Elastic Cross-Layer Orchestration of Network Policies in the Kubernetes Stack

Gerald Budigiri[1], Eddy Truyen[1], Christoph Baumann[2], Jan Tobias Mühlberg[1], and Wouter Joosen[1]

[1] imec-DistriNet, KU Leuven, Belgium
{gerald.budigiri, eddy.truyen, jantobias.muehlberg,wouter.joosen}@kuleuven.be
https://distrinet.cs.kuleuven.be/
[2] Ericsson Research, Stockholm, Sweden
christoph.baumann@ericsson.com

Abstract. Kubernetes (K8s) is a popular container orchestration framework for dynamic and on-demand deployment of service based applications. In a K8s cluster, containers are commonly deployed inside nodes provided as Virtual Machines (VMs) by IaaS providers such as OpenStack. In such layered container-VM model, network policies at the container layer and security groups at the VM layer provide redundant firewall mechanisms that strengthen defenses against attackers attempting to spread to other nodes/containers. However, least-privilege network policies at the container layer may not always be consistent with statically defined security groups at the VM layer. In this paper, we introduce GrassHopper, a fast and dynamic cross-layer enforcement of network policies by automatically generating security group configurations from network policies. Due to the constant changes in the cluster, such as continuous deployment of applications and elastic scaling of containers across VMs, Grasshopper continuously re-adapts security group configurations in function of the actual placement of containers on VMs. The performance overhead of GrassHopper was evaluated on a K8s cluster running on OpenStack using netperf and a synthetic SaaS application. We observed neither overhead on performance nor any impact on the readiness of newly started containers during aggressive auto-scaling.

Keywords: Kubernetes · Network isolation · Network policies · OpenStack · Security groups.

1 Introduction

5G is envisaged to support ultra-low-latency and ultra-high-reliability applications through coupling with the technologies of edge computing and containerization. Edge computing is needed to effectuate lower latencies and increased connectivity for applications such as vehicle-to-everything (V2X) applications and remote surgery. Consider, for example, urgent life-saving manipulations by a surgeon that are performed by a robotic arm to a patient in an ambulance [19].

Containers are preferred over the more resource-intensive, and slower-to-start virtual machines (VMs) due to their lightweight nature and high portability. To meet ultra-high reliability-requirements, however, redundant defense mechanisms are needed across the stack of application code, container and edge platform. With this end-in-view, containers are typically ran inside VMs to protect the infrastructural assets of edge and cloud providers.

This paper argues that least-privilege network policies for restricting inter-container communication should be automatically enforced at the level of security groups of VMs. Without such automated enforcement mechanism in place, any attack that is able to escape a container to the VM, can spread across all the VMs inside the container cluster. This occurs whenever the manually-defined security group of a container cluster is a default, permissive setting that allows communication between all VMs, albeit through specific protocols and port ranges. The lack of automation in such scenarios is cited as the reason for 50% of all network segments, applications or APIs that were inadvertently exposed directly to the public internet, and 99% of all firewall breaches [12, 15].

The main problem is that configurations for attaching security groups to VMs must be updated as fast as possible when applications are continuously deployed and containers are dynamically auto-scaled across VMs in order manage workload deviations [1]. Existing state-of-the-art solutions for managing consistency of network policies employ a verification approach [14, 21, 25] but lack fast enforcement of consistency. Existing container networking solutions [18, 22, 23] and policy generators [13] do not support consistency between network policies and security groups or only do for external cluster traffic, not intra-cluster traffic [23].

We introduce GrassHopper, a solution for cross-layer enforcement of network policies that are verified at run-time. GrassHopper brings much-needed automation to the management of Kubernetes network policies and generates least privilege security groups from verified network policies and enforces these on security group configurations in line with container scheduler actions at run-time. More specifically, we make the following contributions:

- We present an efficient algorithm for generating least-privilege security group configurations from network policies and container scheduling actions so that the number of time-intensive security group operations are maximally reduced.
- We integrate existing network policy checkers into the design of GrassHopper to ensure that network policies at the level of the container cluster can serve as the base truth.
- We demonstrate the applicability of GrassHopper by implementing it on top on Kubernetes (K8s), a popular container orchestration framework, and by evaluating it on a cross-network K8s cluster running on top of a private OpenStack cloud.

The remainder of the paper is organized as follows. Section 2 and 3 provide the background and motivation. Section 4 details the design and methodology of GrassHopper. Section 5 and 6 present the implementation and evaluation. In Section 7 and 8 we discuss related work and conclude our work.

2 Background

This section introduces the background of K8s network policies and verification thereof, security groups and the notation used in the remainder of the paper.

Kubernetes network policies. K8s orchestrates containerized applications, automatically managing their lifecycle including deployment, scaling, and networking. Containerized applications run in pods, the smallest unit of execution in K8s, consisting of one or more tightly coupled containers deployed together. Pods run on nodes, the physical servers or VMs forming a K8s cluster. By default, all pods in the cluster are non-isolated, accepting all ingress and egress traffic. This is precarious from a security perspective, especially in mutually distrusting multi-tenant clusters. Hence, K8s provides network policies to control traffic between pods, tenants, or network endpoints. Such policies provide configurable network isolation by controlling traffic flow at layer 3 and layer 4.

A network policy specification consists of a (*select*) section specifying pods subject to the policy and an *allow* section specifying allowed ingress/egress traffic to/from other pods. Since pod IPs are automatically assigned and change frequently, network policies use pod labels to select pods and namespaces in the cluster, and ipBlocks for external connectivity [11]. K8s uses Container Networking Interface (CNI) network plugins for policy enforcement. In this paper, Calico CNI is used because it uses the extended Berkeley Packet Filter (eBPF) technology for fast policy enforcement and it defaults to to a low overhead pure layer 3 networking solution instead of VXLAN tunneling.

Network policy inconsistency and misconfigurations. Network policy misconfigurations and inconsistencies can be exploited by bad actors to gain illicit access to containerized applications in the cluster, and may result in data breaches, service interruptions, or whole 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 [14] can be used to verify such policies against misconfigurations such as policy conflict, redundancy, and violation of the least privilege principle. Bastion [18] 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 security groups to support configurable inter-node network isolation at the level of VMs. A security group in OpenStack consists of a set of network access filter rules that allow or deny traffic based on port, protocol, IP address or remote security group. The latter notion of remote security group abstracts over IP addresses and instead allows filter rules to refer to other nodes by means of a label.

Notation. In order to simplify the following exposition, we introduce a bit of formalism to describe network policies and security groups. Firstly, in the context of this work it suffices to represent pods scheduled on the cluster by a record

containing a name and a set of labels, i.e., $Pod = \langle\!| name \in Podname; L \subseteq Label |\!\rangle$. Since this work is mainly interested in intra-cluster communication security, labels should be interpreted as the typical key-value pairs used by K8s to tag and match pods, matching IP blocks for external communication is out of scope. Then, network policies are signified by a name, the set of labels they select, and a set of traffic rules specifying allowed ingress or egress for a set of labels:

$$Policy = \langle\!| name \in Polname; sel \subseteq Label; allow \subseteq \mathcal{P}(Label) \times \{\textsc{Ing}, \textsc{Eg}\} |\!\rangle .$$

Here $\mathcal{P}(S)$ is the powerset of S. Similarly, a Security group has a name, and a set of remote allowed ingress or egress security groups:

$$SecGroup = \langle\!| name \in SGname; remotes \subseteq SGname \times \{\textsc{Ing}, \textsc{Eg}\} |\!\rangle .$$

For policies $pol_1, pol_2 \in Policy$ we can define an interpretation $\langle pol_1, pol_2 \rangle \subseteq \mathcal{P}(Label) \times \mathcal{P}(Label)$, s.t. if $(A, B) \in \langle pol_1, pol_2 \rangle$ it means that pol_1 allows egress from pods with labels A to pods having labels B and pol_2 allows ingress to pods with B from pods with A. A pod p_1 can then communicate with a pod p_2, written $p_1 \blacktriangleright p_2$, if there exist policies pol_1, pol_2 in the cluster with $(A, B) \in \langle pol_1, pol_2 \rangle$ s.t. $A \subseteq p_1.L$ and $B \subseteq p_2.L$. On the virtualization layer, a node N_1 may communicate with N_2, i.e., $N_1 \blacktriangleright N_2$, if there exist security groups sg_1^i, sg_1^e and sg_2^i, sg_2^e attached to the respective nodes, such that $(sg_1^i, \textsc{Ing}) \in sg_2^i.remotes$ and $(sg_2^e, \textsc{Eg}) \in sg_1^e.remotes$. 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. Note that allowed protocols and port ranges are omitted in the specification of *remotes*, because these are all the same values. The actual values for protocol and port ranges depend on the particular CNI networking plugin used.

3 Motivation

In this section we exemplify an attack path that allows an external adversary to compromise containers, nodes, and potentially the entire K8s cluster. We assume that the following vulnerabilities and misconfigurations may be present in the infrastructure under attack: 1. Default security groups settings are used in a K8s cluster allowing all of its worker nodes to communicate with each other to ensure a proper functioning cluster. 2. A pod can be accessed by the attacker via the pod network, e.g., it contains a remotely exploitable software vulnerability that allows for a remote code execution attack [6, 16]. 3. The container/VM stack in cluster nodes is vulnerable to container escapes via zero-day exploits or known CVEs [4, 5], or a too permissive Pod access control configuration to the underlying operating system [9].

Following the red attack path shown in Fig. 1 an adversary first gains code execution by exploiting a vulnerability in *PodA* (step 1), then utilizes misconfiguration- or software vulnerabilities in a pod or node to escape from that

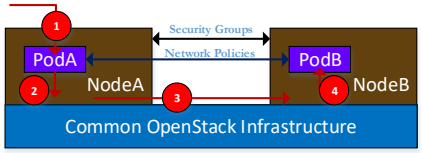


Fig. 1: Attack path (red arrows). Assume that the attacker has gained access to PodA (Step 1). If network policies prevent PodA ▸ PodB, but attached security groups allow NodeA ▸ NodeB, the attacker can escape PodA to NodeA (Step 2), gain access to NodeB (Step 3), and to PodB (Step 4).

pod to the node (step 2). Once a node is compromised, the attacker can use extracted information such as cryptographic keys, tokens, configuration information, along with the default permissive security group configuration to gain access to other nodes (step 3) and potentially compromise the pods on these nodes (step 4). This demonstrates how default security group settings in a K8s cluster can lead to an increased attack surface, potentially resulting in a compromised K8s cluster. The attack scenario described in step 2 can never be completely ruled out by deny-all cluster and pod access controls towards the underlying operating system because many containers require some capabilities, privileges or system calls in order to perform their intended functionality [10,26].

The mostly manual process of configuring default security groups would therefore benefit from being replaced with an automated approach that enforces base-truth network policies on security group configurations. This is a non-trivial problem, given the high dynamicity in cluster and cloud environments where the placement of pods across nodes cannot be predicted in advance. A possible workaround to this problem is to specify placement constraints that limit specific pods to specific subgroup of nodes and then manually add to this subgroup a least privilege security group in line with the network policies for these pods. However, this solution defeats the purpose of flexibility, performance and availability offered by containerization.

4 Design and Methodology

This section presents the main idea and the methodology underlying GrassHopper, elaborating the process of automatic generation and configuration of security groups from network policies.

4.1 Goals

The main goal of GrassHopper is to ensure the consistency between communication rules established by security groups in OpenStack and the network policies of Kubernetes. Moreover, the security group configuration should follow the least privilege principle that no unnecessary communication may be allowed. Formally, if $node : Pod \rightharpoonup Node$ represents the mapping of running pods ($pods \subseteq Pod$) to their compute nodes, we aim for the following properties:

1. *Correctness*: If two running pods are allowed to communicate by a K8s network policy, communication must also be allowed between their nodes.

$$\forall p_1, p_2 \in pods.\ p_1 \blacktriangleright p_2 \implies node(p_1) \blacktriangleright node(p_2)$$

2. *Least privilege*: Communication between nodes is only allowed if pods are running on them and a K8s network policy allows them to communicate.

$$\forall N_1, N_2 \in Node.\ N_1 \blacktriangleright N_2 \implies \exists p_1, p_2.\ node(p_1){=}N_1 \wedge node(p_2){=}N_2 \wedge p_1 \blacktriangleright p_2$$

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

Note that GrassHopper considers the K8s network policies to be 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, GrassHopper first verifies new network policies for misconfigurations by integrating the network policy checker Kano [14]. Kano supports detecting a wide range of bad misconfigurations, specifically: 1. *policy conflict*: two or more policies conflict when they select and allow the same pods but with inconsistent actions; 2. *policy redundancy*: for any two policies, one is considered redundant when either, both select and allow the same traffic connections to the same containers, or connections allowed by one policy are completely covered under the connections allowed by the other; 3. *broad access permissions*: overly permissive network policies specify connections from or to a wide range of pod labels, IP addresses or ports, potentially violating the least privilege principle.

4.2 Labels-Security Group Hashmap

The core component of GrassHopper is the Labels-Security Group hashmap defined as $Map : \mathcal{P}(Label) \rightharpoonup Mapentry$, where *Mapentry* is a record containing a security group name $sgname \in SGname$ for a particular label set $L \in \mathcal{P}(Label)$ and a set of policies $POL \in \mathcal{P}(Policy)$ of which the select label set equals L, the $nodes \subseteq Node$ hosting pods that match label set L, and a flag $remoteSG \in \{\textbf{true}, \textbf{false}\}$ signifying whether *sgname* is present in the *remotes* of other security groups. $Map(L)$ thus represents a security group, to which *remotes* are added from different network policies that all select the same label set L. A *Mapentry* record is only added when pods that match L are actually running.

Figure 2 illustrates how GrassHopper leverages the different sections of a network policy to create new security groups. K8s network policy

$$pol_{test} = \langle\!|\ name := \text{'test-policy'};\ sel := \{\text{'keysA:valuesA'}\};\ allow := \{(\{\text{'keysB:valuesB'}\}, \textsc{Ing}), (\{\text{'keysC:valuesC'}\}, \textsc{Eg})\}\ |\!\rangle$$

allows ingress from a pod p_B on NodeB to pod p_A on NodeA and egress from p_A to a pod p_C on NodeC. In response, security groups SG-A, SG-B and SG-C are created and SG-A is augmented with rules targeting SG-B and SG-C as

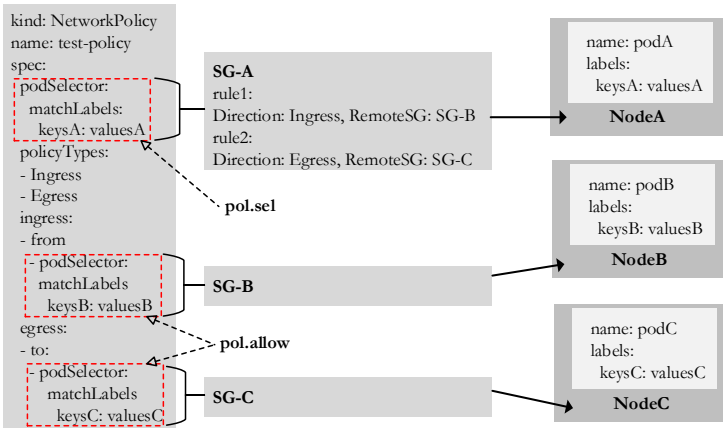


Fig. 2: K8s network policy used to create OpenStack security groups

remote security groups. The corresponding hashmap entry for the select label, *Map*('keysA:valuesA'), is

$$\langle\!| sgname := \text{'SG-A'}, POL := \{pol_{test}\}, nodes := \{\text{'NodeA'}\} |\!\rangle .$$

While label sets in K8s are usually a set of key-value pairs, here we treat them as single string with alphabetically sorted key-value pairs for simplicity.

In our notation we represent security group SG-A as

$$\langle\!| name := \text{'SG-A'}; remotes := \{(\text{'SG-B'}, \textsc{Ing}), (\text{'SG-C'}, \textsc{Eg})\} |\!\rangle .$$

Security group rules in cloud providers specify remote connection targets (sources or destinations) using either a block of IP addresses (CIDR) or a remote security group. GrassHopper uses remote security groups instead of IP blocks to specify remote targets. Thus, GrassHopper only permits connections between nodes to which a security group *sg* is attached and nodes to which security groups specified in *sg.remotes* are attached, in case the group rules allow it.

4.3 Algorithm

To support least privilege network permissions, GrassHopper assumes that there is a permanent *DenyAll* K8s network policy in place that restricts any communication between pods. GrassHopper further assumes that all default security groups are removed from the cluster, except the ones allowing communication between the master and the worker nodes.

The deployer than adds new policies which allow certain communications. GrassHopper reacts to such changes in network policies, and to the deployment or retirement of pods, according to Algorithm 1. Here, for brevity, we omit the field names when representing record values, using a tuple-style notation instead.

At each event, the algorithm traverses the various sections of the resources (pod or network policy) for information to use in matching within the input data

Algorithm 1 Security Group Configuration Algorithm

INPUT: Pods, Nodes, Network Policies
OUTPUT: Security Group Configurations

$pods \subseteq Pod$ ▷ currently running pods
$node : Pod \rightharpoonup Node$ ▷ map pod to host node
$pols \subseteq Policy$ ▷ currently active K8s network policies
$sgd \subseteq SecGroup$ ▷ current security group definitions
$SG : Node \to \mathcal{P}(SecGroup)$ ▷ security groups attached to node
$Map : \mathcal{P}(Label) \to Mapentry$ ▷ labels-security groups hashmap
Wait for event:
if policy $pol_{new} = \langle pn; S; A \rangle$ is added to $pols$ **then**
  for all $(L, type) \in A$ **do** ▷ create remote security groups according to A
    if $Map(L) = \bot$ **then** ▷ no policy selects or allows L (yet)
      Create sg in sgd with $sg.name := SG::L$, $sg.remotes := \varnothing$
      Set $Map(L) := \langle SG::L; \varnothing; \varnothing; \textbf{true} \rangle$ ▷ new remote security group
      for all pods $p \in pods$ such that $L \subseteq p.L$ **do** ▷ p targeted by A
        Add sg to $SG(node(p))$ ▷ attach sg to node hosting p
        Add $node(p)$ to $Map(L).nodes$ ▷ record host nodes
    else Set $Map(L).remoteSG := \textbf{true}$
  if $Map(S) = \bot$ **then** ▷ no policy selects or allows S (yet)
    Create sg in sgd with $sg.name = SG::S$, $sg.remotes := \varnothing$
    Set $Map(S) := \langle SG::S; \varnothing; \varnothing; \textbf{false} \rangle$
    for all pods $p \in pods$ such that $S \subseteq p.L$ **do** ▷ p targetted by S
      Add sg to $SG(node(p))$ ▷ attach sg to node hosting p
      Add $node(p)$ to $Map(S).nodes$ ▷ record host nodes
  Select $sg \in SG$ such that $sg.name = SG::S$ ▷ get security group for S
  Add $SGremotes(pol_{new})$ to $sg.remotes$ ▷ add non-intersecting sg rules according to A
  Add pol_{new} to $Map(S).POL$ ▷ add new policy to $Mapentry$ for S
if policy $pol_{del} = \langle pn; S; A \rangle$ is deleted from $pols$ **then**
  Remove pol_{del} from $Map(S).POL$ and from $pols$ ▷ delete policy
  $R = (Map(S).POL = \varnothing \wedge Map(S).remoteSG = \textbf{false})\ ?\ \{S\} : \varnothing$
  if $R = \varnothing$ **then** ▷ if $Map(S)$ not to be removed, only remove rules
    Remove $SGremotes(pol_{del})$ from $sg.remotes$ where $sg.name = SG::S$
  Set $R := R \cup \{L \mid (L, \_) \in A \wedge \{pol \in pols \mid (L, \_) \in pol.allow\} = \varnothing\}$
  for all L in R **do** ▷ modify all map entries exclusively targeted by pol_{del}
    if $Map(L).POL = \varnothing$ **then** ▷ remove security group if no other policy selects L
      Let sg be the security group with $sg.name = SG::L$
      Remove sg from sgd and from $SG(N)$ for all $N \in Map(L).nodes$
      Remove entry for L from Map
    else Set $Map(L).remoteSG := \textbf{false}$ ▷ unset $remoteSG$ flag
if pod p_{new} is added to $pods$ on $node(p) = N$ **then**
  for all M in dom(Map) **do** ▷ attach matching sg to N if not yet attached
    if $sg \notin SG(N)$ with $sg.name = Map(M).sgname$ **then**
      Add sg to $SG(N)$
      Add N to $Map(M).nodes$
if pod p_{last} on $node(p) = N$ is deleted from $pods$ **then**
  for all M in dom(Map) **do** ▷ only detach if p_{last} is the only pod on N matching M
    if $M \subseteq p_{last}.L$ and $\forall p' \neq p_{last}.\ node(p')=N \Rightarrow M \not\subseteq p'.L$ **then**
      Remove sg with $sg.name = Map(M).sgname$ from $SG(N)$
      Remove N from $Map(M).nodes$
Wait for next event.

and lookups in the hashmap. Since the *DenyAll* rule is in place we only need to consider rules that allow additional communication.

When a new network policy is added, we consider the select and allow sections separately. For every label set L_i specified in the allow section we require a single *remote security group* that is named as $SG{::}L_i$ (:: is string concatenation with implicit conversion) and we attach it to all nodes with matching pods. Similarly, for the select label set S, we only create a single security group, named $SG{::}S$ if it does not exist yet in *Map*. We then add rules to $SG{::}S$ in correspondence with the rules of the network policy. We ensure that only rules are added that have not yet been added by another network policy where the select label set matches S. The corresponding function $SGremotes : Policy \to \mathcal{P}(SGname \times \{\text{ING}, \text{EG}\})$ selects non-intersecting rules as follows:

$$SGremotes(pol) = \{(SG{::}L, type) \mid (L, type) \in pol.allow \setminus \bigcup_{p \in Map(pol.sel).POL} p.allow\}$$

When a network policy is removed, we essentially undo this addition of rules. We only delete rules that have not been added by another network policy. We can also detach and delete a security group *sg* if there are no more rules in *sg* and *sg* does not act as a remote security group of other security groups. Finally, when deleting *sg*, we also need to consider deleting its remote security groups unless these act as remote security groups in other security groups or these have rules of other network policies.

When a new pod is added to the cluster on node N, we need to attach the security groups to N which match with the label set of that pod but only if the security groups are not yet attached to N. Conversely, when a pod is deleted, we only need to detach a matching security group if that pod was the last on its node matching the corresponding label set of that security group.

This summarizes Algorithm 1. Note that migrating a pod to a different node is considered a deletion followed by an addition of the same pod. Moreover, in the above algorithm we assume that all newly added network policies are unique and required, since GrassHopper filters out redundant or conflicting rules before processing them. However it keeps track of them and processes them if they become valid at a later point after the deletion of an interfering policy.

Operation \ No.of SGs	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

Table 1: Security group (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.

Informal complexity analysis. Managing security groups involves execution of several time-intensive operations that may differ for different cloud providers. Table 1 shows the results for a closed lab OpenStack private cloud that has

been used for evaluating GrassHopper. These results highlight the time intensity of creating and deleting a single security group, compared to the lower times needed for adding/removing rules to a security group and (de-)attaching a security group. 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 [3] when adding rules to an already attached security group. As a result, no impact on data plane application performance is expected when adding or removing rules to already attached security groups. The measurements in Table 1 and the evaluation in Section 6.3 confirm this.

GrassHopper, being an event driven tool, creates a small number of security groups per event and thus does not enjoy the benefits of creating them in bulk. This demonstrates the need to reduce the number of security group operations, especially the most time-intensive ones.

The question arises if the algorithm of GrassHopper remains the most optimal versions when other security group operations are more time-intensive than a create operation. For example, consider that in another cloud platform, the addition of rules to security groups would be the most time intensive one. In this case one might consider an alternative algorithm that creates a separate security group per network policy, adds rules to the security group for only this network policy, and then attach it to nodes with matching running pods. Although not shown in Algoritm 1 due to space limitations, these 3 operations are also executed by GrassHopper in that particular order. However, if there would already exist a policy for the same select label set, the 3 operations are also needed in the alternative algorithm, while in GrassHopper's algorithm only adding of rules to an already existing security group would be needed, which does not have an impact on application performance as argued above. Thus, for each unique select label set, there is at most one security group, whereas in the alternative proposal one would need to manage as many security groups as there exist network policies that select that label set. This also means that during creation/deletion of a pod to a node, GrassHopper always requires an equal or less amount of attach/detach operations as compared with the alternative algorithm.

5 Implementation

Based on the aforementioned design, we introduce our implementation. GrassHopper [8] is implemented as a python library invoked by events managed by the K8s API server. If an event pertains to the creation, removal, or update of pods or network policies, GrassHopper is invoked to correspondingly update the security groups in consistence with the event.

Figure 3 shows an overview of the GrassHopper implementation, realizing the design described in the previous section. The central data structure is the HashMap module, which manages per unique label set a *Mapentry* record as defined in Section 4.2. To store different network policies in such record, we use a second nested hash map that maps policy names to policies. The key string used in the HashMap is an alphabetically sorted string of concatenated `key:value;`

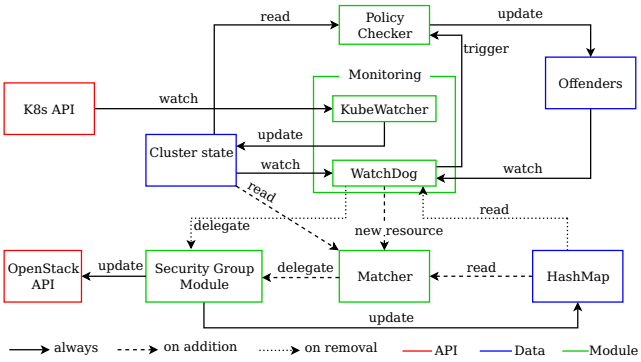


Fig. 3: GrassHopper design overview

pairs. This allows using an actual hashmap data structure to implement a look up time complexity of $O(1)$.

There are mainly four steps involved in the operation of GrassHopper: 1. The `KubeWatcher` module is watching the K8s API server using the `kubectl watch` command in order to detect changes in the K8s cluster w.r.t. network policies and deployment of pods on nodes. It records these changes as input data in in the `Cluster state` data structure. 2. The `WatchDog` module watches the `Cluster state` for changes. In the event that a created resource is a network policy, it first invokes the PolicyChecker module before the Matcher module. This is to verify the consistency of the new policy with already existing policies to avoid policy redundancy, conflicts, and overly permissive policies. It records offending policies in the `Offender` data structure. The Watchdog module also watches this `Offender` data structure for policies that become relevant after deletion of other policies. 3. If a verified resource (pod/policy) is added or removed, the `WatchDog` module invokes the `Matcher` module that reads the `HashMap` and the `Cluster state` data structures and then executes the GrassHopper algorithm. 4. Any changes to HashMap entries are delegated to the `Security Group` module. This module performs via the OpenStack interface all actions on the security groups. A security group is created only if there are matching pods running in the cluster. As such, no unnecessary security groups are created.

GrassHopper starts attaching security groups to nodes at the moment when the `KubeWatcher` module detects on which node a pod is placed by the K8s scheduler. In this way, the configuration of security groups can run in parallel with the process of starting the pods. As a consequence, as long as the time to configure a security group does not exceed the time to start a pod, the effect of GrassHopper on the application deployment and auto-scaling of pods is completely curtailed. This starting time can be considerable, especially in the case of cold starts and aggressive auto-scaling [7]. As shown in Table 2, this starting time can be up to 4 seconds in our experiments. However, research on reducing cold starts may continuously improve the state-of-the-art. Therefore, the design methodology of GrassHopper has been based on reducing the number of time-costly security group operations as much as possible.

There is a specific performance optimization included in the current implementation that ensures that no unnecessary Openstack rules are added to security groups. A *Mapentry* record also stores which other *Mapentry* records are the remote security groups of this record. This information is used to defer addition of remotes to the security group of this record until these remotes in turn refer to this record. For example, in Figure 6, network policies 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. At the time that NP_1 has been created (cfr. Table 3), NP_3 does not exist yet. Therefore, only a security group for NP_1 is created and attached, but the rules of NP_1 are not yet added. When NP_3 is added, the remotes computed from NP_1 and NP_2 will be added to the already created security groups. Similarly, when one of the policies will be deleted, the remotes associated with these policies will be removed as well.

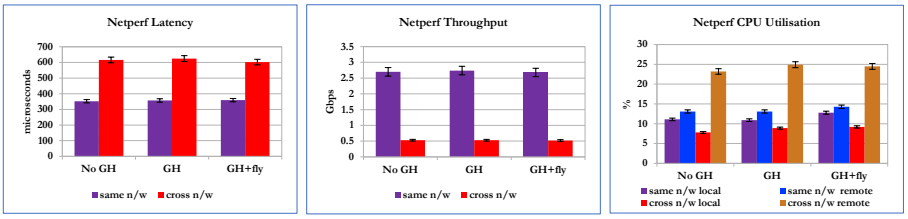
6 Experimentation and Evaluation

In this section we present the results of the performance evaluation of GrassHopper and we discuss limitations and associated future work. We evaluated GrassHopper for two synthetic applications: netperf [2] and an adaptive Software-as-a-Service (SaaS) application [24]. We configured Calico eBPF to use VXLAN only for inter-subnet communication, otherwise pure layer 3 networking was used.

6.1 Experiment Settings

The testbed used for running all the experiments is an isolated part of a private OpenStack cloud, version Liberty. 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 [17]. The OpenStack cloud consists of a master-worker 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 and is configured with `ipset` enabled. The droplets have 16 CPU cores of which 2 are reserved for the operation of the Openstack cloud. The K8s cluster used in the evaluation consists of one master node and three worker nodes spread across two different networks. The cluster was deployed using Kubeadm, running K8s version 1.23.1. 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.

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



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

Fig. 4: Netperf evaluation

mean values are within +/- 2.5% of the real mean values. In this evaluation, we connected to the pods directly using their IP addresses.

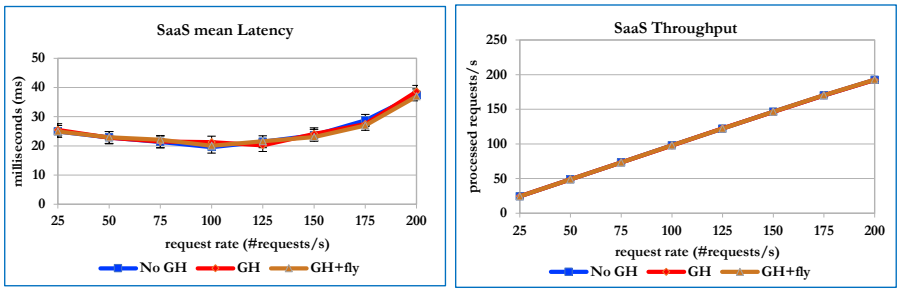
SaaS application evaluation. The SaaS application used for GrassHopper evaluation is written in C++ and is based on the COMITRE approach [20]. It provides a REST API (SaaS API) to which users can send requests. With every user belonging to a tenant, each request has a tenantId field so that the application 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 [24]. For this evaluation, CPU resource was stressed. The default auto-scaler in Kubernetes, Horizontal Pod Autoscaler (HPA) [1], was configured to keep average CPU usage of the service's Pods around 50% so that auto-scaling 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. We report the mean with the confidence interval computed at 95% confidence level.

6.2 Evaluation Results

We answer the following questions regarding GrassHopper's performance:

- Qn1. How does GrassHopper impact performance of container applications with respect to end-to-end latency, throughput, and resource utilization?
- Qn2. How much time does GrassHopper take to configure (or remove) least privilege security groups on addition (or removal) of a network policy or a container to (or from) the cluster?
- Qn3. What is the impact on the performance of container applications when network policies and containers are added on the fly while GrassHopper is running on the cluster?

With respect to Qn1, we compare Netperf and SaaS applications performance without GrassHopper (No GH) to that with GrassHopper (GH) running in the cluster. The results as observed in Fig.4 and Fig. 5 show that GrassHopper does not affect application performance. The variations observed in Fig.5a are due to autoscaling of pods. We also measured the cross n/w scenario where GH needed to add more rules for VXLAN traffic and still observed no overhead.



(a) Latency (b) Throughput

Fig. 5: SaaS evaluation

Table 2: GrassHopper time (s)

	no GH	GH-netperf	GH-SaaS
Time for pod ready	3 to 4	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 hosted on node 1) 0.084 (first scaled SaaS pod hosted on node 2) 0.075 (second scaled SaaS pod hosted on node 3)
Detach and remove from node		0.05	0.05

To answer Qn2, we evaluate time performance for GrassHopper configuration of security groups. We measure the time taken by GrassHopper to detect the addition of a new resource to the K8s API, get required resource information, match and lookup the resource, create a security group and/or add rules, and attach the security group to the pertinent node. As observed in Table 2, 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 SaaS pods is lower because no new security group is created when pods are replicated. Rather, the security group stored in the hashmap corresponding to the first pod is attached to the new nodes hosting the replicated pods. Table 3 further shows the efficiency of adding rules to an already created SG rather than creating and attaching a new SG when a policy (NP4, see Fig.6) with similar select labels is added. The results in both tables indicate that GrassHopper configuration time was much lower than the 3 to 4 seconds that application pods took to start up and attain the `READY` condition. Considering that GrassHopper operations run in parallel with pod start-up, this shows that a security group will be configured before the pod is ready, thereby steering clear of affecting application connection time. Additionally, we measured the time taken by GrassHopper to look up a security group in the hashmap and detach it from the pertinent node when a resource is deleted from the cluster. This time (cf. Table 2) is around 0.05 seconds.

To answer Qn3, we repeated the evaluations for Qn1 while periodically adding new pods with their corresponding network policies to the cluster. A total of 40 pods and network policies was added during each run of the experiment, with a new security group created and attached to the pertinent node for each pod policy pair. The results of this experiment are indicated under the 'GH+fly'

	no GH	GH-netperf
Time for pod ready	3 to 4	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 to NP4 to SG for NP3 Adding rules to SG for NP2		0.249 (addition of NP4)

Table 3: GrassHopper Time (s)

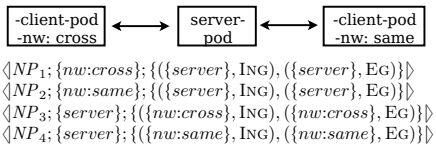


Fig. 6: Policies and Pods setup

labeled results of Fig. 4 and Fig. 5. 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 GrassHopper can configure security groups to multiple applications without affecting their performance.

Scalability. During autoscaling of the SaaS application, pods were scheduled to random nodes in the cluster. Regardless of the number of nodes in the cluster, GrassHopper enforced security groups only to the nodes where replicated SaaS pods were scheduled to be hosted, an indication of GrassHopper scalability with respect to the number of nodes.

6.3 Limitations and Future Work

With enough applications and network policies and a high cluster load, security group configurations in GrassHopper may converge to the default, permissive security group configuration. This scenario does not occur for any fully loaded cluster. For example, with the highest load exposed to the above SaaS application, there are only two connections allowed between worker nodes, namely those from the node with the client pod to those nodes with only SaaS pods. Indeed, connection requests between nodes with only SaaS pods are always refused.

GrassHopper implementation assumes either the same admin for both K8s cluster and OpenStack environment, or mutual trust in the event of different admins. This may however not be the case in the production environment. Hence, as future work, we are looking at extending GrassHopper with support for different mutually distrusting admins.

While the general approach of GrassHopper is platform agnostic, the current implementation is still limited to OpenStack. As future work, we will address this limitation through a more modular design with concrete provisions for supporting other platforms.

7 Related Work

Container networking solutions. AWS' Elastic Kubernetes Service (EKS) offer security groups for pods [22] to govern intra-cluster communication as well as egress communication with AWS services. However, this solution neither preserves consistency with K8s-native network policies nor is it generally portable to other Kubernetes vendors. CNI plugins such as Calico Enterprise [23] integrate network policies and security groups but only concerning egress cluster

traffic. Bastion [18] extends K8s support for network policies towards network-privileged pods that have access to the underlying VM network of a K8s cluster. AutoArmor [13] generates inter-service access policies for Istio and K8s by static code analysis. However, Bastion and AutoArmor do not preserve consistency with security groups. In opposition, GrassHopper enforces a defense-in-depth mechanism so that when an attacker escapes from a pod to a VM, the attacker is still limited by the security group configurations derived from the K8s network policies for the pods actually running on that Node. GrassHopper is also fully portable across Kubernetes vendors.

Verification of container-based network policies. As already stated in section 2, GrassHopper uses the policy checker Kano [14] to verify K8s network policies against inconsistencies, duplication and violation of the principle of least privilege. Although Kano does not itself preserve consistency with security groups, Kano can be extended with such facility. Still, such an extension only detects inconsistency between network policies and security groups, leaving the resolution of the inconsistency to a planning component.

Multi-level consistency management. NFVGuard [21] verifies the security of multi-layer Network Functions Virtualization across an OpenStack platform. TenantGuard [25] offers scalable support for verification of reachability across security groups and virtual routers on a shared Openstack platform. However, these works employ verification as the main mechanism which again requires a planning component to resolve inconsistencies. In the context of ultra-low-latency and ultra-high-reliability applications, such an approach would take too long when user-specific services must be scaled up due to increased workload. As opposed to these works, GrassHopper uses an enforcement and verification approach which does not have an impact on the time to deploy pods.

8 Conclusion

Automated consistency of VM-level and container-level network isolation is important for achieving in-depth defenses for ultra-reliable applications. These benefits are however yet to be reaped because VM-level security group configurations must be adapted in function of actual container placements on nodes. Especially, in the presence of elastic scaling due to fluctuating workloads, security groups must be added to VMs before newly created containers on these VMs are ready to serve traffic. Thus, we have proposed GrassHopper, a novel cross-layer enforcement and verification approach to generate VM-level security groups from runtime verified container-level network policies. We have implemented this solution and integrated it into a K8s cluster running on top of OpenStack. Evaluation through experiments using two container-based applications have shown no overhead on container application performance. These results confirm the efficiency and applicability of GrassHopper for ultra-reliable, and even ultra-low-latency applications.

Acknowledgments 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. Horizontal pod autoscaling. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/, [accessed 2022-05-24]
2. Netperf. https://hewlettpackard.github.io/netperf/, [accessed 2022-04-11]
3. Openstack configuration reference. https://docs.openstack.org/liberty/config-reference/content/networking-options-securitygroups.html (2016), [accessed 2022-05-27]
4. 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]
5. Container breakout vulnerabilities. https://www.container-security.site/attackers/container_breakout_vulnerabilities.html (2022), [accessed 2022-05-12]
6. Baitello, E.: Attacking kubernetes clusters using the kubelet api. https://faun.pub/attacking-kubernetes-clusters-using-the-kubelet-api-abafc36126ca#c0b6 (2021), [accessed 2022-03-29]
7. Beni, E.H., Truyen, E., Lagaisse, B., Joosen, W., Dieltjens, J.: Reducing cold starts during elastic scaling of containers in kubernetes. In: Hung, C., Hong, J., Bechini, A., Song, E. (eds.) SAC '21: The 36th ACM/SIGAPP Symposium on Applied Computing, Virtual Event, Republic of Korea, March 22-26, 2021. pp. 60–68. ACM (2021). https://doi.org/10.1145/3412841.3441887, https://doi.org/10.1145/3412841.3441887
8. Budigiri, G.: k8-scalar/grasshopper. https://github.com/k8-scalar/grasshopper/ (2022), [accessed 2022-05-29]
9. Fox, B.: Unrestricted hostpath. https://github.com/BishopFox/badPods/tree/main/manifests/hostpath (2021), [accessed 2022-03-29]
10. Ghavamnia, S., Palit, T., Benameur, A., Polychronakis, M.: 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)
11. Harrison, R.: 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]
12. Kaur, R., Hils, A., Watts, J.: Technology insight for network security policy management. https://www.gartner.com/en/documents/3902564 (2019), [accessed 2022-03-22]
13. Li, X., Chen, Y., Lin, Z., Wang, X., Chen, J.H.: Automatic policy generation for Inter-Service access control of microservices. In: 30th USENIX Security Symposium (USENIX Security 21). USENIX Association (2021), https://www.usenix.org/conference/usenixsecurity21/presentation/li-xing
14. Li, Y., Jia, C., Hu, X., Li, J.: Kano: Efficient container network policy verification. In: 2020 IEEE Symposium on High-Performance Interconnects (HOTI). pp. 63–70. IEEE (2020)

15. MacDonald, N.: Innovation insight for cloud security posture management. https://www.gartner.com/en/documents/3899373 (2019), [accessed 2022-03-22]
16. Manning, M.: 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]
17. Murugesan, S., Bojanova, I.: Encyclopedia of cloud computing. John Wiley & Sons (2016)
18. Nam, J., Lee, S., Seo, H., Porras, P., Yegneswaran, V., Shin, S.: BASTION: A security enforcement network stack for container networks. In: Gavrilovska, A., Zadok, E. (eds.) 2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020. pp. 81–95. USENIX Association (2020), https://www.usenix.org/conference/atc20/presentation/nam
19. 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]
20. Ochei, L.C., Bass, J.M., Petrovski, A.: Evaluating degrees of multitenancy isolation: A case study of cloud-hosted gsd tools. In: 2015 International Conference on Cloud and Autonomic Computing. pp. 101–112. IEEE (2015)
21. Oqaily, A., T, S.L., Jarraya, Y., Majumdar, S., Zhang, M., Pourzandi, M., Wang, L., Debbabi, M.: Nfvguard: Verifying the security of multilevel network functions virtualization (nfv) stack. In: 2020 IEEE International Conference on Cloud Computing Technology and Science (CloudCom) (2020). https://doi.org/10.1109/CloudCom49646.2020.00003
22. Stefaniak, M., Balaji, S.S.: Introducing security groups for pods. https://aws.amazon.com/blogs/containers/introducing-security-groups-for-pods (2020), [accessed 2022-05-22]
23. Tigera: Configure calico enterprise aws security groups integration. https://docs.tigera.io/security/aws-integration/aws-security-group-integration (2022), [accessed 2022-04-22]
24. Truyen, E., Jacobs, A., Verreydt, S., Beni, E.H., Lagaisse, B., Joosen, W.: Feasibility of container orchestration for adaptive performance isolation in multi-tenant saas applications. In: Proceedings of the 35th Annual ACM Symposium on Applied Computing. pp. 162–169 (2020)
25. Wang, Y., Madi, T., Majumdar, S., Jarraya, Y., Alimohammadifar, A., Pourzandi, M., Wang, L., Debbabi, M.: 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), https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/tenantguard-scalable-runtime-verification-cloud-wide-vm-level-network-isolation/
26. Zhu, H., Gehrmann, C.: Kub-sec, an automatic kubernetes cluster apparmor profile generation engine. In: 2022 14th International Conference on COMmunication Systems NETworkS (COMSNETS). pp. 129–137 (2022). https://doi.org/10.1109/COMSNETS53615.2022.9668504