Zero-cost In-depth Enforcement of Network Policies for Low-latency Cloud-native Systems

1st Gerald Budigiri
KU Leuven
imec-DistriNet
Leuven, Belgium
gerald.budigiri@kuleuven.be

2nd Christoph Baumann
Ericsson Security Research
Stockholm, Sweden
christoph.baumann@ericsson.com

3rd Eddy Truyen
KU Leuven
imec-DistriNet
Leuven, Belgium
eddy.truyen@kuleuven.be

4th Jan Tobias Mühlberg
KU Leuven
Université libre de Bruxelles
Brussels, Belgium
jan.tobias.muehlberg@ulb.be

5th Wouter Joosen
KU Leuven
imec-DistriNet
Leuven, Belgium
wouter.joosen@kuleuven.be

***Abstract*—Packaging applications in containers and managing them dynamically using a cluster orchestrator is the de-facto approach for deployment of cloud-native applications. When containers run inside virtual machines (VMs) to protect infrastructural assets, network policies (NPs) at the container layer and security groups (SGs) at the VM layer provide complementary firewall mechanisms that strengthen defenses against lateral movement of attackers. However, least-privilege network policies at the container layer may not always be consistent with statically defined, over-permissive SGs at the VM layer. This is especially a problem with low-latency configuration of container networking solutions that requires every opened container protocol, port and traffic direction also to be opened at the VM layer. In any post-exploitation scenario where attackers escape from within an already compromised or infected container, such over-permissive SGs do not prevent the attacker from spreading across VMs to find powerful tokens for accessing the cluster orchestrator. In this paper, we introduce GrassHopper (GH), a fast and dynamic cross-layer enforcement approach for NPs, which automatically generates SG configurations from dynamically verified NPs. Given the low-latency context, the design of GH must ensure that dynamically generated SG rules are applied fast before the newly scheduled containers become ready to serve traffic. We evaluate GH on a Kubernetes cluster running on OpenStack. For a wide range of relevant applications and cluster setups, GH can reduce the network attack surface between VMs at a ratio of 70-to-99% while causing no performance overhead at the application-level with respect to latency, throughput, and cpu utilization.**

***Index Terms*—container orchestration, kubernetes, network isolation, network policies, security groups**

I. INTRODUCTION

Containerization and edge computing are vital to support ultra-low-latency and ultra-high-reliability applications such as vehicle-to-everything (V2X) or remote surgery [1]. While edge computing effectuates lower latencies and increased connectivity for applications, containers allow for compartmentalization as well as scalable and fast deployment of microservices. Unlike virtual machines (VMs), containers are lightweight and highly portable. However, to provide defense-in-depth for such applications, containers typically run inside VMs to protect not only the application workloads but also the infrastructural assets of edge and cloud providers.

Container-based network policies (NPs) can be used to restrict communication between containers according to the principle of least-privilege. At the VM level, connectiviy is usually configured using security groups (SGs) that are attached to VM nodes and govern communication with other groups. It is desirable to also configure SGs according to the least-privilege principle in order to prevent unnecessary network attack surface between VMs. Indeed, without such measures, in the event of a container escape [2], [3], an attacker may move laterally across worker nodes and, e.g., obtain API tokens stored in these VMs that give the attacker access to the API of the control plane of the cluster [4].

The common way to configure the VM network for a container orchestrator like Kubernetes (K8s) is to only open those ports and protocols that are required for having an operational container network and cluster control plane, which reduces the attack surface sufficiently. However, opening only such discrete ports is not a feasible solution for low-latency applications which require container networking solutions to be configured without use of network packet encapsulation [5]. In such low-latency configurations of network plugins, which works within a single L3 subnet, the original IP packets of the container are directly sent via the Layer 2 protocol of the VM network, and therefore, a static SG would need to allow all ports and transport protocols that could be used by containers on all the VMs all the time. This leads to an unnecessarily large network attack surface for malicious actors. In fact, a truly least-privilege solution would only open connections between VM nodes if they host containers that require these connections according to their network policies.

In this paper, we introduce GrassHopper (GH), a solution for cross-layer enforcement of NPs in response to dynamic container scheduling. When a new container, say p, is being scheduled on a VM, say N, GH verifies the NPs applicable to p on-the fly to ensure these NPs can be used as the base truth. From these verified NPs, GH automatically generates a consistent and least-privilege set of SG rules for VM N so that N may *only* communicate with VMs that run containers for which communication with p is allowed, and *only* using the specified traffic direction, ports, and protocols.

This mechanism strengthens isolation between containers, micro-services, and applications that are hosted on the same cluster but do not need to communicate with each other, e.g., due to program logic or because they belong to different tenants. Traditionally, such an isolation could be achieved by assigning different namespaces to statically isolated groups of nodes using taints, tolerations, and affinity rules. Our solution, in contrast, automatically and dynamically enforces VM network isolation as defined by such policies on the container orchestration level. As such, it enables better resource utilization and relieves the cloud administrator from the error-prone, manual process of defining static SG rules.

The dynamic SG management by GH involves time-consuming SG operations, e.g., the creation of a SG, addition of a firewall rule to a SG, or attachment of a SG to a node. All these operations must be completed before a new container is fully started, because traffic to and from it will otherwise be blocked. The latency of the overall auto-generation process is thus important, especially when applications are continuously redeployed or when containers are dynamically autoscaled across VMs [6], [7], [8]. We designed GH in a way that minimizes the number of time-consuming SG operations and ensures that such timing constraints can be met for current container orchestrators. On the other hand this means that GH does not introduce any performance overhead or additional latency.

To summarize, this paper makes the following contributions:

- We motivate the importance of in-depth defenses against post-exploitation scenarios with concrete evidence of container-level vulnerabilities and concrete post-exploitation attack scenarios that allow taking over the entire VM network and cluster.
- We present an efficient algorithm for generating least-privilege SG rules from dynamically verified NPs and container scheduling. The design of this algorithm is informed by a study of what are the most time-efficient SG operations of modern `ipset` based firewalls.
- We demonstrate the applicability of GH by implementing it on top of K8s, a popular container orchestration framework, and by evaluating it on two container networking solutions for K8s, three different applications, and the Openstack cloud platform, which has defined the de-facto standardized SG API for all major IaaS cloud providers.

The next section gives background on K8s, NPs, and SGs. In Sect. III we further motivate the need for GH, whose design is explained in Sect. IV. We describe our prototype implementation for K8s and OpenStack in Sect. V and evaluate it experimentally in Sect. VI. Related work is discussed in Sect. VII before we conclude in Sect. VIII.

II. BACKGROUND

In this section we briefly introduce Kubernetes NPs and cloud SGs.

Kubernetes Networking and NPs: K8s manages containerized applications automatically and dynamically. In K8s, containerized applications run in *pods*, the smallest unit of execution that consists of one or more tightly coupled containers. Pods are hosted on physical or VM *nodes*, a group of which forms a K8s *cluster*. By default, all pods in the cluster are non-isolated, accepting all traffic. This is precarious from a security perspective, especially in mutually distrusting multi-tenant clusters. K8s thus provides configurable NPs to restrict communication among pods or tenants by controlling traffic flow at layers 3, 4, (or 7 if used with cilium or a service mesh).

A NP comprises mainly a *select* part specifying pods subject to the policy rules and an *allow* part specifying allowed traffic. Given the ephemeral nature of pod IPs, NPs use pod labels to select pods or namespaces in the cluster, and ipBlocks for external connectivity [9]. K8s needs a Container Networking Interface (CNI) network plugin for policy enforcement, and in this paper Calico and Cilium CNI plugins were used because they support extended Berkeley Packet Filter (eBPF) technology for fast policy enforcement, and without the use of high overhead network encapsulation techniques such as VxLAN or IP-in-IP [5]. Without network encapsulation, IP packets at the container layer are directly wrapped in Ethernet packets of the VM layer. For this to work, each VM of the cluster must belong to the same subnet and be configured to accept packets with a target IP address that is not one of their own. Based on the routing table of each VM, as configured by the CNI plugin, incoming IP packets are directly routed to the appropriate Pod.

Network policy inconsistency and misconfigurations: NPs are however not attacker-proof as any misconfiguration or inconsistency therein can be exploited by bad actors to gain illicit access to containerized applications, leading to data breaches, service interruptions, or cluster compromise. Various approaches already exist to prevent inadvertent exposure of the containerized applications due to errors in manual configuration of network policies. For example Kano[10] can be used to verify against misconfigurations such as policy conflict, redundancy, and violation of the least privilege principle. Bastion [11] is another approach that enforces minimal privileges from a graph of inter-dependent microservices.

Security groups: All mainstream cloud platforms and public cloud providers offer the notion of SGs to support configurable inter-VM isolation. Openstack defined the de-facto standardized SG API for all major IaaS cloud providers. A SG in OpenStack consists of a set of network access filter rules that allow traffic based on port, protocol, IP address or remote SG. The latter notion of remote SG abstracts over IP addresses and instead allows filter rules to refer to other VMs by means of a name. As stated above, by sending container-level transport protocol packets directly over the L2 layer of the VM network, every port and protocol in the transport header must be allowed by the default SG of all cluster nodes, for both egress and ingress traffic, to and from the entire CIDR range of the container network, or by recursively setting itself as remote SG.

III. MOTIVATION

The increased adoption of containerization is accompanied by increased attacks on containerized applications with many potential flaws and vulnerabilities stemming directly from images provided to users from repositories [12]. A study [13] found out that both official and community images contain more than 180 vulnerabilities on average, with more than 80% of the images having at least one high severity vulnerability. A more recent analysis [2] shows an increase in such vulnerabilities to 460 per image, all susceptible to exploitation by remote attackers to execute arbitrary code in the container or to store arbitrary files in the system. According to [3], 82% of certified images contain at least one high or critical vulnerability while it was also found that almost 51% of the Docker Hub images have exploitable critical vulnerabilities [14] and 10-15% of the Docker daemons that were exposed to the internet could be accessed without authentication [15]. A review of Docker CVEs from 2017 to 2021 [16] found privilege escalation and code execution as the most common vulnerability types, both commonly used in conjunction with or to cause a breakout into the host OS.

Judging from these results, it is clear that vulnerabilities in containers and their runtimes exist and will likely be exploited by external attackers to gain a foothold on cloud computing infrastructure. Hence it is all the more important that post-exploitation defenses are in place so that even after a container is infected or a privilege escalation occurs, infection cannot spread to compromise the entire cluster.

To motivate our defense-in-depth solution further we assume the following vulnerabilities in a given K8s cluster:

1) Default SG settings are used for VMs allowing *all* worker nodes to communicate with each other (potentially using only a static range of all required ports and protocols).
2) A pod can be accessed by the attacker via the pod network, i.e., they may schedule a malicious pod of their own or an existing pod contains a remotely exploitable software vulnerability that allows for a remote code execution attack [17], [18].
3) The cluster is susceptible to container breakouts, i.e., containerized applications may escalate privileges to access the host node. This may principally be achieved in three ways [19]: a) exploiting zero-day vulnerabilities or unpatched CVEs of the container runtime [20], [21], [22], [23]; b) exploiting permissive pod access control configurations towards the underlying host system [24]. Strict deny-all pod access controls towards the underlying operating system are not practical since many containers require specific capabilities, privileges, or system calls for their intended functionality [25], [26]; c) exploiting vulnerabilities in the host operating system kernel that are exposed inside of the container [27].

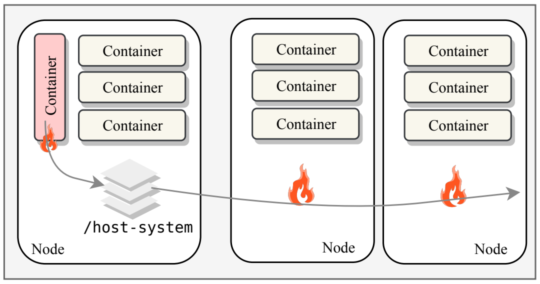


Fig. 1: Attack scenario 1: Assuming that the attacker has gained access to a pod vulnerable to container escape.

The above setting gives the attacker a foothold on the cluster. From a pod, the quickest route to compromise the cluster is to take control over the API server after a container escape, e.g., by exploiting a vulnerability of the server [28], or excessive privileges and credentials of *powerful* pods running on the same node [29]. However, such powerful pods may not always be present on a given node, requiring lateral movement to explore other worker nodes of the cluster as shown in Fig. 1. As the default SG configuration allows all nodes to communicate with each other, an adversary may spread to other nodes using a number of methods: 1) abuse suitable credentials found on the host, e.g., ssh keys; 2) spawn malicious processes on worker nodes to effectively open port ranges and protocols that are allowed by the default SG settings, either directly or via a a reverse shell attack [30]; 3) if local *kubelet* K8s agents are not properly protected via authentication and authorization the attacker may use it to send malicious API requests to other nodes, e.g., to leak credentials or scan for vulnerable pods. Even if authentication is enabled, kubelet authorization may still be misconfigured [31] and certificates found at the current host may be sufficient to authenticate against the remote kubelet's API; 4) attack bare-metal or containerized networked applications running on the other nodes, trying to achieve remote code execution. In all cases, after getting access to another node, more privilege escalations may be necessary to get full control but as we discussed above, suitable software vulnerabilities are likely to be present.

In a second scenario the attacker has gained access to a container but does not escape from it; instead from *within the container's scope*, the attacker scans for nodes whose kubelet agents have no proper API authentication and authorization in place. When such a vulnerable node is discovered, it is also possible to retrieve all its pods' tokens that enable authentication to the cluster API server [32]. This attack scenario cannot be viably prevented by means of a global NP that isolates all pods from the CIDR of the node network, except the master node. This is because we have found that such NP configuration creates errors with mandatory K8s features such as `Services` of type `NodePort` and `LoadBalancer`, and also with the reachability of any pod running in `hostNetwork` mode.

All of the scenarios above would benefit from a least-privilege, in-depth enforcement of container NPs, as such a mechanism may drastically reduce the network attack surface, blocking superfluous ports and protocols or even removing unnecessary access to neighboring nodes entirely. Besides, the automatic generation of least-privilege security groups from container NPs and pod placement simplifies cluster administration and avoids potential inconsistencies and misconfiguration.

IV. GRASSHOPPER DESIGN

This section presents the main idea and the methodology underlying GH, elaborating the process of verification of NPs and automatic generation and configuration of SGs from NPs.

A. Preliminaries

In the context of this work, pods scheduled on a cluster are represented by the set of labels that are associated with the pod. Labels should be interpreted as the typical key-value pairs used by K8s to tag and match pods. If the NPs matching the pod labels include matching IP blocks for external communication, protocols, or ports, these are as well taken into consideration in the generation of SGs. NPs are then signified by the set of labels they select and a set of traffic rules specifying allowed ingress or egress for a set of labels or IP blocks along with respect to a range of ports. For SGs generated by our approach from NPs, we need to store a name and a number of *remote SG rules* that allow ingress or egress to of from other SGs or IP blocks for a given port range.

A pod p_1 can then communicate with a pod p_2, if there exist policies pol_1, pol_2 in the cluster such that pol_1 allows egress from some labels of p_1 to some of the labels of p_2 and pol_2 allows ingress to p_2 from p_1 according to the pod labels. On the virtualization layer, a node N_1 may communicate with N_2, if there exist SGs sg_1^i, sg_1^e and sg_2^i, sg_2^e attached to the respective nodes, such that sg_2^i allows ingress from sg_1^e and sg_1^e allows egress to sg_2^i. Note that this means that communication between nodes need not be governed by a single pair of attached groups (sg_1, sg_2) with matching ingress and egress rules, but can be distributed across separate pairs (sg_1^i, sg_2^i) and (sg_1^e, sg_2^e) of attached groups. Above we described connectivity as prescribed by NPs using label sets. In our approach, ports, protocols, or IP Blocks from the NPs are added to the corresponding created SGs as necessary. With low-latency configuration of network plugins, ports and protocols are the same on the orchestration and virtualization layer (cf. Sect.I and II).

B. Goals

The main goal of GH is to ensure the consistency between communication rules established by SGs in OpenStack and the NPs of K8s. Moreover, the SG configuration should follow the least privilege principle that no unnecessary communication may be allowed. Formally, we aim for the following properties:

1) *Correctness*: If ingress or egress is allowed for a pod p from or to a pod q or IP block I, the same type of traffic must also be allowed between the node hosting p and the node hosting q, or addresses I, respectively.
2) *Least privilege*: Ingress or egress is only allowed between two worker nodes if they host pods between which a matching NP allows such traffic. SG rules only allow a node N to communicate with an IP block I if there is a pod p scheduled on N and a matching NP allows the same type of traffic between p and I.

TABLE I: Judging a new policy pol with given select and allow set if a policy with $\{a, b\}$ and $\{x, y\}$ already exists, meaning that pods with both labels a and b may already communicate with pods that have both labels x and y through at least some of the ports specified by pol. Below, c and z are new labels, different from a, b and x, y, respectively.

Select set	Allow set	Judgement
$\{a, b\}$	$\{x, y\}$	redundant
$\{a, b\}$	$\{x\}$	conflict
$\{a, b\}$	$\{x, y, z\}$	redundant
$\{a, b\}$	$\{z\}$	OK
$\{a\}$	$\{x, y\}$	conflict
$\{a\}$	$\{x\}$	conflict
$\{a\}$	$\{x, y, z\}$	OK
$\{a\}$	$\{z\}$	OK
$\{a, b, c\}$	$\{x, y\}$	redundant
$\{a, b, c\}$	$\{x\}$	OK
$\{a, b, c\}$	$\{x, y, z\}$	redundant
$\{a, b, c\}$	$\{z\}$	OK
$\{c\}$	any	OK

Note that traffic in the above definitions is restricted to the port ranges allowed by the respective network policies. Moreover, there is a caveat to the least-privilege property: cloud platform administrators may define additional remote SG rules that are not directly related to the K8s data plane, but needed for other workloads running on the nodes. We assume here that such additional SGs and their rules do not interfere with our goals.

Non-functional requirements for our approach include: 1) the configuration of SGs is automatic without need for manual intervention, and 2) the mechanism does not introduce significant additional latency to the system.

In the design we describe in the remainder of this section, the K8s NPs are considered the base truth for its operation. This is justified because in modern cloud-native computing and software-defined networking, distributed applications are defined at the container orchestration level by the developers and the underlying virtualization infrastructure should enable running the desired deployment in a zero-touch fashion. Additionally, GH first verifies new network policies for misconfigurations similar to the NP checker Kano [10].

C. Policy checks

Before handling a newly installed NP, GH checks it for misconfiguration against the set of existing policies. In particular, three different types of errors are considered:

1) *policy conflict*: a policy conflicts with an existing one when it weakens that one's traffic restrictions for the same type of traffic, i.e., it allows the same traffic for a subset of the select label set, or it allows access to a superset

TABLE II: SG operations time (ms) in the OpenStack control plane or the total time until the VM becomes reachable in the data plane. The latter is measured for attachment of SGs and adding of rules to attached SGs.

No.of SGs Operation	1	10	50	100
Creation of SG	44	61.5	337	649
Addition of Rules to SG (not attached)	7.4	62	381	698
Attachment of SG to a node	6.8	54.1	327	547
Addition of Rules to SG (attached)	7.2	55	335	709
Detachment of SG from a node	7.3	53.6	314	564
Removal of Rules from SG (not attached)	5.6	56	293	603
Removal of Rules from SG (attached)	5.6	44	275	610
Deletion from OpenStack	5.3	98.6	751	1548

of pods for the same select label set, or a combination of both (cf. Table I for an illustrating example).

2) *policy redundancy*: a new policy is considered redundant when pods selected and corresponding connections allowed by it are completely covered under the connections allowed by an existing one (cf. Table I).
3) *broad access permissions*: overly permissive network policies specify connections from or to a wide range of pod labels, IP addresses or unnecessary ports, potentially violating the least privilege principle. Here, we define permissions of a policy too broad, if it selects all possible label sets, expressed in K8s by an empty selector.

Our approach for generating SGs rejects such offending policies. However, conflicting and redundant policies pol may become non-offending and relevant at some later point, when the active policy that was offended by pol is removed. In principle we could then process such policies, however this might make it hard for an administrator to analyze the system and debug policies. Hence, we dismiss this option here and just report an error if policy checks fail.

In general it should be noted that our checks for conflict and redundancy depend on the order in which policies are applied in the cluster. For instance, given policy pol with $\{a, b\}$ and $\{x, y\}$ from Table I, if a conflicting policy, e.g., the one with select set $\{a, b\}$ and allow set $\{x\}$, was applied before pol, the former would be accepted, while the pol would be judged redundant. Similarly, scheduling a redundant policy before pol, would make pol conflicting.

D. Design motivation and idea

Managing SGs involves execution of several time-intensive operations that may differ for different cloud providers. Table II shows the results for a closed lab OpenStack private cloud that has been used for evaluating GH. These results highlight the time intensity of creating and deleting SGs, compared to the lower times needed for adding/removing rules to a SG and attaching/detaching a SG. Contemporary cloud platforms rely on `ipset` for the `iptables` firewall or on eBPF that do not require costly operations such as a `reload iptables` command [33] when adding rules to an already attached SG. As a result, no impact on data plane application performance is expected when adding or removing rules to already attached SGs. The measurements in Table II and evaluation in Section VI-A confirm this.

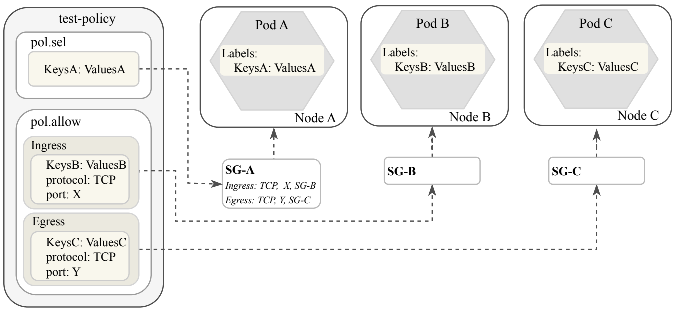


Fig. 2: Policy to SG mapping

Our design aims to reduce the number of SG creations and deletions by mapping several NPs to a single SG corresponding to a common selected label set. Hence, excessive SG creation and deletion, e.g., a dedicated SG per K8s NP, is avoided. Nevertheless, for a different setting than given by Table II, a different optimization strategy may be needed.

E. Labels-Security Group Hashmap

The core component of GH is a hashmap Map that records metadate for a given label set L, namely 1) the name s of the corresponding SG, 2) all policies selecting L including their allow sections with information about the traffic direction, ports, and protocols for targeted label sets or CIDRs, 3) all nodes that host pods matching L, 4) if s is targeted as a remote SG by some other SG. Entry $Map(L)$ thus represents a SG, to which remote SG rules are added for different NPs that all select the same label set L. Label sets are represented here as an ordered string concatenation of all contained labels.

Figure 2 illustrates how GH leverages the different sections of a NP to create new SGs. Given the K8s NP *test-policy* allows TCP ingress on port X from pod B to pod A and TCP egress on port Y from A to pod C. In response, SGs *SG-A*, *SG-B* and *SG-C* are created and *SG-A* is augmented with rules targeting *SG-B* and *SG-C* as *remote SGs*. As pod A is deployed on node A, the corresponding hashmap entry for its select label, Map('keysA:valuesA'), contains SG name *SG-A*, policy *test-policy*, and node A, but is not marked as a remote SG (as opposed to entries for 'keysB:valuesB' and 'keysC:valuesC').

F. Least-Privilege Security Group Management

To support least privilege network permissions, GH assumes that there is a permanent *DenyAll* K8s NP in place that restricts any communication between pods. GH further assumes that all default SGs are removed from the cluster, except for the *baseline* absolutely necessary for operating the cluster, e.g., those allowing communication between the master and the worker nodes or those unrelated to the K8s data plane.

At run-time, GH monitors the deployed pods and NPs and reacts to changes. At each such event, the algorithm traverses the various sections of the resources (pod or NP) and looks up metadata in the hashmap to decide the course of action, i.e., whether to add, remove, attach, detach, or modify SGs.

Adding a NP: When a new NP pol is added, we first check if it violates the constraints outlined in Sect. IV-B and, if so, record it in a special *Offenders* database without processing it further. Otherwise, we consider the newly added policy's select and allow sections separately. For every label set L_i specified in the allow section a single *remote SG* that is named after L_i is created and attached to all nodes with matching pods. The hashmap is updated to reflect all changes. If a SG already exists for L_i, it is just marked as a remote SG. Similarly, for the select label set S, we only create a new SG s if it does not exist yet in Map. We then add rules to s in correspondence with the rules of pol which are guaranteed to be unique, thanks to our redundancy checks.

Removing a NP: When a NP pol is removed, we essentially undo this addition of rules and delete all rules due to pol from the corresponding SG. We can also detach and delete a SG s if there are no more rules in s and s does not act as a remote SG of other SGs. When deleting s, we also need to consider deleting its remote SGs, unless they are still needed to implement other NPs.

Adding a pod: When a new pod p is added to the cluster on node N, we need to attach all SGs which match with the label set of p (if the SGs are not yet attached to N) and we record N in the hashmap entry for each such SG.

Removing a pod: Conversely, when a pod is deleted, we only detach a matching SG if that pod was the last on its node matching the corresponding label set of that SG. Migrating a pod to a different node is considered a deletion followed by an addition of that same pod in our algorithm.

Services: In K8s, a collection of pods can be exposed as a cloud native application to the outside world or other applications running on the same cluster by means of the *service* resource. Besides NPs, our approach also needs to take into account the addition and removal of services which can come in three different flavors. First, a cluster IP can be used as a target for intra-cluster traffic to a service. On each node, a proxy manages load balancing and routing traffic to the corresponding nodes and pods. However, connectivity between pods across services is still governed by NPs exclusively and if SGs connect the nodes hosting these pods accordingly, no additional measures are needed to support cluster IPs.

Exposing a service to both intra-cluster and external traffic can be achieved by assigning it a dedicated *node port*. As K8s assumes this port to be open on all nodes, GH complies by adding a corresponding rule to the baseline SG. A third option to expose services is to connect corresponding pods to a dedicated load balancer provided by the cloud provider. As the details are depending on the specific cloud provider and our prototype runs on a standalone OpenStack cluster, we do not consider this option here, but naturally any implementation of our approach by a cloud provider must enable connectivity between nodes that host service pods and their load balancer.

V. IMPLEMENTATION

Based on the aforementioned design, we introduce the implementation of GH [34] as a python library invoked by K8s API server events. GH is invoked on events pertaining to the creation, removal, or update of pods, NPs, or services to consistently update the SGs.

TABLE III: Application of GH on Fig. 4 policies with a netperf application

operation	**time (s)**
Time for pod ready	3 to 4
Create and attach SG for NP1 and NP2	0.263 (addition of NP1) 0.186 (addition of NP2)
Creating and attaching SG for NP3 Adding rules to SG for NP1	0.622 (addition of NP3)
Adding rules for NP4 to SG for NP3 Adding rules to SG for NP2	0.249 (addition of NP4)

Figure 3 shows an overview of the GH implementation, realizing the design described above. The *HashMap* is the central data structure managing per unique label set an SG metadata record as defined in Section IV-E. To store different NPs in such record, we use a second nested hash map based on policy names. All hash maps allow an $O(1)$ look-up time complexity. The key string used in *HashMap* is an alphabetically sorted string of concatenated `key:value` pairs.

There are mainly four steps involved in the operation of GH: 1) The *KubeWatcher* watches the K8s API server for events pertaining to NPs, pods, or services and records such changes in the *Cluster state*. 2) The *WatchDog* watches the *Cluster state* for changes. If a NP is created, this module invokes the *PolicyChecker* to verify the consistency of the new policy with already existing policies before invoking the *Matcher*. Inconsistent NPs are recorded in the *Offender* data structure and reported to the cluster admin. 3) If a pod, service, or verified NP is added or removed, the *WatchDog* invokes the *Matcher* which reads the *HashMap* and the *Cluster state* data structures and then executes the GH algorithm (Sect. IV-F). 4) *HashMap* updates are delegated to the *Security Group Nodule* which also performs all SG operations via the OpenStack interface. A SG (rule) is created only if there are matched pods in the cluster, thus no unnecessary SGs (rules) are created.

GH attaches SGs to the node the moment *KubeWatcher* detects the K8s scheduler node decision, thereby running SG configurations in parallel with pod starting process. Consequently, as long as the time to configure a SG does not exceed the time to start a pod, the effect of GH on the application is completely curtailed. This starting time, which was on average 4 seconds in our evaluation (cf. Table IV), can be considerable, especially in the case of cold starts and aggressive autoscaling [7]. However, given that research on reducing cold starts may continuously improve the state-of-the-art, the design methodology of GH has been based on reducing the number of time-costly SG operations as much as possible.

There is a specific performance and security optimization in the current implementation that ensures that no unnecessary Openstack rules are added to SGs. To this end, a *HashMap* entry e also stores which other entries represent remote SGs for e. This information is used to defer addition of remotes

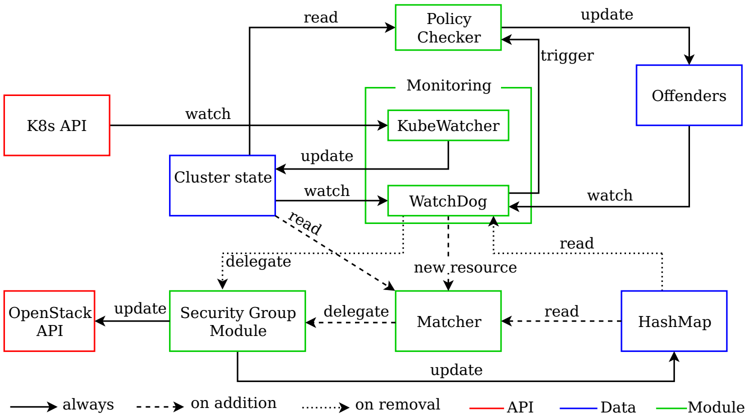
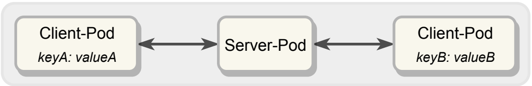


Fig. 3: GH design overview



NP	select	allow
NP_1	$\{keyA{:}valueA\}$	$\{\text{ING}\{server\}, \text{EG}\{server\}\}$
NP_2	$\{keyB{:}valueB\}$	$\{\text{ING}\{server\}, \text{EG}\{server\}\}$
NP_3	$\{server\}$	$\{\text{ING}\{keyA{:}valueA\}, \text{EG}\{keyA{:}valueA\}\}$
NP_4	$\{server\}$	$\{\text{ING}\{keyB{:}valueB\}, \text{EG}\{keyB{:}valueB\}\}$

Fig. 4: NPs for server setup with 2 client pods, both are allowed ingress (ING) from and egress (EG) to the server.

to the SG corresponding to e until these remotes in turn refer to e as well. For example, in Figure 4, NPs NP_1 and NP_3 are both needed to enable communication from a client to a server pod. After all, both an egress to the server pod and an ingress to the client are needed. Assume that at a time NP_1 has been created, but NP_3 does not exist yet. Then, only a security group for NP_1 is created and attached, but no rules are added, yet . When NP_3 is added, its remotes will match NP_1 whose remotes now matches NP_3 and rules will be added to the SG for NP_3 and to the already created SG for NP_1 (cf. Table III). Similarly, when one of the policies is deleted, the remotes associated with these policies will be removed.

VI. EXPERIMENTATION AND EVALUATION

We evaluated GH's performance, security, and efficiency in reducing node connectivity. The results are discussed below.

A. *Performance Evaluation*

In this section we present the results of the performance evaluation of GH. We evaluated GH for three synthetic applications: netperf [35], an adaptive Software-as-a-Service (SaaS) application [36], and teastore application [37], on two eBPF Container Network Interface (CNI) plugins Calico and Cilium. Both Calico and Cilium were configured to run in native mode using pure layer 3 networking.

1) Experiment Settings: The testbed used for running all the experiments is an isolated part of a private OpenStack cloud, version 21.2.4. We run containers on top of OpenStack because it is not only the standard for private clouds and the most widely deployed open source cloud computing software, but also the foundation for public clouds [38]. The OpenStack cloud consists of a master-worker architecture with two controller machines, and droplets on which VMs can be scheduled. The droplets have 2 x Intel(R) Xeon(R) CPU E5-2660 v3 @ 2.60GHz (10 cores, 20 threads) 128 GB RAM. Each droplet has two 10Gbit network interfaces and is configured with `ipset` enabled. The K8s cluster used in the evaluation was deployed using Kubeadm, running K8s version 1.23.1 and consists of one master node and eight worker nodes. All nodes have 4 vCPUs and 8GB RAM, and all were deployed on the same physical droplet to eliminate variations in network delay.

a) Netperf evaluation: With netperf application, we used netperf TCP stream mode for throughput and CPU utilization measurements, and request-response (RR) mode for end-to-end latency measurements. We configured netperf for a test length of 120 seconds with the goal of 99% confidence level that the measured mean values are within +/- 2.5% of the mean values of another sample of the same population. In this evaluation, we connected to the pods directly using their IP addresses.

b) SaaS application evaluation: The SaaS application used for evaluating GH is written in C++ and is based on the COMITRE approach [39]. It provides a REST API (SaaS API) to which users can send requests. With every user belonging to a tenant, each request has a tenantId field so that the application can retrieve the tenant-specific configuration. Parameters can be configured separately for each tenant to determine which resource types (CPU, memory or disk I/O) will be mainly stressed [36]. For this evaluation, CPU resource was stressed. The default auto-scaler in K8s, Horizontal Pod Autoscaler (HPA) [6], was configured to keep average CPU usage of the service's Pods around 50% so that autoscaling happens aggressively when the request rate is linearly increased. The SaaS application is configured to run for 600s for each request rate. Unlike netperf, for this application, pods run behind the built-in K8s load balancer.

c) Teastore evaluation: The teaStore benchmark [37] provides a microservice based software application consisting of five services in additon to a registry necessary for service

TABLE IV: SG operation times (s) by GH for netperf and SaaS application.

	netperf	**SaaS**
Time for pod ready	3 to 4	3 to 4
Create and attach SG to node	0.234 (server pod) 0.333 (two client pods)	0.357 (first SaaS pod on node x) 0.084 (first scaled pod on node y) 0.075 (second scaled pod on node z)
Detach and remove from node	0.05	0.05

discovery. The five services include a *WebUI* providing the user interface serving Java Server Pages and images provided by *Image provider* service, an *Auth* service for the verification of login and session data of a user, the *Recommender* service which uses a rating algorithm to recommend products for the user to purchase, and the *Persistence* service providing access to and caching for the store's relational database. For this evaluation, we measured the total average and median response times for all the services, and the total requests per second and each measurement was repeated 20 times.

2) Evaluation Results: In the evaluation, we measure the performance without GH (No GH), performance with GH (GH) and the performance of GH when arbitrary pods and NPs are added to the cluster requiring several SGs to be created during the experiment (GH+fly). We answer the following questions regarding GH's performance:

Qn1. How does GH impact performance of K8s applications with respect to end-to-end latency, throughput, and resource utilization?

Qn2. How much time does GH take to configure (or remove) least privilege SGs on addition (or removal) of an NP or a Pod scheduling decision?

Qn3. What is the performance impact on container applications when other pods, their NPs, and subsequent SGs, are periodically added by GH?

With respect to Qn1, we compare Netperf, SaaS, and Teastore applications performance without GH to that with GH running in the cluster. The results as observed in Fig. 5, Fig. 6, and Fig. 7 show that GH does not affect application performance for both Calico and Cilium CNI plugins.

To answer Qn2, we measure the time taken by GH to detect the addition of a new resource to the K8s API, get required resource information, match and lookup the resource, create a SG and/or add rules, and attach the SG to the pertinent node. As observed in Table IV, this time was less than 0.4 seconds when creating a pod for the first time and even less when pods are scaled. The time for scaled pods is lower because no new SG is created when pods are replicated. Rather, the SG stored in the hashmap corresponding to the first pod is attached to the new nodes hosting the replicated pods. Additionally, we measured the time taken by GH to look up a SG in the hashmap and detach it from the pertinent node when a resource is deleted from the cluster. This time (cf. Table IV) is around 0.05 seconds. Finally Table III shows the efficiency of adding rules to an already created SG (rather than creating and attaching a new SG). Overall, the results in both tables IV and III indicate that SG configuration time was much lower than the 3 to 4 seconds that application pods took to start up and become ready to serve requests. Considering that GH operations run in parallel with pod start-up, this shows that a SG will be configured before the pod is ready, thereby steering clear of affecting application performance.

To answer Qn3, we repeated the evaluations for Qn1 while periodically adding new pods with their corresponding NPs to the cluster. A total of 40 pods and NPs was added during each run of the experiment, with a new SG created and attached to the pertinent node for each pod policy pair. The results of this experiment are indicated under the 'GH+fly' labeled results of Fig. 5, Fig. 6, and Fig. 7. With the exception of a slight increase in the CPU utilization of the local node (node hosting the server pod) owing to CPU consumed by added pods, there was no observed impact on application performance. This further demonstrates that GH can configure SGs to multiple applications without affecting their performance.

B. Security Validation

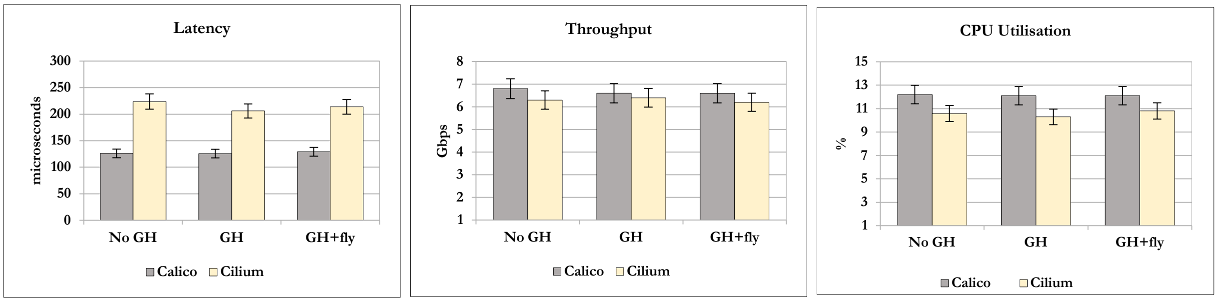
This section uses the second attack scenario introduced in Sect. III where an attacker exploits an accessible kubelet API to gain access to the cluster API server. As explained before, we assume that authentication and authorization of the kubelet agent on a worker node is not functioning correctly, e.g., due to a configuration error.

An attacker who managed to compromise an existing or deploy a malicious pod on a random worker node scans for K8s nodes with an unprotected kubelet API by means of the kubeletctl tool [32], which returns all insecure kubelets in the cluster as shown in Fig. 8a. Then, as observed in Fig. 8b, he is able to see all pods running on the node hosting the insecure kubelet API. Alternatively, as observed in Fig. 8c, the attacker can also find powerful API server tokens of trampoline pods [4] on that node, by means of which he can get access to the API server with the goal to retrieve cluster-admin privileges.

With GH, even if the attacker has a foothold in the cluster and runs arbitrary code in a pod on a worker node, scanning for nodes with insecure kubelet API or searching for the pods that are vulnerable to remote code execution will return no results for dynamically isolated nodes as explained by Fig. 8d.

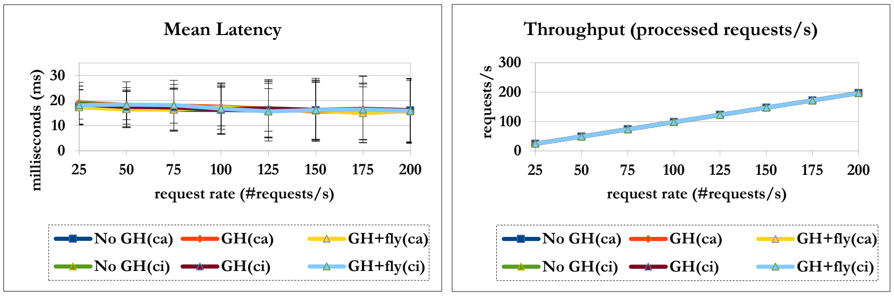
C. Reduction of network attack surface

To evaluate the effectiveness of GH for representative cluster setups and workloads, this section measures the reduction of the network attack surface as a ratio of the *connectivity density* (CD) achieved with GH in comparison to a statically defined SG. CD equals the number of opened directed connections between any pair of nodes of the cluster. A statically defined SG for N cluster nodes (CD_{noGH}) corresponds to opening P ports of J application components of A applications at a cluster, and allowing all possible connections between any pair of its N nodes in both directions (cfr. Eq. 1). For GH, we compute the CD by deriving common Directed Acyclic Graph (DAG) properties for call graphs of micro-service (MSs), found in two recent studies by Aliba [40] and ByteDance [41]. Both studies show that the average call graph within an MS application can be represented as one



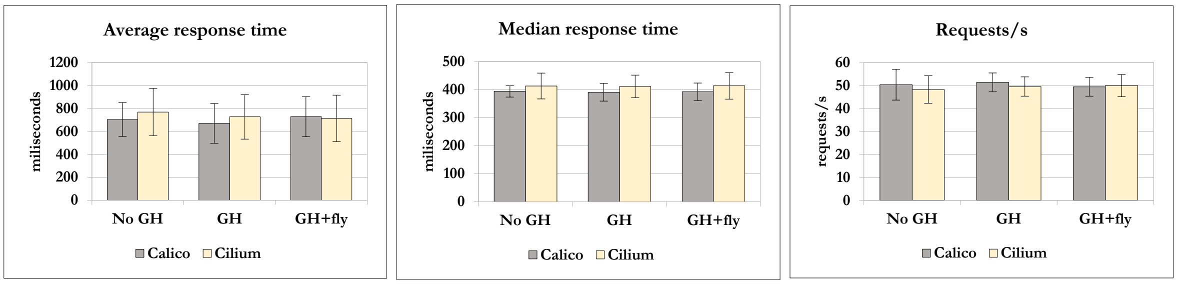
(a) Latency (b) Throughput (c) CPU Utilisation

Fig. 5: The netperf evaluation shows no performance overhead by GH, i.e. error bars, which indicate margin of error, overlap.



(a) Latency (b) Throughput

Fig. 6: Evaluation of GH in an agressive autoscaling scenario of the SaaS app; error bars indicate standard deviation



(a) Average response time (b) Median response time (c) Average requests

Fig. 7: Teastore application evaluation; error bars indicate standard deviation

long chain of nested MSs or as a full binary tree, both of which have $J-1$ edges between J MSs. While a call graph is just a dynamic subset of a full dependency graph, we think $J-1$ is the correct measure for more traditional multi-tier applications. Moreover we assume that I different instances of the same MS are always replicated across different nodes for reasons of reliability and availability. Thus, CD_{GH} can be defined accordingly to Eq. 2. Note that a fair comparison requires to assume a fully-loaded K8s cluster; to this end, we define capacity C as the number of pods that a single node can maximally hold; The number of nodes N is then computed as shown in Eq. 3. Finally, the reduction rate is defined by Eq. 4 as the ratio of the former CD values.

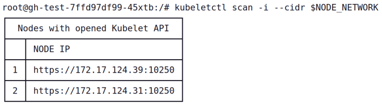
$$CD_{noGH} = 2 \times \binom{N}{2} \times A \times J \times P \tag{1}$$

$$CD_{GH} = (J-1) \times I^2 \times P \times A \tag{2}$$

$$N = \lceil \frac{I \times J \times A}{C} \rceil + 2 \tag{3}$$

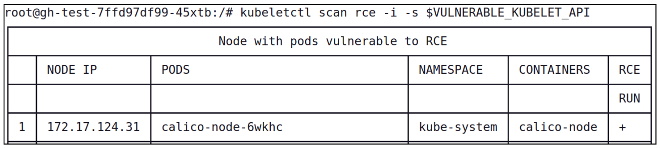
$$Reductionrate = (1 - \frac{CD_{GH}}{CD_{noGH}}) \times 100 \tag{4}$$

Figure 9 shows the resulting reduction rate of 91.94% for the baseline scenario [$J = 2^3 - 1 = 7$, $C = 30$,$I = 2$, $A = 10$]. Maximum values for the variables are: $\bar{J} = 2^5 - 1$ ([41] observed a maximum depth of 5 for the call graph); $\bar{C} = 110$ pods per node[42]; $\bar{I} = 3$ (most production, i.e. low-latency service-oriented workloads at Google[43] have less than 3 instances per service); $\bar{A} = 5911$ as the largest K8s cluster has $N = 5000$ nodes. At these maximum values, the reduction rate approximates 100%. When the variables are



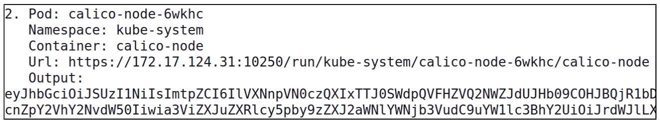
```
root@gh-test-7ffd97df99-45xtb:/# kubeletctl scan -i --cidr $NODE_NETWORK
 Nodes with opened Kubelet API
     NODE IP
 1   https://172.17.124.39:10250
 2   https://172.17.124.31:10250
```

(a) Vulnerable Kubelet APIs



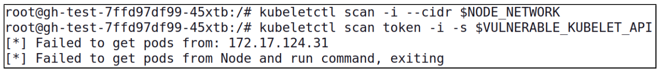
```
root@gh-test-7ffd97df99-45xtb:/# kubeletctl scan rce -i -s $VULNERABLE_KUBELET_API
                         Node with pods vulnerable to RCE
    NODE IP        PODS                        NAMESPACE     CONTAINERS    RCE
                                                                           RUN
 1  172.17.124.31  calico-node-6wkhc           kube-system   calico-node   +
```

(b) Exposed vulnerable Pods



```
2. Pod: calico-node-6wkhc
   Namespace: kube-system
   Container: calico-node
   Url: https://172.17.124.31:10250/run/kube-system/calico-node-6wkhc/calico-node
   Output:
eyJhbGciOiJSUzI1NiIsImtpZCI6IlVXNnpVN0czQXIxTTJ0SWdpQVFHZVQ2NWZJdUJHb09COHJBQjR1bD
cnZpY2VhY2NvdW50Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9uYW1lc3BhY2UiOiJrdWJlLX
```

(c) Powerful token for API server access



```
root@gh-test-7ffd97df99-45xtb:/# kubeletctl scan -i --cidr $NODE_NETWORK
root@gh-test-7ffd97df99-45xtb:/# kubeletctl scan token -i -s $VULNERABLE_KUBELET_API
[*] Failed to get pods from: 172.17.124.31
[*] Failed to get pods from Node and run command, exiting
```

(d) Scanning with kubeletctl blocked by GH

Fig. 8: Without GH, an attacker is able to detect a node/VM with a vulnerable kubelet API from within any pod (cfr. Fig. a) and search for vulnerable pods (Fig. b) or powerful API server tokens (Fig. c). With GH, the attacker is unable to do this within any pod that cannot communicate with any pod on that vulnerable VM (Fig. d); if he can communicate with other pods on that VM, the VM is compromised regardless of the kubelet API vulnerability.

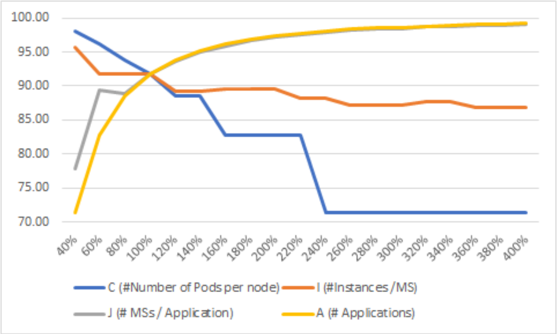


Fig. 9: Reduction rate for the network attack surface with the base-line scenario [$J = 7$, $C = 30$,$I = 2$, $A = 10$] visible at 100% of the X-axis

varied in steps of 20% from the baseline values we can see that the reduction rate vary between 70% and 99% for any node capacity C if there are at least 4 different applications deployed.

VII. RELATED WORK

State-of-the-art solutions for managing consistency of network security policies across different system layers employ a verification approach [44], [45], [10] for detecting inconsistencies but lack procedures for reinforcing consistency in a fully automated manner. Existing container networking solutions [46], [11], [47] and policy generators [48] do not support consistency between NPs and SGs or only do so for external cluster traffic, not intra-cluster traffic [47]. Some microsegmentation vendors [49] allow dividing a K8s cluster in different static segments, but this segmentation of VMs is not informed by or verified against container-level NPs and scheduling decisions. We elaborate upon these differences of GH in the following subsections.

a) Verification of container-based NPs: As already stated in Sect. IV, GH is inspired by the policy checker Kano [10] to verify K8s NPs against inconsistencies, duplication and violation of the principle of least privilege. Although Kano does not itself preserve consistency with SGs, Kano can be extended with such facility.

b) Multi-level consistency management: NFVGuard [44] verifies the security of multi-layer Network Functions Virtualization across an OpenStack platform.It employs verification as the main mechanism which requires a planning component to resolve inconsistencies. For latency-sensitive applications that must be scaled up dynamically, such a component would take take too long. GH thus uses an enforcement approach which does not delay auto-scaling of pods.

c) Least-privilege NP generators: Bastion [11] has developed support for generating least-privilege NPs for microservices based application from higher-level service interaction graphs. AutoArmor [48] generates inter-service access policies for Istio and K8s by static code analysis. While these works do not generate SG configurations at the cloud level as GH does, they offer a complementary defense by relying on OS-based firewalls for DoS attacks.

d) Microsegmentation: Microsegmentation hinders lateral movement of an attacker within a cloud-based application using fine-grained and distributed firewall rules. All major microsegmentation platforms such as Cisco ACI [50], VMware NSX [51], Paloalto's PrismaCloud [49] and Illumio's Core [52] support integration with K8s GH has two unique features that are not present in these platforms: 1) GH supports dynamic orchestration of SGs so that if a Pod is scheduled on a VM that is not reachable, a least privilege SG configuration will be attached to the VM on the fly. In opposition, all microsegmentation platforms separate VM-based and K8s-based firewall rules in different domains that are not dynamically coordinated as GH does. 2) The on-the fly generation of SG configurations without delaying pod readiness is not supported in these platforms. Here the readiness of a pod is delayed until firewall rules have propagated to all its peers. For example, a policy convergence controller in Illumio [53] defaults to a configurable delay between 0 and 300 seconds with a default of 15 seconds.

e) Cloud providers: AWS' Elastic K8s Service (EKS) offer SGs for pods [46] to govern intra-cluster communication and egress communication with AWS services. This solution does not preserve consistency with K8s-native NPs nor is it generally portable to other K8s vendors. In opposition, GH preserves such consistency and is also fully portable across K8s vendors and CNI networking plugins. Network plugins such as Calico Enterprise [47] also integrate K8s NPs and SGs but only concerning traffic from within a cluster to services running outside the cluster.

VIII. CONCLUSION

In this paper we have argued that in the domain of ultra-reliable and low-latency cloud-native systems, there is an inherent conflict between fast container networking and reduction of the network attack surface at the VM level by means of manually defined security groups. Indeed, container network solutions without network encapsulation require that every opened container protocol and ports must also be opened at the VM level. Therefore, to reduce the network attack surface at the VM level, least-privilege security groups must be adapted depending on the dynamic placement of containers on VMs. This adaption must be performed faster than the time it takes for the new container to come online. We have proposed GrassHopper, a novel cross-layer enforcement approach to generate VM-level security groups from at-run-time verified container-level network policies. We have implemented and evaluated GrassHopper on top of a K8s cluster running on OpenStack. Evaluation through experimentation and analysis has shown that for a wide range of cluster setups and applications, the network attack surface between VMs can be reduced at a ratio of 70-to-99% at *zero cost*, i.e there is no significant overhead on any relevant metric for container application performance. These results confirm the efficiency and applicability of GrassHopper for ultra-reliable and low-latency cloud-native systems.

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] NGMN Alliance, "Next generation mobile networks, 5G white paper," https://ngmn.org/wp\-content/uploads/NGMN_5G_White_Paper_V1_0.pdf, 2015, [Accessed: 2022-03-24].

[2] B. Kaur, M. Dugré, A. Hanna, and T. Glatard, "An analysis of security vulnerabilities in container images for scientific data analysis," *GigaScience*, vol. 10, no. 6, p. giab025, 2021.

[3] K. Wist, M. Helsem, and D. Gligoroski, "Vulnerability analysis of 2500 docker hub images," in *Advances in Security, Networks, and Internet of Things: Proceedings from SAM'20, ICWN'20, ICOMP'20, and ESCS'20*. Springer, 2021, pp. 307–327.

[4] Y. Avrahami and S. Ben Hai, "Trampoline pods: Node to admin privesc built into popular k8s platforms," https://www.youtube.com/watch?v=PGsJ4QTlKlQ, 2022, [Accessed: 2022-12-09].

[5] G. Budigiri, C. Baumann, J. T. Mühlberg, E. Truyen, and W. Joosen, "Network policies in kubernetes: Performance evaluation and security analysis," in *2021 Joint European Conference on Networks and Communications & 6G Summit (EuCNC/6G Summit)*. IEEE, 2021, pp. 407–412.

[6] "Horizontal pod autoscaling," https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/, 2022, [Accessed: 2022-05-24].

[7] E. H. Beni, E. Truyen, B. Lagaisse, W. Joosen, and J. Dieltjens, "Reducing cold starts during elastic scaling of containers in kubernetes," in *SAC '21: The 36th ACM/SIGAPP Symposium on Applied Computing, Virtual Event, Republic of Korea, March 22-26, 2021*, C. Hung, J. Hong, A. Bechini, and E. Song, Eds. ACM, 2021, pp. 60–68.

[8] V. Kjorveziroski and S. Filiposka, "Kubernetes distributions for the edge: serverless performance evaluation," *The Journal of Supercomputing*, vol. 78, pp. 13 728–13 755, 2022.

[9] R. Harrison, "An introduction to kubernetes network policies for security people," https://reuvenharrison.medium.com/an-introduction-to-kubernetes-network-policies-for-security-people-ba92dd4c809d, 2019, [Accessed: 2022-04-21].

[10] Y. Li, C. Jia, X. Hu, and J. Li, "Kano: Efficient container network policy verification," in *2020 IEEE Symposium on High-Performance Interconnects (HOTI)*. IEEE, 2020, pp. 63–70.

[11] 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 Conference, USENIX ATC 2020, July 15-17, 2020*, A. Gavrilovska and E. Zadok, Eds. USENIX Association, 2020, pp. 81–95.

[12] L. Chen, Y. Xia, Z. Ma, R. Zhao, Y. Wang, Y. Liu, W. Sun, and Z. Xue, "Seaf: A scalable, efficient, and application-independent framework for container security detection," *Journal of Information Security and Applications*, vol. 71, p. 103351, 2022.

[13] R. Shu, X. Gu, and W. Enck, "A study of security vulnerabilities on docker hub," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, 2017, pp. 269–280.

[14] H. Barua, "Half of 4 million public docker hub images found to have critical vulnerabilities," https://www.infoq.com/news/2020/12/dockerhub-image-vulnerabilities/, 2020, [Accessed: 2023-03-20].

[15] J. Chen, "Attacker's tactics and techniques in unsecured docker daemons revealed," https://unit42.paloaltonetworks.com/attackers-tactics-and-techniques-in-unsecured-docker-daemons-revealed/, 2020, [Accessed: 2023-03-20].

[16] M. MacLeod, "Escaping from a virtualised environment: An evaluation of container breakout techniques," 2021.

[17] M. Manning, "Deep dive into real-world kubernetes threats – ncc group research," https://research.nccgroup.com/2020/02/12/command-and-kubectl-talk-follow-up/, 2020, [Accessed: 2022-03-29].

[18] E. Baitello, "Attacking kubernetes clusters using the kubelet api," https://faun.pub/attacking-kubernetes-clusters-using-the-kubelet-api-abafc36126ca\#c0b6, 2021, [Accessed: 2022-03-29].

[19] B. Edwards and N. Freeman, "A compendium of container escapes," https://i.blackhat.com/USA-19/Thursday/us-19-Edwards-Compendium-Of-Container-Escapes-up.pdf, 2019, presentation at Blackhat USA 2019 [Accessed: 2022-12-15].

[20] "CVE-2020-15257: Privilege escalation from containerd," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-15257, 2020, [Accessed: 2022-12-15].

[21] "CVE-2019-5736: Privilege escalation from runc," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-5736, 2019, [Accessed: 2022-12-15].

[22] M. J. Reeves, "Investigating escape vulnerabilities in container runtimes," Master's thesis, Purdue University Graduate School, 2021.

[23] "Container breakout vulnerabilities," https://www.container-security.site/attackers/container_breakout_vulnerabilities.html, 2022, [Accessed: 2022-05-12].

[24] B. Fox, "Unrestricted hostpath," https://github.com/BishopFox/badPods/tree/main/manifests/hostpath, 2021, [Accessed: 2022-03-29].

[25] H. Zhu and C. Gehrmann, "Kub-sec, an automatic kubernetes cluster apparmor profile generation engine," in *2022 14th International Conference on COMmunication Systems NETworkS (COMSNETS)*, 2022, pp. 129–137.

[26] S. Ghavamnia, T. Palit, A. Benameur, and M. Polychronakis, "Confine: Automated system call policy generation for container attack surface reduction," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. USENIX Association, 2020.

[27] N. Stoler, "The route to root: Container escape using kernel exploitation," https://www.cyberark.com/resources/threat-research-blog/the-route-to-root-container-escape-using-kernel-exploitation, 2019, [Accessed: 2022-12-15].

[28] "CVE-2020-8559: Privilege escalation from compromised node to cluster," https://groups.google.com/g/kubernetes-security-announce/c/JAIGG5yNROs?pli=1, 2020, [Accessed: 2022-05-12].

[29] Palo Alto Networks, "Kubernetes Privilege Escalation: Excessive Permissions in Popular Platforms," https://www.paloaltonetworks.com/resources/whitepapers/kubernetes-privilege-escalation-excessive-permissions-in-popular-platforms, 2022, [Accessed: 2022-12-07].

[30] Imperva, "What is a reverse shell?" https://www.imperva.com/learn/application-security/reverse-shell/, [Accessed: 2023-03-25].

[31] "Kubelet authentication/authorization," https://kubernetes.io/docs/reference/access-authn-authz/kubelet-authn-authz/, 2022, [Accessed: 2022-12-15].

[32] CyberArk, "Kubeletctl tool," https://github.com/cyberark/kubeletctl, 2022, [Accessed: 2022-12-15].

[33] "Openstack configuration reference," https://docs.openstack.org/liberty/config-reference/content/networking-options-securitygroups.html, 2016, [Accessed: 2022-05-27].

[34] Anonymized, "Grasshopper artifact," https://anonymous.4open.science/r/grasshopper-E806/, 2022, we will release this artifact publicly upon acceptance of the paper.

[35] HewlettPackard, "Netperf," https://github.com/HewlettPackard/netperf/, 2022, [Accessed: 2022-04-11].

[36] E. Truyen, A. Jacobs, S. Verreydt, E. H. Beni, B. Lagaisse, and W. Joosen, "Feasibility of container orchestration for adaptive performance isolation in multi-tenant saas applications," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, 2020, pp. 162–169.

[37] J. Von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, and S. Kounev, "Teastore: A micro-service reference application for benchmarking, modeling and resource management research," in *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2018, pp. 223–236.

[38] S. Murugesan and I. Bojanova, *Encyclopedia of cloud computing*. John Wiley & Sons, 2016.

[39] L. C. Ochei, J. M. Bass, and A. Petrovski, "Evaluating degrees of multitenancy isolation: A case study of cloud-hosted gsd tools," in *2015 International Conference on Cloud and Autonomic Computing*. IEEE, 2015, pp. 101–112.

[40] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, J. He, and C. Xu, "An in-depth study of microservice call graph and runtime performance," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, 2022.

[41] Y. Wen, G. Cheng, S. Deng, and J. Yin, "Characterizing and synthesizing the workflow structure of microservices in bytedance cloud," *Journal of Software: Evolution and Process*, vol. 34, no. 8, p. e2467, 2022.

[42] C. N. C. Foundation, "Considerations for large clusters," https://kubernetes.io/docs/setup/best-practices/cluster-large/, 2023, [Accessed: 2023-03-25].

[43] M. Tirmazi, A. Barker, N. Deng, M. E. Haque, Z. G. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes, "Borg: The next generation," in *Proceedings of the Fifteenth European Conference on Computer Systems*. Association for Computing Machinery, 2020.

[44] A. Oqaily, S. L. T, Y. Jarraya, S. Majumdar, M. Zhang, M. Pourzandi, L. Wang, and M. Debbabi, "Nfvguard: Verifying the security of multilevel network functions virtualization (nfv) stack," in *2020 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2020.

[45] Y. Wang, T. Madi, S. Majumdar, Y. Jarraya, A. Alimohammadifar, M. Pourzandi, L. Wang, and M. Debbabi, "Tenantguard: Scalable runtime verification of cloud-wide vm-level network isolation," in *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017.

[46] M. Stefaniak and S. S. Balaji, "Introducing security groups for pods," https://aws.amazon.com/blogs/containers/introducing-security-groups-for-pods, 2020, [Accessed: 2022-05-22].

[47] Tigera, "Configure calico enterprise aws security groups integration," https://docs.tigera.io/security/aws-integration/aws-security-group-integration, 2022, [Accessed: 2022-04-22].

[48] X. Li, Y. Chen, Z. Lin, X. Wang, and J. H. Chen, "Automatic policy generation for Inter-Service access control of microservices," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2021.

[49] Paloalto, "Prisma Cloud Microsegmentation Administrator's Guide," https://docs.paloaltonetworks.com/content/dam/techdocs/en_US/pdf/prisma/prisma-cloud/prisma-cloud-admin-microsegmentation/prisma-cloud-admin-microsegmentation.pdf, 2022, [Accessed: 2022-12-02].

[50] Cisco, "Cisco ACI and Kubernetes Integration," https://www.cisco.com/c/en/us/td/docs/switches/datacenter/aci/apic/sw/kb/b_Kubernetes_Integration_with_ACI.html, 2022, [Accessed: 2022-11-13].

[51] VMware, "NSX Container Plugin for Kubernetes and Tanzu Application Service - Installation and Administration Guide," https://docs.vmware.com/en/VMware-NSX-Container-Plugin/4.0/ncp_40_kubernetes.pdf, 2022, [Accessed: 2022-11-13].

[52] Illumio, "Illumio Core for Kubernetes and OpenShift," https://docs.illumio.com/core/22.4/Content/Resources/PDF/Illumio_Core_for_Kubernetes_and_OpenShift_21.5.18.pdf, 2022, [Accessed: 2022-11-13].

[53] illumio, "Local policy convergence controller," https://docs.illumio.com/core/21.2/Content/Guides/kubernetes-and-openshift/security-policies/local-policy-convergence-controller.htm, 2022, [Accessed: 2022-12-15].