An intent-based solution for network isolation in Kubernetes

An intent-based solution for network isolation in Kubernetes

Francesco Pizzato, Daniele Bringhenti, Riccardo Sisto, Fulvio Valenza
Dip. Automatica e Informatica, Politecnico di Torino, Torino, Italy, Emails: {first.last}@polito.it

***Abstract*—Cloud computing has transformed the landscape of application delivery, offering an enormous pool of devices with a wide-spread geographical distribution. In this context, liquid computing is a novel paradigm that aims to avoid that available resources are underutilized, by facilitating their seamless sharing among different tenants and administrative domains. Nevertheless, liquid computing introduces new security challenges, particularly related to network isolation, which traditional approaches are inadequate to address. Therefore, this paper proposes a security orchestrator to automate the configuration of network isolation primitives across a multi-domain and multi-tenant cloud environment, simplifying the implementation of security patterns like zero trust and least privilege. The proposed solution is intent-driven, because users define their requirements in terms of desired and prohibited network communications through a user-friendly language. In our implemented proposal, intents expressed by different users are harmonized to avoid discordances among them, and then they are translated into Kubernetes Network Policies as isolation primitives.**

***Index Terms*—cloud security, Kubernetes, security automation**

I. Introduction

Cloud computing is a revolutionizing paradigm that reshaped the traditional ways to deliver applications. Its main principle consists in enabling network access to a shared pool of computing resources that can be rapidly provisioned and released with minimal management effort, thus allowing to build distributed systems reaching world-wide scale. In order to further expand this feature, several initiatives are researching a way to avoid that cloud servers use less computational resources with respect to what they have been equipped with, as they often leave a large portion of the allocated resources unused. A noteworthy concept is liquid computing [1], a novel cloud paradigm that allows a seamless sharing of resources (e.g., network, computation, data) between different tenants and administrative domains, thus generating a seamless pool of resources that can be borrowed or lent in a dynamic way. Despite the many advantages of this approach in terms of scalability and costs [2], it introduces additional security risks and an increased complexity in security management. In fact, in such multi-cloud and multi-domain scenario, each physical node could be possibly shared by multiple and heterogeneous users belonging to different administrative domains.

A significant problem to be addressed in the context of liquid computing is network isolation. This problem has been often overlooked in traditional cloud environments, despite its relevance. As reported in [3], most of the analyzed clusters (>90%) have not configured any Network Policy in their namespaces, lacking an important isolation layer that could prevent attackers from performing lateral movements within the cluster. All the more, the problem of network isolation cannot be neglected anymore in liquid computing, because the conventional notion of a physical boundary that must be protected is no more valid. The single physical server is substituted by the virtual cluster, which describes the concept of a physical machine that is extended to one or more remote ones by buying resources from them. The boundary of this virtual cluster is a dynamic virtual boundary capable of shrinking and expanding in a "liquid" way.

Specifically, ensuring network isolation in the shared resource continuum serves multiple purposes. First, it is necessary to protect the hosting cluster against potential harm from guest applications deployed by a different tenant, possibly belonging to a different administrative domain. Second, it is equally important to protect guest applications against potential stealing of data or code or unauthorized interference from the host. Third, there is the need to allow application owners to specify precisely what interactions their applications may have with their environments, and what the hosting environments are willing to allow. However, no solution is available in literature to address network isolation in liquid computing, so currently these three goals have not yet been achieved.

In view of all these considerations, this paper proposes the design of a security orchestrator to automate the configuration of network isolation primitives, i.e., Kubernetes Network Policies, across a multi-domain and multi-tenant cluster, so as to ease the implementation of common security patterns such as zero trust and least privilege. The proposed solution is intent-driven, because desired and prohibited network communications are expressed by the user with a user-friendly high-level language that does not require excessive technical knowledge, and then these intents are automatically translated into the specific low-level configurations used for enforcement.

The remainder of this paper is structured as follows. Section II discusses related work. Section III describes the proposed approach, detailing its main features. Section IV discusses how this approach has been implemented and validated. Section V concludes the paper and outlines future research work.

II. Related Work

Network security automation has been explored in recent years to make use of the improved dynamism of virtual

networks, as shown in a comprehensive analysis of the state of the art about this topic [4]. Specifically, some studies, such as [5]–[8], combine configuration automation with policy-based management, an approach that brings different benefits, such as reducing the risk of misconfigurations and introducing the usage of high-level intents. However, those solutions cannot be applied to cloud computing environments to configure it so as to provide network isolation.

Intent-based security automatic approaches within the cloud thus represent a natural follow-up to what has already been investigated in that field. In this context there have been different studies trying to automate security related tasks. In particular, a relevant number of solutions [9]–[11] have been proposed for the automatic verification of user-defined security intents over formal models generated starting from configuration files of a cloud system. Other studies [12], [13] focus on the extraction of an enriched model from Kubernetes configuration files, so as to use it to solve different automated reasoning problems, such as attack graph generation and threat analysis. However, all of them are applying automation only for the verification and compliance of the security posture of the system, lacking the ability of automatically generate a security configuration from scratch. From this point of view, there are just few studies [14]–[16] that try to automate the enforcement of low-level security configuration aspects starting from the analysis of running systems. Nevertheless, none of them is providing a solution capable of integrating higher-level security features, e.g., Network Policies, they are all missing the support for user-defined intents, and they are not designed to work with the liquid computing paradigm.

III. The Proposed Approach

In a multi-domain and multi-tenant cloud environment, the resource sharing process determined by liquid computing involves two roles: the consumer and the provider. The former uses resources of a remote cluster, and the latter supplies resources to other clusters. A resource, which resides physically in the provider's cluster, once shared will be seen by the consumer as part of their own local cluster. The remote resources can be consumed in a transparent way, meaning that any workload targeting these resources will be automatically executed in the hosting cluster (i.e., the provider's remote cluster) and will be seen as a workload hosted by the provider. The technology enabling the creation of this seamless continuum of resources is Liqo [17], and the process of connecting two clusters is called peering. Instead, the process of allocating a workflow in a remote cluster is called offloading.

In this context, this paper proposes an automatic approach for the enforcement of a network boundary between the consumer and the provider, enabling both parties involved in the peering to define finely-grained intents to request and authorize connections. Specifically, at the beginning of the process, different sets of intents are exchanged during the offloading (Subsection III-A). Then, an harmonization is performed between the intents defined by both tenants taking part in the peering to intelligently select the resulting set of approved intents, which are later translated and enforced in the appropriate locations (Subsection III-B).

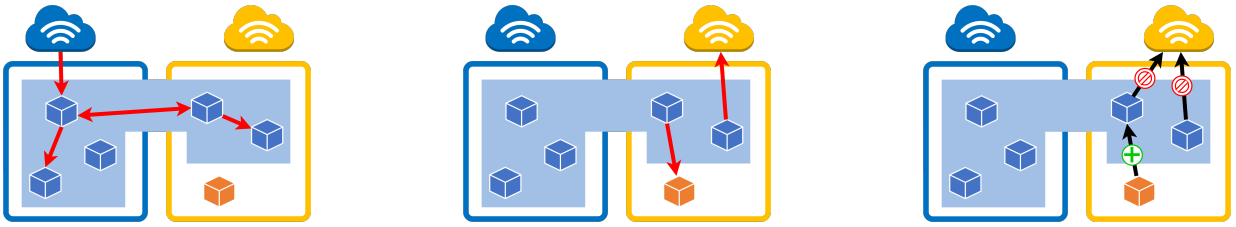
A. Intents

In defining the types of intents a user can formulate as input to our proposed approach, we considered security isolation requirements that are specific to liquid computing. For example, a user may have the need of allowing the communications for pods in the same virtual cluster but distributed over many physical clusters. This is the deployment scenario called "elastic cluster", which could be adopted to absorb a cloud bursting, during which some pods are moved to a remote cluster due to the physical limitations of the one on premise. As an another example, a user may also have to restrict communications for pods belonging to different virtual clusters but sharing the same physical one. This other situation is motivated by the principle of data gravity, requiring that the processing is moved where the data is located for improved latency, or also to be compliant with regulation policies like GDPR, which requires that data cannot be moved outside a specific geographical location.

In order to express all these possible scenarios, we have envisioned three main types of intents a user may express, each tailored to achieve different objectives and definable only if the user has a certain role, i.e., consumer or provider. When users assume the consumer role, their main security objective is to safeguard communications within their local cluster (*Private intents*) and protecting communications among resources offloaded to remote clusters (*Request intents*). Conversely, when users assume the provider role, their main security goal is to limit communications involving the hosted resources on the one hand, its own services or the external network (i.e., the Internet) on the other hand (*Authorization intents*).

Fig. 1 presents an example of the different communications that could be expressed with these three types of intents. Each sub-figure includes two thick-bordered boxes representing two different physical clusters owned by the blue and yellow tenants, some pods represented by filled boxes, and a virtual cluster owned by the blue tenant represented by a region with blue background that spans the blue physical cluster (the home portion of the virtual cluster) and the yellow cluster (the offloaded portion of the virtual cluster). The different communications are represented by arrows. Given such example, the possible intent types are explained in the following:

- *Private intents*: they are related to communications happening within the virtual cluster, i.e., intra-virtual cluster, involving both local and remote resources. These are not subject to the authorizations of the host(s), following the principle that each user should have unlimited control over his own resources. Some possible communications expressed with Private intents are represented in Fig. 1a.
- *Request intents*: they are related to inter-virtual cluster communications, i.e., those that are crossing the virtual border of an extended cluster. These could be configured to target services offered by the hosting cluster or addresses on the external network. Some examples are



(a) Private Intents (b) Request Intents (c) Authorization Intents

Fig. 1: User-defined sets of intents

shown in Fig. 1b. Within these intents, the consumer could tune some parameters to refine his will in terms of isolation with respect to the hosting cluster (e.g., to accept or refuse to be monitored by the provider).

- *Authorization intents*: they are defined by each user to represent their authorization policies with respect to any possible guest. Their scope is to regulate the inter-virtual cluster communications for all virtual clusters that are hosted in the local physical cluster. Specifically, they target two communication types:
  1) *DeniedCommunications* that must be blocked or filtered for all hosts. For instance the provider could choose to deny the connections to some blacklisted URLs, or to a subset of his resources.
  2) *MandatoryCommunications* that must be injected to all hosts. A possible usage is to whitelist the communications needed for the provider to collect the logs of each hosted application.

  Fig. 1c presents some examples of communications that can be expressed with Authorization intents.

All these intents can be expressed by the user with a format that provides a similar degree of expressiveness as the one achieved with selectors in Kubernetes Network Policies, so as to allow the specification of the information needed to select specific traffic. The envisioned structure is the following one: "*from SRC to DST*, *protocol*[: *port*[−*endPort*]]".

- *SRC* and *DST* can be either a pod or a group of pods with the same label and an associated namespace, or an address or a group of addresses defined through CIDR (at most one could be a CIDR address). For them, the wildcard symbol "*" can be used to target the whole (virtual) cluster if "*" is the value assigned to both pod and namespace, or to select all pods in a specific namespace if it assigned only to the pod.
- *protocol* can be any transport protocol (TCP, UDP, SCTP, etc.) or the value "ALL" to represents all of them.
- *port* can be a port, or a range of ports.

Finally, the Request and Private intents can be expressed only in whitelisting, so as to be compliant with the default behavior of Kubernetes Network Policies, which define the set of permitted communications and all the other ones are consequently blocked. Instead, the Authorization intents allow for more expressiveness and flexibility. The *DeniedCommunications* are expressed with a blacklisting approach, i.e., the user can define only denied communications and the remaining ones are allowed. Conversely, the *MandatoryCommunications* are simply allowed communications, since they express the connections that must be imposed to all guest.

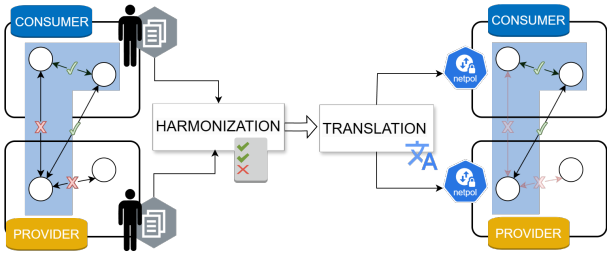


Fig. 2: Schema of the intent-based network isolation workflow.

B. Workflow

The workflow envisioned for the orchestration of network isolation intents is integrated in the resource acquisition process. During this operation, the offloading cluster, i.e., consumer, and the hosting cluster, i.e., provider, interact exchanging the previously defined sets of intents and performing the harmonization, translation, and enforcement. In general, this process can be performed multiple times, one for each candidate provider, with the scope of finding the best one with respect to the consumer's intents. The core process is the harmonization, which produces a new set of *Harmonized intents* by comparing the provider's and consumer's sets, solving all the detected discordance between them. The result of the harmonization is used by the consumer to choose the compatibility level of the provider accordingly to the approved, forced, or denied intents. The user could combine this result with other parameters, coming from heterogeneous sources, such as latency, geographical position, or energy consumption. However, for the scope of this article, the consumer chooses a provider only based on the offered set of network authorizations and the compatibility with his defined intents.

Fig. 2 graphically represents the workflow employed in the situation of a peering request performed by a consumer cluster towards a single provider cluster. The process starts when the consumer creates a request to acquire some resources, either computational resources (e.g., VMs, Kubernetes Slice) or services (e.g., database, training data-set). The request is enriched with the sets of user-defined Private and Request

intents, as explained in Section III-A. The local orchestration module handle this request and it is responsible for contacting all the providers to gather their offer, which is enriched, among the others, with the set of Authorization intents. For each candidate, the consumer must perform the harmonization, and, once a specific provider has been selected, the intents are translated into low-level configurations and properly enforced. The modules handling these two phases are presented in greater details in the following.

1) Harmonization: An algorithm has been designed to perform the harmonization between different sets of intents established by the users engaged in a peering process, providing an intelligent resolution of the possible discordance between them. The general principle behind this algorithm is that the hosting cluster has the decision power: it chooses which Request intents, defined by the consumer, can or cannot be enforced while possibly forcing some new ones. Anyhow, the host can impose his authority only over the inter-virtual cluster communications, while the intra-virtual cluster communications, even if happening on the host's cluster, should be fully determined by the consumer who acquired the usage of those resources and has the right to configure it freely. In this context, a discordance arises when an intent defined by one user is not authorized or coherent with intents defined by another user with whom peering is requested.

These discordance can be classified in three main types, related to three possible cases: 1) when a Request intent defined by the consumer is not (fully or partially) authorized by the Authorization intents of the provider; 2) when a MandatoryCommunication defined by the provider's Authorization intents is not satisfied (fully or partially) by the Request intents of the consumer; 3) when a Request intent defined by the consumer has been authorized, but does not have a symmetrical Request intent in the provider. This last case is needed to have coherence between the consumer and provider intent sets, since it is not enough that the consumer's Request intent is authorized but a similar provider's intent should be defined in order to fully allow the communication. These three discordance types are represented in Fig. 3 through an example. For the first case, a user performing the offloading is requesting that his offloaded entity A can contact a malicious website, but the Authorization intents defined by the hosting user deny all connections to the Internet for all offloaded pods, thus causing a discordance. Second, the provider defined a MandatoryCommunication from his resource M, a monitoring endpoint, to all offloaded pods. This is not yet satisfied by the intents defined by the consumer, causing another discordance. Third, the consumer requests that the same offloaded entity A can contact entity B, which is part of the hosting cluster. However, the hosting user has not defined an intent allowing B to be contacted by A, thus resulting in another discordance.

2) Translation: After the Harmonized set of intents has been computed, and the consumer has selected a provider, the translation module translates the high-level intents, agnostic to the actual implementation, into the low-level configuration of the Kubernetes Network Policies required to enforce network

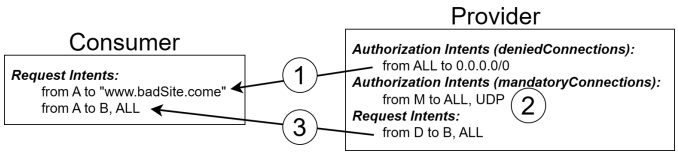


Fig. 3: Example of possible discordance between intents.

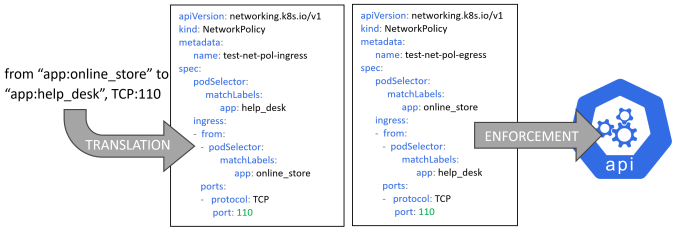


Fig. 4: Example of translation of an intent.

isolation between workloads. The complexity of translation depends on the selected CNI (e.g., not all CNI support cluster-wide Network Policy, thus requiring in some cases a one-to-many translation for the intents). The presence of this translation module is crucial for the interoperability with multiple clusters that are possibly using multiple technologies. Different version of this translation module could adapt the solution to different vendor-specific solutions, e.g., different CNIs and Network Policy formats. Finally, this module is also responsible for the enforcement of the generated Kubernetes Network Policies by communicating with the API server of the cluster hosting the targeted resources. Fig. 4 represents an example of such workflow considering a single intent that is translated to two Network Policies, one allowing the ingress traffic at the destination from the source, and another allowing the egress traffic for the source toward the destination.

IV. Implementation and Validation

The proposed approach has been implemented as a proof-of-concept Java module, which authenticates itself and communicates with the API server of different Kubernetes clusters. For what concerns the CNI, we implemented a translator working with Calico because of its full support to the native Kubernetes Network Policy format. Moreover, the security intents are expressed using an extended version of the MSPL (i.e., Medium-level Security Policy Language) language, which is characterized by a generic syntax that abstracts the vendor-specific configuration. This choice is motivated by the fact that MSPL has already been successfully used and validated by multiple European research project, e.g., ANASTACIA and SECURED, and research papers [18], [19].

This implementation has been validated with different scenarios and sets of user-defined intents in the context of the EU project FLUIDOS. However, for the scope of this paper, we present a representative validating use case.

The scenario considered in the use case is shown in Fig. 5. It is characterized by two domains, represented by the cluster with a blue border and the one with an orange border, each

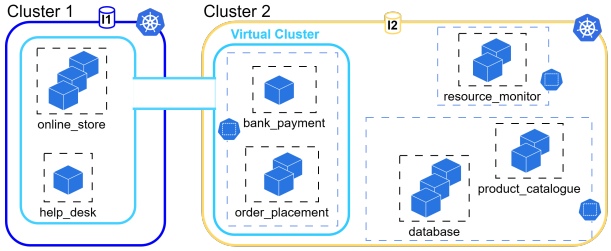


Fig. 5: Example scenario.

```
[INFO]    Received the following Request intents (CONSUMER):
 (*) src:[ app:order_placement - name:fluidos ], sPort:[*],
      dst:[ *:* - name:default ], dPort:[*], protocol:[ALL]
 (*) src:[ app:bank_payment - name:fluidos ], sPort:[*],
      dst:[0.0.0.0/0], dPort:[*], protocol:[ALL]

[INFO]    Received the following Authorization Intents (PROVIDER):
.-> ForbiddenCommunicationList:
|(*) src:[ *:* - *:* ], sPort:[*],
      dst:[0.0.0.0/4], dPort:[*], protocol:[ALL]
|(*) src:[ *:* - *:* ], sPort:[*],
      dst:[ app:product_catalogue - name:default ], dPort:[*], protocol:[STCP]
|(*) src:[ *:* - *:* ], sPort:[*],
      dst:[ app:product_catalogue - name:default ], dPort:[*], protocol:[UDP]
|(*) src:[ *:* - *:* ], sPort:[*],
      dst:[ app:product_catalogue - name:default ], dPort:[0-79], protocol:[TCP]
|(*) src:[ *:* - *:* ], sPort:[*],
      dst:[ app:product_catalogue - name:default ], dPort:[81-65535], protocol:[TCP]
|(*) src:[ *:* - *:* ], sPort:[*],
      dst:[ app:database - name:default ], dPort:[*], protocol:[ALL]
|(*) src:[ *:* - *:* ], sPort:[*],
      dst:[ app:resource_monitor - name:monitoring ], dPort:[*], protocol:[ALL]
.-> MandatoryCommunicationList:
 (*) src:[ app:resource_monitor - name:monitoring ], sPort:[*],
      dst:[ *:* - name:fluidos ], dPort:[43], protocol:[TCP]
```

Fig. 6: Simplified visualization of intent sets.

hosting a different service, composed of multiple applications. The first hosts a simplified e-commerce service composed of four applications. The second hosts a warehouse management service, composed of two applications, and a monitoring agent to keep track of cluster events and application-specific logs. All the resources are labeled at the application level as represented within the image, which means that all pods running the same application will have the same label.

In this scenario, the user of the blue cluster wants to use an external warehouse service, which is offered by the user of the orange cluster. The goal is to handle the resource acquisition process and automatically enforce isolation between the two tenants over the border of the virtual cluster, represented with a light blue line extending from the physical cluster of the consumer to the remote one of the provider. At the same time, some connections should be opened for different reasons:

- the consumer wants to use the Internet connection of the hosting cluster for the application handling payments, which must be able to communicate with the external bank's payment network.
- the consumer wants to limit the communications with the provider's service, i.e., warehouse management service, only to a specific application, i.e., order placement, blocking all the others communications.
- the provider wants that each application in their cluster, even the guest ones, can be contacted by the monitoring agent to retrieve application's logs.

All the other communications are by default blocked, following the least privilege principle. Note that for the consumer only the Private and Request intents are important, whereas

```
[INFO]    List of harmonized Request intents (CONSUER) after type-1 resolution:
 (*) src:[ app:order_placement - name:fluidos ], sPort:[*],
      dst:[ app:product_catalogue - name:default ], dPort:[80], protocol:[TCP]
 (*) src:[ app:bank_payment - name:fluidos ], sPort:[*],
      dst:[128.0.0.0/1], dPort:[*], protocol:[ALL]
 (*) src:[ app:bank_payment - name:fluidos ], sPort:[*],
      dst:[64.0.0.0/2], dPort:[*], protocol:[ALL]
 (*) src:[ app:bank_payment - name:fluidos ], sPort:[*],
      dst:[32.0.0.0/3], dPort:[*], protocol:[ALL]
 (*) src:[ app:bank_payment - name:fluidos ], sPort:[*],
      dst:[16.0.0.0/4], dPort:[*], protocol:[ALL]
```

Fig. 7: Result of type-1 discordance resolution.

```
[INFO]    List of harmonized Request intents (CONSUMER) after type-2 resolution:
 (*) src:[ app:order_placement - name:fluidos ], sPort:[*],
      dst:[ app:product_catalogue - name:default ], dPort:[80], protocol:[TCP]
 (*) src:[ app:bank_payment - name:fluidos ], sPort:[*],
      dst:[128.0.0.0/1], dPort:[*], protocol:[ALL]
 (*) src:[ app:bank_payment - name:fluidos ], sPort:[*],
      dst:[64.0.0.0/2], dPort:[*], protocol:[ALL]
 (*) src:[ app:bank_payment - name:fluidos ], sPort:[*],
      dst:[32.0.0.0/3], dPort:[*], protocol:[ALL]
 (*) src:[ app:bank_payment - name:fluidos ], sPort:[*],
      dst:[16.0.0.0/4], dPort:[*], protocol:[ALL]
 (*) src:[ app:resource_monitor - name:monitoring ], sPort:[*],
      dst:[ *:* - name:fluidos ], dPort:[43], protocol:[TCP]
```

Fig. 8: Result of type-2 discordance resolution.

for the provider only the Authorization and Request ones. The complete set of intents defined by both parties is presented in a simplified form in Fig. 6.

Starting from this initial situation, the user sends a request for the resources and the service to an orchestration agent. The offers of all possible providers are collected and the consumer has to pick one of them. This process might involve many factors, but the only one we consider in this case is network authorizations. In order to compute the compatibility of the provider's intents with his set, the consumer has to perform the harmonization of the two. Within this use case, the consumer defined two Request intents:

1) allow traffic from $app : order\_placement$ to all destinations in the hosting cluster $* : *$, any port and protocol.
2) allow traffic from $app : bank\_payment$ to all IP addresses, i.e., $0.0.0.0/0$, any port and protocol.

In the harmonization process, each type of discordance is solved separately. Starting from the first discordance type, the first intent is overlapping with the authorizations defined by the provider, which exposed only the application $app : product\_catalogue$ at port 80 TCP and blocks all the other communications. Moreover, the second intent is partially overlapping with another Authorization intent of the provider, which blocks the $0.0.0.0/4$ range of addresses. The result of this passage is shown in Fig. 7. Continuing with the second discordance type, the provider defined only one intent within the mandatory list of communications. Moreover, that intent describes a communication that has no intersection with the ones in the current Harmonized set of consumer's Request intents. Consequently, the mandatory communication is added to the final Harmonized set with no modification. This is shown in Fig. 8. Then, concerning the third discordance type, as initially the set of provider's Request intents is empty, the consumer's Request intents are simply opportunely modified and added to the provider's set because there is no possible overlap. This is represented in Fig. 9.

```
[INFO]    List of harmonized Request intents (PROVIDER) after type-3 resolution:
 (*) src:[ app:order_placement - name:fluidos ], sPort:[*],
     dst:[ app:product_catalogue - name:default ], dPort:[80], protocol:[TCP]
 (*) src:[ app:bank_payment - name:fluidos ], sPort:[*],
     dst:[128.0.0.0/1], dPort:[*], protocol:[ALL]
 (*) src:[ app:bank_payment - name:fluidos ], sPort:[*],
     dst:[64.0.0.0/2], dPort:[*], protocol:[ALL]
 (*) src:[ app:bank_payment - name:fluidos ], sPort:[*],
     dst:[32.0.0.0/3], dPort:[*], protocol:[ALL]
 (*) src:[ app:bank_payment - name:fluidos ], sPort:[*],
     dst:[16.0.0.0/4], dPort:[*], protocol:[ALL]
 (*) src:[ app:resource_monitor - name:monitoring ], sPort:[*],
     dst:[ *:* - name:fluidos ], dPort:[43], protocol:[TCP]
```

Fig. 9: Result of type-3 discordance resolution.

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: request_1_egress
  namespace: fluidos
spec:
  policyTypes:
  - Egress
  podSelector:
    matchLabels:
      app: order_placement
  egress:
    to:
    - namespaceSelector:
        matchLabels:
          name: default
      podSelector:
        matchLabels:
          app: product-catalogue
```

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: request_1_ingress
  namespace: default
spec:
  podSelector:
    matchLabels:
      app: product_catalogue
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          name: fluidos
      podSelector:
        matchLabels:
          app: order_placement
    ports:
    - port: 80
      protocol: TCP
```

Fig. 10: Resulting Network Policy.

After having performed the same harmonization procedure for each possible candidate provider, the consumer has to pick one of them. Once this has been done, the peering is completed and the harmonized set is passed to the translator which must enforce the isolation as soon as the application are deployed in the remote cluster. Taking as example just the first harmonized Request intent, i.e., allow communication from $app : order\_placement$ to $app : product\_catalogue$ at port 80 TCP, its translation into a Network Policy is shown in Fig. 10. This file is then pushed to the API server of the provider cluster for the actual enforcement.

V. Conclusion and Future Work

This paper presented a novel approach for the automated orchestration of network isolation in a liquid computing environment. By using user-defined intents, the approach streamlines the configuration of network security primitives across multiple clusters, simplifying the implementation of common security patterns. The proposed approach has been implemented and validated on different use cases, showing its applicability to the presented problem.

Future work will focus on refining the capabilities of the security orchestrator. This includes the integration of the solution within the resource acquisition workflow, currently under development within the FLUIDOS project, to ease the lease of resources. Finally, we plan on conducting further validation to ensure the effectiveness and scalability of the solution in real-world deployments.

Acknowledgment

This work was partly supported by EU Horizon project FLUIDOS, under grant agreement 101070473.

References

[1] M. Iorio, F. Risso, A. Palesandro, L. Camiciotti, and A. Manzalini, "Computing without borders: The way towards liquid computing," *IEEE Trans. Cloud Comput.*, vol. 11, no. 3, pp. 2820–2838, 2023.

[2] S. Galantino, F. Risso, V. C. Coroama, and A. Manzalini, "Assessing the potential energy savings of a fluidified infrastructure," *Computer*, vol. 56, no. 6, pp. 26–34, 2023.

[3] Wiz, "The 2023 Kubernetes Security Report," Available: https://www.wiz.io/lp/the-2023-kubernetes-security-report, Visited: 2024-01-15.

[4] D. Bringhenti, G. Marchetto, R. Sisto, and F. Valenza, "Automation for network security configuration: State of the art and research trends," *ACM Comput. Surv.*, vol. 56, no. 3, pp. 57:1–57:37, 2024.

[5] E. J. Scheid, C. C. Machado, M. F. Franco, R. L. dos Santos, R. J. Pfitscher, A. E. S. Filho, and L. Z. Granville, "Inspire: Integrated nfv-based intent refinement environment," in *Proc. of the IFIP/IEEE Symp. on Integrated Network and Service Management (IM17)*, 2017.

[6] T. Szyrkowiec, M. Santuari, M. Chamania, D. Siracusa, A. Autenrieth, V. López, J. Y. Cho, and W. Kellerer, "Automatic intent-based secure service creation through a multilayer SDN network orchestration," *J. Opt. Commun. Netw.*, vol. 10, no. 4, pp. 289–297, 2018.

[7] D. Bringhenti, G. Marchetto, R. Sisto, F. Valenza, and J. Yusupov, "Automated firewall configuration in virtual networks," *IEEE Trans. Dependable Secur. Comput.*, vol. 20, no. 2, 2023.

[8] D. Bringhenti, R. Sisto, and F. Valenza, "A novel abstraction for security configuration in virtual networks," *Comput. Networks*, vol. 228, p. 109745, 2023.

[9] C. Cauli, M. Li, N. Piterman, and O. Tkachuk, "Pre-deployment security assessment for cloud services through semantic reasoning," in *Proc. of Computer Aided Verification (CAV) - 33rd International Conference, Virtual Event, July 20-23, 2021*, ser. Lecture Notes in Computer Science, vol. 12759. Springer, 2021, pp. 767–780.

[10] J. Backes *et al.*, "Semantic-based automated reasoning for aws access policies using smt," in *Formal Methods in Computer Aided Design (FMCAD)*, 2018, pp. 1–9.

[11] ——, "Reachability analysis for aws-based networks," in *Proc. of Computer Aided Verification (CAV) - 31st International Conference, New York City, NY, USA, July 15-18, 2019*, ser. Lecture Notes in Computer Science, vol. 11562. Springer, 2019, pp. 231–241.

[12] A. Blaise and F. Rebecchi, "Stay at the helm: secure kubernetes deployments via graph generation and attack reconstruction," in *IEEE 15th Int. Conf. on Cloud Computing, Barcelona, Spain, July 10-16, 2022*. IEEE, 2022, pp. 59–69.

[13] F. Minna, F. Massacci, and K. Tuma, "Towards a security stress-test for cloud configurations," in *IEEE 15th Int. Conf. on Cloud Computing, Barcelona, Spain, July 10-16, 2022*. IEEE, 2022, pp. 191–196.

[14] H. Zhu and C. Gehrmann, "Kub-Sec, an automatic Kubernetes cluster AppArmor profile generation engine," in *14th International Conference on COMmunication Systems & NETworkS, COMSNETS 2022, Bangalore, India, January 4-8, 2022*. IEEE, 2022, pp. 129–137.

[15] M. U. Haque, M. M. Kholoosi, and M. A. Babar, "KGSecConfig: A Knowledge Graph Based Approach for Secured Container orchestrator configuration," in *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, Honolulu, HI, USA, March 15-18, 2022*. IEEE, 2022, pp. 420–431.

[16] 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 2021, August 11-13, 2021*, 2021, pp. 3971–3988.

[17] Liqo, Available: https://liqo.io, Visited: 2024-03-27.

[18] D. Bringhenti, F. Valenza, and C. Basile, "Toward cybersecurity personalization in smart homes," *IEEE Secur. Priv.*, vol. 20, no. 1, pp. 45–53, 2022.

[19] A. M. Zarca, M. Bagaa, J. B. Bernabé, T. Taleb, and A. F. Skarmeta, "Semantic-aware security orchestration in sdn/nfv-enabled iot systems," *Sensors*, vol. 20, no. 13, p. 3622, 2020.