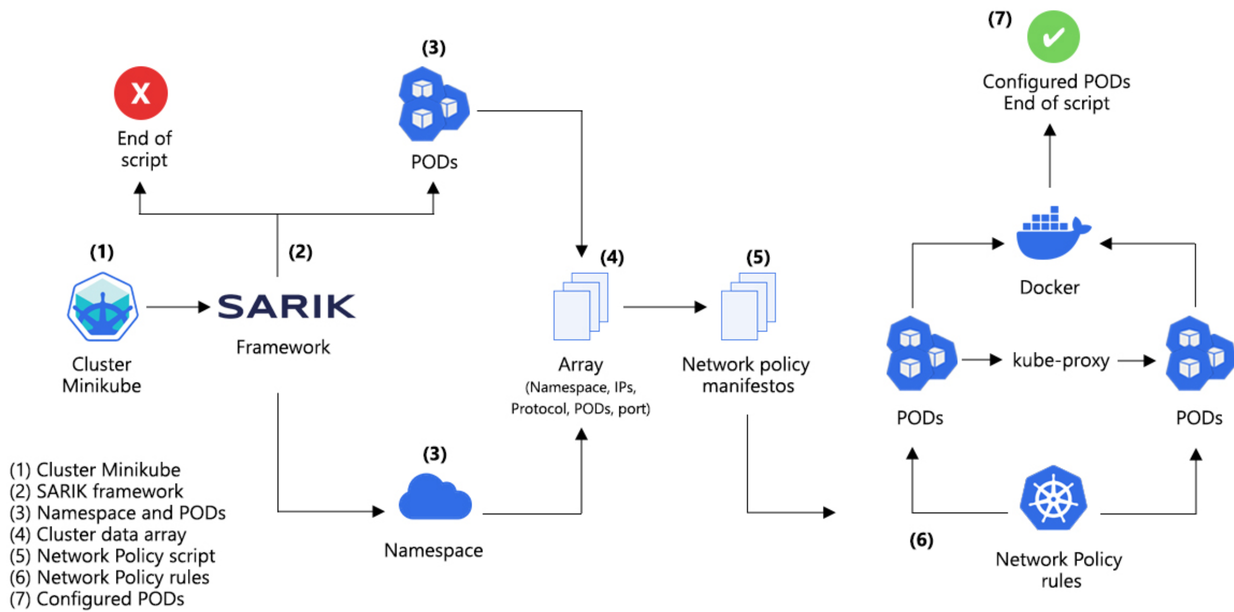


Figure 1: SARIK execution diagram

between Pods (within and/or across namespaces) [22] to create security rules in the cluster. In Kubernetes, NetworkPolicies are a way to allow security rules to control the traffic between Pods in a cluster. According to Liz Rice [23], "Network Policy can be used to implement progressive security policies, allowing teams to add security restrictions as needed, rather than having to design and implement all security policies at once." This means that you can start with basic rules and gradually add more restrictions as needed, ensuring increasing security. Network Policies are applied at the transport layer, specifically at Layer 3 or 4 of the OSI model (Network layer). They are used to control the traffic flow between Pods in the cluster, by allowing or denying access between different Pods. They are implemented at the cluster level, yet they are applied in conjunction with the Container Network Interface (CNI) plugin and kube-proxy.

Unfortunately, by default, Minikube [24] /Kubernetes does not support network policies due to the use of the CNI plugin, which is the cluster's standard. The CNI plug-in is responsible for providing the network infrastructure of the Pods and ensuring properly configured IP addresses, which prevents the acceptance of network policies in this plug-in.

Reference [25] investigated various options to overcome this limitation, including container network interface plugins such as Flannel [26], Weave [27], Cilium [28], and Calico [29]. In this paper, we have chosen Calico plugin, as it is the first option suggested on the Minikube page. However, it is important to mention that SARIK framework is capable of automatically generating network policies for Kubernetes clusters regardless of the network plugin. In the following we present Calico and Kube-proxy and make considerations about their use in the SARIK framework.

3.1.1. Enabling Calico in the cluster

Calico is a CNI plugin that applies NetworkPolicy rules at Layer 3 of the OSI model (i.e., the network layer), using technologies such as iptables or eBPF (Berkeley Packet Filter). This plugin enables the enforcement of NetworkPolicy rules to control traffic flow between Pods, allowing or denying access among different Pods based on the rules defined in NetworkPolicies.

The choice of Calico as a CNI plugin in Minikube is significant, as it offers a more direct and effective approach to implementing and managing network policies in Kubernetes environments, especially compared to other available options. The command minikube start –network-plugin=cni –cni=calico is crucial in this context, as it modifies the default CNI configuration in Minikube to use Calico. This allows Minikube to configure the user-defined network policies in the cluster environment, which is essential for the tests and validations conducted in this study.

Iptables is a Linux packet filtering table manipulation tool that enables the creation of firewall and NAT rules. The use of iptables in Calico allows for fine-grained control over network traffic, offering security and efficiency. Alternatively, eBPF, a more recent technology, offers advanced capabilities for packet filtering and performance monitoring with lower overhead, increasingly being adopted in modern network security scenarios.

Studies, such as those presented in [30] "eBPF: A New Approach to Cloud-Native Observability, Networking and Security for Current (5G) and Future Mobile Networks (6G and Beyond)" and [31] "A methodology for using eBPF to efficiently monitor network behavior in Linux Kubernetes clusters", highlight the capabilities of iptables and eBPF

in network traffic management and security policy implementation in distributed environments. These technologies are fundamental for the effective implementation of network policies in Kubernetes and provide a relevant context for choosing Calico as a CNI plugin in this work.

3.1.2. Kube-proxy

The kube-proxy is the component of a Kubernetes system responsible for monitoring network resource changes, such as Pods, services, and IP addresses, and configuring firewall and routing rules to reflect these modifications. When a Network Policy is applied, the kube-proxy is responsible for configuring the firewall and routing rules to ensure compliance with the Network Policy rules, i.e., the kube-proxy is a central hub of cluster communication. Kube-proxy offers a wide range of services including load balancing, traffic forwarding, high availability, reverse proxy, and IP address management [32]. Thus, kube-proxy is a key component, which uses iptables to configure the necessary network rules to ensure connectivity between the cluster components.

Algorithm 1 : Network Policy

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
 name: block-port-egress-db
 namespace: vote
spec:
 podSelector:
  matchLabels:
   app: db
 policyTypes:
 - Egress
 egress:
 - ports:
  - protocol: TCP
   port: 22
  - to:
   - podSelector:
     matchLabels:
      app: db
```

3.2. SARIK and Network Policies

We present the generation of the files during the execution of the SARIK framework (Figure 1). A detailed explanation of how this process occurs and how SARIK, CNI, and kube-proxy ensure the security of the infrastructure through network policies is provided. In Algorithm 1, we detail each field (selectors) and how SARIK handles these fields.

The default header of the manifest consists of the ApiVersion and Kind fields. The Metadata section contains information such as the name and namespace, which are filled based on the Pods ID, name, and the network in which the Pods reside. The spec field contains information about the Pod's specifications, such as the podSelector and the corresponding labels (matchLabels). The policyTypes section provides restriction options, as shown in the case of Algorithm 1 the "egress" was chosen. Based on a mapping (step (3) in Figure 1), a series of protocols and ports are blocked using the egress, ports, protocol, and port fields. The podSelector section again refers to the selection of Pods and their corresponding labels.

In summary, SARIK populates the selection fields of the manifest shown in Algorithm 1 through a mapping, resulting in N manifests that block outbound communication of each Pod in their respective namespaces. After this step, a additional step is performed to apply the network policies in the cluster. The command "kubectl apply -f rules" is used to this end.

The Kubernetes ecosystem, along with the selected CNI network plugin, performs several procedures to direct specific network policy requests from the manifests generated by SARIK to the kube-proxy. Within kube-proxy, chains and rules are created to reflect the communication blocking. Algorithm 2 presents an example of iptables configuration that is executed throughout the process. These rules aim to allow incoming traffic on a specific TCP port (443) and mark the network traffic arriving at that port with a mark value (0x10000/0x10000).

Algorithm 2 : Iptables rules in kube-proxy

```
[1] -A cali-po-_dUocG07UHnoYGHwzj1q -p tcp
    -m comment --comment "cali:
   RUtNW8LPCr3T_zE7" -m multiport --
   dports 443 -j MARK --set-xmark 0x10000
   /0x10000
[2] -A cali-po-_dUocG07UHnoYGHwzj1q -m
   comment --comment "cali:
   AX7qYuyJPSjHM8J7" -m mark --mark 0
   x10000/0x10000 -j RETURN
[3] -A cali-po-_dUocG07UHnoYGHwzj1q -m
   comment --comment "cali:
   KVjlLm06huMbM5nt" -m set --
match-set cali40s:CVtTjVoVYfBCPNXq7iYGrlr
   dst -j MARK --set-xmark 0x10000/0
   x10000
[4] -A cali-po-_dUocG07UHnoYGHwzj1q -m
   comment --comment "cali:5
   EN0O5PuSQjj41WP" -m mark --mark 0
   x10000/0x10000 -j RETURN
```

The first rule of Algorithm 2 defines the TCP port (443) and the mark that will be set for the incoming traffic on this port. The second rule verify if the received traffic has the mark defined by the first rule, and then it returns the traffic control to the originating process. The third rule defines a set IP addresses that is allowed to access the port defined by the first rule and applies the previously defined mark. The last rule checks the mark defined in the first rule again, and then it returns the traffic control to the originating process.

The rule mentioned above are part of a larger set of network security policies, which can be used to ensure authorization and security of network traffic.

In Algorithm 3, a list of iptables rules is created to enhance the security and efficiency of data management on Linux systems. In summary, the rules are as follows:

1. Rule [1]: Allows related and established traffic, using the conntrack module to verify the connection state of packets. The ACCEPT action allows packets that

meet these conditions to be forwarded for further processing.
2. Rule [2]: Blocks invalid traffic, identifying packets marked as "INVALID". The DROP action discards these packets, preventing their propagation on the network.
3. Rule [3]: Applies a mark (xmark) to packets, using the MARK action with the parameter –set-xmark to set the mark as 0x0/0x10000. This mark can be used later to apply other specific rules or policies.
4. Rule [4]: Blocks UDP packets encapsulated in VXLAN using port 4789. This rule prevents encapsulated VXLAN packets, originated from specific workloads, from being forwarded, adding an additional layer of security to the network.
5. Rule [5]: Blocks IP-in-IP encapsulated packets originated from specific workloads. This rule discards packets that match this condition, using the -p ipencap parameter to identify the IP-in-IP encapsulation protocol.
6. Rule [6]: Directs traffic to the cali-pro-kns.vote chain without specifying a protocol or any related traffic condition. The chain likely contains other rules to process the packet.
7. Rule [7]: Checks if the packet has been accepted by the security profile by verifying if the packet's mark matches 0x10000/0x10000. If the condition is met, the packet is returned (RETURN) to the normal processing flow.
8. Rule [8]: Routes the packet to the default voting of the Calico authorization service (cali-pro-ksa.vote). The decision to accept or block the packet is determined by the policies and permissions configured for the authorization service.
9. Rule [9]: Performs an immediate return (RETURN) if the associated profile of the packet is accepted. This rule is triggered when the packet has a mark that matches 0x10000/0x10000, interrupting the evaluation of subsequent rules in the chain.
10. Rule [10]: Drops (DROP) packets that do not match any defined profiles. This rule is triggered when no traffic profile is found for the packet, ensuring that only packets meeting specific criteria are allowed.
11. Rule [11]: Utilizes marking (MARK) at the beginning of the firewall policies, using the mark set (–set-xmark) with the value 0x0/0x20000. This marking serves as an indicator to identify the starting point of the applied firewall policies.

In Algorithm 4, the process of blocking ports and protocols is detailed. In summary, the rules are as follows:

1. Rule [1]: Checks if a policy has been accepted. If positive, the packet is returned (RETURN) using the marking (MARK) 0x10000/0x10000. This approach allows the packet to return to the normal processing flow after being processed by a specific policy, assisting in the control and proper routing of traffic based on established policies.

Algorithm 3 : Calico iptables rules in kube-proxy

```
[1] -A cali-fw-cali56dde0ed24e -m comment
    --comment "cali:ZuYBI2b8cXggjkHK" -m
    conntrack --ctstate RELATED,
    ESTABLISHED -j ACCEPT
[2] -A cali-fw-cali56dde0ed24e -m comment
    --comment "cali:HbLByr_5h7-iVibx" -m
    conntrack --ctstate INVALID -j DROP
[3] -A cali-fw-cali56dde0ed24e -m comment
    --comment "cali:QuLrKssNqb4cPtof" -j
    MARK --set-xmark 0x0/0x10000
[4] -A cali-fw-cali56dde0ed24e -p udp -m
    comment --comment "cali:
    vXfahtOQI3y4FFWb" -m comment --comment
    "Drop VXLAN encapped packets
    originating in workloads" -m multiport
    --dports 4789 -j DROP
[5] -A cali-fw-cali56dde0ed24e -p ipencap
    -m comment --comment "cali:6
    H8FrGk4Wk6Ve17w" -m comment --comment
    "Drop IPinIP encapped packets
    originating in workloads" -j DROP
[6] --A cali-fw-cali56dde0ed24e -m comment
    --comment "cali:-4b85r6VZynUeOd4" -j
    cali-pro-kns.vote
[7] --A cali-fw-cali56dde0ed24e -m comment
    --comment "cali:h5PhrImYCuzL6qET" -m
    comment --comment "Return if profile
    accepted" -m mark --mark 0x10000/0
    x10000 -j RETURN
[8] --A cali-fw-cali56dde0ed24e -m comment
    --comment "cali:E4Gsbi4OgMASZZUh" -j
    cali-pro-ksa.vote.default
[9] --A cali-fw-cali56dde0ed24e -m comment
    --comment "cali:0vRfCptgGHBh-1l-" -m
    comment --comment "Return if profile
    accepted" -m mark --mark 0x10000/0
    x10000 -j RETURN
[10] --A cali-fw-cali56dde0ed24e -m
    comment --comment "cali:
    tYr2s6kuv49jm0yz" -m comment --comment
    "Drop if no profiles matched" -j DROP
[11] +-A cali-fw-cali56dde0ed24e -m
    comment --comment "cali:80
    ryJuwKsNJyBhFx" -m comment --comment "
    Start of policies" -j MARK --set-xmark
    0x0/0x20000
```

2. Rule [2]: Marks (MARK) packets with the value 0x0/0x20000, directing them to the specific policy cali-po-_Ipke1sZDAZREYnL9Cs8. This marking allows the packets to be processed according to the rules defined in that policy, facilitating traffic management based on the established configurations.
3. Rule [3]: Checks if the associated policy has been accepted by verifying the marking (MARK) of packets with the value 0x10000/0x10000. If the marking matches, the packets are directed to the RETURN action, continuing their processing according to the established policy.
4. Rule [4]: Aims to discard packets that have not been approved by any policy. This is done by checking the marking (MARK) of the packets and comparing it to the value 0x0/0x20000. If the marking does not match, the packets are directed to the DROP action, indicating that they should be discarded.

5. Rule [5]: Related to the policy voting process in a specific namespace. This rule directs packets to the cali-pro-kns.vote table, where evaluation and decision-making regarding the forwarding of these packets based on the policies defined for the namespace take place.
6. Rule [6]: Related to the packet return process after accepting a security profile. It checks if the marker (mark) assigned to the packets is equal to 0x10000/0x10000, indicating that the security profile has been accepted. In this case, the packets are returned to their source without any additional action.
7. Rule [7]: Related to the voting process to determine whether a packet is allowed or not based on the security policies defined for the default Kubernetes Service Account (KSA) namespace. This rule forwards the packet to the voting table of the default KSA namespace, where the policies are evaluated, and a decision is made.
8. Rule [8]: Aims to return the packet if the associated profile is accepted. It uses marking (mark) to identify packets that have gone through the profiling process and had their profile accepted. When a packet matches this condition, it is returned, meaning it will proceed to subsequent steps of firewall processing or be forwarded to the specified final action in the chain.
9. Rule [9]: Aims to discard the packet if no corresponding profile is found. This rule is applied when no profile associated with the packet is identified, indicating that there is no match with the defined policies or rules. The packet is immediately discarded, preventing any further processing and ensuring it is not allowed in the network.

Algorithm 4 : Calico iptables block rules in kube-proxy

```
[1] -A cali-fw-cali56dde0ed24e -m comment
    --comment
  "cali:ZWtp0l5QKTr_VPXU" -m comment --
    comment
  "Return if policy accepted" -m mark --
    mark
  0x10000/0x10000 -j RETURN
[2] -A cali-fw-cali56dde0ed24e -m comment
    --comment
  "cali:v3-lAQEi-QRHNeZ-" -m mark --mark 0
    x0/0x20000
  -j cali-po-_Ipke1sZDAZREYnL9Cs8
[3] -A cali-fw-cali56dde0ed24e -m comment
    --comment
  "cali:56JZ_F-iaRRshpsl" -m comment --
    comment
  "Return if policy accepted" -m mark --
    mark
  0x10000/0x10000 -j RETURN
[4] -A cali-fw-cali56dde0ed24e -m comment
    --comment
  "cali:dv6l1Ow96eEW40m0" -m comment --
    comment
  "Drop if no policies passed packet" -m
    mark
  --mark 0x0/0x20000 -j DROP
[5] -A cali-fw-cali56dde0ed24e -m comment
    --comment
  "cali:mB6IM8g-g1Nz-OS6" -j cali-pro-kns.
    vote
[6] -A cali-fw-cali56dde0ed24e -m comment
    --comment
  "cali:RLFlTbD-Wm0owtSk" -m comment --
    comment
  "Return if profile accepted" -m mark --
    mark
  0x10000/0x10000 -j RETURN
[7] -A cali-fw-cali56dde0ed24e -m comment
    --comment
  "cali:qKtZvJTKIWtHBwfX" -j cali-pro-ksa.
    vote.default
[8] -A cali-fw-cali56dde0ed24e -m comment
    --comment
  "cali:NKUtuKuCn6gyOB8m" -m comment --
    comment
  "Return if profile accepted" -m mark --
    mark
  0x10000/0x10000 -j RETURN
[9] -A cali-fw-cali56dde0ed24e -m comment
    --comment
  "cali:fH_OupKrlVyQyFBo" -m comment --
    comment
  "Drop if no profiles matched" -j DROP
```

4. Experiment and use case

This section presents experiments using a minikube cluster divided into secure and insecure environments. We conducted experiments in the insecure environment using the default configuration of Kubernetes. While in the secure environment, the same experiments were performed with the addition of SARIK to protect the Pods.

The proposed solution was developed in a controlled environment using a minikube cluster with the following configurations: **Processor:** Intel Core i5-3360M CPU @ 2.80 GHz; Memory: 16 GB DDR3; **Network interface:** Intel 82579LM Gigabit Ethernet; **Storage:** 256 GB SSD; **Operating System:** Ubuntu Mint. In the development of the SARIK framework, shell script language was chosen due to its presence in most Unix systems and cloud computing platforms.

Algorithm 5 presents the kube-system namespace, which is responsible for maintaining key components of Kubernetes, such as: the Calico network controller, CoreDNS, etcd, among others. These components are necessary for the proper functioning of the Kubernetes cluster. The vote namespace is used to maintain the Pods of a voting application. In this experiment, we have the db, redis, result, vote, and worker Pods running in this namespace. Additionally, we can observe in Algorithm 5 network policies applied to these Pods, blocking ports 22, 443, 7, and 80 for each of the Pods that make up the voting application. Finally, all Pods communicate with each other through the Kubernetes Service, which is responsible for exposing these Pods within the cluster, allowing other components of the cluster to communicate with them.

The conducted experiments aim to test the secure and insecure scenarios to evaluate the time required to execute

Algorithm 5 : Pods execution diagram

```
NAME                                        READY   STATUS    RESTARTS        AGE
calico-kube-controllers-58497c65d5-9cd9p    1/1     Running   0               88s
calico-node-wsx4z                           1/1     Running   0               88s
coredns-78fcd69978-kw5xj                    1/1     Running   0               88s
etcd-minikube                               1/1     Running   0               108s
kube-apiserver-minikube                     1/1     Running   0               103s
kube-controller-manager-minikube            1/1     Running   0               101s
kube-proxy-cs58s                            1/1     Running   0               88s
kube-scheduler-minikube                     1/1     Running   0               109s
storage-provisioner                         1/1     Running   1 (46s ago)     83s

NAME                          READY   STATUS    RESTARTS   AGE
db-684b9b49fd-45h76           1/1     Running   0          2m3s
redis-67db9bd79b-9x66h        1/1     Running   0          2m3s
result-77f68799c4-wr5p6       1/1     Running   0          2m3s
vote-79787c6c8b-fz44b         1/1     Running   0          2m2s
worker-78b9ff59fc-58nk6       1/1     Running   0          2m2s

NAME                                          POD-SELECTOR   AGE
block-ports-egress-db-684b9b49fd-45h76-22     app=db         53s
block-ports-egress-db-684b9b49fd-45h76-443    app=db         53s
block-ports-egress-db-684b9b49fd-45h76-7      app=db         53s
block-ports-egress-db-684b9b49fd-45h76-80     app=db         53s
block-ports-egress-redis-67db9bd79b-9x66h-22  app=redis      53s
block-ports-egress-redis-67db9bd79b-9x66h-443 app=redis      53s
block-ports-egress-redis-67db9bd79b-9x66h-7   app=redis      53s
block-ports-egress-redis-67db9bd79b-9x66h-80  app=redis      53s
block-ports-egress-result-77f68799c4-wr5p6-22 app=result     53s
block-ports-egress-result-77f68799c4-wr5p6-443 app=result    53s
block-ports-egress-result-77f68799c4-wr5p6-7  app=result     53s
block-ports-egress-result-77f68799c4-wr5p6-80 app=result     53s
block-ports-egress-vote-79787c6c8b-fz44b-22   app=vote       53s
block-ports-egress-vote-79787c6c8b-fz44b-443  app=vote       53s
block-ports-egress-vote-79787c6c8b-fz44b-7    app=vote       53s
block-ports-egress-vote-79787c6c8b-fz44b-80   app=vote       53s
```

each command and, based on this information, quantify and assess the security of each analyzed group.

In the first experiments, the Pods named "db," "redis," "result," and "vote" were accessed, from which the commands "ping," "curl," "apt," and "wget" were executed within the "vote" Pod and outside the cluster; using the Google website to download and copy files. During the experiments in the insecure scenario, it was observed that the communication between these Pods and outside the cluster occurred correctly, that is, they were able to ping each other and download data from the internet. This shows that, in a default Kubernetes environment, adjacent Pods are vulnerable to attacks that map existing Pods on the node, and the Pods can download content. These attacks can be exploited by attackers to gather information about the cluster's structure, as well as to install malware, posing various security risks. Figure 2 illustrates the communication between the Pods and highlights the vulnerability of adjacent Pods to Pod mapping attacks. It is important to note that, to mitigate this type of vulnerability, network policies can be implemented to block such communication, as done in this work.

In the next experiments, the ports 22, 443, 7, and 80 were blocked in each of the Pods that make up the voting application using network policies to prevent communication between them. During the experiments in the secure scenario, it was observed that the communication between the Pods and outside the cluster was successfully blocked, i.e., the commands "ping," "curl," "apt," and "wget" were not executed. This shows that by implementing network policies to block communication on vulnerable ports, it is possible to increase the security of the Pods and reduce the risk of malicious invasions. Figure 3 illustrates the blocked communication between the Pods and highlights the effectiveness of the network policies in preventing Pod mapping attacks

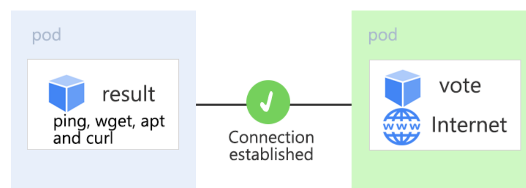


Figure 2: Connection established

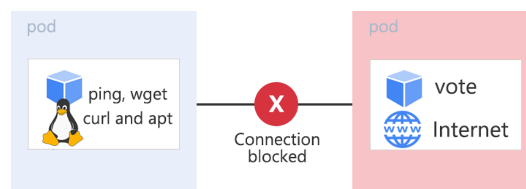


Figure 3: Connection block

	Latency metric		Response rate metric		Transmission rate metric	
Day	**Group 1**	**Group 2**	**Group 1**	**Group 2**	**Group 1**	**Group 2**
1	1.5425	21.1753	3.38343	336.25	10.0823	31.831
2	1.62673	21.1511	3.28344	336.283	10.0817	31.8277
3	1.60383	21.2002	5.68344	469.55	10.0833	31.827
4	1.62517	21.1992	3.38344	336.35	10.082	31.829
5	1.5257	21.1454	2.81678	334.917	10.082	31.8533
6	1.69003	21.153	3.28344	336.283	10.0827	31.847
7	1.60157	21.1489	3.31678	334.95	10.083	31.8473
8	1.52497	21.1483	3.28344	334.917	10.085	31.8537
9	1.54467	21.139	3.38345	336.25	10.0817	31.833
10	1.59287	21.1842	3.35011	336.283	10.0813	31.8423

Table 2
The average of groups 1 (insecure) and 2 (secure) of the metrics from the experiments

and the installation of malware through the "curl," "apt," and "wget" commands.

The experiments generated several tables containing the average execution times of each command according to the metrics detailed in Section 5. These values were collected over ten consecutive days using various scripts that performed the tests and gathered the information. To ensure the accuracy of the results, each command was tested 30 times for both the insecure and secure groups.

In this work, we conducted experiments to evaluate three metrics: transmission rate, latency, and response rate. The transmission rate metric was tested using iperf, while the latency and response rate metrics were assessed by executing four commands, namely “apt,” “curl,” “ping,” and “wget,” on each study Pod. For a more detailed analysis, we took the “db” Pod and the “apt” command as a reference, using Table 2. The results indicate that the average execution time of the commands in Group 1 (insecure), shown in the second column of the latency metric, is lower compared to Group 2 (secure), presented in the third column. This behavior can be attributed to the activation of blocking rules in Group 2, which hinder the quick completion of the commands. This finding highlights the effectiveness of the implemented blocking rules in Group 2, contributing to a more secure environment.

In Group 1 (insecure), tests were conducted without network policies, which means that all Pods in the cluster had unrestricted access to all network resources. The results obtained showed an average time of 1.5878 seconds with a time variation ranging from 1 to 21 seconds for the latency metric. Regarding the response rate metric, the results showed an average time of 3.5167 seconds with a time variation ranging from 3 to 5 seconds. In the transmission rate metric, the average time was 10.0825 seconds with a time variation of 10 seconds, as observed in Table 2. The experiments involved performing certain actions to simulate atypical behavior of an application, such as the Pod downloading or updating the system without the administrator’s authorization (malicious script that accesses the internet to send and receive data).

In Group 2, network policies were enabled, which means that all Pods in the cluster lost network communication access. The results obtained showed an average time of 21.1644 seconds with a time variation of 21 seconds for the latency metric. Regarding the response rate metric, the results showed an average time of 338.0741 seconds with a time variation ranging from 334 to 469 seconds. In the transmission rate metric, the average time was 31.8391 seconds with a time variation of 31 seconds, as observed in 2. The experiments involved blocking the communication of Pods to prevent the previously reported atypical behaviors.

5. Performance evaluation

This section summarizes the performance evaluation results of several tests applied to the experimental data. The findings were based on a confidence interval of 95% ($p < 0.05$). The performance of four Pods was evaluated to measure response rate, latency, and transmission rate in experiments conducted on each Pod. The overall objective of these experiments was to measure the execution time of each metric in seconds and assess whether the Pods are connected or if the traffic is being blocked by active network rules.

After an exploratory analysis of the data, outliers were identified in data from both groups, and upon further review, they were identified as measurement errors. As recommended by Hair et al. [33], the outlier were excluded to avoid influences in the statistical analysis. Also, the exclusion of these data points allowed the data to meet the normality criterion. By improving the fit of the data to a normal distribution, it was possible to use more robust parametric tests.

The data in this study were analyzed using tests for normality and variance homogeneity, using the Jamovi software [34]. Some variables were transformed using the natural logarithm (LOG) to improve their fit to the normality criteria. The next subsections presents a summary of results and findings, additional details of the static analysis are available on the SARIK[1] project page.

5.1. Latency metric

Latency, a key measure for in distributed systems and computer networks, was the first analysed metric, as it represents the delay time between sending a data packet and

[1]SARIK project validation: https://github.com/jonathamgg/

Table 3
Comparison of Statistical Tests in Pods

Comparison of Statistical Tests for Latency Metrics in the Pod db						
Variables	Shapiro-Wilk test	Transformation (LOG)	Levene test	Student's t-test	Welch's t-test	p-value.
apt	$p \geq 0.05$	NO	$p < 0.05$*	NO	YES	$p < 0.05$
curl***	$p \geq 0.05$**	NO	$p \geq 0.05$	YES	NO	$p < 0.05$
ping	$p \geq 0.05$	NO	$p \geq 0.05$	YES	NO	$p < 0.05$
wget***	$p \geq 0.05$**	NO	$p < 0.05$*	NO	YES	$p < 0.05$
Comparison of Statistical Tests for Latency Metrics in the Pod redis						
Variables	Shapiro-Wilk test	Transformation (LOG)	Levene test	Student's t-test	Welch's t-test	p-value.
apt***	$p \geq 0.05$**	NO	$p \geq 0.05$	YES	NO	$p < 0.05$
curl	$p \geq 0.05$	NO	$p < 0.05$*	NO	YES	$p < 0.05$
ping	$p \geq 0.05$	NO	$p \geq 0.05$	YES	NO	$p < 0.05$
wget***	$p \geq 0.05$**	NO	$p \geq 0.05$	YES	NO	$p < 0.05$
Comparison of Statistical Tests for Latency Metrics in the Pod result						
Variables	Shapiro-Wilk test	Transformation (LOG)	Levene test	Student's t-test	Welch's t-test	p-value.
apt***	$p \geq 0.05$**	NO	$p \geq 0.05$	YES	NO	$p < 0.05$
curl***	$p \geq 0.05$**	NO	$p \geq 0.05$	YES	NO	$p < 0.05$
ping	$p \geq 0.05$	NO	$p \geq 0.05$	YES	NO	$p < 0.05$
wget***	$p \geq 0.05$**	NO	$p < 0.05$*	NO	YES	$p < 0.05$
Comparison of Statistical Tests for Latency Metrics in the Pod vote						
Variables	Shapiro-Wilk test	Transformation (LOG)	Levene test	Student's t-test	Welch's t-test	p-value.
apt	$p \geq 0.05$	NO	$p < 0.05$*	NO	YES	$p < 0.05$
curl***	$p \geq 0.05$**	NO	$p < 0.05$*	NO	YES	$p < 0.05$
ping	$p \geq 0.05$	NO	$p \geq 0.05$	YES	NO	$p < 0.05$
wget***	$p \geq 0.05$**	NO	$p < 0.05$*	NO	YES	$p < 0.05$

* The Levene test is significant, suggesting a violation of the assumption of homogeneity of variances. ** Significant normality test indicates a violation of the normality assumption. *** An outlier was excluded to correct the normality in the distribution of the data.

receiving the response by the system. In latency analysis, we used the TIME tool in each Pod, applying regular expressions to filter the corresponding field and obtain the measurement time in seconds, as observed in the latency[2] metric available on the validation project page. This approach allows us to obtain accurate and relevant results for evaluating the performance of network policies in the cluster.

A customized approach using a script that executes the 'kubectl exec' command on each Pod was used to measure the latency in the proposed Kubernetes cluster. Specifically, the 'time apk update' command was used to measure the system's response time. The time data collected were then analyzed using regular expressions, allowing for the precise extraction of the latency metric, represented in seconds. This methodology provided detailed and accurate measurements of the delay in network communication within the cluster, essential for assessing the impact of network policies on system performance.

As shown in Figure 4, the Pod db in Group 1 achieved the average latency of less than 1.5 seconds in three variables, and 7.5 seconds in the PING variable, considering a cycle of 30 experiment repetitions. While Group 2 achieved approximately 20 seconds of latency in the same test cycle. It is worth noting that the CURL and WGET variables in Group 1 exhibited outlier values during the experiment.

[2]Script latency: https://github.com/jonathamgg/

Furthermore, it is noticeable that Group 1 exhibited faster command execution speed as compared with Group 2. This occurred because, unlike Group 2, Group 1 does not have any network policies to be applied and, consequently, affect the latency of data packets.

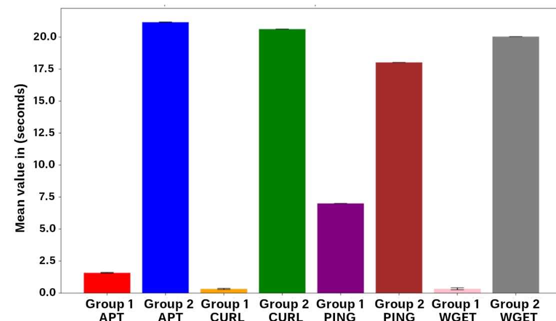


Figure 4: Comparison of mean values between Groups 1 and 2 with 95% confidence intervals in the Pod db.

According to Table 3, related to the Pod db, only the Curl and Wget variables exhibited outliers and non-normal distribution ($p < 0.05$). After excluding outliers, a normal distribution was obtained, and it was not necessary to perform logarithmic transformation. In the Levene's test, only

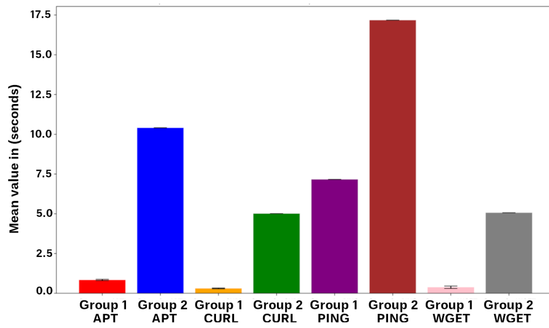


Figure 5: Comparison of mean values between Groups 1 and 2 with 95% confidence intervals in the Pod redis.

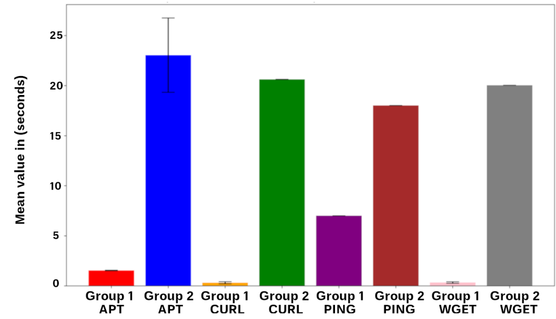


Figure 6: Comparison of mean values between Groups 1 and 2 with 95% confidence intervals in the Pod result.

Curl and Ping showed homogeneous distribution, so they were submitted to the t-test. The other two variables - Apt and Wget - did not have homogeneous distribution, requiring the application of Welch's t-test. The results of the experiments indicated a p-value less than 0.05 for all variables, indicating statistically significant differences between the means of Groups 1 and 2.

As shown in Figure 5, the Pod redis in Group 1 achieved the average latency of less than 1 second in three variables and 7.5 seconds in the PING variable, considering a cycle of 30 experiment repetitions. While Group 2 achieved approximately 5 to 17.5 seconds of average latency in the same test cycle. It is worth noting that the WGET variable in Group 1 exhibited outlier values during the experiment.

According to Table 3, related to the Pod redis, only the Apt and Wget variables exhibited outlier values and non-normal distribution ($p < 0.05$). After excluding outliers, a normal distribution was obtained, without the need to perform a logarithmic transformation. In Levene's test, the Apt, Ping, and Wget variables showed homogeneous distribution, so they were submitted to the t-test. The Curl variable did not have a homogeneous distribution, requiring the application of the t-test with Welch correction. The results of the experiments indicated that the p-value was lower for all variables. All the results showed statistically significant differences ($p < 0.05$).

As shown in Figure 6, the Pod result in Group 1 achieved the average latency of less than 1 second in three variables and 7 seconds in the PING variable, considering a cycle of 30 experiment repetitions. While Group 2 achieved approximately 7 to 21.5 seconds of average latency in the same test cycle. It is worth noting that the CURL and WGET variables in Group 1 and the APT variable in Group 2 exhibited outlier values during the experiment.

According to Table 3, related to the Pod result, the variables Apt, Curl, and Wget exhibited outlier values and non-normal distribution ($p < 0.05$). After excluding the outlier values, the distribution was normalized. It was not necessary to perform a logarithmic transformation. In Levene's test, the variables Apt, Curl, and Ping had homogeneous distribution, so they were submitted to the t-test. However, the variable Wget did not have a homogeneous distribution, which required the application of the t-test with Welch correction. The results of the experiments indicated that the p-value was lower for all variables. All results showed statistically significant differences ($p < 0.05$).

As shown in Figure 7, the Pod vote in Group 1 achieved the average latency below 1 second in three variables and 7.5 seconds in the PING variable, considering a cycle of 30 experiment repetitions. While Group 2 achieved approximately an average latency of approximately 5 to 17.5 seconds in the same test cycle. It is worth noting that the WGET variable in Group 1 exhibited outlier values during the experiment.

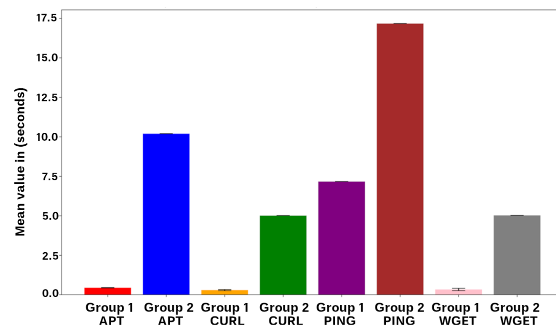


Figure 7: Comparison of mean values between Groups 1 and 2 with 95% confidence intervals in the Pod vote.

According to Table 3, related to the Pod vote, the variables Curl and Wget exhibited outlier values and non-normal distribution ($p < 0.05$). After excluding outliers, a normal distribution was obtained, without the need to perform a logarithmic transformation. In Levene's test, the Ping variable had a homogeneous distribution, so they were submitted to the t-test. The variables Apt, Curl, and Wget did not have a homogeneous distribution, requiring the application of the t-test with Welch correction. The results of the experiments indicated that the p-value was lower for all variables. All the results showed statistically significant differences ($p < 0.05$).

Table 4
Comparison of Statistical Tests in Pods

Comparison of Statistical Tests for Response Rate Metrics in the Pod db						
Variables	Shapiro-Wilk test	Transformation (LOG)	Levene test	Student's t-test	Welch's t-test	p-value.
apt***	$p < 0.05$	YES	$p < 0.05$*	NO	YES	$p < 0.05$
curl***	$p < 0.05$	YES	$p < 0.05$*	NO	YES	$p < 0.05$
ping***	$p < 0.05$	YES	$p < 0.05$*	NO	YES	$p < 0.05$
wget***	$p < 0.05$**	YES	$p < 0.05$*	NO	YES	$p < 0.05$
Comparison of Statistical Tests for Response Rate Metrics in the Pod redis						
Variables	Shapiro-Wilk test	Transformation (LOG)	Levene test	Student's t-test	Welch's t-test	p-value.
apt***	$p < 0.05$	YES	$p < 0.05$*	NO	YES	$p < 0.05$
curl***	$p < 0.05$	YES	$p < 0.05$*	NO	YES	$p < 0.05$
ping***	$p < 0.05$	YES	$p < 0.05$*	NO	YES	$p < 0.05$
wget***	$p < 0.05$	YES	$p < 0.05$*	NO	YES	$p < 0.05$
Comparison of Statistical Tests for Response Rate Metrics in the Pod result						
Variables	Shapiro-Wilk test	Transformation (LOG)	Levene test	Student's t-test	Welch's t-test	p-value.
apt***	$p < 0.05$	YES	$p < 0.05$*	NO	YES	$p < 0.05$
curl***	$p < 0.05$	YES	$p < 0.05$*	NO	YES	$p < 0.05$
ping***	$p < 0.05$	YES	$p < 0.05$*	NO	YES	$p < 0.05$
wget***	$p < 0.05$	YES	$p < 0.05$*	NO	YES	$p < 0.05$
Comparison of Statistical Tests for Response Rate Metrics in the Pod vote						
Variables	Shapiro-Wilk test	Transformation (LOG)	Levene test	Student's t-test	Welch's t-test	p-value.
apt***	$p < 0.05$	YES	$p < 0.05$*	NO	YES	$p < 0.05$
curl***	$p < 0.05$	YES	$p < 0.05$*	NO	YES	$p < 0.05$
ping***	$p < 0.05$	YES	$p < 0.05$*	NO	YES	$p < 0.05$
wget***	$p < 0.05$	YES	$p < 0.05$*	NO	YES	$p < 0.05$

* The Levene test is significant, suggesting a violation of the assumption of homogeneity of variances. ** Significant normality test indicates a violation of the normality assumption. *** An outlier was excluded to correct the normality in the distribution of the data.

5.2. Response rate metric

Just like latency, the response rate is an key metric in the study of distributed systems and computer networks' performance, as it represents the system's ability to respond to requests efficiently. In this analysis, we measured the response rate using DATE tool for each Pod of the application. We used DATE command to obtain and register the start timestamp before initiating the command executed in the Pod. Then, the DATE command is executed in the Pod to send a test request and receive the response. After the DATE command, we record the current time again to obtain the end timestamp. Thus, the response time is the difference between the start and end timestamps. By examining the obtained results, we can evaluate the server's processing capacity, and the quality of the network connection, and identify possible bottlenecks that may affect the response rate. It is important to note that the use of regular expressions is applied to filter the relevant data from the DATE response field, as observed in the response rate metric[3].

As Figure 8 illustrates, the db Pod in Group 1 achieved an average response time of less than 2 seconds for the variables – apt, curl, ping, and wget - considering a cycle of 30 experiment repetitions. While Group 2 recorded an average response time of approximately 350 seconds for the apt variable, and an average of 50 seconds for the other variables in the same test cycle. It is worth noting that the wget variable in Group 1 presented outliers, and in Group 2, the apt, ping, and wget variables showed outliers during the experiment.

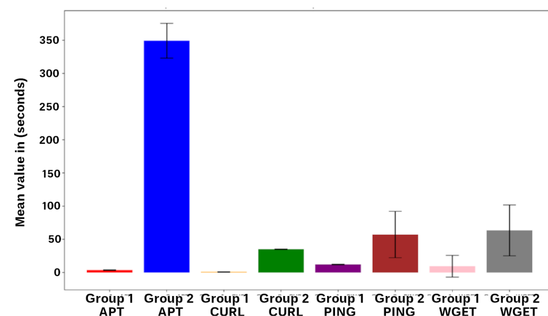


Figure 8: Comparison of mean values between Groups 1 and 2 with 95% confidence intervals in the Pod db in response rate metric.

As Figure 9 illustrates, the redis Pod in Group 1 achieved an average response time of less than 5 seconds for the variables – apt, curl, and wget - and 24 seconds for the ping variable, considering a cycle of 30 experiment repetitions.

[3]Script response rate metric: https://github.com/jonathamgg/

While Group 2 recorded an average response time of approximately 29 seconds for the ping variable, and an average of 9 to 19 seconds for the other variables in the same test cycle. It is worth noting that all variables in both Group 1 and Group 2 presented outliers during the experiment.

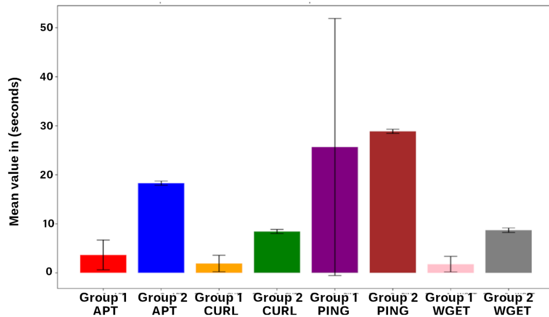


Figure 9: Comparison of mean values between Groups 1 and 2 with 95% confidence intervals in the Pod redis in response rate metric.

As Figure 10 illustrates, the result Pod in Group 1 achieved an average response time of less than 1 second for the variables – apt, curl, and wget - and 25 seconds for the ping variable, considering a cycle of 30 experiment repetitions. While Group 2 recorded an average response time of approximately 65 seconds for the variables – curl, ping, and wget - and 340 seconds for apt in the same test cycle. It is worth noting that the variables apt and curl in Group 1 did not have outliers during the experiment, while the other variables presented outliers.

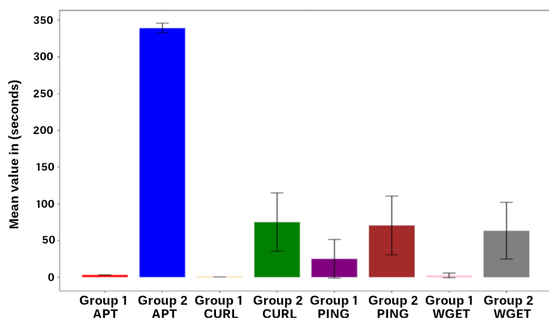


Figure 10: Comparison of mean values between Groups 1 and 2 with 95% confidence intervals in the Pod result in response rate metric.

As Figure 11 illustrates, the vote Pod in Group 1 achieved the average response time ranging from 2 to 11 seconds for the variables - apt, curl, ping, and wget - considering a cycle of 30 experiment repetitions. While Group 2 recorded an average response time of approximately 9 to 41 seconds for the variables - apt, curl, and wget - and 58 seconds for ping in the same test cycle. It is worth noting that all variables presented outliers during the experiment.

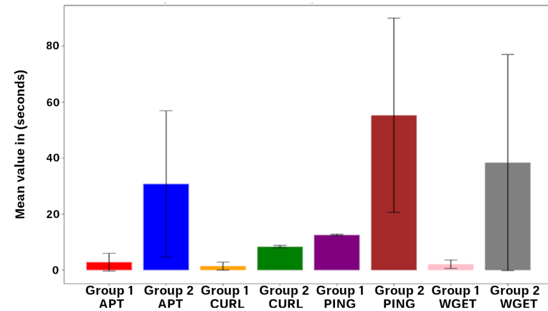


Figure 11: Comparison of mean values between Groups 1 and 2 with 95% confidence intervals in the Pod vote in response rate metric.

According to Table 4, related to the db, redis, result, and vote Pods, all variables presented outliers and non-normal distribution ($p < 0.05$). Thus, it was necessary to perform logarithmic transformations. In the Levene test, the variables did not have homogeneous distribution, which required the application of the t-test with Welch correction. The results of the experiments indicated that the p-value was less than 0.05 for all variables in all Pods. All results showed statistically significant differences ($p < 0.05$).

5.3. Transmission rate metric

The transmission rate is another key metric for evaluating the data transfer speed between devices in a network. In the context of this study, the iperf3 tool was used in the cluster and application Pod's to measure this rate. When performing the transmission test, the "time" command is used to record the start and end times of the test, which allows the calculation of the time difference between them. This difference provides us with the necessary information to determine the transmission rate. The analysis of the results, obtained through regular expressions applied to the iperf3 output, is essential for a precise and detailed understanding of the system's transmission rate.

The results presented in Table 5 indicate that the assumption of normality for the Pod redis was violated ($p < 0.05$), therefore, a logarithmic transformation was applied. Levene's test indicated heterogeneity of variances for all variable, requiring the application of the t-test with Welch's correction. All the results showed statistically significant differences ($p < 0.05$).

Figure 12 shows that the Pods in Group 1 achieved the average transmission time of around 10 seconds, with a transmission rate of 10 Gbps in a cycle of 30 repetitions. While Group 2 achieved the average transmission time of approximately 30 seconds in the same test cycle, maintaining a transmission rate of 10 Gbps. It is important to note that the redis Pod in Group 1 presented outliers, preventing the execution of the t-test for this specific Pod.

Table 5
Comparison of Statistical Tests for Transmission Rate Metrics in Pod

Variables	Shapiro-Wilk test	Transformation (LOG)	Levene test	Student's t-test	Welch's t-test	p-value.
db	$p \geq 0.05$	NO	$p < 0.05$*	NO	YES	$p < 0.05$
redis	$p < 0.05$	YES	$p < 0.05$*	NO	YES	$p < 0.05$
result	$p \geq 0.05$	NO	$p < 0.05$*	NO	YES	$p < 0.05$
vote	$p \geq 0.05$	NO	$p < 0.05$*	NO	YES	$p < 0.05$

* The Levene test is significant, suggesting a violation of the assumption of homogeneity of variances. ** Significant normality test indicates a violation of the normality assumption. *** An outlier was excluded to correct the normality in the distribution of the data.

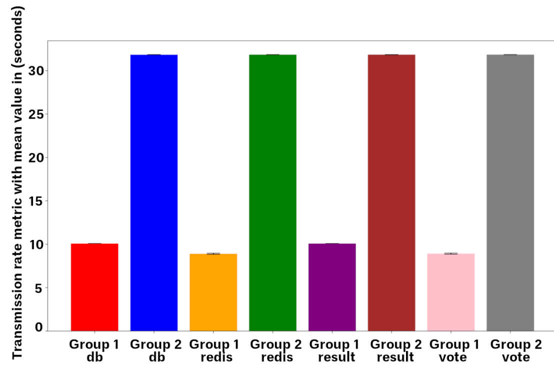


Figure 12: Comparison of mean values between Groups 1 and 2 with 95% confidence intervals in the Pods

In the context of this study, it was observed that the use of network policies led to statistical disparities in the analyzed metrics. The policies influence the data flow, affecting latency, response rate, and transmission rate. However, the graphs consistently demonstrated that the network policies effectively restricted unwanted traffic. Despite the statistical differences, a clear distinction can be observed between the groups with and without network policies, highlighting the ability of these policies to control and direct traffic efficiently. This analysis of the graphs reinforces the importance of network policies in ensuring security and quality of service in distributed environments, even if some metrics may exhibit disparate values.

6. Discussions and Implications

This section covers the discussion of the results obtained in this research, presenting the limitations encountered, the strengths identified, as well as the future challenges that arose during the study. Additionally, the main results of the performance evaluation of the analyzed metrics are summarized. The objective of this section is to conduct a critical analysis of the impact of security rules in low-workload environments, as well as in high-demand environments. To support the discussions and gain meaningful insights, monitoring tools such as Grafana and Prometheus were used. These tools provided valuable data that contributed to the understanding of the results obtained in the experiment.

The default configuration of Minikube disables all "addons," making it challenging to analyze data through tools like Grafana and Prometheus. To ensure more accurate analysis and discussions, it was necessary to enable the metrics-server and pod-security-policy addons using the commands "minikube addons enable metrics-server" and "minikube addons enable pod-security-policy." These traffic monitoring tools allow monitoring of CPU, memory, and network consumption in the cluster.

A namespace called "monitoring" was created to host two essential Pods for the experiment monitoring: Grafana and Prometheus, as observed in Algorithm 6. The Prometheus Pod is responsible for collecting data from the cluster, while the Grafana Pod displays the results. A sample dashboard was developed to visualize this data appropriately, as seen in Figure 13.

Algorithm 6 : Prometheus and grafana

```
jonathan@jonathan-HP:~/monitoring$ minikube service list
|-------------|--------------------|-------------------|----------------------------|
| NAMESPACE   | NAME               | TARGET PORT       | URL                        |
|-------------|--------------------|-------------------|----------------------------|
| default     | kubernetes         | No node port      |
| kube-system | kube-dns           | No node port      |
| kube-system | kube-state-metrics | No node port      |
| kube-system | metrics-server     | No node port      |
| monitoring  | grafana            |              3000 | http://192.168.39.131:32000 |
| monitoring  | prometheus-service |              8080 | http://192.168.39.131:30000 |
| vote        | db                 | No node port      |
| vote        | redis              | No node port      |
| vote        | result             | result-service/5001 | http://192.168.39.131:31001 |
| vote        | vote               | vote-service/5000 | http://192.168.39.131:31000 |
|-------------|--------------------|-------------------|----------------------------|
```

It is important to highlight that a new test cycle was conducted with the addition of these add-ons and the monitoring of cluster resources. This approach allowed for a more comprehensive and accurate analysis of the system's performance.

In this work, the curl and wget commands were used to retrieve data from the internet, encompassing the latency and response rate metrics. In order to determine the number of hops required to reach the target domain with these commands, the traceroute command was used to obtain information about the hops made in the network. However, it is important to note that the db and result Pods were not evaluated due to the inability to install either of these tools, "traceroute" or "mtr" (My TraceRoute), in these Pods as they use the Debian operating system. Only the redis and vote Pods, which operate with the Alpine operating system, were evaluated.

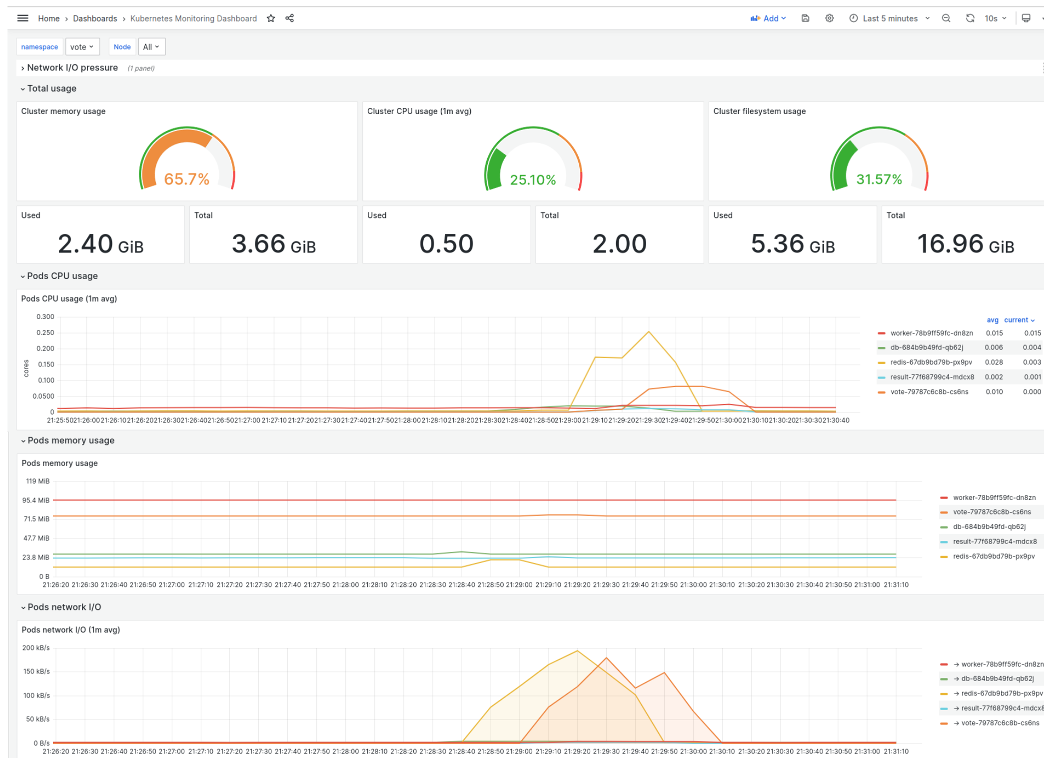


Figure 13: Dashboard grafana

The obtained results presented a sequence of hops, representing the path taken by the data packets, as illustrated in Algorithm 7 and 8. Initially, the packets were sent to IP addresses in the local Kubernetes network, possibly indicating internal nodes. They then went through local routers and default gateways of the local network. However, some hops did not receive a response, which could be attributed to specific router or firewall configurations. After these hops, the packets proceed to a service provider network or an internet service provider, traversing multiple intermediate routers. Subsequently, more hops were observed through routers and intermediate nodes until they reached the servers of the internet providers. In one of the hops, there was again no response from the packet. Finally, the packet reached the desired destination, the domain www.google.com.

Based on the collected information, it was determined that the total number of hops to the destination was 12 and 13, respectively. These results demonstrate that the latency and response rate metrics showed an average time variation of less than 1 second, considering a network connection of 400 Mbps download and 200 Mbps upload for this cluster. During the execution of the experiment, it was observed through Grafana that the CPU and memory usage in the two evaluated Pods remained stable. These insights provided valuable information about the network infrastructure used to establish the connection to the destination domain, as well as highlighting the elapsed time between the source and the destination.

Considering the same scenario with blocked ports and protocols and the same number of hops to the destination for the two described metrics, a time variation of less than 5 seconds was evidenced, considering a network connection of 400 Mbps download and 200 Mbps upload for this cluster. The 4-second delay compared to the first scenario can be attributed to the TCP protocol, which is connection-oriented. Even if there is a blocking rule in the packet's path, the TCP protocol will retransmit the packet or wait for a determined time until the request is completed or discarded. Therefore, during the execution of the experiment with blocked ports and protocols, it was observed through Grafana that the CPU and memory usage in the two evaluated Pods remained stable.

Iperf3 was used to evaluate the throughput metric in the context of communication between Kubernetes cluster devices. The results in Figure 12 revealed significant differences between groups 1 (unsecured) and 2 (secured) in the cluster. In group 1, where blocking rules were not active, we recorded a constant throughput of 10 Gbps, with an average of 10 seconds to complete the tests, suggesting a potential vulnerability to unauthorized traffic. On the other hand, in group 2, where blocking rules were in effect, the throughput remained stable but with longer transmission times, averaging 30 seconds, due to packet retransmission until a timeout occurred, highlighting the effectiveness of network policies in containing unwanted traffic. These results emphasize the critical importance of network security policies in the Kubernetes environment, not only influencing throughput but also ensuring safer communication among cluster devices.

Algorithm 7 : Traceroute in Pod redis

```
traceroute to www.google.com
    (142.251.129.100), 30 hops max, 46
    byte packets
[1] 192-168-39-131.kubernetes.default.svc.
    cluster.local (192.168.39.131)
0.010 ms 0.009 ms 0.008 ms
[2] jonathan-HP.local (192.168.122.1)
    0.301 ms 0.124 ms 0.115 ms
[3] _gateway (192.168.100.1) 0.747 ms
    0.599 ms 0.555 ms
[4] * * *
[5] 100.120.68.108 (100.120.68.108) 5.053
    ms 100.120.67.180 (100.120.67.180)
5.047 ms 5.150 ms
[6] 100.120.21.85 (100.120.21.85) 4.897 ms
     100.120.21.79 (100.120.21.79)
7.757 ms 100.120.21.85 (100.120.21.85)
    4.863 ms
[7] 100.120.23.69 (100.120.23.69) 27.228
    ms 100.120.26.67 (100.120.26.67)
20.607 ms 100.120.31.157 (100.120.31.157)
    24.641 ms
[8] 100.120.20.182 (100.120.20.182) 28.057
     ms 100.120.26.190 (100.120.26.190)
24.244 ms 100.120.31.134 (100.120.31.134)
    20.532 ms
[9] 72.14.198.152 (72.14.198.152) 24.869
    ms 201.10.242.247 (201.10.242.247)
25.122 ms 24.872 ms
[10] * 74.125.243.65 (74.125.243.65)
    26.657 ms 28.055 ms
[11] 209.85.143.205 (209.85.143.205)
    21.479 ms 24.758 ms
216.239.46.49 (216.239.46.49) 21.913 ms
[12] gru14s30-in-f4.1e100.net
    (142.251.129.100) 21.440 ms 21.772 ms
216.239.46.49 (216.239.46.49) 21.682 ms
```

Due to the distinct characteristics of the Alpine Linux distributions used for the implementation of the system tested in the "redis" and "vote" Pods and the Debian distribution in the "db" and "result" Pods, potential limitations such as kernel configurations, drivers, compilation optimizations, software versions, network settings, system load, resource management, security policies, and the balance between lightweight and resource-intensive approaches may have caused disparities in execution times. These distortions can be attributed to the diverse nature of the distributions, each adopting its own approach to lightweights, optimization, network configurations, security policies, and resource allocation.

This demonstrates the influence of the choice of distribution on the performance of applications and processes in a Kubernetes environment, highlighting the importance of carefully considering the characteristics of the operating system during the planning and optimization of container deployments. In the proposed scenario, Group 1 involved commands (apt, ping, curl, and wget) that were executed without interference. The scripts in this group can successfully interact with the resources of the "Alpine" and "Debian" operating systems. The execution times of these commands can be influenced by various factors, such as script efficiency and network speed, among other aspects.

Algorithm 8 : Traceroute in Pod vote

```
traceroute to www.google.com
    (142.250.219.4), 30 hops max, 46 byte
    packets
[1] 192-168-39-131.kubernetes.default.svc.
    cluster.local (192.168.39.131)
0.006 ms 0.006 ms 0.005 ms
[2] jonathan-HP.local (192.168.122.1)
    0.156 ms 0.202 ms 0.142 ms
[3] _gateway (192.168.100.1) 0.654 ms
    0.837 ms 1.696 ms
[4] * * *
[5] 100.120.68.108 (100.120.68.108) 4.830
    ms 100.120.68.106 (100.120.68.106)
5.977 ms 100.120.71.154 (100.120.71.154)
    3.693 ms
[6] 100.120.18.199 (100.120.18.199) 5.485
    ms 177.2.210.53 (177.2.210.53)
21.800 ms 100.120.18.201 (100.120.18.201)
    4.522 ms
[7] 100.120.23.67 (100.120.23.67) 19.498
    ms 100.120.23.69 (100.120.23.69)
20.107 ms 100.120.22.206 (100.120.22.206)
    25.430 ms
[8] 100.120.25.64 (100.120.25.64) 28.710
    ms 100.120.20.240 (100.120.20.240)
42.026 ms 100.120.25.62 (100.120.25.62)
    28.727 ms
[9] 72.14.198.152 (72.14.198.152) 24.475
    ms 20.694 ms 201.10.242.247
(201.10.242.247) 30.837 ms
[10] 74.125.243.65 (74.125.243.65) 25.074
    ms 74.125.243.1 (74.125.243.1)
27.314 ms 26.561 ms
[11] 209.85.251.5 (209.85.251.5) 29.130 ms
     209.85.250.243 (209.85.250.243)
28.443 ms 216.239.56.46 (216.239.56.46)
    34.628 ms
[12] 209.85.251.5 (209.85.251.5) 28.446 ms
     108.170.245.141 (108.170.245.141)
28.963 ms 209.85.251.5 (209.85.251.5)
    28.735 ms
[13] gru14s27-in-f4.1e100.net
    (142.250.219.4) 24.919 ms
    142.250.227.231
(142.250.227.231) 25.498 ms 216.239.54.143
     (216.239.54.143) 25.919 ms
```

In contrast, the same commands in Group 2 are subject to restrictions that increase the execution time. The blocking of these commands in Group 2 is due to security policy configurations implemented in kube-proxy that affect the "Alpine" and "Debian" distributions. This approach proves to be intriguing when simulating the functionalities of the SARIK framework, providing a deeper understanding of how this framework can impact response times in each command.

The distribution of the "Alpine" operating system also plays a significant role in the observed disparity in execution times between the groups. Minimalistic distributions like Alpine Linux are known for being lightweight and efficient. However, the absence of certain resources or libraries can impact the performance of specific commands, leading to variations in execution times.

Regarding the Debian operating system distribution, it is more comprehensive and includes more features compared

to Alpine Linux. This can result in different execution times for commands, as Debian is a more complete distribution, potentially heavier. Additional libraries and dependencies on Debian can also influence command execution times.

The analysis of the results obtained in this study reveals important discussions and implications related to the implementation of firewall rules in the Kubernetes cluster and their impact on system performance and security. When introducing new rules into iptables to control network traffic and strengthen security, it is crucial to consider the side effects that these changes may have.

A relevant discussion is the impact on cluster resource consumption, such as CPU and memory. Adding firewall rules can increase the processing load required to inspect and filter network traffic. This can result in higher CPU usage, affecting the overall system performance. Additionally, firewall rules may require additional memory allocation to store state information and connection contexts. Therefore, it is crucial to conduct careful testing and tuning to ensure that firewall rules are efficient in terms of resource consumption.

Another important implication is the potential latency that firewall rules can introduce into the processing of network requests. As network traffic is inspected and filtered based on the defined rules, there can be a slight additional latency. While this delay may be negligible in some cases, in time-sensitive environments such as real-time systems or low-latency communication, it can significantly impact performance and user experience. Therefore, a careful balance is required between the security provided by firewall rules and the acceptable latency for the specific application.

Furthermore, it's essential to consider the scalability and maintenance of firewall rules. As the Kubernetes cluster grows in size and complexity, with a higher number of Pods and nodes, managing firewall rules can become a challenge. Adding or removing rules in a large-scale environment requires a proper approach to avoid conflicts, inconsistencies, and to ensure compliance with security policies. The use of automated tools and the adoption of best practices in firewall management are crucial for maintaining the security and integrity of the cluster in the long term.

Therefore, it is crucial to acknowledge the presence of various security risks that can affect Pods if not properly planned and managed. These risks encompass a wide range of considerations, from the careful choice of the operating system to the use of compromised container images, the implementation of individual namespaces, and the possibility of privilege escalation within the cluster, among other potential threats.

According to the authors Shamim et al. [35], whose systematic review emphasized the best security practices in Kubernetes environments, it is of paramount importance to incorporate these practices from the beginning to the completion of any Kubernetes implementation. This proactive approach not only helps to prevent security breaches, but also significantly contributes to risk mitigation in distributed environments. Careful consideration of these best practices is essential to ensure the ongoing robustness and security of Kubernetes-based systems.

6.1. Practical guidelines for implementing Network Policies in Kubernetes

Effectively implementing network policies in Kubernetes environments is a complex task that requires a clear understanding of the processes and challenges involved. In this subsection, we detail a step-by-step guide, inspired by best practices and the experience gained from developing the SARIK framework:

Understanding Network Policies in Kubernetes: It's fundamental to start by comprehending the concept of network policies in Kubernetes, which are rules defining the communication between Pods and other network endpoints, applied at the namespace level.

Checking System Prerequisites: Ensure that your Kubernetes cluster is operational and that a Container Network Interface (CNI) compatible network plugin, such as Calico or Cilium, is available.

Selecting and Configuring the CNI Plugin: Choose a CNI plugin that meets the specific needs of your environment and configure it according to its documentation.

Developing a Network Policy Plan: Create a detailed plan for the network policies, identifying the communication requirements between the Pods and defining the policies in YAML object format.

Implementing and Applying Network Policies: Use the kubectl tool to apply the network policies to your cluster. This step is crucial and should be done carefully to avoid disrupting existing services.

Testing and Validating Implemented Policies: After applying the policies, it's vital to test them to ensure they are functioning as expected. Adjust them based on the test results and feedback.

Ongoing Maintenance and Policy Review: Network policies should be regularly reviewed and updated to ensure they continue to be effective and aligned with changes in the environment and security needs.

This guide provides a framework to assist system administrators and developers in implementing network policies in Kubernetes effectively and securely, taking into account the complexity and variability of modern IT environments.

6.2. Challenges in Implementing Network Policies and Solutions Proposed by SARIK

Effective implementation of network policies in Kubernetes involves several complex challenges, which the SARIK framework seeks to solve:

Complexity and Scalability: In Kubernetes environments, especially in large-scale clusters, managing network policies can become a complex task. SARIK simplifies this process by introducing an intuitive interface and automation mechanisms that facilitate the configuration and management of policies. This approach significantly reduces the margin for error and enables system administrators to cope with the inherent complexity of large infrastructures.

Continuous Monitoring and Maintenance: Constant monitoring of network policies is crucial for system security. SARIK incorporates advanced real-time monitoring features, allowing administrators to track and respond quickly to changes in the environment. These monitoring tools assist in identifying and addressing security issues, ensuring that policies remain effective and up-to-date.

Container Security and Network Isolation: Proper security and isolation of containers are fundamental in protecting Kubernetes environments from external and internal threats. SARIK strengthens network isolation by implementing security rules that rigorously control access and communication between containers. This approach minimizes the attack surface and reduces the likelihood of system compromise.

Implementation and Testing Challenges: Applying network policies can be complex, especially in production environments where errors can have serious consequences. SARIK provides a safe and controlled environment for the implementation and testing of policies, ensuring that changes can be safely and effectively applied before being rolled out to the production environment.

Adaptation to Different Operational Scenarios: Each Kubernetes environment has its own specific needs and challenges. SARIK offers the necessary flexibility to adapt to a variety of operational scenarios, allowing administrators and developers to configure policies that meet the specific needs of their environments.

6.3. Practical and theoretical implications of policy-based network controls in Kubernetes and IoT security

This subsection delves into the practical and theoretical implications of implementing policy-based network controls in Kubernetes environments, focusing on enhanced security for IoT devices.

Enhancing Kubernetes security: The implementation of detailed network policies, as facilitated by SARIK, significantly improves security in Kubernetes environments. This approach not only allows for stricter network segmentation, crucial in mitigating targeted attacks and ensuring service integrity but also emphasizes automation as a force multiplier. Automation in the creation and application of network policies provides an efficient mechanism to maintain security in dynamic environments, essential in scenarios with frequent changes in applications and network configurations.

Relevance to IoT device security: The increasing integration of IoT devices into Kubernetes infrastructures highlights the need for robust network policies. Implementing these policies ensures that IoT devices operate in a secure environment, protecting them from intrusions and unauthorized access. Furthermore, the evolving nature of the IoT ecosystem demands a solution that can adapt and respond swiftly to changes, a key aspect of SARIK's design. This ensures effective protection against emerging threats and maintains the security of IoT devices.

Final considerations: The use of automated policy-based network controls, particularly in the context of Kubernetes, brings significant implications for IoT device security. Solutions like SARIK enable achieving a balance between robust security and operational flexibility. This not only enhances the protection of Kubernetes environments but also sets new security standards for IoT ecosystems, characterized by their diversity and constant evolution. This integrated approach is essential for ensuring a more secure and resilient future for IT infrastructures encompassing both Kubernetes and IoT.

7. Conclusion and Future Work

In this work, we used and evaluated SARIK, a framework developed for the automatic configuration of network policies in Kubernetes clusters. The main goal of SARIK was to ensure the protection of Pods within the Kubernetes cluster by applying blocking rules to each Pod. These rules were implemented to block undesired communication through the outbound interface of the Pods, ensuring the security and integrity of the Pods in the Kubernetes environment. During the conducted experiments and analyses, the importance of performance metrics such as latency, response rate, and transmission rate became evident in the evaluation and optimization of distributed systems and computer networks. A precise measurements and analysis of these metrics provided valuable insights into the system's behavior and the effectiveness of the network policies implemented by SARIK.

It is important to note that statistical discrepancies in the metrics may occur due to the influence of network policies on the system's performance. However, the consistent graphs and results obtained demonstrated that the network policies implemented by SARIK were effective in restricting unwanted traffic, highlighting the importance of their implementation to ensure security and service quality. In future work, the need to implement and analyze network policies for the inbound interface of Pods is emphasized. In the scope of this work, we focused on blocking the outbound interface, so it is crucial to explore rules that can prevent denial-of-service attacks and further improve the system's protection and performance in Kubernetes environments. Finally, recognizing the importance of deepening our understanding and evaluation, we intend to expand our experimental setup to analyze the SARIK in greater detail, adding new metrics to the scenario analyzed.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Please see the link:
https://github.com/jonathamgg/sarik_validation_graphics

References

[1] Andreas Bardoutsos, Gabriel Filios, Ioannis Katsidimas, Thomas Krousarlis, Sotiris Nikoletseas, and Pantelis Tzamalis. A multidimensional human-centric framework for environmental intelligence: Air pollution and noise in smart cities. In *2020 16th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pages 155–164. IEEE, 2020.

[2] Muntaha Alawneh and Imad M Abbadi. Expanding devsecops practices and clarifying the concepts within kubernetes ecosystem. In *2022 Ninth International Conference on Software Defined Systems (SDS)*, pages 1–7. IEEE, 2022.

[3] Shazibul Islam Shamim. Mitigating security attacks in kubernetes manifests for security best practices violation. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1689–1690, 2021.

[4] Hui Zhu and Christian Gehrmann. Kub-sec, an automatic kubernetes cluster apparmor profile generation engine. In *2022 14th International Conference on COMmunication Systems & NETworkS (COMSNETS)*, pages 129–137. IEEE, 2022.

[5] The Kernel security. Apparmor. https://www.kernel.org/doc/html/latest/admin-guide/LSM/apparmor.html. Access Date March, 2023. [Online].

[6] Sysdig secure. https://docs.sysdig.com/en/docs/sysdig-secure. Access Date March, 2023. [Online].

[7] Ruriko Kudo, Hirokuni Kitahara, Kugamoorthy Gajananan, and Yuji Watanabe. Integrity protection for kubernetes resource based on digital signature. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, pages 288–296. IEEE, 2021.

[8] Jonathan GP dos Santos, Geraldo P Rocha Filho, and Vinícius P Goncalves. Sarik-framework para automatizar a segurança em ambientes de orquestracao kubernetes. In *Anais Estendidos do XL Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, pages 57–64. SBC, 2022.

[9] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes: Lessons learned from three container-management systems over a decade. *Queue*, 14(1):70–93, 2016.

[10] Víctor Medel, Omer Rana, José Ángel Bañares, and Unai Arronategui. Modelling performance & resource management in kubernetes. In *Proceedings of the 9th International Conference on Utility and Cloud Computing*, pages 257–262, 2016.

[11] Chia-Chen Chang, Shun-Ren Yang, En-Hau Yeh, Phone Lin, and Jeu-Yih Jeng. A kubernetes-based monitoring platform for dynamic cloud resource provisioning. In *GLOBECOM 2017-2017 IEEE Global Communications Conference*, pages 1–6. IEEE, 2017.

[12] Leila Abdollahi Vayghan, Mohamed Aymen Saied, Maria Toeroe, and Ferhat Khendek. Deploying microservice based applications with kubernetes: Experiments and lessons learned. In *2018 IEEE 11th international conference on cloud computing (CLOUD)*, pages 970–973. IEEE, 2018.

[13] Shapna Muralidharan, Gyuwon Song, and Heedong Ko. Monitoring and managing iot applications in smart cities using kubernetes. *Cloud Computing*, 11, 2019.

[14] Zhang Wei-guo, Ma Xi-lin, and Zhang Jin-zhong. Research on kubernetes' resource scheduling scheme. In *Proceedings of the 8th International Conference on Communication and Network Security*, pages 144–148, 2018.

[15] Felipe Balabanian and Marco Henriques. Tocker: framework para a segurança de containers docker. In *Anais Estendidos do XIX Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais*, pages 145–154. SBC, 2019.

[16] Jaehyun Nam, Seungsoo Lee, Hyunmin Seo, Phil Porras, Vinod Yegneswaran, and Seungwon Shin. {BASTION}: A security enforcement network stack for container networks. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 81–95, 2020.

[17] RGKP Kulathunga. *Dynamic security model for container orchestration platform*. PhD thesis, 2021.

[18] Savio Levy Rocha, Fabio Lucio Lopes de Mendonca, Ricardo Staciarini Puttini, Rafael Rabelo Nunes, and Georges Daniel Amvame Nze. Dcids—distributed container ids. *Applied Sciences*, 13(16):9301, 2023.

[19] Daniele Bringhenti, Riccardo Sisto, and Fulvio Valenza. Security automation for multi-cluster orchestration in kubernetes. In *2023 IEEE 9th International Conference on Network Softwarization (NetSoft)*, pages 480–485. IEEE, 2023.

[20] Yifan Li, Xiaohe Hu, Chengjun Jia, Kai Wang, and Jun Li. Kano: Efficient cloud native network policy verification. *IEEE Transactions on Network and Service Management*, 2022.

[21] Seungsoo Lee and Jaehyun Nam. Kunerva: Automated network policy discovery framework for containers. *IEEE Access*, 2023.

[22] Gerald Budigiri, Christoph Baumann, Jan Tobias Mühlberg, Eddy Truyen, and Wouter Joosen. Network policies in kubernetes: Performance evaluation and security analysis. In *2021 Joint European Conference on Networks and Communications & 6G Summit (EuCNC/6G Summit)*, pages 407–412. IEEE, 2021.

[23] Liz Rice. *Container security: Fundamental technology concepts that protect containerized applications*. " O'Reilly Media, Inc.", 2020.

[24] Minikube - network policy. https://minikube.sigs.k8s.io/docs/handbook/network_policy/. Access Date March, 2023. [Online].

[25] Shixiong Qi, Sameer G Kulkarni, and KK Ramakrishnan. Understanding container network interface plugins: design considerations and performance. In *2020 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN*, pages 1–6. IEEE, 2020.

[26] Flannel - network plugin. https://github.com/flannel-io/flannel. Access Date March, 2023. [Online].

[27] Weave - network plugin. https://github.com/weaveworks/weave. Access Date March, 2023. [Online].

[28] Cilium - network plugin. https://github.com/cilium/cilium. Access Date March, 2023. [Online].

[29] Calico - network plugin. https://github.com/projectcalico/calico. Access Date March, 2023. [Online].

[30] David Soldani, Petrit Nahi, Hami Bour, Saber Jafarizadeh, Mohammed Soliman, Leonardo Di Giovanna, Francesco Monaco, Giuseppe Ognibene, and Fulvio Risso. ebpf: A new approach to cloud-native observability, networking and security for current (5g) and future mobile networks (6g and beyond). *IEEE Access*, 2023.

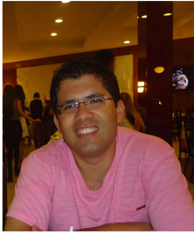
[31] Timothy D Zavarella. *A methodology for using eBPF to efficiently monitor network behavior in Linux Kubernetes clusters*. PhD thesis, Massachusetts Institute of Technology, 2022.

[32] J.F.N. Vitalino. *Descomplicando o Kubernetes [Online]*. LINUXtips, 2020.

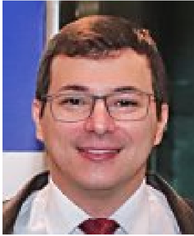
[33] Joseph F Hair, William C Black, Barry J Babin, Rolph E Anderson, and Ronald L Tatham. *Análise multivariada de dados*. Bookman editora, 2009.

[34] Jamovi open statistical software for the desktop and cloud. https://www.jamovi.org/. Access Date March, 2023. [Online].

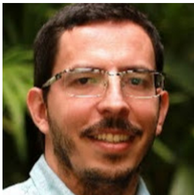
[35] Md Shazibul Islam Shamim, Farzana Ahamed Bhuiyan, and Akond Rahman. Xi commandments of kubernetes security: A systematization of knowledge related to kubernetes security practices. *2020 IEEE Secure Development (SecDev)*, pages 58–64, 2020.



Jonathan Santos (https://orcid.org/0000-0003-1830-0055) Currently, a master's student in the Professional Postgraduate Program in Electrical Engineering - PPEE at the University of Brasília. Possess a Postgraduate degree in Information Security Management from the University of Brasília, an MBA in IT Governance from UNIESP, and an undergraduate degree in Computer Network Technology from FAJESU. Currently working as an Information Technology Technician at the University of Brasília. Interests: Internet of things, networks, cybersecurity, micro-services with container.



Geraldo P. Rocha Filho (https://orcid.org/0000-0001-6795-2768) is a Professor at the Department of Exact and Technological Sciences at the State University of Southwest Bahia (UESB). He was an effective Professor (2019-2022) at the computer science department at the University of Brasília (UnB). He was Researcher at the Institute of Computing at UNICAMP through the Post-Doctorate funded by FAPESP. He obtained the title of Doctor and Master in Computer Science and Computational Mathematics from ICMC-USP with a FAPESP scholarship. In the last five years, he has obtained 24+ publications in international journals and 32+ publications in conferences. His research interests are wireless sensor networks, vehicular networks, smart grids, smart home, and machine learning.



Rodolfo I. Meneguette (https://orcid.org/0000-0003-2982-4006) is a professor at University of São Paulo (USP). He received his Bachelor's degree in Computer Science from the Paulista University (UNIP), Brazil, in 2006. He received his master's degree in 2009 from the Federal University of São Carlos (UFSCar). He received his doctorate from the University of Campinas (Unicamp), Brazil, in 2013. In 2017 he did his post-doctorate in the PARADISE Research Laboratory, University of Ottawa, Canada. His research interest are in the areas of vehicular networks, resources management, flow of mobility, and vehicular clouds.

Rodrigo Bonacin (https://orcid.org/0000-0003-3441-0887) holds a Ph.D. in Computer Science from UNICAMP, Brazil, and did postdoctoral at the Luxembourg Institute of Science and Technology. He is a Senior Researcher and the Head of Computing Methodologies Division at Renato Archer Information Technology Center, Brazil, and professor at UNIFACCAMP, Brazil. His research interests include Human-Computer Interaction, Semantic Web, Artificial Intelligence, Organizational Semiotics, Informatics in Education, and Medical Informatics.

Gustavo Pessin got his D.Sc. in Computer Science at the University of Sao Paulo as a member of the Mobile Robotics Lab. During his D.Sc. Pessin carried out research within the Robotics Lab, at the Heriot-Watt University, Edinburgh, UK and the Communication and Distributed Systems Group, at the Universität Bern, Switzerland. In 2015, Pessin had a Visiting Scholar position within the Media Lab at the Massachusetts Institute of Technology. Currently, Pessin is an Full Researcher within the Robotics Lab, at the Vale Institute of Technology. The bulk of his research is related to intelligent systems and mobile robots.

Vinícius P. Gonçalves (https://orcid.org/0000-0002-3771-2605) has a Ph.D. in Computer Science and Computational Mathematics (2016) from the University of São Paulo (USP). He was also a research fellow at the University of Arizona (USA) before joining the University of Brasília (UnB). Dr. Gonçalves was a Postdoctoral Researcher at the USP Medical School, with a CAPES Fellowship. Currently, he is an Assistant Professor in the Electrical Engineering Department (ENE) at UnB, Brasília, Brazil, where he is a member of the Graduate Programs in Electrical Engineering (PPGEE and PPEE). Dr. Gonçalves is a researcher and member of the AQUARELA Group; his main research interests include Human–Computer Interaction, Internet of Things, Cybersecurity, Mobile Health, Image Processing and Machine Learning.