POLITECNICO DI TORINO
Repository ISTITUZIONALE

An intent-based solution for network isolation in Kubernetes

(Article begins on next page)

14 October 2024

Security automation for multi-cluster orchestration in Kubernetes

Daniele Bringhenti, Riccardo Sisto, Fulvio Valenza

Dip. Automatica e Informatica, Politecnico di Torino, Torino, Italy, Emails: {first.last}@polito.it

Abstract—In the latest years, multi-domain Kubernetes architectures composed of multiple clusters have been getting more frequent, so as to provide higher workload isolation, resource availability flexibility and scalability for application deployment. However, manually configuring their security may lead to inconsistencies among policies defined in different clusters, or it may require knowledge that the administrator of each domain cannot have. Therefore, this paper proposes an automatic approach for the automatic generation of the network security policies to be deployed in each cluster of a multi-domain Kubernetes deployment. The objectives of this approach are to reduce of configuration errors that human administrators commonly make, and to create transparent cross-cluster communications. This approach has been implemented as a framework named Multi-Cluster Orchestrator, which has been validated in realistic use cases to assess its benefits to Kubernetes orchestration.

Index Terms—security automation, cloud orchestration, Kubernetes

I. INTRODUCTION

In recent years, containerization has been emerging as a dominant cloud technology in the field of software development. The advent of this paradigm has changed how users perceive the development, deployment, and maintenance of software applications [1]. Indeed, the success of container technologies has determined a general shift from monolithic applications to cloud-native applications, composed of loosely-coupled microservices which can be deployed and managed independently from each other. In a microservice-based architecture, orchestrators are the element in charge of automating operations related to cloud service management, e.g., microservice deployment, resource optimization, and security managament.

Among the different orchestrators that have been proposed, Kubernetes is nowadays considered the de-facto tool for container orchestration, and it is widely adopted by organizations such as Adidas, Nokia, Spotify, and the U.S. Department of Defense [2]. Since its first release, the design of the orchestration provided by Kubernetes mainly focused on the management of a single group of computing nodes, named cluster. The security management of the pods (i.e., single containers or a group of them) deployed by Kubernetes in a cluster is based on the definition of application-centric constructs, named network policies. These expressions specify how a pod is allowed to communicate with other network entities [3].

Even if a single Kubernetes cluster is extremely flexible by itself, as a multi-tenant Kubernetes environment may be created inside it with various techniques, over the years multi-cluster Kubernetes architectures have started to be more frequently created, so as to take advantage of the best features this technology can offer. Some benefits are improved workload isolation, resource availability flexibility and scalability. Multi-cluster Kubernetes architectures are therefore common when the clusters composing them belong to different companies, whose services must communicate with each other. These architectures are also known as multi-domain architectures, where a domain is a set of clusters managed by the same company.

However, if the complexity of setting up network policies for a single Kubernetes cluster is already high, it is even increased for a multi-cluster architecture [4]. On the one hand, as a specific security configuration must be set up in each cluster, inconsistencies may be introduced in the policy definition, and they may lead to service disruptions. On the other hand, inter-domain communication would require that the person in charge of a domain knows IP addresses, routing rules, and DNS settings related to services in other clusters. Otherwise, such sharing of knowledge is not always feasible.

In light of these open problems, this paper proposes a novel approach for automating the security configuration of a multi-cluster and multi-domain Kubernetes environment. This approach is based on the creation of a top-level entity, named Multi-Cluster Orchestrator, which refines user-specified security requirements into the concrete configuration, comprised of Kubernetes network policies, to be deployed on each cluster of the involved domains. The presence of this entity contributes to the reduction of configuration errors that human administrators commonly make, and to the creation of transparent cross-cluster communications. A Java-based implementation of this module has been implemented and validated in use cases, showcasing the benefits of this approach in such environment.

The remainder of this paper is structured as follows. Section II discusses related work. Section III presents the approach that we propose for automatic security orchestration in multi-cluster environments managed by Kubernetes. Section IV describes the implementation of the Multi-Cluster Orchestrator and discusses its validation. Finally, Section V draws conclusions and outlines future work.

II. RELATED WORK

Automation has been investigated in literature for improving security configuration in computer networks, as it con-

tributes to avoiding human errors and sub-optimizations. It has been also leveraged to enable "Security as a Service" in security orchestration, e.g., in the European Union projects projects INSPIRE-5Gplus [5], MonB5G [6] and FLUIDOS [7]. A paradigm that is commonly pursued within automation is policy-based management. The main idea behind this paradigm is that human users should only specify network security requirements with sentences expressed with a user-friendly high-level language, as automatic tools are later in charge of refining those expressions into concrete security configurations. In the wake of this idea, automatic approaches have been presented in literature for the automatic configuration of different virtual function types: packet filtering firewalls [8]–[12], channel protection systems such as VPN gateways [13], [14], IoT devices [15], [16], SDN switches [17], [18]. However, all these approaches have been designed to work either in physical networks, or in virtual networks based on the Network Functions Virtualization and Software-Defined Networks technologies. Therefore, they are not suitable for the management of cloud-based networks, especially in the context of multi-cluster Kubernetes architectures.

At the same time, even if some solutions exist for multi-cluster Kubernetes network interconnection, they are limited in managing network security configuration in a user-friendly and efficient way. Three main classes of solutions are currently used for network management of multi-cluster architectures: solutions related to a specific Container Network Interface (CNI) such as Ciulium Cluster Mesh [19], CNI-agnostic technologies such as [20] and [21], and service mesh technologies such as Istio [22] and Linkerd [23]. However, all these solutions require coordination from all domain managers of the clusters participating in the mesh, but a mutual share of knowledge about their domains is not always feasible. In fact, they do not implement a cross-cluster control plane and they lack a single management point for the whole architecture.

The top-level entity proposed in this paper, i.e., the Multi-Cluster Orchestrator, has the objective to overcome the limitations of the state of the art of both network security automation for cloud architectures and security management for multi-cluster architectures. On the one hand, it aims to leverage a policy-based management approach, while customizing it for the peculiar features of cloud environments. On the other hand, it represents a single management point that multiple domain managers can interact with, so as to express security requirements related to cross-cluster communications.

III. THE PROPOSED APPROACH

The approach proposed in this paper for the automation of security management in a multi-cluster environment orchestrated by Kubernetes envisions the presence of a top-level entity, named Multi-Cluster Orchestrator. This entity is in charge of automatically computing the network policies for each cluster, by refining high-level security requirements defined by a network manager. These requirements express how clusters are interconnected and how the services deployed in those clusters must be protected from all unwanted requests
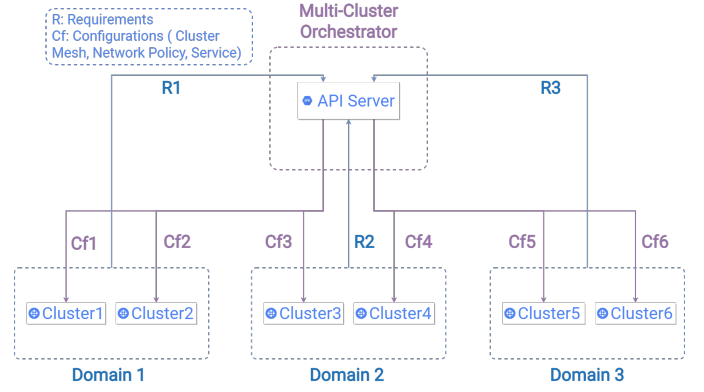

Fig. 1. Multi-Cluster Orchestrator interactions with other entities

for incoming and outgoing service connections. The Multi-Cluster Orchestrator can speed up operations that would have to be performed manually otherwise, and at the same time it prevents human errors that commonly occur during the configuration operations. Furthermore, it allows communications to service in clusters where some pieces of knowledge are missing, e.g., the position of a service in terms of cluster or namespace may be unknown.

In our vision, the clusters whose security is managed by the Multi-Cluster Orchestrator can be grouped into entities named domains. We define *domain* a group of clusters belonging to the same company and under the control of a single network manager, called domain manager. Grouping clusters into domains is helpful in differentiating security requirements expressed for intra-domain communications and those expressed for inter-domain service communications. Indeed, these two classes of requirements have different objectives. On the one hand, requirements expressed for intra-domain communications are useful to manage clusters belonging to different sections of the company (e.g., to express that the cluster of the testing department must be able to communicate with the cluster of the development department, but not with the cluster of the human resources section). In this way, the security of these clusters can be automatically managed in a way that is independent of other domains that may belong to the environment handled by the Multi-Cluster Orchestrator. On the other hand, requirements expressed for inter-domain communications are also useful to allow each domain manager to specify how their clusters must communicate with clusters of other domains, even if the manager does not have full information about them (e.g., the domain manager may not know in which cluster of an external domain the service whose communication must be allowed is located). In its refinement process, the Multi-Cluster Orchestrator has full knowledge of all the managed domains, and it can thus create all required network policies, which could not be directly written by domain managers because of their partial knowledge.

Fig. 1 shows how the Multi-Cluster Orchestrator works with the other entities of the cloud environment for computing the security configuration of the Kubernetes domains. First,

it receives information about the domain structure and the network security requirements from each domain composing the environment. This information represents the input for the refinement process performed by the orchestrator and it will be detailed in Subsection III-A. After elaborating the received input, the Multi-Cluster Orchestrator automatically creates a Global Configuration, which is then further refined into Single Configurations composed of the network policies to be applied to each cluster. More details about the automatically produced output are provided in Subsection III-B.

A. Input: Domain Structure and Security Requirements

The Multi-Cluster Orchestrator requires two inputs from each domain composing the environment it manages: 1) a description of the domain structure; 2) a list of security requirements expressing the allowed communications.

1) Domain structure: The description of the domain structure must specify both the list of clusters belonging to the domain, and the list of all services that the domain exposes. The first element (i.e., the list of clusters) is required by the Multi-Cluster Orchestrator as the clusters are the entities to which the orchestrator will directly apply the output Single Configurations, computed at the end of its refinement process. The description of each cluster must at least include the cluster name and the API address, which represents the location where Kubernetes has its REST service API and can be used to create services and apply network policies. Instead, the second element (i.e., the list of exposed services, and the information about the clusters where they are located) is required by the Multi-Cluster Orchestrator for completing the partial information that it may receive from other domain managers, when they request connections to those specific services. In fact, some pieces of information may be unknown by domain managers of other domains like the namespace to which a service belongs, the used port and protocol, the cluster where the pods linked to the service are deployed, or the selector that links pods to that particular service. Feeding the Multi-Cluster Orchestrator with this information, provides transparently for starting secure communications between services.

2) Security requirements: Each security requirement that can be specified as input to the Multi-Cluster Orchestrator represents a request for security policies that must be applied to allow selected communications between services. By default, all other communications must be denied. In each requirement, the domain manager can specify three different target types for the communications:

- Services belonging to the same domain: it is possible to select one or more services that belong to the same domain of the service, but that may be deployed in different clusters.
- Services belonging to a different domain: it is possible to select services of different domains. In this case, the Multi-Cluster Orchestrator will add the missing information about that service in a transparent way.
- External IP addresses not belonging to a domain: it is possible to specify a range or a single IP address not
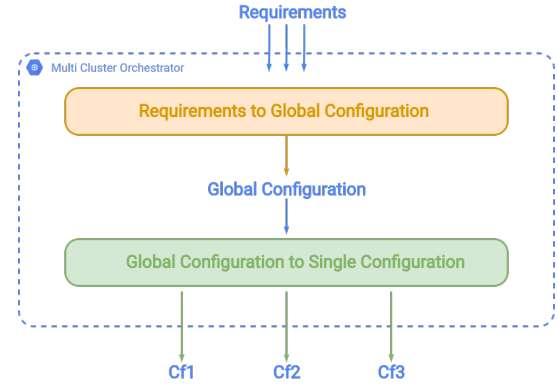

Fig. 2. Refinement process

belonging to any domain managed by the Multi-Cluster Orchestrator.

The communications can be allowed for incoming and outgoing traffic independently. For example, it is possible to request that a service named *service1* can start a communication with a service named *service2*, but *service2* cannot start a communication with the *service1*.

B. Output: Global Configuration and Single Configurations

The refinement process executed by the Multi-Cluster Orchestrator is composed of two steps, as shown in Fig. 2. First, the orchestrator automatically creates a Global Configuration, characterized by general information about all the communications that must be allowed. Then, it derives a Single Configuration for each cluster, refining the elements of the Global Configuration that pertain the specific cluster.

1) Global Configuration: After receiving the required input from all domains, the Multi-Cluster Orchestrator creates a Global Configuration that keeps tracking of the following aspects:

- the pairs of services that must be in communication;
- the pairs of clusters that must be linked together.

On the one hand, the list describing the pairs of services that must be in communication is useful for the optimization process of the security configuration for clusters inside the same domain. For example, if two services are deployed in the same cluster and both need to talk with the same service outside their cluster, it is preferable to apply one single configuration that works for both, as long as some conditions are met (e.g., if the two services are in the same namespace). The presence of this list can also help in identifying cases the operation of creating a path between two services can be skipped, if a path between them was previously created before for another communication.

On the other hand, the list describing the pairs of clusters that must be linked together is useful for the optimization process of the cluster mesh, i.e., the interconnection among the clusters of the different domains. Once the cluster mesh is initially created, it is later updated by the Multi-Cluster Orchestrator only when actually required on the basis of the information provided by this list, e.g., every time a new
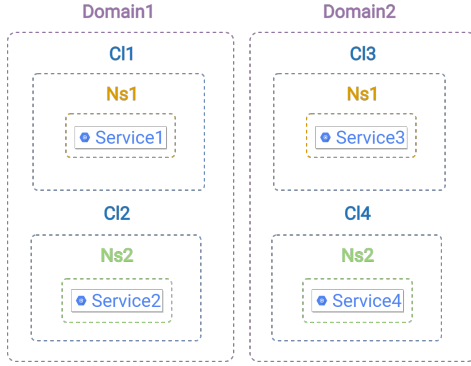
Fig. 3. Use Case

domain must be included or a new link from a service in a cluster that does not currently belong to the cluster mesh to a service in another domain must be created.

2) Single Configurations: Once a Global Configuration is created, the Multi-Cluster Orchestrator creates a Single Configuration for every cluster and applies it to them. Each Single Configuration includes:

- the parameters that are required to set up the presence of the cluster in the cluster mesh;
- the network policies to be installed for the services or pods of the cluster;
- the commands to be used to create new services in the cluster, with the objective resolve names of services deployed in external clusters.

First, for the connection with other clusters, some parameters need to be set in the cluster and some operations need to be executed in order to secure connections between all clusters. These parameters and operations are different according to the technology chosen for the specific implementation of the cluster mesh. In this work, as we will discuss in Section IV, the mesh creation is created using the Cilium Cluster Mesh technology, which requires parameters such as a cluster id and a cluster name. These names and ids may be different from the ones specified by the domain manager in the formulation of the input security requirements. In that case, the Multi-Cluster Orchestrator is in charge of transparently mapping them to the actual values.

Second, for the creation of the network policies, the Multi-Cluster Orchestrator performs a refinement process. Starting from the high-level security requirements, it automatically computes the different network policies for all services or pods that must be protected, so as to allow communications only to the services specified by the domain manager for both incoming and outgoing traffic. At the end of the refinement process, the Single Configuration accordingly contains all the network policies for that specific cluster, which can be then applied using the Kubernetes Cluster's API.

Third, for the name resolution of services deployed in external clusters, Multi-Cluster Orchestrator does not require the modification of the deployed applications. Instead, for each cluster, the Multi-Cluster Orchestrator automatically deploys

new services, which simply refer to the original services deployed in other clusters. In this way, applications inside that cluster can resolve the name of the service they want to connect, as it belongs to the same namespace, in a completely transparent way. The Single Configuration contains all the commands for the setup of these services, which are then created using again the Kubernetes Cluster's API.

IV. IMPLEMENTATION AND VALIDATION

This section describes the implementation of the Multi-Cluster Orchestrator in Subsection IV-A. Then, it discusses its validation in Subsection IV-B, presenting a use case that clarifies how the Multi-Cluster Orchestrator works in a multi-domain environment, which operations it performs and which final configurations it applies to each cluster.

A. Implementation

The implementation of the Multi-Cluster Orchestrator has been developed with the Java language. As an official Java client for the interaction with the Kubernetes Cluster's API was already available [24], we started from that implementation, and we extended it with missing Java methods, such as the ones for the creation of network policies.

The Multi-Cluster Orchestrator exposes REST APIs, so as to allow interactions with human users or external tools which may be components of a more complex architecture. Besides, it employs the Cilium Cluster Mesh technology for the creation of the mesh interconnecting clusters of different domains.

The security requirements that describe which communications must be allowed are formulated with an extended version of the YAML syntax that is used for the definition of the network policies. In particular, special labels can be used (service, cluster, and domain) to specify services or groups of services belonging to a cluster or a domain whose ingress or egress traffic must be allowed, without the need to specify their selector or namespace. These labels are used by the Multi-Cluster Orchestrator for the refinement process described in Section III.

B. Validation

The Multi-Cluster Orchestrator has been validated in multiple multi-domain environments. For each scenario, we have checked that the cluster configurations automatically computed by the orchestrator are correct by verifying if the communications to be allowed were not blocked, and vice versa. Here, we only describe a representative use case due to page limitation.

The environment of this use case, represented in Fig. 3, is characterized by two domains, managed by different companies. The domain managers want to allow communications from some services of their cluster to services belonging to the other one automatically and securely, so as not to have configuration problems that can lead to disservices or breaches. The main characteristics of this environment are as follows:

- Both domains, i.e., *domain1* and *domain2*, are registered to the Multi-Cluster Orchestrator, after interacting with its REST APS.

```
apiVersion: v1
kind: Service
metadata:
  name: service
  namespace: service—namespace
  annotations:
    io.cilium/global—service: "true"
spec:
  type: ClusterIP
  ports:
  - port: 80
  selector:
    name: service
```

Fig. 4. Service Definition

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: service
spec:
  selector:
    matchLabels:
      name: service
  replicas: 2
  template:
    metadata:
      labels:
        name: service
    spec:
      containers:
      - name: rebel—base
        image: docker.io/nginx:1.15.8
        volumeMounts:
          - name: html
            mountPath: /usr/share/nginx/html/
        ...
      - name: html
        configMap:
          name: service—response
          items:
            - key: message
              path: index.html
```

Fig. 5. Service Deployment

- *domani1* is composed of two clusters named *cl1* and *cl2*, *domain2* is composed of two clusters named *cl3* and *cl4*.
- In *domain1*, there are two services: *service1* is located in namespace *ns1* and deployed in cluster *cl1*, whereas *service2* is located in namespace *ns2* namespace and deployed in cluster *cl2*.
- In *domain2*, there are two services: *service3* is located in namespace *ns1* and deployed in cluster *cl3*, whereas *service4* is located in namespace *ns2* and deployed in cluster *cl4*.

Each service of this use case is defined as the one in the YAML file of Fig. 4. For each service, the corresponding deployment of Fig. 5 creates two pods running that service.

At the beginning of this use case, when no inter-domain security requirement has been yet specified, services of different domains are not able to interact with each other. In order to check this property and later also to test which communications are allowed or blocked in this environment, we used a client, created with the YAML file of Fig. 6, describing a deployment with two pods that are able to request HTML pages using the curl command. For example, we tried

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: client
spec:
  selector:
    matchLabels:
      name: client
  replicas: 2
  template:
    metadata:
      labels:
        name: client
    spec:
      containers:
      - name: x—wing—container
        image: docker.io/cilium/json—mock:1.2
        livenessProbe:
          exec:
            command:
            - curl
            - —sS
            - —o
            - /dev/null
            - localhost
```

Fig. 6. Client Deployment

```
$ kubectl exec -ti client-644c86b6b-5wppx -- curl service2
curl: (6) Could not resolve host: service2
command terminated with exit code 6
```

Fig. 7. Request failure due to missing service in the cl3

to deploy the client in the *default* namespace of cluster *cl3*, and we checked that it is not able to find and contact *service2* of *domain1*, as shown in Fig. 7.

Then, we request the application of the security requirement described in Fig. 8. The Multi-Cluster Orchestrator creates the services and namespaces needed to start the communication with services *service1* and *service2* in cluster *cl3*, as shown in Fig. 9 and Fig. 10. Fig. 11 also shows the status of the namespaces in cluster *cl3*, after the creation of *service2*.

```
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: "allow—domain1"
  spec:
    endpointSelector:
      matchLabels:
        name: client
    egress:
    - toEndpoints:
      - matchLabels:
          service: "*"
          cluster: "*"
          domain: "domain1"
```

Fig. 8. Service Deployment

```
$ kubectl -n ns2 get service
NAME       TYPE       CLUSTER-IP    EXTERNAL-IP   PORT(S)   AGE
service2   ClusterIP  10.5.25.91    <none>        80/TCP    91s
```

Fig. 9. Service2 created in cl3

```
$ kubectl get service
NAME        TYPE          CLUSTER-IP   EXTERNAL-IP                     PORT(S)   AGE
kubernetes  ClusterIP     10.5.0.1     <none>                          443/TCP   41m
service1    ExternalName  <none>       service1.ns1.svc.cluster.local  80/TCP    53s
service2    ExternalName  <none>       service2.ns2.svc.cluster.local  80/TCP    53s
```

Fig. 10. Service2 External Service created in the default namespace

```
$ kubectl get namespace
NAME                STATUS   AGE
default             Active   42m
kube-node-lease     Active   42m
kube-public         Active   42m
kube-system         Active   42m
local-path-storage  Active   42m
ns1                 Active   28m
ns2                 Active   103s
```

Fig. 11. Namespaces of cl3 after Service2 creation

At that point, the Multi-Cluster Orchestrator automatically creates a Cilium Network Policy, named *"allow-domain1"*, which contains the labels of the pods running services *service1* and *service2*, so as to allow only communications with these two services for the pods of the client deployment. When the client thus requests again to communicate with *service2*, it is able to contact it and receive a response, as shown in Fig. 12.

```
$ kubectl exec -ti client-644c86b6b-5wppx -- curl service2
{"Domain": "1", "Cluster": "Cluster-2"}
```

Fig. 12. Successful request to Service2

If this Cilium Network Policy is later deleted, also the services and namespaces created are deleted, because there are no other policies applied in the clusters that require the presence of those services and namespaces. Fig. 13, Fig. 14, and Fig. 15 respectively show how the services and namespaces are deleted, after the policy removal.

```
$ kubectl get service
NAME         TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
kubernetes   ClusterIP   10.5.0.1     <none>        443/TCP    51m
```

Fig. 13. Services in default namespace of cl3 after policy removal

```
$ kubectl -n ns2 get service
No resources found in ns2 namespace.
```

Fig. 14. Services in ns2 namespace of cl3 after policy removal

```
$ kubectl get namespace
NAME                STATUS   AGE
default             Active   52m
kube-node-lease     Active   52m
kube-public         Active   52m
kube-system         Active   52m
local-path-storage  Active   52m
ns1                 Active   37m
```

Fig. 15. Namespaces of cl3 after policy removal

V. CONCLUSION AND FUTURE WORK

This paper proposes an automatic approach, based on an entity named Multi-Cluster Orchestrator, to simplify and automate the security operations related to the configuration of a Kubernetes-based multi-domain environment. The Multi-Cluster Orchestrator has a complete overview of all the clusters composing the domains of the environment, and therefore it can refine the security policies specified by the domain managers into the concrete configuration. The implementation of this entity has been validated on use cases, showcasing the applicability of this approach for cluster mesh management.

Future work envisions an extensive validation of the Multi-Cliuster Orchestrator, so as to assess its performance according to measurable metrics such as deployment latency and overhead of the communication. This validation will also allow a comparison with existing solutions. Another future work consists in the creation of an interface that may help domain managers in setting the security requests in a more intuitive way, so that they can easily decide which internal services must be able to communicate with external services.

REFERENCES

[1] J. Watada, A. Roy, R. Kadikar, H. Pham, and B. Xu, "Emerging trends, techniques and open issues of containerization: A review," *IEEE Access*, vol. 7, pp. 152 443–152 472, 2019.

[2] A. Rahman, S. I. Shamim, D. B. Bose, and R. Pandita, "Security misconfigurations in open source kubernetes manifests: An empirical study," *ACM Trans. Softw. Eng. Methodol.*, 2023, in press.

[3] G. Budigiri, C. Baumann, J. T. Mühlberg, E. Truyen, and W. Joosen, "Network policies in kubernetes: Performance evaluation and security analysis," in *Proc. of EuCNC/6G Summit 2021, Porto, Portugal, June 8-11, 2021*. IEEE, 2021, pp. 407–412.

[4] M. S. I. Shamim, F. A. Bhuiyan, and A. Rahman, "XI commandments of kubernetes security: A systematization of knowledge related to kubernetes security practices," in *Proc. of IEEE SecDev 2020, Atlanta, GA, USA, September 28-30, 2020*. IEEE, 2020, pp. 58–64.

[5] https://www.inspire-5gplus.eu, Visited: 2023-04-24.

[6] https://www.monb5g.eu, Visited: 2023-04-24.

[7] https://www.fluidos.eu, Visited: 2023-04-24.

[8] A. El-Hassany, P. Tsankov, L. Vanbever, and M. T. Vechev, "Netcomplete: Practical network-wide configuration synthesis with autocompletion," in *Proc. of the USENIX Symp. on Networked Systems Design and Implementation*, S. Banerjee and S. Seshan, Eds., 2018.

[9] C. Bodei, P. Degano, L. Galletta, R. Focardi, M. Tempesta, and L. Veronese, "Language-independent synthesis of firewall policies," in *Proc. of the IEEE European Symp. on Security and Privacy*, 2018.

[10] D. Bringhenti, G. Marchetto, R. Sisto, F. Valenza, and J. Yusupov, "Automated optimal firewall orchestration and configuration in virtualized networks," in *Proc. of the IEEE/IFIP Network Operations and Management Symposium*, 2020.

[11] ——, "Introducing programmability and automation in the synthesis of virtual firewall rules," in *Proc. of 6th IEEE Conference on Network Softwarization, NetSoft 2020, Ghent, Belgium, June 29 - July 3, 2020*. IEEE, 2020, pp. 473–478.

[12] ——, "Automated firewall configuration in virtual networks," *IEEE Trans. Dependable Secur. Comput.*, vol. 20, no. 2, pp. 1559–1576, 2023.

[13] D. Bringhenti, G. Marchetto, R. Sisto, and F. Valenza, "Short paper: Automatic configuration for an optimal channel protection in virtualized networks," in *Proc. of the Work. on Cyber-Security Arms Race*, 2020.

[14] L. Firdaouss, A. Bahnasse, B. Manal, and Y. Ikrame, "Automated VPN configuration using devops," in *Proc. of (EUSPN 2021), Leuven, Belgium, November 1-4, 2021*, ser. Procedia Computer Science, vol. 198. Elsevier, 2021, pp. 632–637.

[15] A. M. Zarca, J. B. Bernabé, R. Trapero, D. Rivera, J. Villalobos, A. F. Skarmeta, A. Bianchi, A. Zafeiropoulos, and P. Gouvas, "Security management architecture for nfv/sdn-aware iot systems," *IEEE Internet Things J.*, vol. 6, no. 5, pp. 8005–8020, 2019.

[16] M. A. Rahman, A. Datta, and E. Al-Shaer, "Automated configuration synthesis for resilient smart metering infrastructure," *EAI Endorsed Trans. Security Safety*, vol. 8, no. 28, p. e4, 2021.

[17] D. Bringhenti, J. Yusupov, A. M. Zarca, F. Valenza, R. Sisto, J. B. Bernabé, and A. F. Skarmeta, "Automatic, verifiable and optimized policy-based security enforcement for sdn-aware iot networks," *Comput. Networks*, vol. 213, p. 109123, 2022.

[18] J. Kim, Y. Kim, V. Yegneswaran, P. A. Porras, S. Shin, and T. Park, "Extended data plane architecture for in-network security services in software-defined networks," *Comput. Secur.*, vol. 124, p. 102976, 2023.

[19] https://cilium.io/blog/2019/03/12/clustermesh/, Visited: 2023-04-24.

[20] https://submariner.io/, Visited: 2023-04-24.

[21] https://skupper.io/, Visited: 2023-04-24.

[22] https://istio.io/, Visited: 2023-04-24.

[23] https://linkerd.io/, Visited: 2023-04-24.

[24] https://github.com/kubernetes-client/java, Visited: 2023-04-24.