2021 Joint European Conference on Networks and Communications & 6G Summit (EuCNC/6G Summit) | 978-1-6654-1526-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/EUCNC/6GSUMMIT51104.2021.9482526

Network Policies in Kubernetes: Performance Evaluation and Security Analysis

Gerald Budigiri*, Christoph Baumann†, Jan Tobias Mühlberg‡, Eddy Truyen§, Wouter Joosen¶
*‡§¶imec-DistriNet, KU Leuven, Belgium †Ericsson Research, Stockholm, Sweden
Email:{*gerald.budigiri, ‡jantobias.muehlberg, §eddy.truyen, ¶wouter.joosen}@kuleuven.be, †christoph.baumann@ericsson.com

***Abstract*—5G applications with ultra-high reliability and low latency requirements necessitate the adoption of edge computing solutions in mobile networks. Container orchestration frameworks like Kubernetes (K8s) have further emerged as the preferred standard to dynamically deploy edge applications on demand of end-users and third-party companies. Unfortunately, complex networking and security concerns have been highlighted as challenges that impede the successful adoption of container technology by the industry. The security challenge is exacerbated by (mis-)conceptions that secure inter-container communication comes at the cost of performance, yet both requirements are vital for 5G edge-computing use cases. Pursuing low-overhead security solutions, this paper investigates network policies, the K8s concept for controlling network isolation between tenants. We evaluate performance overheads of eBPF-based solutions by Calico and Cilium, and analyze the security of network policies, highlighting security threats to network policies and outline corresponding state-of-the-art solutions. Our assessment shows that network policies are a suitable low-overhead security solution for low-latency inter-container communication.**

***Index Terms*—container orchestration, edge computing, network isolation, microservices, tenants, eBPF, Calico, Cilium.**

I. Introduction

The advent of 5G opened the door to several new Ultra-Reliable Low-Latency Communication (URLLC) applications and use cases such as Virtual/Augmented Reality (VR/AR), Vehicle-to-Everything (V2X), and Remote Surgery (RS), poised to increase demand significantly for both computational and communication resources. Even with recent advances in hardware capability, the stringent performance requirements for these applications are still not matched by the actual device and network capabilities [1]. To address such mismatches, several technological concepts like edge computing and Network Function Virtualization (NFV) have been adopted by 5G providers. With NFV, the edge computing platform virtualizes network function modules and extends cloud computing capabilities to edge devices in close vicinity of end users, providing flexibility and agility in multi-tenant edge ecosystems.

This development is met by the cloud-native computing trend, where applications are composed of microservices that use stateless APIs to interact with each other. Using lightweight and portable containers on top of flexible orchestration frameworks like Kubernetes (K8s) instead of more resource-intensive and slow-to-start VM-based solutions, makes Containerized Network Functions (CNFs) a seemingly better choice for 5G edge computing URLLC applications, more so now that the microservice architecture is being investigated as a possible solution for NFV in the edge [2].

However, complex networking and security concerns have been highlighted as challenges for container adoption in industry [3]–[5]. Potential security vulnerabilities have even higher severity in multi-tenant edge computing cloud environments where microservices belonging to different parties need to be isolated so that they can only interact when necessary. In addition, for mission-critical 5G applications like RS and V2X, neither communication performance nor security should be compromised. This makes it imperative to find low overhead security solutions for inter-container communication. While there has been a multitude of efforts [3], [5]–[11] to improve container security by both industry and research communities, most of these works focus on container image, runtime, kernel OS, and K8s configuration, with little or no consideration for network security in low latency applications. K8s provides *Network Policies* that allow to restrict on inter-container communication. Although lacking advanced features of modern firewalls like intrusion detection, different network policy recipes [12] still provide a reasonable level of network security. In a multi-tenant environment, they provide configurable network isolation between tenants by restricting traffic to only those microservices allowed to communicate with each other. While being defined via the K8s Network Policy API, enforcement of policies is handled by customizable Container Network Interface (CNI) plugins. Previous work has compared different CNI plugins [13] and discussed their use for multi-tenancy [14]. In contrast, this paper is the first to investigate both performance overheads and security implications of K8s network policies specifically, and to explore the suitability of network policies to protect URLLC applications at the edge. We make the following contributions:

- *Performance:* We evaluate suitable CNI plugins and show that network policies incur a negligible performance overhead which only varies slightly with the number of policies and for different policy recipes.
- *Security:* We define an attacker model for network policies and analyze corresponding threats and vulnerabilities. Furthermore we propose suitable state-of-the-art, low-latency solutions addressing these threats.

Before presenting these results, we outline the technical foundations, features, and challenges of K8s network policies.

II. Background and Motivation

This section introduces K8s, network policies in K8s, and why they are important in multi-tenant edge platforms.

Furthermore, we compare CNI plugins with network policy support and explain policy enforcement in Calico.

A. Kubernetes (K8s)

Kubernetes is the de-facto standard for container orchestration. It supports deploying, scaling, and managing containerized microservice applications, and provides networking concepts and support for inter-connecting microservices at a large scale. A single microservice application runs in a container, and one or more tightly coupled containers run in a *pod*, the smallest unit of deployment in K8s, while pods run on a *node* (host), controlled via an *API server* running on the K8s master node. Since pods are ephemeral, dynamically launched or killed during scaling, service objects are used to offer a stable endpoint for pods. Such and other configuration objects are stored in the distributed *etcd* database along with state information controlling the K8s orchestration process.

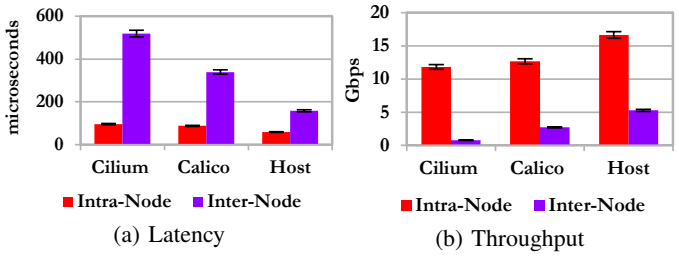
B. Kubernetes Network Policies

In multi-tenant edge platforms, it is important to protect edge user privacy. Even though K8s offers cluster-wide namespaces [15] to provide isolation between tenants for administration and resource quota management purposes, this support is not sufficient to prevent network traversal. Yet lack of sufficient network segmentation is an often cited high-risk vulnerability that has been exploited in large cybercrimes such as the Equifax data breach [16].

Network policies help to provide the guardrails needed to restrict traffic between pods (in and/or across namepaces) as well as between pods and external networks, by explicitly specifying allowed and denied connections. A network policy specification consists of a `podSelector` to specify pods that will be subject to the policy and `policyTypes` to specify the types of policies, i.e., ingress and/or egress. Ingress rules specify allowed inbound traffic to the target pods, and egress rules specify allowed outbound traffic from the target pods. Each rule is comprised of a `NetworkPolicyPeer` for selecting pods on the other side of the connection to/from which traffic is allowed, through a Classless Inter-Domain Routing (CIDR) notation that specifies IP address blocks, namespaces, or pod labels; and a `NetworkPolicyPort` that allows to explicitly specify ports or protocols that may communicate with the pod. Network policies are additive, and if multiple policies select a pod, traffic is restricted to what is allowed by the union of those policies' ingress/egress rules.

C. CNI Plugins and Networking Modes

With no built-in capability to enforce network policies, K8s relies on CNI plugins for enforcement. A CNI plugin is installed as an add-on that runs either as a pod or relies on open-source K8s components. While many CNI plugins exist, only Calico and Cilium support enforcement of network policies with eBPF, a high-performance alternative to iptables. Indeed, when comparing eBPF with iptables, we observed a latency alleviation of 0.7X and 0.8X, and a throughput improvement of 3.5X and 1.2X for inter- and intra-node scenarios



(a) Latency (b) Throughput

Fig. 1: eBPF: Calico vs Cilium

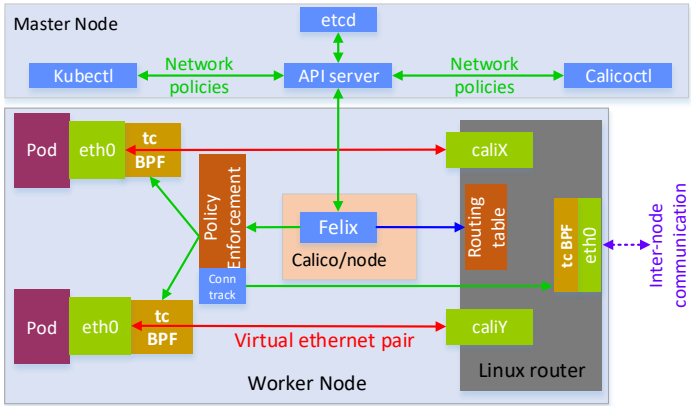


Fig. 2: Calico eBPF policy implementation

respectively. The measurements were performed using Calico on an OpenStack testbed in a closed lab (cf. Section III-A and III-B for the experimental setup and specification of the testbed). Both Calico and Cilium use traffic control (`tc`) hooks [17], [18] for attaching BPF programs at both ingress and egress of a pod's virtual ethernet (`veth`) interface.

Calico eBPF and Cilium eBPF performances were then evaluated against host mode where the benchmark is run on OpenStack VM instances. The results (see Fig.1) indicate that despite the benefits of eBPF data plane, the overheads of CNI plugins (and possibly the entire K8s architecture) remain significant, especially for inter-node communication. The results further show that Calico eBPF outperforms Cilium eBPF notably in the inter-node scenario. This is because Cilium by default runs in tunneling mode which incurs encapsulation headers while Calico uses the Linux kernel routing support on each node to provide a pure Layer 3 network solution, allowing for lower overheads. Thus, Calico eBPF was used for the rest of the performance evaluations in this paper. Calico and host intra-node measurements were repeated on bare-metal, for which host results were better by 8.7% and 10.7% for latency and throughput, respectively, as compared to 33% and 31% for OpenStack. The higher overhead on OpenStack indicates a likely negative interference between the OpenStack-based network configuration of the VM and the CNI-based network configuration for the pods.

D. Calico eBPF Policy Implementation

The Calico CNI plugin deploys to every node a Felix daemon (see Fig.2) that programs the kernel routes to local pods and distributes the routing information using Border Gateway

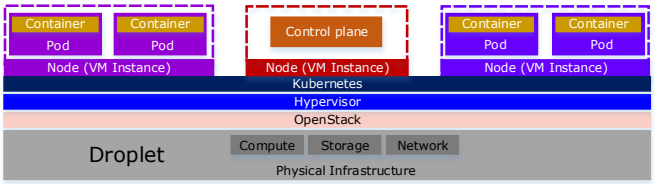


Fig. 3: Experimental setup

Protocol (BGP). Felix gets the network policy definitions from the K8s etcd via the API server and translates them to BPF programs that are loaded into the kernel and attached to the `tc` hooks of each Pods' `veth` interface. Only rules from network policies that match the pod's labels will be attached, leading to a scalable implementation, which is reinforced by Linux's `conntrack`. Calico also implements its own network policy model in the form of a Custom Resource Definition (CRD) which offers an extended range of actions such as prioritization of the rules based on their order. Felix further uses express Data Path (xDP) to implement packet filtering at the node to protect against Denial of Service (DoS) attacks.

E. Network Policies in Multi-Tenant Platforms

Microservice-based applications can contain hundreds of microservices, with companies like NetFlix using more than 500 microservices [19] for running its movie streaming system. In multi-tenant edge platforms where different applications may be from different providers with a need for strong network isolation, network policies must be explicitly specified for each intended inter-tenant interaction between microservices, following the least-privilege principle, which aims to reduce the security attack interface for attackers [4]. Similarly, explicit allow-policies are required for all intra-tenant container communications. Hence, the number of policies in multi-tenant platforms is likely to increase dramatically with the number and size of tenant applications. As a consequence, for 5G edge deployments, it becomes imperative to not only benchmark networking performance of K8s, but also to evaluate the performance overhead of network policies.

III. Overhead Performance Evaluation

This section summarizes evaluation results for the network latency overheads of Calico network policies for the intra- and inter-node scenarios under various conditions.

A. Benchmark and Evaluation Architecture

We used TCP stream mode and request-response (RR) mode of netperf [20] for throughput and end-to-end latency measurements, respectively. We configured netperf for a test length of 120 seconds with the goal of 99% confidence that the measured mean values are within +/- 2.5% of the real mean values. We do not report throughput and CPU Utilization results, as they follow the same pattern as latency.

B. Experimental Setup

The testbed for running all the experiments is an isolated part of a private OpenStack cloud, version Liberty, as illustrated in Fig.3. We used OpenStack since in public edge clouds

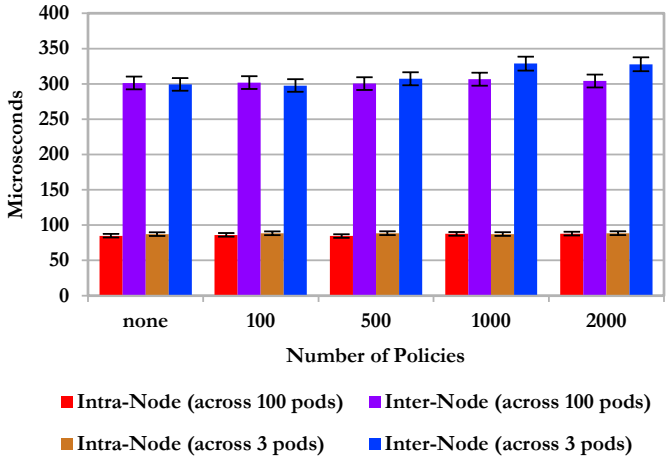


Fig. 4: Latency overhead for increasing number of policies

it is often preferred to run containers in a VM to protect the assets of the cloud provider. The OpenStack cloud consists of a master-slave architecture with two controller machines, and droplets on which VMs can be scheduled. The droplets have Intel(R) Xeon(R) CPUE5-2650 2.00GHz processors and 64GB DIMM DDR3 memory with Ubuntu xenial. Each droplet has two 10Gbit network interfaces. The droplets have 16 CPU cores of which 2 are reserved for the operation of the Openstack cloud. A K8s cluster, consisting of one master node and two worker nodes, was deployed using Kubeadm, running K8s version 1.19.2. All nodes have 4 vCPUs and 8GB RAM, and all were deployed on the same physical droplet to eliminate variations in network delay. Moreover, the vCPU cores of each node are exclusively pinned to physical cores that belong to the same motherboard socket of the droplet.

1) Effect of Increasing the Number of Policies: We created 20 namespaces each with 5 microservices (pods). Three pods were used for performance measurements: one local and two remote pods (one for each scenario, i.e., intra-node and inter-node), and all were assigned the same number of policies. Calico provides a policyOrder feature for prioritizing policy enforcement, with lower orders taking precedence. The lowest order was given to a policy that does not allow any inter-pod traffic followed by policies allowing communication with unrelated pods until the required number of policies. The highest order was then assigned to the policies that allow communication between the three selected pods. The number of network policies was then varied from zero to 2000 and the results are shown in Fig. 4 (across 100 pods). For none of the evaluated numbers of network policies we observed a significant effect on performance.

2) Network Policy Recipes: Different policy recipes [12] covering a wide range of K8s network policy features are outlined in Table I. While individual evaluations were done for each of the recipes numbered 5, 11, 13, and 14 in the table, only one test was done for recipes with similar structure, i.e., 2, 8, 9, and 12. For the rest of the recipes, i.e., 1, 3, 4, 6, 7, and 10, it is not plausible having many policies since the desired traffic isolation functionality from these policies can

TABLE I: Network policy recipes, NS=namespace

#	Deny traffic	#	Allow traffic
1	to a pod	8	to a pod
2	limiting traffic to a pod	9	to a pod from all NSs
3	to a NS	10	from a NS
4	from other NSs	11	from pods in another NS
5	from a pod	12	from external clients
6	non-whitelisted from a NS	13	only to a port
7	external egress	14	using multiple selectors

be achieved by just one policy in the namespace or even in the whole cluster for the case of `GlobalNetworkPolicy`. Evaluations provided similar results as observed in Fig. 4 (across 100 pods), indicating no significant overheads.

C. Scalability of Network Policies

From all the evaluations above, no significant overhead was observed on enforcement of network policies. We can attribute this to the decentralized Calico implementation where only relevant policies are evaluated at the `veth` interface of each pod. Even though there may be 2000 policies in the cluster, only those policies whose selectors match communicating pods are evaluated at these pods. In addition, the order feature of Calico is enforced at policy creation or modification time rather than packet time. To evaluate the scalability of network policies, we deployed three pods in the cluster and labeled them such that they match all policies in the cluster. The results as observed in Fig.4 (across 3 pods) still show no significant policy overheads, at least for intra-node traffic. This indicates that Calico's use of eBPF for implementation of network policies is scalable. A reason for that may be the fact that Calico only checks its policies for the first packet in a new flow; afterwards `conntrack` automatically allows subsequent packets in the same flow, without rechecking every packet.

IV. Network Policy Security Analysis

Network policies are constraining in-bound and out-bound network connections between containers across different pods, denying any connection that is not explicitly allowed. Below we discuss the security implications and challenges of network policies for inter-container communication as perceived by edge platform providers that host microservices of different edge applications in the same K8s cluster.

A. Attacker Model

We consider an attacker has compromised a microservice in one of the edge application namespaces. This could have been achieved by a malicious or compromised edge application provider, by compromising the software supply chain that deploys services on the platform, or by exploiting vulnerabilities in an externally accessible microservice.

Assuming that the compromised application runs in an unprivileged (non-root) container, the attacker is limited by the capabilities and resource restrictions applicable to that container and its corresponding pod. Still, the attacker may execute arbitrary code and access all permitted memory regions, files, and network connections of the container. Thus, the attacker may connect to other reachable microservices in the cluster and exploit any present vulnerabilities to compromise them as well. On the other hand, we require that the cluster virtualization infrastructure is trustworthy and sufficiently hardened such that the attacker cannot escape its container sandbox and gain root privileges on its hosting node.

The goal of the attacker is to spread in the cluster by compromising other services, especially those that are privileged to access sensitive information or mission-critical services to disrupt their functionality. A secondary goal is to stay undetected. Then, knowing any pod's network policies and thus the corresponding reachable sub-network may help an attacker to avoid triggering alarms. In this sense the network policies applicable in a cluster can be seen as sensitive information.

B. Security Implications of Network Policies

Network policies are an effective tool to hinder the attacker described above, if policies are defined according to the least privilege principle, allowing only necessary connections. The resulting network segmentation reduces the attack surface, protecting potentially vulnerable co-hosted applications and preventing arbitrary lateral attacker movement.

Nevertheless, network policies are not a silver bullet. Attackers can still abuse permitted connections and target the corresponding microservices in order to achieve remote code execution or leak sensitive information. While policies allow blocking connections to certain ports, the simple rules from Table I do not offer any fine-grained control over the traffic between admissible communication partners. Other solutions, like network monitoring [21] and anomaly detection [22], can be employed to detect and stop such attacks. Once suspicious actors are identified, network policies can be used to isolate them in a "quarantined" part of the cluster. However, newly enacted network policies are only enforced for new connections and do not cut existing connections automatically.

C. Network Policy Security Vulnerabilities and Threats

The use of network policies itself has several pitfalls. Below we report common vulnerabilities and potential attacks as well as available remedies from the literature.

1) Misconfiguration: A StackRox survey [23] revealed that 91% of K8s users had experienced security issues of which 67% was attributed to misconfigurations. Moreover, misconfiguration is still considered the top cause of data breaches in the cloud [24]. When configuring network policies, there are plenty of opportunities for weak or faulty configurations that indubitably threaten container network isolation. These include wrong pod label specifications, manual errors like typos, or even completely forgetting to enforce network policies after writing them. K8s does not warn if network policies are disabled but just accepts and silently ignores the configuration.

State-of-the-art solutions: The Open Policy Agent (OPA) helps to alleviate misconfiguration risks such as typos or oversights in network policies, forgetting to enforce them, or even deleting them by mistake, by enforcing suitable OPA requirements on network policies through custom admission controllers during pod creation [25]. However, OPA does

not verify that the configured policy requirements establish a higher level security goal, such as network isolation.

2) Weak Policy Design: All security policies, including but not limited to network policies, need to be designed and configured according to principles such as least privilege [26] and/or zero trust [27]. Communication between containers should be explicitly allowed and kept to the bare minimum required for the applications to do their job. For example, an application that needs only to read from the database should not inadvertently be given write permission. When specifying network policies, an application container should be given sets of privileges each minimized to the smallest set of permissions needed for the service. Allow policies should allow the bare minimum by for example making use of `NetworkPolicyPort` while deny policies should block all unauthorized traffic to the maximum. However, writing a least-privilege policy is a daunting task since a typical application in K8s requires hundreds of connections, between services, from the outside (ingress) and to external end-points (egress).

State-of-the-art solutions: When enforcing network policy rules, OPA can limit the number of allowed ingress rules to the protected container to a specific value to avoid granting additional containers access [25]. BASTION [4] enforces least privileged network access for containers by ensuring that their connectivity is limited to inter-dependencies between itself and those containers required to compose a service.

3) Tenant Administrators: While the edge infrastructure is usually trusted, tenants and their administrators may pose certain risks. A rogue admin with sufficient privileges may collude with the attacker to weaken network policies and isolation, stealthily allowing illegitimate network connections. In particular, regular K8s network policies are bound to tenant namespaces and may be changed by tenant admins [14].

State-of-the-art solutions: In a multi-tenant environment, best practices of authentication, access control, and auditing in the control plane must be followed to limit the impact of malicious tenants and insider attacks [11]. The problem of tenant admins changing cluster network policies is solved in Calico via the concept of global policies and prioritization [14]. Just as for misconfiguration, OPA also offers protection against launching containers with illegitimate network policies [25].

4) Privileged Networking: It is unfortunately common that containers are run with insecure settings, granting unnecessary or inadvertent privileges, posing risks to the node OS and other containers on the node. A network-privileged container is directly exposed on the LAN network, bypassing pod network interfaces and network policies. Such a container can access all other nodes through the gateway IP address assigned to them [4] or by guessing the corresponding IP addresses, compromising the desired network isolation.

State-of-the-art solutions: A high-performance security enforcement network stack is presented in [4], which addresses this vulnerability by enforcing fine-grained access control over network-privilege-enabled containers that share the host network namespace. In addition, although Calico does not provide security mechanisms to guard against host-network namespace abuse, it prevents privileged containers from abusing the virtual gateway IP address. Pod Security Policies ensure that no container is spawned with inadvertent capabilities or access privileges [28]. They can also be enforced by OPA [29].

5) Vulnerable Implementations: In K8s, the enforcement of network policies depends on the CNI plugin components. These components and their dependencies may contain vulnerabilities themselves, that may compromise policy enforcement or allow attackers to intercept and redirect network traffic, e.g., CVE-2019-9946, CVE-2020-10749, and CVE-2020-13597.

State-of-the-art solutions: Keeping software up-to-date mitigates the impact of vulnerabilities, but does not solve the root problem. Formal verification techniques and safe programming languages may contribute to improving software security.

6) eBPF Exploits: In our setup, the calico/node agent translates network policies to eBPF programs for enforcement. Hence eBPF kernel capabilities need to be enabled on the node in order to use network policies, potentially increasing the node's attack surface. Prior to Linux 4.4 all `bpf()` commands required the caller to have the `CAP_SYS_ADMIN` capability, currently however, unprivileged users may create and run limited eBPF programs by attaching them to a socket they own [30]. As exemplified in CVE-2020-8835, this feature has not only been exploited to perform out-of-bounds reads and writes in kernel memory but also to achieve privilege escalation [31].

State-of-the-art solutions: Such threats can be mitigated by disabling unprivileged access to the `bpf()` syscall entirely by setting the `kernel.unprivileged_bpf_disabled` sysctl to `1`. The eBPF in-kernel verifier can restrict which kernel functions and data structures may be accessed from eBPF programs [32]. Seccomp-BPF restricts the set of system calls and arguments available to user-space programs [33].

7) Leaking Network Policies: As discussed above, the network policies themselves may be valuable information for an attacker keen on avoiding detection. The easiest way to obtain them is by requesting them from K8s API server or etcd, but these APIs are not normally accessible to a user plane attacker. An adversary with the capability to monitor control plane traffic may obtain configuration information such as network policies if sent in the clear. Additional memory read capabilities via exploits such as [31] would also be needed by an attacker to leak the policies from the local calico/node agent or the eBPF kernel facility. An unprivileged attacker may probe the network to infer allowed connections, but this may defeat the initial goal of staying stealthy.

State-of-the-art solutions: Primitive attempts to leak policy information as described above can easily be hindered by best practices, such as enabling mutual Transport Layer Security (TLS) encryption, authentication, and RBAC for the control plane [11]. Although Calico supports TLS to secure Calico components and control plane communication, this security is not configured by default in most of manifests provided to make Calico deployments easy [34]. For Calico, TLS is recommended for communication with its datastore along with the use of the client certificates and JSON web token (JWT) authentication modules [35]. For both confidentiality

and integrity reasons it should be considered to run mission-critical applications and control plane components in Trusted Execution Environments (TEE) such as Intel SGX to create isolated enclave environments in the nodes between each container application [36]. Intel SGX enclaves isolate applications residing in the container by means of encrypted memory, shielding application memory from malicious and/or privileged containers, as well as from compromised node operating systems. Although, SGX incurs a performance overhead when executing enclave code [10], [37], applying it to control plane components to protect confidentiality and integrity of network policies would not add any overhead to user plane traffic.

V. Conclusion

Edge computing and containerization are unequivocally important technologies in multi-tenant 5G cloud environments. However, security requirements still impede their widespread adoption especially in 5G URLLC edge applications with high performance and security requirements, since most security solutions compromise performance. This makes finding a low overhead container security solution vital. Having evaluated network policies, a configurable network isolation security solution for K8s and observed no significant performance overheads, we demonstrate that they are a suitable security solution for such applications. As a general concern, the interacting microservices should be deployed on the same node in order to ensure highest performance, which however increases the need for proper isolation of different tenants.

We call attention to security challenges that may be exploited to subvert the isolation of containers by compromising, or even bypassing network policies, and correspondingly highlight current possible solutions. For future work, we point out the need to investigate the negative interference between Openstack network configuration and CNI-based network configuration to reduce performance overhead of CNI plugins as observed in Fig.1. Other possible research directions include a formal treatment of network policies to verify network policies against desirable security properties, hardening network policies against highlighted attacks with low overhead solutions, as well as the application of trusted execution in policy enforcement and the control plane in general.

Acknowledgment

This research is partially funded by the Research Fund KU Leuven, and by the Flemish Research Programme Cybersecurity. This research has received funding under EU H2020 MSCA-ITN action 5GhOSTS, grant agreement no. 814035.

References

[1] Q.-V. Pham *et al.*, "A survey of multi-access edge computing in 5G and beyond: Fundamentals, technology integration, and state-of-the-art," *IEEE Access*, 2020.

[2] H. Hawilo *et al.*, "Exploring microservices as the architecture of choice for network function virtualization platforms," *IEEE Network*, 2019.

[3] J. Watada, A. Roy, R. Kadikar, H. Pham, and B. Xu, "Emerging trends, techniques and open issues of containerization," *IEEE Access*, 2019.

[4] J. Nam, S. Lee, H. Seo, P. Porras, V. Yegneswaran, and S. Shin, "BASTION: A security enforcement network stack for container networks," in *2020 USENIX Annual Technical Conf. (USENIXATC 20)*, 2020.

[5] S. Sultan, I. Ahmad, and T. Dimitriou, "Container security: Issues, challenges, and the road ahead," *IEEE Access*, 2019.

[6] M. Souppaya, J. Morello, and K. Scarfone, "Application container security guide (2nd draft)," NIST, Tech. Rep., 2017.

[7] E. Reshetova, J. Karhunen, T. Nyman, and N. Asokan, "Security of os-level virtualization technologies: Tech. report," *arXiv:1407.4245*, 2014.

[8] A. Grattafiori, "Understanding and hardening linux containers," *Whitepaper, NCC Group*, 2016.

[9] S. Vaucher, R. Pires, P. Felber, M. Pasin, V. Schiavoni, and C. Fetzer, "SGX-aware container orchestration for heterogeneous clusters," in *2018 IEEE 38th Conf. on Dist. Computing Syst. (ICDCS)*. IEEE, 2018.

[10] S. Arnautov *et al.*, "SCONE: Secure linux containers with intel SGX," in *12th USENIX OSDI 16*, 2016.

[11] M. S. I. Shamim, F. A. Bhuiyan, and A. Rahman, "XI commandments of Kubernetes security: A systematization of knowledge related to Kubernetes security practices," in *2020 IEEE SecDev*. IEEE, 2020.

[12] A. Balkan, "Kubernetes network policy recipes," https://github.com/ahmetb/kubernetes-network-policy-recipes/, [accessed 2021-01-28].

[13] S. Qi *et al.*, "Understanding container network interface plugins: design considerations and performance," in *2020 IEEE Int. Symp. on Local and Metropolitan Area Networks (LANMAN)*. IEEE, 2020.

[14] X. Nguyen, "Network isolation for K8s hard multi-tenancy," 2020. [Online]. Available: https://aaltodoc.aalto.fi/handle/123456789/46078

[15] "Kubernetes namespaces," https://kubernetes.io/blog/2016/08/kubernetes-namespaces-use-cases-insights/, [accessed 2020-11-24].

[16] "How to stop the next equifax-style megabreach," https://www.wired.com/story/how-to-stop-breaches-equifax/, 2017, [accessed 2021-01-14].

[17] "About eBPF," https://docs.projectcalico.org/about/about-ebpf/, [accessed 2020-12-18].

[18] "eBPF datapath," https://docs.cilium.io/en/v1.9/concepts/ebpf/, 2020, [accessed 2020-12-18].

[19] CED, "How it's built," https://medium.com/the-andela-way/how-its-built-netflix-8f62ab329011/, 2019, [accessed 2020-12-18].

[20] "Care and Feeding of Netperf 2.7.X," https://hewlettpackard.github.io/netperf/doc/netperf.html/, [accessed 2020-10-23].

[21] https://github.com/falcosecurity/falco/, 2020, [accessed 2021-01-27].

[22] C.-W. Tien *et al.*, "KubAnomaly: Anomaly detection for Docker orchestration platform with neural network approaches," *Eng. Reports*, 2019.

[23] "Kubernetes adoption, security, and market share trends report," https://www.stackrox.com/kubernetes-adoption-security-and-market-share-for-containers/, 2020, [accessed 2020-12-18].

[24] F. Truta, "Misconfig. remains #1 cause of data breaches in the cloud," https://securityboulevard.com/2020/04/misconfiguration-remains-the-1-cause-of-data-breaches-in-the-cloud/, [accessed 2021-01-14].

[25] M. Ahmed, "How to enforce Kubernetes network security policies using OPA," https://www.magalix.com/blog/how-to-enforce-kubernetes-network-security-policies-using-opa/, 2020, [accessed 2020-11-02].

[26] D. Mónica, "Least priv. cont. orchestration," https://www.docker.com/blog/least-privilege-container-orchestration/, [accessed 2020-12-17].

[27] "Zero-trust networks in Kubernetes, cloud-native applications," https://www.stackrox.com/wiki/zero-trust-networks-in-kubernetes-cloud-native-applications/, 2020, [accessed 2020-12-17].

[28] "Pod security policies," https://kubernetes.io/docs/concepts/policy/pod-security-policy/#host-namespaces/, 2020, [accessed 2021-01-27].

[29] M. Ahmed, "Enforce pod security policies in Kubernetes using OPA," https://www.magalix.com/blog/enforce-pod-security-policies-in-kubernetes-using-opa/, 2020, [accessed 2021-01-27].

[30] "bpf(2) - linux manual page," https://man7.org/linux/man-pages/man2/bpf.2.html/, 2020, [accessed 2020-12-21].

[31] P. Manfred, "Kernel privilege escalation via improper eBPF program verification," https://www.thezdi.com/, 2020, [accessed 2020-12-17].

[32] M. Fleming, "A thorough introduction to ebpf," https://lwn.net/Articles/740157/, 2017, [accessed 2020-12-21].

[33] "Using eBPF in Kubernetes," https://kubernetes.io/blog/2017/12/using-ebpf-in-kubernetes/, 2017, [accessed 2021-01-28].

[34] "Customize the manifests," https://docs.projectcalico.org/getting-started/kubernetes/installation/config-options/, 2020, [accessed 2020-12-01].

[35] "Configure encryption and authentication," https://docs.projectcalico.org/security/comms/crypto-auth/, 2020, [accessed 2020-12-01].

[36] A. Gowda and D. Coulter, "Microsoft Docs:Enclave aware containers on Azure ," https://docs.microsoft.com/en-us/azure/confidential-computing/enclave-aware-containers/, 2020, [accessed 2020-10-30].

[37] P.-L. Aublin *et al.*, "TaLoS: Secure and transparent TLS termination inside SGX enclaves," *Imperial College London, Tech. Rep*, vol. 5, 2017.