## SecConNet

*Smart and secure container networks for trusted big data sharing*

Shakeri, S.

**Publication date**
2024

**Document Version**
Final published version

SECCONNET SMART AND SECURE CONTAINER NETWORKS FOR TRUSETED BIG DATA SHARING SARA SHAKERI

# SecConNet

## Smart and Secure Container Networks for Trusted Big Data Sharing

Sara Shakeri

# SecConNet
# Smart and Secure Container Networks for Trusted Big Data Sharing

**Sara Shakeri**

This work is supported by the Netherlands eScience Center and NWO under the project SecConNet.


Cover image credits to Tavakoli Amir
Printed by Ridderprint, The Netherlands

ISBN: 978-94-6483-983-8

SecConNet
Smart and Secure Container Networks for Trusted Big Data Sharing

# ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr. ir. P.P.C.C. Verbeek
ten overstaan van een door het College voor Promoties ingestelde commissie,
in het openbaar te verdedigen in de Agnietenkapel
op woensdag 22 mei 2024, te 10.00 uur

door Sara Shakeri
geboren te Arak

***Promotiecommissie***

| | | |
|---|---|---|
| *Promotores:* | prof. dr. ir. C.T.A.M. de Laat | Universiteit van Amsterdam |
| | prof. dr. P. Grosso | Universiteit van Amsterdam |
| *Copromotores:* | prof. dr. ing. L.H.M. Gommans | KLM |
| *Overige leden:* | prof. dr. ir. F.A. Kuipers | TU Delft |
| | prof. dr. R.V. van Nieuwpoort | Universiteit van Amsterdam |
| | dr. Z.A. Mann | Universiteit van Amsterdam |
| | dr. C. Papagianni | Universiteit van Amsterdam |
| | prof. dr. A.D. Pimentel | Universiteit van Amsterdam |

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

# Contents

# Chapter 1

# Introduction

Data sharing is becoming increasingly important in science as well as in industry. Combining shared data allows for richer analysis and deeper insights, such as in many Machine Learning applications. For example, at KLM, the flight safety department owns data monitored on the aircraft fleet in order to improve flight safety and efficiency. Such data also allows maintenance engineers to monitor the health of certain aircraft components allowing their maintenance to be predicted. Moreover, the amount of monitored aircraft data will exponentially grow, hence its benefits for the industry in general. In 2026, the worldwide fleet of 20,000 increasingly advanced aircraft will collect 98 Exabytes per year [1]. This increasing amount of collected data might be of mutual benefit when allowed to be shared within the industry to, for example, develop standards to collect data that prove new ways of scheduling particular maintenance are safe to continue airworthiness.

Sharing nonetheless needs to satisfy constraints regarding the amount or type of data exposed/provided to other parties, e.g., only part of the whole dataset is made available, or anonymization is required before sharing. Private sector enterprises are reluctant to share their data assets with other enterprises unless access and use of such assets happen in a platform that conforms to adequately defined, enforced, and audited *sharing policies* [2–4]. Digital Data Marketplaces (DDMs) constitute a novel framework for secure data sharing among organizations, and they are governed by agreed sharing policies among participating parties. Only if the DDM strictly enforces the sharing policies organizations will trust to share their data on it. More explanation about DDMs is presented in Sec. 1.1

In the SecConNet project[1] we contribute to the development of DDMs focusing on identifying the foundational elements needed for the creation of the infrastructure of a DDM. Our emphasis is on building secure, agile, and dynamic network connections among the endpoints organizations, i.e., domains. In particular, we research if containers can be used to guarantee quality- and policy-based access and use of the shared data assets. We research novel container network architec-

[1]Secure Container Networks: https://www.esciencecenter.nl/project/secconnet

tures, which utilize programmable infrastructures and virtualization technologies across multiple administrative domains.

At the time of starting this work, lots of research had already been published on the security aspects of executing the containers [5–8]. However, there was little work that focused on the security of container network connections. The most basic way to interconnect containers is to use kernel modules. [9] has shown that kernel modules are well-performing ways to support basic interconnections. A more sophisticated method to provide inter-container communication is to create overlay networks. Container overlay networks are virtual networks on top of physical networks that connect containers through virtual links. They play an essential role in policy enforcement between containers. Open questions related to overlay creations have to do with the isolation, exposure, and visibility of containers to each other; as well as performance. The applicability of the available overlay networks in a DDM infrastructure has to be evaluated.

In addition, a data-sharing container network is a dynamic environment. Containers may move across the network, and their addresses may change. Moreover, according to the sharing policies, the rules of container connections may change over time. A container network has to support changes in network configuration when needed. We consider using P4 for building a programmable container overlay and studied which capabilities it can provide to DDM infrastructure.

## 1.1 Digital Data Marketplaces (DDMs)

Fig. 1.1 shows the general architecture of Digital Data Marketplaces(DDMs) and the position of SecConNet in this framework. A DDM is a distributed platform with the goal of creating a trusted data processing infrastructure for data providers and data consumers [2]. In general, the participating organizations in a DDM can share two kinds of assets: algorithms and data [10, 11].

The major consideration in a DDM is that all of the transactions between algorithm suppliers and data suppliers and their customers have to be done based on the agreements established by all participating parties. The agreement, for example, can be based on a GDPR (General Data Protection Regulation) data processing agreement. The sharing policies are according to these pre-established agreements [12, 13]. To make the collaboration of different participating parties in a DDM more efficient, a general semantic model has to be used to describe the sharing policies in a standard way. The policies regulate any movement of data or execution of algorithms. In addition, several other factors like authorization, auditing, and accounting should be taken into consideration. However, the architectures for secure data-sharing platforms and methods for enforcing the high-level policies in the infrastructure in practice are still a matter of research.

As shown in Fig. 1.1 the focus of SecConNet is on the methods of building the infrastructure of a DDM for performing data sharing. Our research focuses on

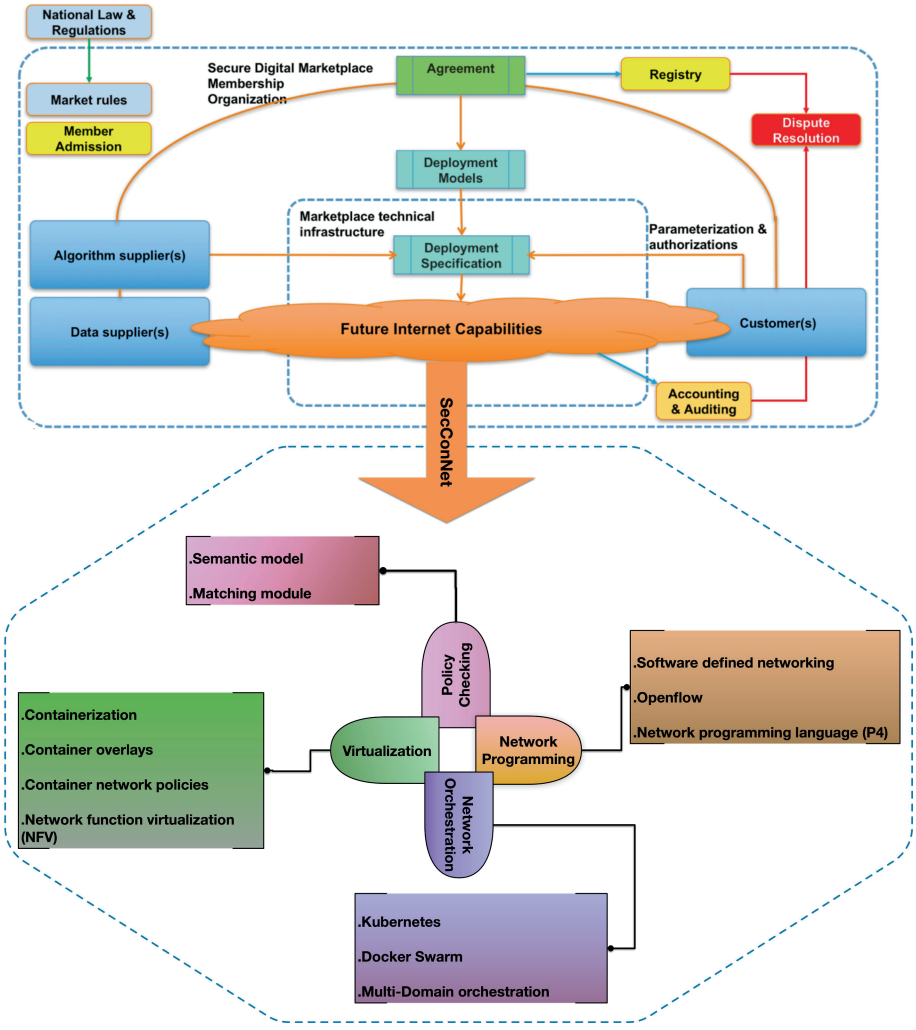



Figure 1.1: Secure Digital Data Marketplace Framework [2] and the position of SecConNet in DDM framework; SecConNet focuses on constructing the DDM infrastructure using container-based technologies.

four primary areas: Policy checking, Virtualization, Network orchestration, and Network programming. Policy checking refers to the process of evaluating and enforcing high-level data-sharing policies within the network infrastructure. In this regard, a semantic model can be defined as a structured framework that allows us to express the sharing policies and their corresponding permission and prohibition

rules. Virtualization mainly refers to using containers as the main building block of the DDM that performs the data sharing. Containerization can provide a consistent and portable runtime environment, enabling efficient resource utilization and easy deployment across different systems. By utilizing containers, we can create isolated network environments for running the sharing requests. Network orchestration explores different methods for orchestrating a container-based network and studies how they can support single and multi-domain network infrastructure. Finally, network programmability allows programming and automating network operations and the management of network behavior. Especially, we investigate how we can provide isolation and enforce the sharing policies between containers by programming the network. We explore how these technologies can improve the construction of a DDM infrastructure.

### 1.1.1 Layers of DDM architecture

Authors in [14] presented the architecture of a DDM in six layers . They defined DDM as a group of actors with specific roles and categorized the roles into different layers, as shown in Fig. 1.2. The lower layers deal with connectivity and infrastructure implementation, and the upper layers deal with sharing agreements and regulations. Each actor is identified by public and private keys. The private

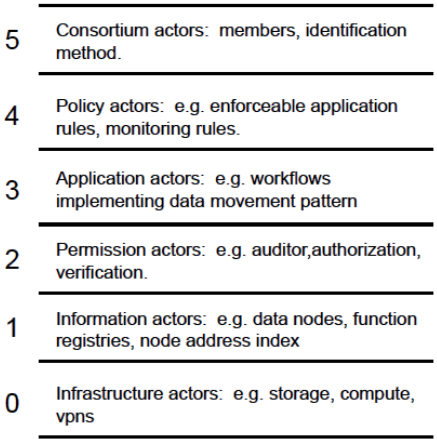



Figure 1.2: Actors and their roles in different layers of DDM [14]

key is used to sign the transactions performed by the actor and the public key

is used to verify the actor of the transaction. In addition, they used this cryptographic addressing for assigning an address to each shared asset. The shared assets are addressed by *publickey/datahash*. Each asset belongs to a data collection that has a public key and each data asset in the collection is identified by its data hash. This information is used for monitoring and auditing the activities of each node or tracking the movement of shared assets.

The focus of SecConNet is on the infrastructure layer (layer0) in Fig. 1.2. This layer deals with the operational aspects of moving data in the network infrastructure. The methods presented in [14] can be combined with the SecConNet outcome to complete building the layers of DDM.

### 1.1.2 Data-sharing related Projects

In the following, we summarize related projects which aimed to build data-sharing platforms for scientific and commercial collaborations. The focus of these projects is on methodologies for establishing agreements and defining the sharing policies between organizations. For implementing the operational infrastructure of these projects, the methodology presented in SecConNet can be considered.

**Data Logistics for Logistics Data (DL4LD)** is going to provide an effective solution that allows organizations to agree on how data is shared and exchanged along with deploying a controllable, enforceable, and goal-oriented method. The project is based on the architecture shown in Fig. 1.1. It offers different components, including agreements, deployment models, suppliers of digital resources, and customers. As depicted in this figure, the agreement must be deployed and mapped on the infrastructure before exchanging the digital resources can take place [2].

**International Data Spaces (IDS)** is a project initiated by the European Union with the goal of building a secure and sovereign system of data sharing in which the data provider can control the use of its data [15]. IDS defines data exchange protocols and contains a data broker, clearing house, identity provider, app store, and vocabulary provider [16]. Data are requested from a data provider, optionally processed, and returned to the data consumer. Several IDS-based or IDS-supporting data exchange systems are coming online. One is the Smart Connected Supplier Network(SCSN) [17]. SCSN defines a universal language to be used by all organizations to provide seamless data in the supply chain and uses a standardized API to send orders, invoices, and dispatch information. It offers a new data standard and technical infrastructure to make sharing data in chains much more efficient.

**Enabling Personalized Interventions (EPI)** aims to develop a healthcare platform based upon a secure and trustworthy distributed data infrastructure to create new, actionable, and personalized insights for providers and patients. It benefits from the collaboration of medical professionals, data scientists, ICT-infrastructure experts, and machine learning researchers [3].

**Green Village Data Sharing Platform** is a project defined by the Green Village with the cooperation of the ICT innovation department of TU Delft and SURFsara. It aims to provide a reliable and highly available sharing platform to exchange data among different organizations [18].

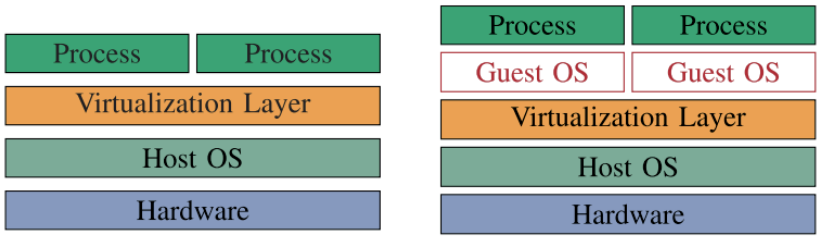
**The Neutral Logistics Information Platform (NLIP)** is a part of the Netherlands' Logistics Top Sector program, a leading platform promoting data exchange in the transport and logistics sector. One of the projects defined in this context is iSHARE. It is an appointment system for identification, authentication, and authorization to share the logistics data in a safe and controlled fashion. This system can be used by all parties which have activity in the logistics sector. Overall, the NLIP project is trying to facilitate the development of tools and digital standards for accessing and sharing data sources to eliminate data sharing barriers and reduce the pressure on physical infrastructure [19].

## 1.2 Containerization

In e-Infrastructures and clouds, there is a transition from virtual machines to containers, such as Docker. Containers are a lightweight virtualization solution that shares the OS kernel [20]. In this thesis, we explore the adaptation of software containers as elements for creating and delivering data sharing application services. In a distributed data sharing platform, containers act on behalf of the participating parties. Containers can provide the required isolation between sharing transactions in a DDM. When a container is deployed on a host, each container's resources, such as its file system and network namespace, are placed in an isolated environment that no other container can access [21]. Containers belonging to different domains must connect to each other securely so data can be transferred from one party to the other. Containers are connected to each other through overlay networks. We look into different configurations of overlay networks to explore how to provide secure connections that preserve the required isolation.

### Comparing Containers with Virtual Machines

In Fig. 1.3 we show the architecture of containers and VMs. Different containers on the same machine share the same OS (Fig. 1.3a) while each VM contains a full set of OS on virtualized hardware and runs a complete copy of the OS (Fig. 1.3b). In VMs, a hypervisor separates the VM from its host. It translates the VM instructions to the instructions that are executable by the host, whereas a container communicates with its host through the system calls. Although these layers in VM setup impose overhead in virtualized applications when compared with applications running on bare metal systems, host OS kernel sharing introduces many security issues, making containers less secure than VMs.




(a) Container architecture (b) VM architecture

Figure 1.3: Architecture of container vs. VM [6]

Authors in [6] classify the possible attacks in a container-based system to three different groups:

- Application to container: In this case, an application could gain control over the container within it's running. Attacks like remote code execution, unauthorized access, or embedded malware can happen in this group.
- Container to host: In this case, a malicious container may have access to confidential data of its host or even target the information integrity. It also can consume the available resources of the target host. Attacks like data tampering, container escape, and DoS may happen in this group.
- Container to container: In this case, one or more containers attack the other containers on the same or different hosts. A malicious container could be able to access confidential data of other containers, learn resource usage patterns, or target another container's availability. Attacks like DoS on other containers, port scanning, ARP spoofing, and MAC flooding can happen in this group.

The focus of SecConNet is on the container to container and container to host type of attacks. We study how to improve the security of container networks by researching container connectivity methods, making isolation between containers when needed, and enforcing the provided network policies.

### Container Orchestration

For orchestrating containerized applications across a cluster, we can use Kubernetes [22]. A Kubernetes cluster consists of one master and several worker nodes. The master manages tasks like scheduling and scaling containers. Each worker node has an agent called Kubelet for communication with the master. Deploying containers goes through the master. After a deployment request, the master will schedule the container to run on its cluster's worker nodes [20, 23]. Kubernetes

uses a concept called pods to manage containers. A pod is formed by one container or a group of containers. Containers in the same pod are scheduled on the same node and share resources like storage, network namespaces (an IP address), and container specific information like image versions. Containers in separate pods are isolated from each other.

## 1.3 Software Defined Networking and P4

In SecConNet we explore the question of how to deploy the network programmability to support agile and secure (scientific and business) Big Data sharing across domains. A data sharing network infrastructure needs intelligent network reconfiguration to support dynamic container network topologies, adaptation to changing network conditions, and instantiating the sharing policies. Software Defined Networking (SDN) is a promising technology that provides programmability by separating the control plane from the data plane. The control plane defines the configuration of the network and routing of the traffic [24]. The data plane physically handles the traffic based on the configuration that is set by the control plane. A centralized controller manages the data plane switches. In SDN architecture, as the controller has a global view of the network based on its communication with the data plane, network management is more efficient compared to traditional networks. Moreover, SDN provides more flexibility as there are programmable switches that use open source network operating systems and the vendor-specific devices are not needed. However, despite the flexibility of SDN, the Openflow protocol, which was defined for the communication between the control plane and the data plane, has a fixed specification [25]. The specification is characterised by a predefined set of header fields according to the Openflow version. On the other hand, the SDN switches are heavily dependent on the controller, which entails lots of communication overhead. To resolve these issues, P4 programmable data plane switches can be used. This allows for transferring some dynamic functions from the controller to the data plane.

P4 is a high-level language that is designed to program the data plane of packet forwarding devices. In this thesis, we use the P4 capability to program and monitor the containers' connections. P4 is more flexible than Openflow offering the capability of defining new headers and as a result making protocol-independent programs that can be deployed on different forwarding hardware. A compiler is used to map those programs to target devices [26].

Fig. 1.4 shows a typical packet processing pipeline in P4. It includes a parser, ingress match-action, egress match-action, and deparser. The parser is a stage that parses the packet according to the header definition that is written by a user in the program. Both ingress and egress match-actions include logical tables that match the packet against the protocol field and determine the required actions to apply to the packet. Actions determine common packet modifications, such

as changing header fields, adding/removing headers, or packet cloning. It performs the modification based on forwarding rules. In P4, it is possible to gather telemetry metadata for each packet, such as ingress and egress timestamps, the latency of the packet, the link utilization, and the routing path of the packet. Information can also be embedded into the packets and removed at the endpoints. P4 supports constants, variables, and registers. Variables have local scope, and they do not keep any state. Registers, on the other hand, keep the state between various network packets.

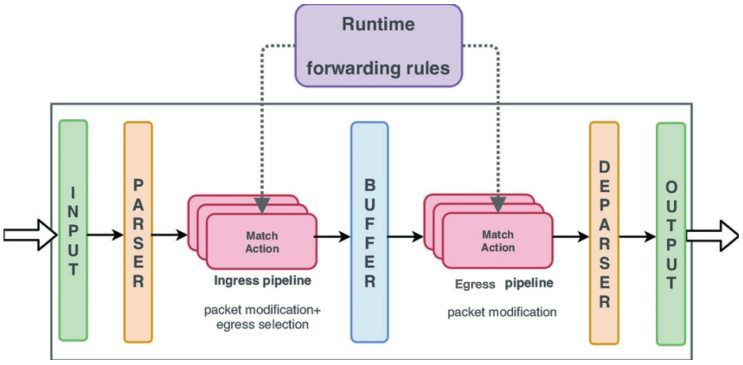



Figure 1.4: P4 processing pipeline [27]

## 1.4 Research Questions

In this thesis, we investigate the capability of containers in providing a secure data sharing infrastructure with our main research question (RQ) as follows:

***RQ: How does a container-based infrastructure guarantee secure and high-performance data sharing among organizations?***

To answer this question, we first need to understand how the sharing policies are defined in a DDM. How do we describe the shared assets, sharing requests, and matching module for checking the sharing requests? Therefore, we define our first sub-question as follows:

***RQ1: How to describe high-level data sharing policies, and how to build a module for authorizing the sharing requests?***

Once we define and describe the sharing policies, we can study how these policies can be enforced in a container-based network. We first need to investi-

gate different available overlay technologies that connect containers and support access control between containers. We study their method of implementation and compare their performance with each other.

As our next step, we study how these available technologies can support the DDM's requirements. We define different connectivity types of containers with different levels of isolation for running sharing requests in a DDM and explain how these connectivity types can be implemented with overlay technologies.

***RQ2: How can overlay network technologies provide the required policy enforcement and isolation, while maintaining quality in a sharing environment?***

*RQ2.1: What are the functionalities of the available overlay technologies for managing container connectivity and enforcing sharing policies?*

*RQ2.2: Can different configurations of available container overlays meet the requirements of a DDM?*

The network infrastructure of a DDM can be managed by a centralized orchestrator (single-domain DDM) or each of the resources in a DDM can be controlled by independent administration systems (multi-domain DDM). In RQ2, we study the functionality of container networks in a single domain environment. In the next research question, we aim our attention on designing a multi-domain container-based network. In a multi-domain DDM each domain administrator manages the connectivity of its own containers while collaborating with other domains to run the sharing requests.

***RQ3: How to build a containerized multi-domain DDM on a programmable network infrastructure to enforce sharing policies?***

For building a multi-domain container-based overlay, we study programmable switches and P4. In this way, the whole connectivity between containers can be managed by the controller of the switch that is under the control of the administrator. After studying the security aspects of our design, in our next question, we study how we can use the capability of the P4 program to be able to monitor the sharing activities between containers.

***RQ4: How can P4-based network capabilities assist the data sharing management system in providing security and maintaining quality?***

By using P4, we can track the container connections and reconfigure the connections between containers when needed.

## 1.5 Thesis at a Glance

In Fig. 1.5, we show an overview of the structure of the thesis. In Chapter 2, we answer RQ1 by introducing the ODRL model for describing the sharing policies and a matching module for handling the sharing requests. After describing the policies, we focus on studying the container overlay networks and constructing a DDM container-based infrastructure for data sharing operations. With RQ2.1, we first investigate the available container overlay technologies to evaluate how they can support creating containers, connecting containers, and enforcing network policies. In Chapter 4, according to the DDM requirements, we define specific container connectivity types, implement the overlay setups based on these connectivity types and compare the implementations from performance and security perspectives (RQ2.2).

In Chapter 5, we research building a multi-domain DDM in which each domain administrator can control its own container connectivity. We use virtual switches to connect containers and use P4 for programming these switches to answer RQ3. We demonstrate how this method can isolate the connection between containers. In Chapter 6, we show how using a programmable switch and the P4 language can assist in monitoring the behavior of containers and their connections to answer RQ4.

## 1.6 Publications

Listed below are the author's contributions to the publications that are used in the chapters.

**Ch 2.** ***Shakeri, S.****, Maccatrozzo, V., Veen, L., Bakhshi, R., Gommans, L., de Laat, C., and Grosso, P. "Modeling and Matching Digital Data Marketplace Policies". In 2019 15th International Conference on eScience (eScience), pp. 570–577.*
Shakeri designed the architecture. Shakeri, Maccatrozzo, and Veen wrote the paper. Maccatrozzo described the SecConNet ODRL semantic model and defined the policy matching module. Veen edited the written work. Grosso advised throughout the study. The remaining authors supervised.

**Ch 3.** ***Shakeri, S.****, van Noort, N., and Grosso, P. "Scalability of Container Overlays for Policy Enforcement in Digital Marketplaces". In 2019 IEEE 8th International Conference on Cloud Networking (CloudNet), pp. 1–4.*
Shakeri designed the experiments and wrote the paper. Van Noort set up and ran the experiments. Shakeri and van Noort analyzed the data. Grosso consulted the study and edited the written work.

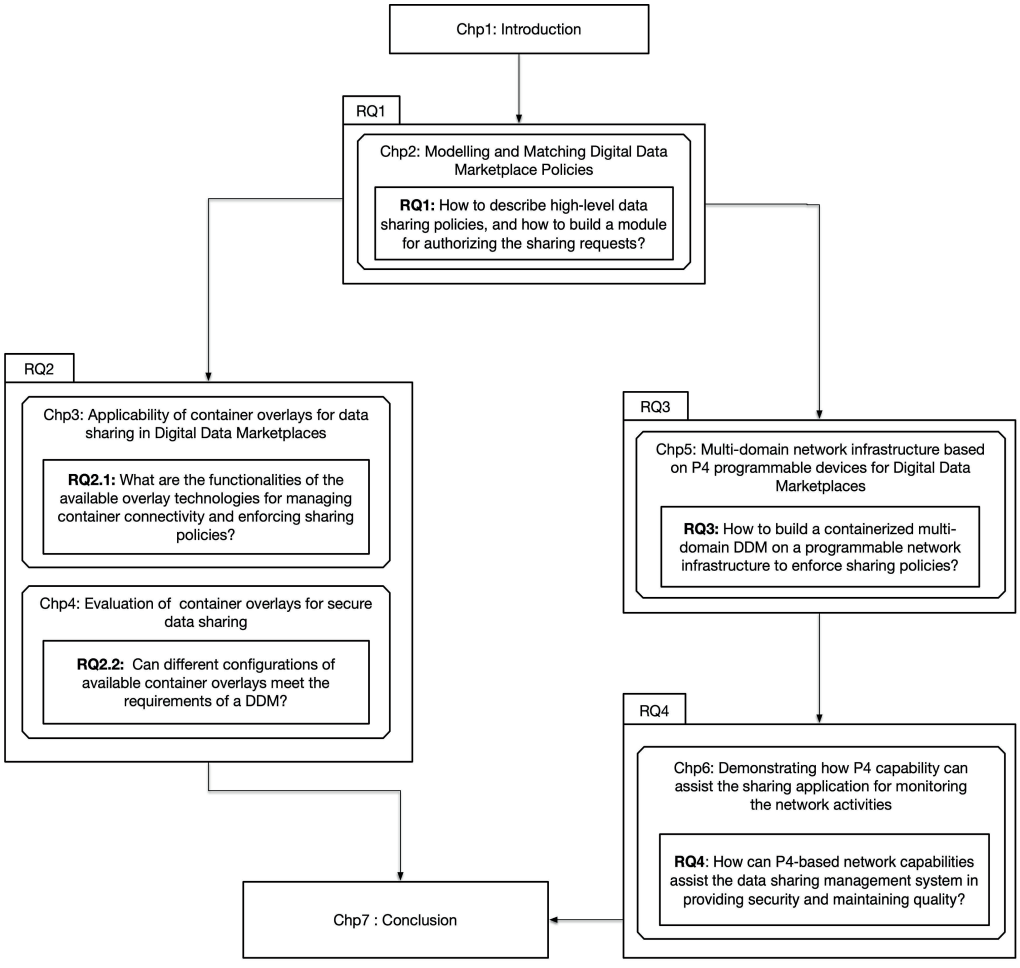



Figure 1.5: The overview of the thesis, including the chapters and research questions

**Ch 4.** ***Shakeri, S.****, Veen, L., and Grosso, P. "Evaluation of Container Overlays for Secure Data Sharing". In 2020 IEEE 45th LCN Symposium on Emerging Topics in Networking (LCN Symposium), pp. 99–108.*
Shakeri designed the architecture, defined the different connectivity types, implemented the overlay setups, and ran the experiments. Veen extended the experiments and analyzed the data. Veen contributed to editing the written work. Grosso consulted the study and edited the written work.

**Ch 5.** ***Shakeri, S.****, Veen, L., and Grosso, P. "Multi-domain network infrastructure based on P4 programmable devices for Digital Data Marketplaces". Cluster Computing (2022).*
Shakeri designed and implemented the proposed method with P4. Veen defined the methodology for calculating request setup time. Shakeri performed the experiments and analyzed the data. Grosso consulted the study and edited the written work.

**Ch 6.** ***Shakeri, S.****, Veen, L., and Grosso, P. "Tracking container network connections in a Digital Data Marketplace with P4". In 2022 International Conference on Computer Information and Telecommunication Systems (CITS), pp. 1-8.*
Shakeri designed the architecture, implemented the method with P4, and wrote the program for running the tracking scenarios. Veen consulted the experiments and edited the written work. Grosso supervised and consulted the study.

**Ch 6.** *Veen, L.,* ***Shakeri, S.****, and Grosso, P. "Mahiru: a federated, policy-driven data processing and exchange system". Submitted to arXiv:2210.17155*
Veen wrote the paper, designed the methodology, and developed the prototype. Shakeri worked on the implementation part. Grosso consulted the study.

# Chapter 2

# Modeling and Matching Digital Data Marketplace Policies

In this chapter, we introduce the concept of sharing policies and sharing requests in Digital Data Marketplaces. We explain the methods for sharing data in DDMs and discuss how they can be built up on a container-based network architecture. This chapter is related to RQ1: *How to describe high-level data sharing policies, and how to build a module for authorizing the sharing requests?*

In particular, we present a semantic model for describing sharing policies by means of semantic web technologies. We use and extend the Open Digital Rights Language (ODRL). The ODRL information model allows for a flexible description of policies by modeling what is permitted and what is not, as well as other terms, requirements, and parties involved. In the DDM, users can submit requests to use specific datasets or algorithms and specify the location of execution. We introduce a matching module to allow for the automatic management of user requests.

This chapter is based on:

**Shakeri, S.**, Maccatrozzo, V., Veen, L., Bakhshi, R., Gommans, L., de Laat, C., and Grosso, P. "Modeling and Matching Digital Data Marketplace Policies". In 2019 15th International Conference on eScience (eScience), pp. 570–577.

## 2.1 Introduction

Digital data marketplaces provide a distributed alternative to data silos run by a small number of large parties or organizations. In a distributed system, control over data is kept with the owner or subject of the data. This benefits privacy, increases business interest of the owner, and reduces market-distorting monopolies. Accordingly, providing a data sharing platform among different participating parties is of paramount importance. However, there are still many challenges about the methods for bringing the required trust and security in DDMs as sharing environments [28, 29]. Arranging appropriate sharing policies in the platform plays a very important role in the functionality of digital collaborations. By describing and implementing these policies, the owners of the data can be sure that only authorized users can access their data and this will increase trust in DDMs.

In this chapter, we use the terms party and organization interchangeably. We define them as the members of the DDM that own data and play a role in determining the agreements for sharing the data.

A sharing policy is a set of access rules that determine permission and prohibition related to a specific object in a specific location. The first major step for developing a secure data sharing platform is describing these rules and regulations in an efficient way. A full description will make the rules more clear and then implementing them in the infrastructure will be more straight-forward.

We built a data sharing platform to support automatic handling of users' requests. We do this by means of Semantic Web technologies. In particular, we use and extend the Open Digital Rights Language (ODRL) to describe sharing policies between participating organizations and to model users' sharing requests.

In this chapter, we present the following:

- An introduction to the DDM sharing application handling mechanism.
- A generic semantic model that leverages the ODRL ontology and extends it specifically for data sharing applications in DDMs.
- The deployment of the semantic model for automatic handling of users' sharing requests in the data sharing platform.

## 2.2 DDM Sharing Policies

A sharing policy is a set of rules including the permitted transmission of shared digital resources. Fig. 2.1 shows examples of policies with possible use cases, which can be defined in a DDM. We categorize the policies into two types: Type A and Type B; Type A policies describe processing a data set using an algorithm when two parties are involved. Type B policies involve two parties supplying data and software to be combined, and a trusted third party (TTP) controlling

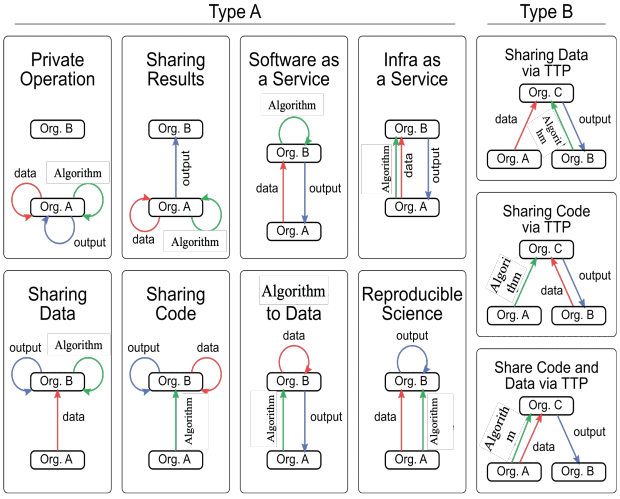



Figure 2.1: DDM policies; Type A with two parties involved; and Type B: with two parties and a TTP (Trusted Third Party) involved;

execution. Corresponding policies may be constructed in a similar fashion as for Type A. Following is an explanation of each use case in this figure.

- **Private operation** is a degenerate case, in which no exchange takes place or is permitted.
- **Sharing results** is when only the result of an operation is shared from one organization to the other organization. This can be part of a chain of policies that allow the use of an algorithm's operation output on data as the input for the next operation.
- **Software as a service** is when an organization uses the algorithm of another organization as a service for performing operations on its data.
- **Infrastructure as a service** is when an organization is using the infrastructure of the other organization for performing the operation. It sends its algorithm and data to the other organization that is providing resources including storage, compute, and network.
- **Sharing data** is when an organization is selling its data to be used by other organizations.
- **Sharing code** is when an organization is selling its algorithm to be used by the other organization.

- **Algorithm to data** is when an organization does not agree to share its data. It takes other algorithms, runs the algorithm on its data, and then sends the results back.
- **Reproduce science** is when the experiments of a scientific use case must be repeated. The other scientists can use the data and algorithm of the specific use case to reproduce the results.
- **Sharing data/code via TTP** is when both the data provider and algorithm provider do not want to send their data or algorithm to the other organization. They send their shared assets to a trusted third party and the results will be used by data provider or algorithm provider according to the use case.
- **Sharing data and code via TTP** is when an organization uses a trusted third party infrastructure as a service and then the output is shared through the TTP.

In a specific scenario, the corresponding rules can be formulated permitting exactly the transmission of a digital object with specific functionality (algorithm or data) to a specific location. For example, "Sharing Data Via TTP" use case in Fig. 2.1, depicts a simple sharing policy including three sharing rules. A rule that allows transmitting the input data from *Org. A* to *Org. C*. The other rule allows the transmission of the algorithm from *Org. B* to *Org. C*, and the last rule allows the output to be transferred to *Org. B*. Any other transfer of data and algorithm is denied.

## 2.3 Request handling in DDM

Digital data sharing and digital collaboration in a DDM rely on the sharing policies and agreements between parties. Ensuring the enforcement of these policies introduces the need for representing an efficient and secure data sharing system in a DDM. Fig. 2.2 shows the mechanism of accepting or rejecting a sharing request in a DDM. By construction, it relies on four components: 1) Sharing application; 2) Sharing policies; 3) Matching Module; and, 4) Container-based network infrastructure. In the following, we describe each of these components.

- **Sharing application** Two kinds of digital resources can be shared in the proposed DDM system.
  - Digital Algorithm: A program that operates on data.
  - Digital Data, including:
    - Input Data: Input data of the algorithm.

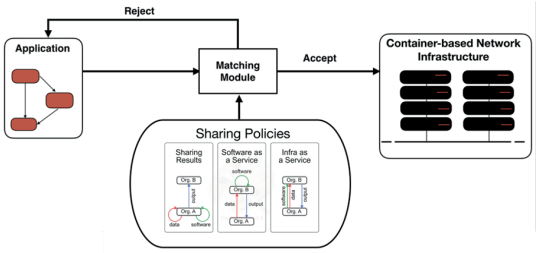



Figure 2.2: Digital Data Marketplace request handling mechanism

∗ Output Data: Result of executing the algorithm.

We consider an application as a set of sharing requests. In each sharing request the operation that the algorithm performs on the input data and the generation of the output is defined. Fig. 2.3 shows an application encompassing two different sharing requests. Here the output of sharing request 1 is used as an input for sharing request 2.

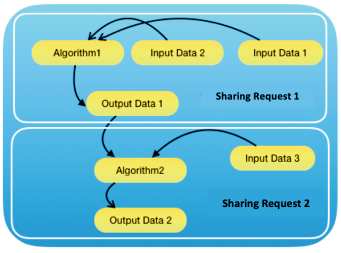



Figure 2.3: Digital Data Marketplace sharing application; it includes two sharing requests and the output of the first sharing request is the input for the second sharing request.

- **Sharing policies** As explained in Sec. 2.2, the policies describe rules to apply how and where data and algorithms may be processed.
- **Matching module** As shown in Fig. 2.2, the matching module automatically verifies whether the sharing application satisfies one of the available sharing policies in the DDM or not. Let's consider a scenario in which a user from a specific organization has a request based on the application in Fig. 2.3. If this sharing application matches one of the available sharing policies, the system will set up the infrastructure based on the sharing policies and will start performing the sharing application. Otherwise, the request will be rejected.

- **Container-based network infrastructure** This is the platform that provides the connections amongst different parties. Several infrastructure architectures are possible. Containers proved their flexibility and applicability in data sharing platforms in other projects like DL4LD and EPI. In this thesis, we looked into how we can build the network infrastructure with containers considering sharing security and performance.

## 2.4 Semantic Model

In this section, we describe the guiding principles of the semantic model, the Open Digital Rights Language (ODRL) ontology, and the extensions needed.

The main goal of our model is to describe how digital resources can be transferred and/or shared among different parties in a DDM.

### 2.4.1 Model requirement

We need a simple language to describe how shared resources can be used by different parties. ODRL is a W3C recommendation language designed to model permissions, obligations, and prohibitions regarding digital resources. Through these rules, the model allows describing the terms of use and reuse of digital content. This model shares many similarities with our requirements, and we needed only a few extensions to customize it for our needs. In the next subsections we introduce the ODRL information model[1] and, propose the extensions for a DDM.

### 2.4.2 ODRL Information Model

The ODRL information model allows for a flexible description of policies by modeling what is allowed and what is not, as well as other terms, requirements, and parties involved. The classes from the ODRL information model that we use are:

- *Asset*: a digital resource, e.g., data, and algorithms.
  - *AssetCollection* (sub-class of *Asset*): a group of Assets with common characteristics. This is used to describe categories of assets. Note that ODRL allows both extensional and intensional definitions of AssetCollections.
- *Action*: an activity performed on an Asset. This is used to describe how the Assets can be shared in the DDM.
- *Rule*: description of an action to be performed over an *Asset*.

[1] https://www.w3.org/TR/odrl-model/

  - *Permission* (sub-class of *Rule*): a description about what is allowed to do with an *Asset*.
  - *Duty* (sub-class of *Rule*): a description of an action a party is obliged to perform with an *Asset*.

- *Party*: entity that assumes a role in a rule. A party is a member of the DDM. The DDM agreements are pre-approved by party.
  - *PartyCollection* (sub-class of *Party*): a group of parties with common characteristics (e.g. employees of the same company).

- *Constraint*: Refinement of an Action, a Rule, and a Party/Asset collection. This is used, for instance, to refine the movement of an asset to a specific digital location.

- *Policy*: a group of rules. Policies are defined by means of rules about the usage of digital resources (i.e., assets).
  - *Agreement* (sub-class of *Policy*): granting of Rules from assigner to assignee parties. This class is used to describe the contract signed by the parties in the DDM.

### 2.4.3 SecConNet semantic model

In Fig. 2.4, we present an example of a policy, representing the policy "sharing data via TTP" shown in Fig. 2.1. The figure includes three main boxes, from top to bottom: Algorithm, Input, and Output. In every box, we can see the Asset description on the left (the Asset Collection box). Asset Collections are used to group together assets that share the same rule set. In particular, given the fact that contracts in DDMs define how assets can be used by the parties and that there could possibly be an unlimited number of parties, with an unlimited number of assets to share, it would be rather inconvenient to define rules for every asset in the DDM. To overcome this issue, we propose to define categories of assets, i.e. Asset Collections, hence a party needs only to declare which category its assets belong to. Every box also includes the description of the actions allowed on that specific asset. Most of these actions are described by a rule of the type permission. In the Output box, there is a rule of the type duty. This rule describes the fact that the output generated by the algorithm has to be moved to location *Org. B.*

Data in DDMs is shared for use as input to other parties' algorithms. While the concept of output (defined as the asset that is created from the output of an action, see Output box in Fig. 2.4) is included in ODRL, the concept of input is missing. When the target of action is an algorithm and the action is "execute", we have no way to define the data used as input. So, we extend the ODRL model by adding the "input" property, as we show in Fig. 2.4 in the input box.

In the SecConNet semantic model, we also use other models, such as the provenance Ontology (PROV-O)[2] to describe and record every action performed in the DDM, friend of a friend (FOAF)[3] to describe parties, and the data catalog vocabulary (DCAT)[4] to describe datasets. However, these models are not included in Fig. 2.4, as the focus of this chapter is on the modeling of the policy.

## 2.5 Matching Module

The matching module's main goal is to allow for automatic management of user requests. In the DDM, users can submit requests to use specific datasets or algorithms, specifying the location of execution. When sending an application, users, have to specify:

- the dataset they want to use;
- the algorithm they want to use;
- the location of execution of the application;
- the location where the results of the application have to be sent.

Finally, using SPARQL queries, the matching module will verify whether the request is doable and approve or reject it.

While the main goal of the matching model is to verify the applicability of the request, it can be easily extended and used to guide the user in the submission of the request. For instance, considering the user's credentials, only the datasets that the user is allowed to use will be listed. Once the dataset is selected, only allowed locations will be shown, and so on, until all the fields are filled in.

We illustrate the behavior of the matching module with the following example.

### 2.5.1 Example

We show the functioning of the matching module by testing the applicability of the sharing policy "sharing data via TTP" in Fig. 2.1. As a contract defining the DDM, we use the one described in Fig. 2.4.

The application request consists of:

- use an *Org. A* dataset (Data1);
- use an *Org. B* algorithm (Algorithm1);

[2]https://www.w3.org/TR/prov-o/
[3]http://xmlns.com/foaf/spec/
[4]https://www.w3.org/TR/vocab-dcat/

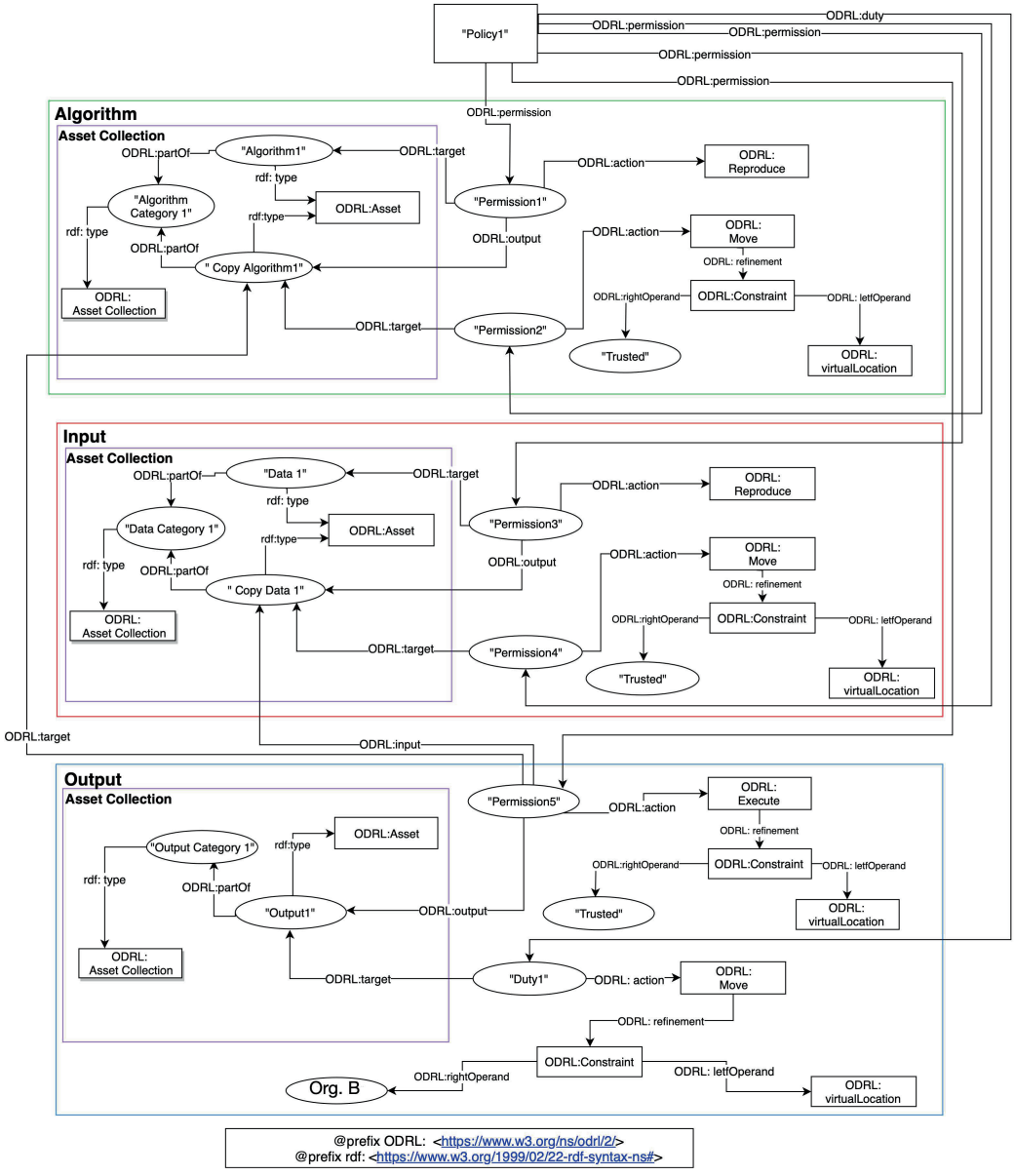



Figure 2.4: An example of a policy. The figure translates the policy "sharing data via TTP" in Fig. 2.1 into our semantic model.

```
SELECT ?location
      WHERE {SECCONET:algorithmAsset/Algorithm1 PROV:atLocation ?location.
      }

SELECT ?location
      WHERE {SECCONET:dataAsset/Data1 PROV:atLocation ?location.
      }
```

Figure 2.5: SPARQL queries to verify where Algorithm1 and Data1 are located.

- *trusted* is the location of execution;
- the results need to be sent to *Org. B*.

With the help of Algorithm 1, we guide the reader through the steps the module performs. The first step is to verify where the dataset and the algorithm to be used in the application are located (see Algorithm 1 line 8). In Fig. 2.5, we show the SPARQL queries to find the location of Algorithm1 and Data1. Location of all the assets in the DDM is recorded using the property *PROV:atLocation*. In case the assets are not in the requested execution location, we need to verify whether they could be moved (see Algorithm 1 line 3). In Fig. 2.6, we show the SPARQL queries to verify whether the assets can be moved to the requested location of execution, after being copied.

**Algorithm 1** Matching module algorithm to check if a sharing request is accepted or rejected according to sharing policies

**Input:** Algorithm1, Data1, Execution Location, Output Location

```
1  Function moveAllowed(Asset, Location)
2  |  if location(Asset) ≠ Location then
3  |  |  return whether Asset may be moved to Location
4  |  end
5  |  return (True)
6  end
7  Output1 = outputOf(Algorithm1, Data1)
8  if moveAllowed(Algorithm1, Execution Location) and moveAllowed(Data1, Execution Location) and moveAllowed(Output1, Output Location) then
9  |  Accept the Request
10 else
11 |  Reject the Request
12 end
```

The last step for completing the matching is to verify whether the results could be moved to the location *Org. B* (see Algorithm 1 line 8-9). In Fig. 2.7 we show the SPARQL query to verify whether Algorithm 1 can be executed in location *trusted* with input Data1 and if the results could be moved to location *Org. B*.

```
SELECT ?moveAlgorithm
         WHERE {
          ?ruleCopy ODRL:action "reproduce".
          ?ruleCopy ODRL:target SECCONET:algorithmAsset/Algorithm1.
          ?ruleCopy ODRL:output ?moveAlgorithm.
          ?rule ODRL:target ?moveAlgorithm.
          ?rule ODRL:action "move".
          ?rule ODRL:constraint ?constraint.
          ?constraint ODRL:virtualLocation SECCONET:location/trusted.
         }

SELECT ?moveData
         WHERE {
          ?ruleCopy ODRL:action "reproduce".
          ?ruleCopy ODRL:target SECCONET:dataAsset/Data1.
          ?ruleCopy ODRL:output ?moveData.
          ?rule ODRL:target ?moveData.
          ?rule ODRL:action "move".
          ?rule ODRL:constraint ?constraint.
          ?constraint ODRL:virtualLocation SECCONET:location/trusted.
         }
```

Figure 2.6: SPARQL queries to verify whether it is possible to move Algorithm1 and Data1 to the requested location.

```
SELECT ?ruleMoveOutput
         WHERE {
          ?rule ODRL:action "execute".
          ?rule ODRL:target SECCONET:AlgorithmAsset/Algorithm1.
          ?rule ODRL:output ?output.
          ?rule ODRL:input SECCONET:DataAsset/Data1.
          ?rule ODRL:constraint ?constraint .
          ?constraint ODRL:virtualLocation SECCONET:location/trusted.
          ?ruleMoveOutput ODRL:action "move".
          ?ruleMoveOutput ODRL:target ?output.
          ?ruleMoveOutput ODRL:constraint ?constraint2.
          ?constraint2 ODRL:virtualLocation SECCONET:location/ Org. B.
         }
```

Figure 2.7: SPARQL query to verify whether it is possible to move the results of the application to the requested location.

## 2.6 Discussion

The example presented in Sec. 2.5.1 shows how we use the semantic model of a policy to perform an automatic matching of the application request with the rules of the DDM.

Mello et al. [30] provides three requirements for infrastructures for sharing clinical trial data: 1) the system must provide sufficiently broad access, 2) it must ensure accountability of all parties involved, 3) and it must be practicable. As both commercial and privacy aspects play a role for clinical trial data, we believe that their criteria apply more broadly, and provide a good reference to measure our approach against.

A system that can accommodate complicated requirements with respect to access to and use of data and algorithms will arguably allow more parties to participate. ODRL is a powerful right description language, and the use of semantic technology makes it easy to extend the ontology if needed. It is flexible enough to be extended to support different types of policies when it is necessary.

From a technical perspective, a DDM consists of users, data and algorithm

providers, and infrastructure providers. To ensure accountability of users, requests need to be matched against the sharing policies specified in the contracts as demonstrated above. Some improvement can still be made in describing data sets and algorithms, however. Algorithms (or specifically implementations) will have to be audited by a human auditor, as automatic software verification is unlikely to be feasible in daily practice. The system should provide digital signature verification to support this as demonstrated before by Cushing et al. [31]. The semantic description can also be used as a machine-readable base for auditing network configuration and performance, in order to ensure accountability of the infrastructure providers. Complex constraints can be verified by querying or theorem proving. Finally, provenance recording can be done for after-the-fact auditing, as well as for reproducibility in a scientific context.

With respect to practicality, automatic request validation as demonstrated here is a necessity for providing a timely response to user requests. The present implementation could be improved upon by support for more sharing policies (as in [31]) and more complex workflows, and by leaving more of the details of where and how to execute to the system, rather than the user. This needs more advanced algorithms for matching and scheduling, however, which we plan to develop. From the system administration perspective, RDF's flexibility allows for putting fewer constraints on users. Our approach allows for the translation of a simple human-understandable concept like the sharing policies presented in Fig. 2.1, into a machine-understandable concept such as the model presented in Fig. 2.4, removing the burden of the translation from the system's administrator.

## 2.7 Related Work

One of the first works in semantic policy management was proposed by Uszok *et al.* in [32]. KAOS is composed of two core ontologies: the actor ontology, which describes people and software subjects of an action, and the action ontology, which provides support for describing actions and related context. There are four types of policies: positive or negative authorization and positive or negative obligations. This model seems to be deprecated.

The Legal Knowledge Interchange Format (LKIF) presented by Hoekstra *et al.* [33] includes a legal core ontology and a legal rule language that can be used to deploy comprehensive legal knowledge management solutions. This model lacks a proper representation of the temporal aspects. Gandon *et al.* present in [34] an extension of LegalRuleML [35] for deontic reasoning on normative requirements and rules. LegalRuleML is a rule interchange language proposed by OASIS, based on RuleML (Rule Markup Language). RuleML is a unifying system of families of languages for Web rules specified through schema languages for Web documents and data. These models allow for very specific logic reasoning, which is not required by our model. We prefer to keep the modeling lighter, to allow for more

flexibility.

XACML (eXtensible Access Control Markup Language) [36], is an OASIS industry standard language for access control requests and policies. It provides a common ground regarding terminology and workflow between multiple vendors building implementations of access control using XACML and interoperability between the implementations. XACML is a general policy language model, while ORDL focuses on modeling digital rights over assets [37].

L4LOD is a lightweight vocabulary for expressing licensing terms in Linked Open Data [38]. Its aim is to provide the means to represent existing licensing models in RDF. However, we are using ODRL which already provides support for RDF representation of the contracts.

Palmirani *et al.* [39] introduce one of the first GDPR inspired ontologies integrated with deontic logic model, called PrOnto. PrOnto allows for privacy and data protection regulation in order to define the legal concepts in legal frameworks and the relationships among them. In the context of medical data privacy, Li and Samavi propose Data Sharing Agreement Privacy Ontology (DSAP) [40]. This ontology is specific to the medical domain, and it is not widely applicable. Our work focuses on sharing datasets, and these models do not allow for modeling the business aspects, e.g. sharing data.

## 2.8 Conclusion

In this chapter, we presented the notion of sharing policies and sharing requests within the concept of DDMs (Fig. 2.1). We introduced an architecture for request handling and explained its components to show how a sharing request can be executed when it is matched with one of the sharing policies. For describing the sharing policies we used a semantic model, instrumenting an example of data sharing in the business domain. We modeled the policies using and extending the Open Digital Rights Language (ODRL) and defined a matching module that checks if a request is allowed to be executed or not.

In Chapter 3, we focus on implementing the described sharing policies in a DDM network infrastructure. We introduce containers as the main components of the network infrastructure that contain the shared assets and explain how the sharing policies are enforced between containers.

# Chapter 3

# Applicability of Container Overlays for Data Sharing in Digital Data Marketplaces

In this chapter, we evaluate the capabilities of container overlays in constructing a container-based DDM, with a focus on enforcing high-level DDM sharing policies. Secure data sharing in a DDM relies on ensuring the implementation of sharing policies in the infrastructure. However, converting high-level sharing policies in a DDM into operational infrastructure is still a matter of challenge. Container-based overlays are a promising approach for making virtual connections between containers and managing filtering rules. There are multiple available container overlay models with different methods of implementation and functionalities. In this chapter, we first show how a sharing request can be mapped to a container-based network and how high-level sharing policies can be translated to container overlay network policies. We then select Calico [41] and Cilium [42] as container overlays that have better support for creating a secure environment and compare their performance by measuring the network throughput when the number of network policies and the number of pods increases. To this end, we set up a container-based sharing platform for emulating a DDM and building a Kubernetes cluster implementing the aforementioned overlay technologies. This chapter is related to RQ2.1: *What are the functionalities of the available overlay technologies for managing container connectivity and enforcing sharing policies?*

This chapter is based on:

**Shakeri, S.**, van Noort, N., and Grosso, P. "Scalability of Container Overlays for Policy Enforcement in Digital Marketplaces". In 2019 IEEE 8th International Conference on Cloud Networking (CloudNet), pp. 1–4.

## 3.1 Introduction

Agreements between DDM participating parties need to be converted into deployment models and specifications in the infrastructure. A container-based solution, e.g., Docker [20] can be deployed to construct the sharing application platform in which the Docker container can act as a participating party in a DDM and support data sharing. However, the security of the connections between containers and the method of policy enforcement has to be investigated.

For executing a data sharing request, the containers of the participating parties have to be connected, and at the same time, all of the high-level sharing policies have to be imposed between them. Container overlay network technologies are the available approaches that put the containers in connection with each other and enforce the network policies between them. In this chapter, we investigate the capability of overlay network technologies in providing a secure connection and enforcing the sharing policies.

There are multiple container network overlay technologies with different features and implementation methods, and selecting the proper one is dependent on the application's workload and its requirements. We consider making a secure connection, ability to enforce sharing policies, and data transfer throughput between containers as the essential KPIs of a sharing platform. We then compare different container overlay network technologies to evaluate how they can fulfill these KPIs.

We set up a container-based sharing platform for emulating a DDM and running sharing requests. We then present the methods by which the sharing policies that are explained in Chapter 2 can be mapped to the network policies of overlays. We explain the method of implementation of the four most popular container overlays: Weave [43], Flannel [44], Calico [41], and Cilium [42] and present their features in supporting the policy enforcement and making secure connections. For throughput experiments, we select Cilium [42] and Calico [41] as we conclude they have the best support for enforcing the policies in the network. To compare Cilium and Calico performance, we observe how their throughput in data transfer scales well with an increasing number of policies and pods. The main contributions of this chapter are:

- Presenting an overview of available container overlay technologies and their capabilities in supporting container-based DDM
- Demonstrating how the sharing policies can be mapped to the network filtering rules between containers using overlays
- Comparing the data transfer throughput of Calico and Cilium with an increasing number of policies and pods

## 3.2 Container Overlay Technologies

The concept of an overlay network is not a new idea. It is a virtual network on top of the physical network to build virtual links among containers [45, 46]. In container network overlays, each container has a private IP address for its communication. The mapping between the container's private IP addresses and their host IP addresses will be saved in a distributed key-value (KV) store, that is accessible by all joined nodes in the overlay network. Most overlay technologies use etcd as KV store [47]. When a packet is sent to another container in a different physical machine, the overlay network uses the KV store to find the destination host IP address. The original packet will be encapsulated in a packet with the host source and destination IP address and then it will be sent to the destination. The encapsulation can be done in different layers depending on the implementation of the overlay technology.

There exist various implementations of overlay networks for Docker containers that are also integrated with Kubernetes. However, their capability of deploying the sharing policies in a DDM has to be investigated. In this section, we introduce four popular container overlay technologies: Weave, Flannel, Cilium, and Calico. We explain their method of making the connection between containers and enforcing network policies.

### 3.2.1 Weave

Weave is a virtual network solution developed by Weavework [43]. Weave deploys a weave router container on each Docker host. Weave creates a mesh overlay network between each of the nodes in the cluster, allowing for flexible routing between participants. It uses a custom encapsulation method, and each packet is encapsulated in a tunnel protocol header and sent to the destination host, where the header is removed. The communications among weave endpoints can be encrypted using the NaCl crypto libraries [48] to enhance data security. Weave uses iptable for implementing the network policies and supports Kubernetes network policy enforcement.

### 3.2.2 Flannel

Flannel is a virtual network developed by CoreOS [44]. It inserts a virtual network interface, *flannel0*, between docker bridge and the physical interface of the docker host and gives a subnet to each node to allocate IP addresses internally. Containers within the same host can communicate using the Docker bridge, while containers on different hosts will have their traffic encapsulated in VXLAN packets for routing to the appropriate destination. A distributed KV store, etcd [47], is maintained to store network configuration and address mappings. Project flannel is focused on networking and does not support network policy enforcement.

### 3.2.3 Cilium

Cilium is an open source technology that is developed for securing the network connectivity of the Linux containers which are being managed by Docker and Kubernetes [42]. It leverages eBPF [49] as a technology for filtering and security policy enforcement. In the Cilium architecture, the Cilium agent runs in the user space of the host which interacts with the orchestration systems like Kubernetes. It will set up the connectivity and networking among containers in a cluster and also is responsible for deploying the network security policies. Linux kernel eBPF runs the bytecodes which are compiled by Cilium in order to enforce the security and policies over the traffic among containers from within the kernel. In Cilium, all the packets which are sent by a container to an endpoint in the overlay network, are encapsulated by VXLAN. Cilium uses IPsec for encrypting the container to container traffic.

The policy enforcement in Cilium is based on the labels that are dedicated to the containers or the pods. Each container or pod has its own label and all of the rules and policies are based on these labels. Defining the identity of each container based on labeling provides dynamic policy enforcement and makes the security independent of addressing. Moreover, Cilium provides the HTTP layer filtering in addition to L3-L4 operation. Cilium has full support for Kubernetes Network Policy based on a modern identity-based implementation built entirely in eBPF. Extensive visibility functionality eases problem troubleshooting and compliance monitoring. It supports simulation and policy audit in a way that the effect of network policy changes can be inspected before dropping live traffic.

### 3.2.4 Calico

Calico is used to create overlay networks and establish connections between containers across the nodes [41]. The Calico node agent consists of three main components: *felix*, *bird*, and *confd*. *felix* is responsible for providing the connectivity and policy enforcement by programming the routes and iptable on the host. *bird* distributes the routing information which is programmed by *felix* between hosts as a Linux BGP agent. In case of any changes to the BGP configuration in the etcd datastore *confd* triggers *bird* to reload the changes on each host.

Generally, there are two methods for bringing connectivity between multiple hosts in Calico: BGP and IPIP. In the first method, BGP should be enabled on the underlying router so that the nodes can be added as the BGP peer in the cluster. On the other hand, in IPIP the original packet with the container IP addresses will be encapsulated at the network layer with the host IP address. Calico uses IPsec for encrypting the traffic between hosts [41]. In Calico, the policy will be translated to the host iptable rules. *felix* is responsible for deploying the policies in the iptable. All pods' traffic traverses iptables rules before they are routed to their destination. Calico has an anomaly detection feature analyzing

Table 3.1: Features of four different container overlay technologies

| Overlay technology | How it works | Policy enforcement method | Encryption | Ingress/Egress policy | Other features |
|---|---|---|---|---|---|
| Weave | Custom encapsulation | iptable | NaCl | Yes | Troubleshooting [50]<br>Auditing [51] |
| Flannel | VXLAN | – | No | No | – |
| Cilium | VXLAN | eBpf | IPSec | Yes | Control on application protocol [42]<br>Simulation and audit [52]<br>Flow logs at L3-L7 [42]<br>Troubleshooting [53] |
| Calico | IPIP/VXLAN | iptable | Wireguard | Yes | Policy monitoring [54]<br>Attack mitigation [55]<br>Troubleshooting [56]<br>External firewall integration [57]<br>Image assurance [58] |

network activity and identifying anomalous and suspicious behavior detected in the cluster.

Table 3.1 shows a summary of the features of these technologies. Except for Flannel, the other technologies support the policy enforcement. In the next section, we present the way that the sharing policies in a DDM can be mapped to the network policies of these overlay network technologies.

## 3.3 Sharing policy enforcement in DDM using container overlays

According to the agreements established in a DDM, various sharing policies can be defined for sharing the data and algorithms. Four of them are depicted in Fig. 3.1. Each sharing policy determines the permitted and prohibited traffic flow among the organizations, which can be translated to network policy and then implemented in the container-based infrastructure. Let's consider scenario

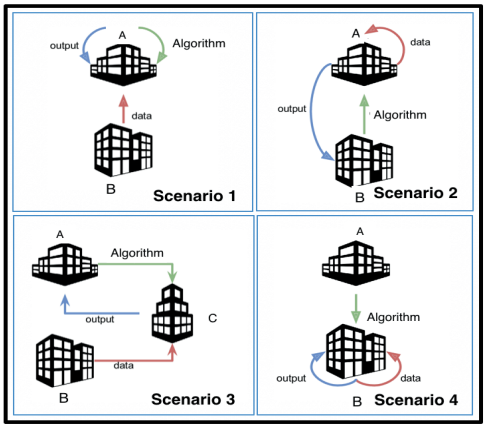



Figure 3.1: Examples of sharing policies in a DDM

1 in Fig. 3.1 as the target scenario which should be implemented in the infrastructure. It shows the algorithm of organization A will be executed on data of organization B at the location belonging to organization A, and then organization A will use the output of the operation. To implement this, it is necessary to define a policy in the network that permits the connections from organization B to organization A on the required port numbers. As an example, Listing. 3.1 shows the Kubernetes network policy that allows the *ingress* connection from organization B to organization A on port 80 of protocol TCP. Let's denote that only the connections that are defined in the policies are allowed. Therefore, a connection that does not match any of the defined policies will be rejected. However, running the policy imposes an overhead on the network and may affect the network performance, especially when the number of policies increases. It becomes therefore very important to quantify **policy scalability**.

```
kind: NetworkPolicy
metadata:
  name: Network_Policy_Example
spec:
  podSelector:
    matchLabels:
      id.pod : OrganizationA
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
     - podSelector:
         matchLabels:
           id.pod : OrganizationB
    ports:
     - protocol: TCP
       port: 80
```

Listing 3.1: A container network policy, specifying connection rules between organization A and organization B

Another important factor is the network performance when the number of pods increases. In many cases in a sharing platform, it is necessary to run multiple numbers of pods at the same time to handle the applications' requests. Therefore, it is worth it to investigate the DDM performance in handling concurrent communications between multiple pods, i.e., **pod scalability**.

## 3.4 Experiments

Considering the capabilities provided by each of the four technologies, we can conclude that Calico and Cilium are more suited to our purposes than Flannel and Weave. Especially useful for us is the fact that Calico provides an attack mitigation method [59] and Cilium supports L3-L7 filtering rules [60]. Therefore, to compare the performance of these two technologies we set up a Kubernetes container cluster deploying Cilium and Calico and evaluate them in terms of policy scalability and pod scalability.

### 3.4.1 Experiment Setup

We have set up the sharing platform as a Kubernetes cluster, utilizing three VMs running on separate physical machines. The VMs are connected via 10Gbps Ethernet link and each VM is running Ubuntu 18.04 and Linux kernel 4.15 as the operating system and has access to one CPU core. The MTU of the network interfaces is 1500 bytes. One of the VMs is functioning as a master node (VM 1). The other two VMs are joined to the cluster as worker nodes and run Kubernetes pods (VM 2 and VM 3). We use Cilium version 1.6 and Calico version 3.5 as overlay networks in the experiments. Fig. 3.2 depicts the experimental setup.

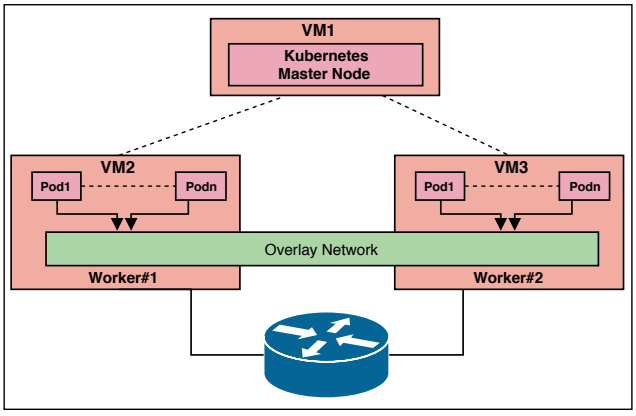



Figure 3.2: Experimental setup using container overlays

In our experiments, we use iperf3 as the throughput testing tool [61]. Listing. 3.2 shows how the servers and clients started on the respective VMs.

```
# Start iperf3 server
$ iperf3 --server --json --logfile result.json

# Start TCP client
$ iperf3 -c <IP pod-x> -b 0 --json --logfile result.json -t
   200
```

Listing 3.2: Iperf3 commands to start a server and connect a client.

### 3.4.2 Basic Experiments

We define two basic experiments. In the first we measure the TCP throughput of the network between two worker nodes of the setup in Fig. 3.2 (VM-to-VM Experiment). In the second experiment, we measure the throughput between containers on top of the VMs when Calico or Cilium are deployed as the container network overlay (Container-to-Container Experiment: C-to-C). Figure 3.3 shows the results of the experiments, both for the VM-to-VM and the C-to-C cases. We can observe that the VMs reached a TCP throughput of 9.31 Gbit/s. However, we see that there is a throughput loss when using containers. Cilium only reaches 20.9% TCP throughput of what the VMs reached. Calico does better with 26.7%.

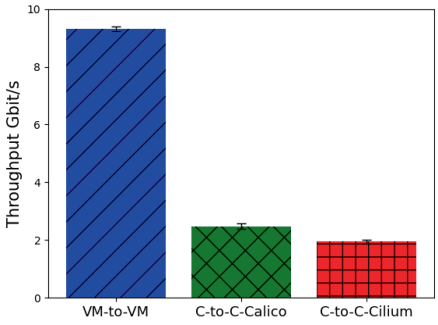



Figure 3.3: The TCP throughput in the experiment setup of Fig. 3.2: from VM 2 to VM 3 (VM-to-VM), from a container on VM 2 to a container on VM 3 using Calico (C-to-C Calico), and from a container on VM 2 to a container on VM 3 using Cilium (C-to-C Cilium).

We were interested in understanding the substantial throughput degradation in both Calico and Cilium. We determined that this was related to **Linux interrupts**; In Linux, sending and receiving packets is partly handled by kernel

interrupts [62]. Linux's interrupt handler knows two types of interrupts: hardware interrupts and software interrupts. After a packet is sent or received by the NIC, a hardware interrupt is raised. This hardware interrupt will also cause a software interrupt. Linux uses softirq as a software interrupt. On the sender side, softirq is used to free up the resources utilized by the packet after sending the packet. On the receiver side, after the NIC receives a packet, it is up to softirqs to move the packet through the network stack and get it to the application socket. In the case of TCP, softirq plays a role in buffer management when buffering is required, it also has the task of handling the ACK messages. If the softirq processes need more time to finish, they are moved to the ksoftirqd thread [62]. Thus when the CPU usage of ksoftirqd increases, it takes more time to send and receive a packet.

Calico and Cilium use IPIP and VXLAN respectively for encapsulating the packets and transferring data between containers. **Encapsulation and decapsulation** cause overhead in processing the network packets. On the receiving side, after decapsulation the packet must start all over again by being processed through the network stack, introducing more CPU usage of ksoftirqd.

To show the impact of ksoftirqd, we measured the mean CPU usage of ksoftirqd threads during the execution of an iperf3 measurement between two VMs and two containers using Calico and Cilium. Fig. 3.4 shows the results. The containers using Calico and Cilium consume much more CPU via ksoftirqd than the VMs. This reflects back in the performance degradation we saw in Fig. 3.3. We can also see a difference between the CPU usage of Calico and Cilium. This is likely due to the difference between the encapsulation methods. It is explained more in Sec. 3.4.3.

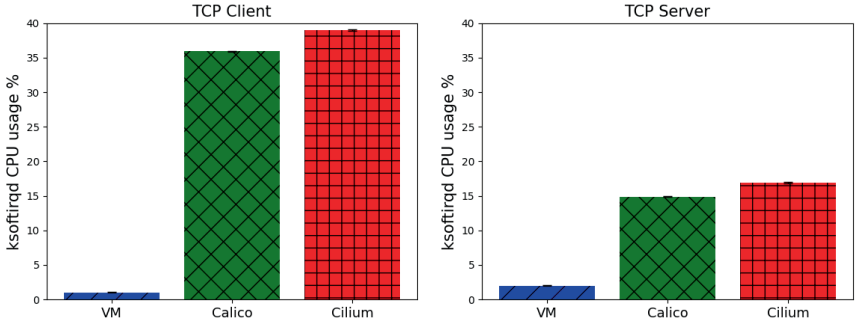



Figure 3.4: The mean CPU usage of ksoftirqd threads during an iperf3 TCP measurement. This is done between two VMs and between two containers using Calico and Cilium.

### 3.4.3 DDM Related Experiments

When the need for transferring more data between organizations in a container-based DDM increases, more pods are required and consequently, the number of policies between those pods will increase. To compare Calico and Cilium performance and evaluate their applicability in DDMs we measure the policy and pod scalability in our experimental setup.

**Policy scalability experiment** – We evaluate the network policy scalability by measuring the throughput of the network when the number of policies increases. A policy is defined using the generic template introduced in Listing. 3.1. To create a new policy we change the port number used to filter traffic. The policies are ordered in such a way that the incoming traffic will match the last policy. This ensures that all incoming traffic has to traverse the entire policy list before making a decision.

Fig. 3.5 depicts the result of our experiment measuring the throughput of the network from a running pod in a worker node to another when the number of policies goes from zero to 4000 policies. We increase the number of policies in steps of 100 from zero policy to 1000 policies and in steps of 1000 from 1000 policies to 4000 policies. We run each experiment three times using iperf3 for a duration each time of 300 seconds. We take the average of the three runs as the representative throughput.

As the results in Fig. 3.5 show overall both Calico and Cilium perform well in policy scalability. There is 6% throughput degradation in Calico and 4% throughput degradation in Cilium.

**Pod scalability experiment** – The pod scalability will be tested by increasing the number of pods and measuring the throughput in the network. In every single experiment carried out in our setup (see Fig. 3.2), we run a certain number of pods in one VM functioning as a client and the same number of pods in the other VM functioning as a server. We increase the number of pods in each VM in steps of 1 from 1 to 10 pods and in steps of 10 from 10 to 40 pods. For each experiment, we have calculated the total throughput that is achieved in the network and compared it to the expected throughput. The expected throughput is the maximum throughput that can be achieved in the network when running one single client-server stream. We run each experiment three times using iperf3 for a duration each time of 300 seconds and take the average of the three runs as the representative throughput.

The result of pod scalability is depicted in Fig. 3.6. The throughput achieved with Calico is higher than the one with Cilium. However, there is throughput degradation for both Calico and Cilium. Calico has 39% Gbps throughput loss, and Cilium has 51% drop for 40 pods. We can conclude that both Calico and Cilium do not scale well with an increasing number of pods.

To explain the difference between Cilium and Calico performance in pod and policy scalability, we consider two major implementation differences between Cil-

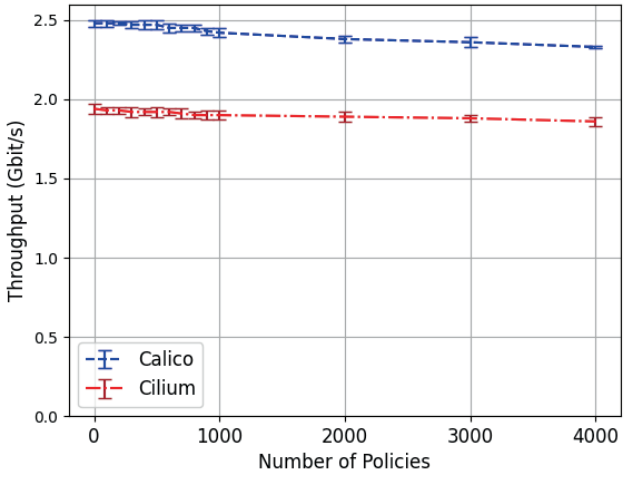



Figure 3.5: The mean throughput as a function of the number of network policies using Calico and Cilium

ium and Calico:

- **Encapsulation method** As seen in Sec. 3.4.2, Calico has always better throughput than Cilium. This is mainly because of the different encapsulation methods. Calico uses IPIP which imposes less overhead than VXLAN. In fact, VXLAN will increase the packet size more than IPIP and consequently, because of the limited MTU in the network interface, the packet will be fragmented. Therefore, the number of packets that should be handled increases. This increases the CPU usage in the host and creates additional interrupts in the network interface.

- **Policy translation method** As is shown in Fig. 3.5, the throughput drop in Calico is higher than Cilium as we increase the number of policies. The reason is the difference in the policy translation method and their way of filtering the packets. Cilium uses BPF program running inside the host kernel for policy enforcement. It defines the identity of each pod based on its label and then uses a hash table for storing the policies. However, Calico will translate the policies to the iptable rules of the host. In Cilium increasing the number of policies does not significantly affect the filtering process due to the fact that Cilium checks the policies for every packet with a hashtable lookup. However, in Calico as the rules are translated to the

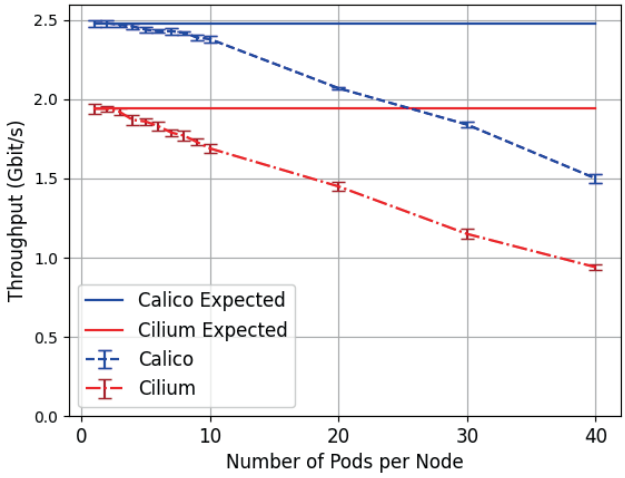



Figure 3.6: The cumulative throughput as a function of the number of pods

host iptable rules, more rules should be checked by increasing the number of the policies and this will negatively affect the throughput. Therefore, the throughput reduction will be more in Calico than in Cilium.

## 3.5 Related Work

There are multiple studies and projects about setting up a secure sharing platform. For example, iSHARE [63] is an appointment system for identification, authentication, and authorization to share the logistics data in a safe and controlled fashion. This system can be used by all parties which have activity in the logistics sector.

DL4LD project [2] focuses on providing an effective solution that allows organizations to agree on how data is shared and exchanged along with deploying a controllable, enforceable, and goal-oriented method. Here we specifically investigate running a container-based data sharing platform utilizing container overlay network technologies.

Moreover, there are multiple works that investigate container network technologies and evaluate their performance. For example, in [5] authors presented an empirical study about different methods of container networks. They conducted a qualitative comparison of available methods regarding their different levels of isolation and overhead. Then, they have done lots of experiments for evaluating

the performance of different container networks.

The authors in [64], investigated the possibility of deploying Osmotic Computing environments in order to deploy distributed microservices among Cloud, Edge and IoT devices. In particular, they deployed two different microservices: FTP and CoAP inside Docker containers orchestrated by Kubernetes. In order to find the best overlay solution they performed scalability analysis on four different network overlays: OVN, Calico, Weave, and Flannel. Their results show the difference between the overlays is not substantial.

Also, [7] represented a performance evaluation of three different container network technologies: Flannel, Docker Swarm, and Calico. Moreover, they presented a comparison of the configuration setup of each implementation.

[8] proposed a method for specifying IP addresses to the containers utilizing EVPN and ILA as overlay technologies. In addition, the authors evaluated Cilium/eBPF performance in network filtering, specifically in multi-tenant environments.

All of the above studies have considered different implementations of container networking and evaluated each of them. However, in this work, we focus on the applicability of overlays in a DDM and show how a sharing request can be mapped to a container network and how the sharing policies can be deployed between containers. We investigate the capability of overlays in providing more security in the network and compare their performance when the number of policies and pods increases.

## 3.6 Conclusion

In this chapter, we presented how a sharing request can be mapped to a container based network and how the sharing policies in a request could be translated to the network policies between containers. As the connection between containers and enforcing the network policies happens by container overlays, we studied the implementation method of four different available overlay technologies: Flannel, Weave, Cilium, and Calico. We first compared these overlays from security aspects. With the focus on the capability to support the network policies and the features they can have for providing better security we chose Calico and Cilium. We then compared them in terms of the throughput of the network with increasing the number of policies (policy scalability) and the number of pods (pod scalability) in a Kubernetes cluster.

We established that both Cilium and Calico scale well in policy scalability as a function of the number of policies in the network. In fact, the number of policies applied to a cluster has little effect on the throughput, especially when Cilium is used. However, there is a substantial throughput degradation in both technologies in pod scalability. This introduces a challenge for deploying these technologies when there is a requirement for running a large number of pods. It

depends on the application of the data sharing platform whether the throughput losses are acceptable. Overall, Calico had better throughput in all experiments. Therefore, for our next experiments we choose Calico as the overlay technology.

In Chapter 4, we define different connectivity types for different types of sharing requests in a DDM. We then use the available container overlay technologies to implement the network according to these connectivity types. We compare the isolation level of each implementation and perform experiments to measure their performance.

# Chapter 4

# Evaluation of Container Overlays for Secure Data Sharing

In this chapter, we define possible container connectivity methods in a container-based DDM by deploying available overlay technologies. Our focus is on improving security by controlling inter-container connectivity and providing isolation between sharing requests of a DDM. To this end, we implement three different overlay setups according to container connectivity types and study how they provide isolation between containers. We investigate which type of attacks are possible in each method. We also measure the time required to complete a sharing request in each method. The results show that providing higher isolation between containers can lead to a longer time for completing a sharing request. This chapter is related to RQ2.2: *Can different configurations of available container overlays meet the requirements of a DDM?*

This chapter is based on:

**Shakeri, S.**, Veen, L., and Grosso, P. "Evaluation of Container Overlays for Secure Data Sharing". In 2020 IEEE 45th LCN Symposium on Emerging Topics in Networking (LCN Symposium), pp. 99–108.

## 4.1 Introduction

Data exchange entails copying data (sub)sets and algorithms from one system to another. In a container-based DDM, each sharing request consists of a number of containers (depending on the number of participating parties) that are connected together for transferring data or algorithms. However, due to a lack of isolation in container-based setups, in a sharing environment constructed from containers, data confidentiality is at risk. Providing more isolation in a container-based network will decrease the probability of specific kinds of attacks, and this will improve network security [6].

Two types of isolation can be provided in a container-based network: 1) isolation between containers and their host and 2) isolation between containers themselves [6]. Multiple studies have focused on bringing isolation between containers and their host suggesting hardware and software solutions by utilizing Linux kernel security modules [65–67]. However, an in-depth study for providing isolation between containers themselves is missing. This is of prime importance for improving security, especially in a data sharing platform. In fact, lack of isolation between containers of distinct sharing requests may lead to different kinds of attacks between containers like ARP spoofing or MAC flooding that will affect the shared data confidentiality [68, 69]. In this chapter, we define three different container connectivity types in a container-based DDM with the goal of improving security by controlling inter-container connectivity and providing isolation between sharing requests.

We then implement three different overlay setup methods according to container connectivity types using Kubernetes, Calico, and Docker Swarm technologies [22, 41, 70]. We study how each implementation provides isolation to improve security between containers of sharing requests. We also take the performance of each overlay setup into consideration by measuring the required time to complete a sharing request in each method.

More specifically, the main contributions of this chapter are:

- Presenting a container-based architecture for data sharing infrastructure that can translate high-level DDM policies to respective network configurations and run data sharing requests in practice.

- Defining three types of container connectivity with the goal of improving security through providing higher isolation between containers of sharing requests. The connectivity types are then implemented by container overlays and are called Overlay per DDM, Overlay per request, and Overlay per group.

- Studying the security aspects of the proposed methods with respect to how they are secure against inter-container types of attacks. In addition, we

present a performance evaluation regarding the time taken to complete a sharing request.

## 4.2 Container-based DDM architecture

DDM sharing policies describe agreements between parties. Each party in a DDM can define a desired service from the DDM as a ***sharing request***. A sharing request can be defined in the same format as DDM sharing policies including participating parties and the requested flow of data or algorithm (see Sec. 2.2). We define two types of sharing requests based on the number of parties that are involved in that sharing request.

- *Type A* for requests that two organizations are involved.
- *Type B* for requests that three organizations are involved. In this case, two organizations share their data or algorithm with a third organization which is called a trusted third party (TTP).

The organization that has control over the connectivity between containers of a request is defined as the ***owner*** of the request.

Fig. 4.1 shows the proposed architecture for constructing a container-based DDM. A sharing request is executed by creating requests' containers and setting up the connection between them by means of overlay setup considering DDM sharing policies of that request. Executing a sharing request is handled in three main steps, and each step contains different modules in the architecture.

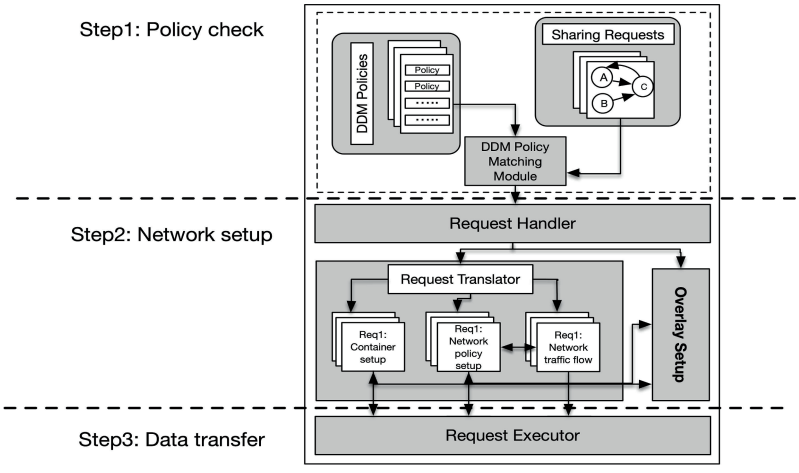



Figure 4.1: Container-based DDM architecture

- **Step1 - Policy check** that verifies if the request is matched with available policies in DDM. It is handled by DDM Policy Matching Module in the architecture.
- **Step2 - Network setup** that builds the elements in the network for executing a sharing request. It is handled by Request Handler, Overlay Setup, and Request Translator modules.
- **Step3 - Data transfer** that transfers data between containers of the sharing request according to the traffic flow and is done by the request executor module in the architecture.

  The functionality of each module is explained in the following:

  - **DDM Policy Matching Module**: A sharing request needs to be matched with one of the pre-established DDM policies that are described by Open Digital Rights Language (ODRL) [71] in a DDM as explained in Chapter 2. When a sharing request comes in, the policy matching module searches for a policy that is matched with the requested sharing scenario. If a match is found, the request will be authorized to be executed on the infrastructure and sent to the request handler. Otherwise, the request will be rejected at this level.

  - **Request Handler**: Request handler is responsible for orchestrating the execution of necessary modules for running sharing requests, i.e., *overlay setup* and *request translator*.
  - **Overlay Setup**: In the proposed container-based DDM, overlays provide the connection between containers. Different types of connectivity between containers can be implemented by means of overlays that lead to different levels of isolation between sharing requests. In this work, we propose three overlay setups in a DDM that are described in Sec. 4.4.
  - **Request Translator**: For implementing a DDM in practice, the high-level described DDM policies and sharing requests should be translated to the network configurations. Therefore, after receiving a sharing request, its corresponding ***containers***, ***network policies***, and ***traffic flow*** will be generated in the Request translator. Fig. 4.2 shows an example of the traffic flow of a sharing request between three organizations A, B, and C.

  - **Request Executor**: In this module, the overlay is set up, containers and their corresponding policies are created on the overlay, and the request is executed by sending traffic between containers based on the

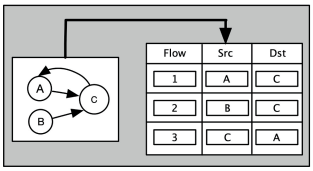



Figure 4.2: A sharing request's traffic flow.

requested traffic flow. As an example, for the request shown in Fig. 4.2, three containers will be created for this request, the network policies will be deployed to allow required data transfer, and then data will be transferred between these containers according to the steps of the traffic flow.

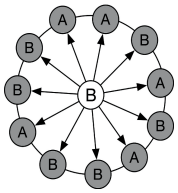
## 4.3 Container Connectivity Types

With the goal of improving security in a DDM by isolating sharing requests' containers, different possible container connectivity types in a DDM have to be investigated. As a matter of fact, with less connectivity comes a higher level of isolation and consequently, the security will be improved [6].

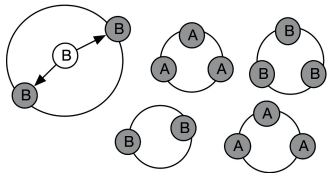
In this section, we define three different container connectivity types in a DDM based on the level of accessibility of a container to other containers. We do this by considering that a container's network accessibility depends on overlay network configuration.

**DDM connectivity (Fig. 4.3a)**: In this type of connectivity, all of the containers are allocated to one overlay network and therefore, all of them are connected together. In this type, if for example, a container of a request of organization B is compromised (the white circle in the center) all of the other containers in the DDM are at risk. This type has the highest attack surface and the lowest level of isolation between containers. For running sharing requests of this type just one overlay needs to be set up, and all the containers are joined to this overlay.

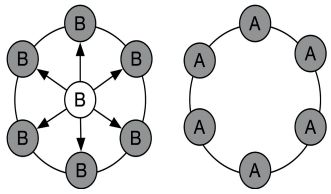
**Request connectivity (Fig. 4.3b)**: In this type, there is one overlay for each single sharing request. Therefore, only the containers related to the same request are in the same overlay. As is shown in Fig. 4.3b, the connection between containers is automatically limited by just being in the same overlay. In this connectivity type, if a container of a sharing request is compromised, only the container of that sharing request will be affected. Therefore, this method has the highest level of isolation. However, in this type, one overlay has to be set up for each single sharing request to connect its containers together. This will increase the network setup time and negatively affects the time of completing a sharing request. By increasing the number of sharing requests, this item can disrupt the




(a) DDM Connectivity type: all of the containers are connected to each other.




(b) Request Connectivity type: containers of the same request are connected to each other.




(c) Group Connectivity type: containers of the same group are connected to each other.

Figure 4.3: Three container connectivity types in a DDM

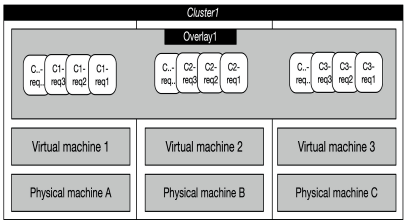
network availability that in delay-sensitive requests is not negligible.

**Group connectivity (Fig. 4.3c)**: As an intermediate type between the two previous ones, we define the Group connectivity type. In this type, the requests and their respective containers will be assigned to different groups. Containers related to the same group will be in the same overlay network. For grouping the requests, we define two characteristics:

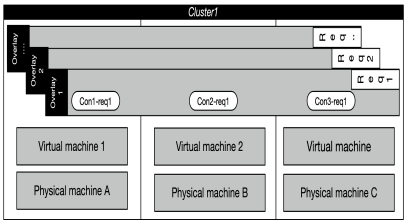
- The set of participating organizations in the DDM: this is the list of organizations in a DDM that are involved in sharing transactions in a DDM. For example in a DDM with three participating organizations (Org. A, Org. B, Org. C) and two types of requests (*Type A*, *Type B*), there will be two lists of participating organizations; (Org. A, Org. B) for sharing requests of *Type A* and (Org. A, Org. B, Org. C) for sharing requests of *Type B*.

- The owner of the request: this is the organization that has control over the connectivity of containers of a request.

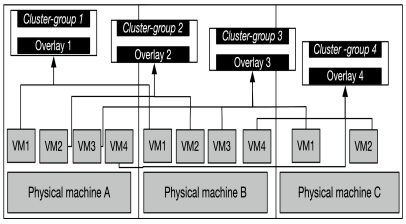
All sharing requests can be expressed with the same formalism, as Group ((Set of participating parties), owner of the request). For example Group ((Org. A,Org. B),Org. A) represents two organizations A, B are involved in data or algorithm exchanges and organization A is the owner of the request. Therefore, in a DDM with two organizations A, B, and a TTP (organization C), four different groups are defined. As is shown in Fig. 4.3c, the containers of requests of organization A are isolated from the containers of requests of organization B. In this case, if a container is compromised (the white circle) only the containers in the same group will be affected, which means that the attack surface is less than the DDM connectivity type. In addition, the number of overlay networks that have to be set up matches the number of groups.




(a) Overlay per DDM network setup, one cluster consisting of three virtual machines




(b) Overlay per Request network setup, one cluster consisting of three virtual machines




(c) Overlay per Group network setup, four clusters consisting of one or two virtual machines

Figure 4.4: Overlay network setup in a DDM

## 4.4 Overlay Setup

In this section, we present the method of setting up DDMs according to container connectivity types by means of container overlays and explain the policy enforcement method in each setup. In the following, we focus on DDMs with three participating organizations, but this is just to show the overall operations of the system and that the insights are clearly applicable to other topologies. In all setups, for constructing a DDM with organizations A, B, and C, we consider three physical machines acting as the organization's node in the DDM. Containers of each organization will be created on its own node.

**Method 1: Overlay per DDM (Fig. 4.4a)**

This method constructs the DDM according to the ***DDM connectivity*** type. In this method, all containers related to different sharing requests will be running inside one overlay network for the whole DDM. For implementing the configuration, we created one Kubernetes [22] cluster and connected all of the containers in the cluster by a Calico overlay network [41]. We selected Calico considering the results of Chapter 3. In addition, it is deployable in most cloud environments that

is integrated with Kubernetes. A Calico node contains two processes: Felix and Bird. Felix programs host route tables and Bird is responsible for route sharing among nodes [72]. After installing Calico, it uses IP-in-IP for encapsulating container's packets, which are then routed by the host through a specific interface. With this implementation, all containers are connected to each other inside one overlay network.

**Policy enforcement method**: In this method, DDM policies are enforced by Calico network policy rules. Calico filters the traffic between containers by generating *iptable* rules in the host machine of containers. Considering data flow in Fig. 4.2 as an example, the policy rules allow any traffic according to request traffic flow (source and destination) and forbid any traffic from any other container.

### Method 2: Overlay per Request (Fig. 4.4b)

In this method, all of the connections are based on the ***Request connectivity type***. We used Docker Swarm cluster technology for implementing this method as it is the available technology for running multiple overlays between the same nodes in one cluster [70, 73]. We first created a specific overlay for each request and then created related containers inside that overlay network. Each overlay has one docker bridge and containers of the same request are connected to each other through this bridge. Accordingly, containers of different requests are separated from each other. Fig. 4.4b shows that as an example four different overlay networks have been created for running four sharing requests in DDM. Docker Swarm uses VXLAN for building overlay networks.

**Policy enforcement method**: DDM policies are enforced by separating sharing requests via one overlay per request. In this method, defining firewalling rules between containers is not possible and the connection should be confined by overlays. Unlike in the overlay per DDM method, where containers are connected to each other in the network layer but traffic is controlled by filtering rules, in this method, there is no network connectivity between containers of two different requests.

### Method 3: Overlay per Group (Fig. 4.4c)

This method implements the ***Group connectivity type***. A separate overlay was created for each group and the traffic of containers of each group has to be filtered by network filtering rules. We could not use Swarm as it can not implement the filtering rules between containers of a group. Therefore, we used Kubernetes. Considering the four groups that are needed for constructing a DDM with two participating organizations and one other organization as a TTP (Trusted Third Party), we need four overlay networks. As we can just create one overlay in each Kubernetes cluster, we created four Kubernetes clusters, one for each group.

Depending on the participating organizations, two or three virtual machines are involved in each cluster. We used Calico as overlay technology in each cluster. As a result, all of the containers related to the same group were connected via one overlay network, but there was no connection between the containers in different groups.

**Policy enforcement method**: Policy enforcement in this method was implemented using Calico network policies. We defined the Calico network policy based on the permitted traffic flow of a request for containers inside a group.

## 4.5 Security

For providing security, three main aspects of security should be considered in a DDM:

1. **Availability:** The resources required to run authorized sharing transactions should be readily available to organizations. This means that during the processing of multiple sharing requests on different hosts, all relevant containers must be accessible for legitimate traffic transmission and inaccessible for illegitimate transactions.

2. **Confidentiality:** Unauthorized access to data must be strictly forbidden. If a container is compromised, it may gain unauthorized access to data related to another request.

3. **Integrity:** Compromised containers may gain access to data related to other organizations, allowing the attacker to change the data and thereby compromising the integrity of the data.

As is explained in Chapter 1, authors in [6] provide a clear classification of all kinds of attacks that can happen in a container-based platform: application to container, host to container, container to host, and container to container. Given our focus in this chapter, we consider the container-to-container attack types.

In ***Container to container*** kind of attacks, a compromised container attacks containers of other sharing requests. Other containers can be in the same or in a different host. We classify the container to container type of attacks into three main attack scenarios:

**ARP Spoofing:** A compromised container may gain access to confidential data via an ARP (Address Resolution Protocol) spoofing attack on other requests' containers. Method 1 is secure against this kind of attack because Calico makes the connection between containers on layer three and therefore, ARP spoofing can not happen [74]. This is also true about method 3, as it also uses Calico. In method 2 as the containers are connected to different bridges and they are not in the same local area network, ARP spoofing can not happen between them. Therefore, these three methods are all secure against ARP Spoofing.

Table 4.1: Inter-container attack analysis of method 1(Overlay per DDM), method 2(Overlay per request), and method 3(Overlay per group)

| Attack Type | Attack Scenario | Security | | |
|---|---|---|---|---|
| | | Method1 | Method2 | Method3 |
| ARP Spoofing | The compromised container can have unauthorized access to data sets which are related to other requests | High | High | High |
| Malware Spread | A malicious container may spread a malware across container that is connected to | Low | High | Medium |
| L3 DoS Attack | A compromised container might flood the other container related to another request at layer 3 | Low | High | Medium |
| Application DoS Attack | A compromised container might flood the other container related to another request at the application layer | High | High | High |

**Malware Spread:** A malicious container may spread malware across multiple containers that are connected to it. This is container-to-container traffic that may not be detected by network policies because there is no inspection of the content of transferred data among containers. In this type of attack, all of the containers that are connected to the malicious container are at risk and the *confidentiality* and *integrity* of DDM are affected. Comparing the three different methods, as there is no network connection between every two requests' containers in method 2, it can provide more security than method 1. As in method 3 containers are distributed in groups, its security against this kind of attack is higher than method 1 and lower than method 2.

**Denial of Service (DoS) Attacks:** In DoS attacks in a container-based infrastructure, a compromised container can send a huge amount of traffic to other containers, interrupt their service, and affect the *availability* of DDM. We classify possible DoS attacks into two categories: "L3 DoS attacks" and "Application layer DoS attacks". In L3 DoS attacks, for example, a malicious container overwhelms the other container by sending a large number of echo requests to affect its functionality. The isolation level between containers of different requests plays a major role in mitigating this kind of attack. As there is no network connection between different requests' containers in method 2, it is the most secure method compared to the two other methods. However, in method 1 containers are all connected together and it does not have a mechanism for mitigating this kind of attack. Method 3 is more secure than method 1. It takes advantage of the distribution of requests between different overlay groups and this will decrease the number of requests that may be affected by this kind of attack.

In the application layer DoS attacks, due to the fact that in the current setup of all methods no session can be established between containers of different requests, belonging to different transactions, all of the methods are secure.

Table 4.1 summarizes the security analysis of proposed methods. It shows the degree of security of each method. We defined high, medium, and low as qualitative metrics where high means more protection against the attack considered. Method 2 provides the most security against all kinds of attacks because setting up an overlay network for every single request increases inter-container isolation between requests' containers.

## 4.6 Performance analysis

In this section, we analyze the performance of the proposed methods by measuring the time taken to complete a sharing request in each method.

### 4.6.1 Experiment settings

**Hardware specification**: Our experiments were performed on three servers, connected by 10 Gigabit Ethernet. Each server is equipped with a dual 10-core Intel Xeon E5-2690 2.9GHz processor and 8GB memory.

**Software specification**: We used Ubuntu 18.04 and Linux kernel 4.15.0 as the host OS, Docker Community Edition 18.09, Kubernetes 1.18 for managing containers, and Calico version 3.8 to implement the overlay networks.

We performed a number of experiments aimed at assessing the times required to complete a request in the proposed methods. In each experiment, we first selected the type of requests, either *Type A* or *Type B* based on the pattern shown in Fig. 2.1. We then, simultaneously, submitted a group of requests consisting of a mix of sharing scenarios from the chosen type. The following group sizes are considered:

$$group\_size \in \{10, 20, 30, 40, 50\}$$

As discussed in Sec. 4.2, completing the execution of sharing requests includes three steps: policy check, network setup, and data transfer. As in this chapter, the focus is on network overlay setup, in our experiments, we consider two steps of network setup and data transfer and skip the policy check step. To make a comparison between different connectivity methods, we measured the average network setup time, the average transfer time, and finally the average total time for completing a request. We repeated every experiment three times to ensure the consistency of the results. For every repeat, the mean of the quantity of interest was calculated across the group of requests. In the plots, the mean and standard deviation of these means is shown.

Setting up the network in methods 1 and 3 involves creating the request's containers and deploying the network policies between them. In method 2, an overlay is established for each request and the request's containers are then allocated to the overlay. In all methods, the shared data transfer is executed by sending 1 GByte data for each traffic flow of the request using *iperf3* [61].

### 4.6.2 Experiment results

- *Setup time*: Fig. 4.5 shows the average setup time of a request in each connectivity method. Setup time increases in all three methods with the increasing number of requests. Method 2 has the largest setup time and method 3 has the smallest. In our experimental setup, in method 3 two

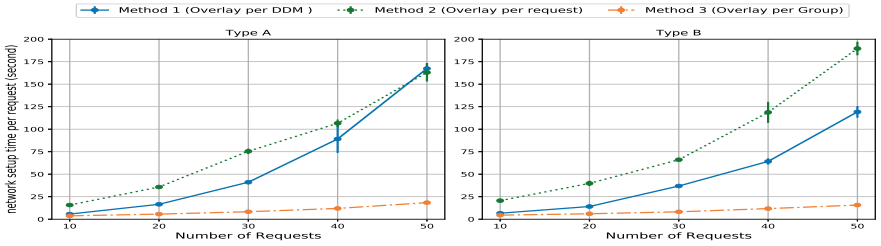



Figure 4.5: Network setup time as a function of the number of requests for the three methods. Setting up an overlay network (green) takes more time than configuring traffic filters (blue) within an existing overlay network. Using one overlay per group is the fastest, but most of the apparent difference is due to having more resources available (see main text).

clusters are used and half of the group size is running in each cluster. Therefore, the setup time of method 3 should be compared to the setup time of method 1 at half of the group size. For example, the setup time of 40 requests in method 3 is almost equal to the setup time of 20 requests in method 1.

- *Transfer time*: Fig. 4.6 shows the average transfer time. Transfer time in Type B is larger than in Type A for all methods. That is because of the difference between the number of transactions in requests of Type A (1.43 on average) and requests of Type B (3). As for the setup time, method 3 is faster than method 1, however, it has more resources available.

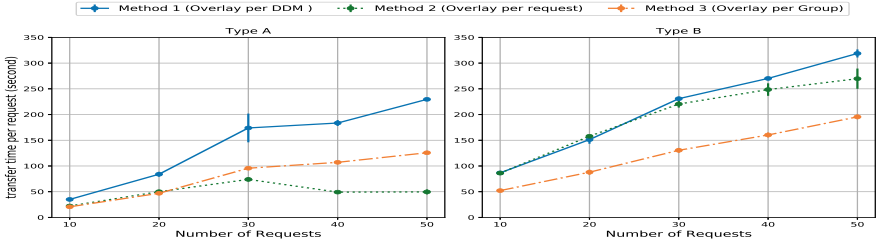



Figure 4.6: Transfer time as a function of the number of requests in three methods. A separate overlay network per request appears to be much faster than using a single network but is mostly a result of the requests being scheduled differently. See Fig. 4.7 and the text.

These performance results differ between requests of Type A and Type B. Also, for Type A and method 2, transfer time decreases with increasing

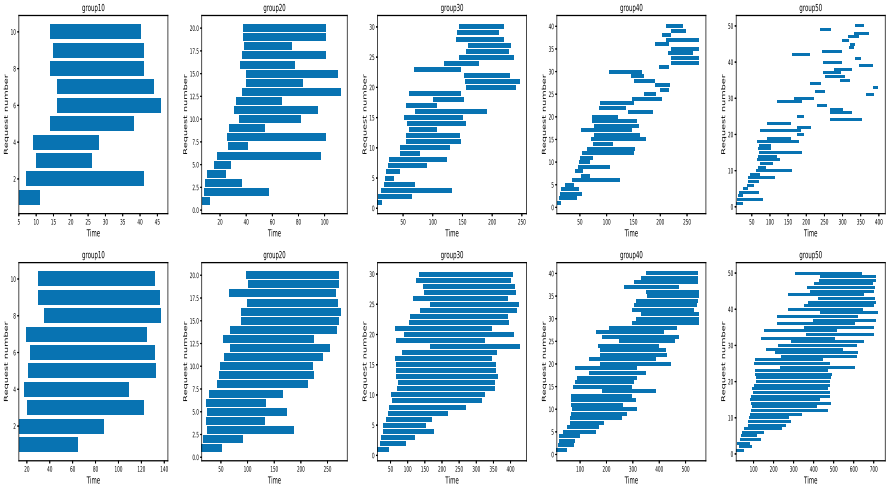



Figure 4.7: Transfer time of each request for Method 2 and different group sizes, for Type A (top row) and Type B (bottom row). The requests are submitted at time zero. Each request's bar starts when the setup step of the request is done and the transfer step starts and ends when the transfer finishes. For large group sizes (rightmost plots), there is on average less overlap between the requests, causing them to be completed more quickly. The total time required to execute all requests is still larger for larger groups of requests. The effect is much stronger for Type A (top row) than for Type B (bottom row) requests.

group size, which is unexpected. For Type B, a decrease relative to method 1 is also visible for larger group sizes.

To investigate this further, we plot the exact transfer time of each individual request for different group sizes for both type A and type B of Method 2 (Fig. 4.7). When increasing the number of requests and having more containers to set up at the same time, the average setup time increases, as was shown in Fig. 4.5. As the group size becomes larger more requests are running in parallel, which increases resource contention and slows down the requests.

For Type A (Fig. 4.7, top row), less data is transferred and the transfer step is shorter. As a result, the cluster spends more time setting up and shutting down the networks. Therefore, the requests are scattered in time, which leads to fewer transfers running in parallel and a lower average transfer time. This effect becomes larger for larger group sizes and it explains the decrease in transfer time for Type A groups over 30.

- *Total time*: Fig. 4.8 shows the average total time of completing a request. The overall time for Type B is more than Type A in all methods. For Type A, the average total time in method 2 is less than method 1, whereas for Type B method 1 is faster. In both types of requests method 3 takes less time, and roughly half of the time of method 1, that is due to the fact that it has more resources.

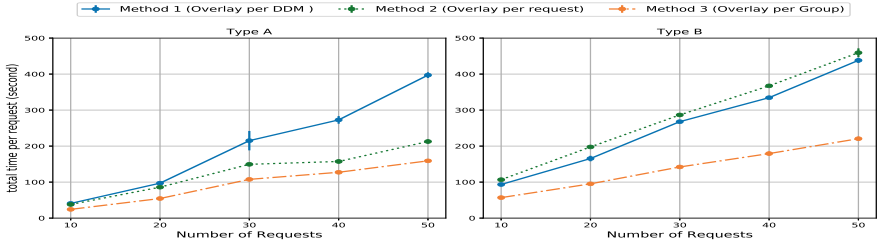



Figure 4.8: Total time to complete a request as a function of the number of requests in three methods. For Type A requests, an overlay per request was measured to be faster, due mainly to the way the requests were scheduled here. The results for Type B are more representative and show that an overlay per request is slower than using a single overlay network for the whole DDM. Overlay per group is on par with overlay per DDM when resource differences are taken into account.

## 4.7 Discussion

In this section, we will discuss in more detail the implications of our findings. We are particularly interested in how the three methods compare with respect to security and performance, so an optimal setup can be chosen when implementing a DDM. For Type A, in our experiment method 2 was faster than the other methods. However, our results indicate that this may be partially caused by the Swarm scheduler, and more research is needed into the scheduling behaviour of Kubernetes and Swarm to see how this affects performance for this use case. For Type B, method 2 is slightly slower than the others, and this can be considered as a more general result.

While method 2 is the slowest method, it is also the most secure one against the mentioned types of attacks. Method 1 is the fastest but the least secure, and method 3 is in between from both perspectives. In general, the performance difference between methods is small however, in most cases method 2 will be preferred.

The presented experiments show the performance of the proposed methods (time to complete a request) when the system is under pressure. In a real-world

system, loads will vary with time. In these experiments, all of the requests arrived at the same time, which can be considered a worst-case scenario. However, the results are consistent across different load levels, which suggests that the conclusions will hold for lower load levels as well.

## 4.8 Related Work

We expect the adoption of DDMs to increase in the coming years. The research and efforts to arrive at working platforms are ongoing. Several initiatives tackle the problem of establishing contracts between parties. For example, the Dutch logistics sector has defined iSHARE [63]. iSHARE is a uniform set of agreements for identification, authentication, and authorization to share logistics data in a safe and controlled fashion. This system can be used by all parties which have activity in the logistics sector. However, efforts like this do not define, as we do, an effective architecture for deploying the contracts and agreements in infrastructure to make a DDM work in practice by using container overlay networks.

In regard to the evaluation of overlay technologies' performance, there is a number of previous studies that have provided us with guidelines. The authors in [64] investigated the possibility of deploying Osmotic Computing environments in order to deploy distributed microservices among Cloud, Edge, and IoT devices. In particular, they deployed two different microservices: FTP and CoAP inside Docker containers orchestrated by Kubernetes. In order to find the best overlay solution, they performed scalability analysis on four different network overlays: OVN [75], Calico, Weave [43], and Flannel [44]. [8] proposes a solution for connecting containers utilizing EVPN and ILA as overlay technologies. They study the performance of Cilium/eBPF in network filtering. Authors in [76], evaluate the scalability of Calico and Cilium [42] as two popular overlay technologies by measuring the network throughput with increasing the number of containers and the number of deployed network policies between containers. Finally, the work in [5] presents a performance analysis of different methods of implementing the network connectivity between containers including overlays. In our work, we do consider these efforts and we move further to identify the better-suited overlay setups depending on relation to the data sharing request characteristics.

Different methods have been studied for providing security in Docker containers. For example, [68] and [77] utilize Linux Kernel security modules like Apparmor [65] and SELinux [66] to enhance the access control mechanism of the containers and provide more protection of the host against a malicious container. [67] studies a virtualized trusted platform (vTPM) in a container-based architecture for protecting containers from a malicious host. The main focus of these works is on limiting the container's accessibility to the resources of the host but not on the connection between containers in the network layer.

Authors in [69] discuss important security issues of Docker containers and

proposed solutions. They also propose an algorithm to tackle Dos attack issues by limiting container resources. [78] performs a comprehensive study about the security of Docker containers and denotes the possible vulnerabilities in Docker containers and the available solutions in literature works. It also specifically investigates the inter-container attacks and suggests the container network separation method as a solution, however, no practical solution is presented. In this chapter, we focus on providing network layer isolation between containers by means of overlay setups specifically for data sharing services in a DDM.

## 4.9 Conclusion

In this chapter, we proposed an architecture for implementing sharing requests and deploying high-level DDM policies in container-based network infrastructure.

We defined three different connectivity types that are different in isolation they provide between containers of a sharing request. We implemented these connectivity types by setting up the overlay networks between containers using Kubernetes and Docker Swarm. The implementations are called overlay per DDM, overlay per group, and overlay per request. To evaluate each method, we studied how they are secure against inter-container types of attacks. This work reasoned that the number of attacks that can happen in 'Overlay per request' method is less than the other methods as it provides better isolation between requests. However, to further validate the security of the different methods against potential attacks, future work could involve conducting actual attack experiments on the methods and measuring their effectiveness against various types of attacks. This will provide a more comprehensive assessment of the security of each method.

We also compared the time required to complete a sharing request between the methods. The three methods performed similarly, although the "Overlay per request" method is slower than the others in larger types of requests.

In Chapter 5, we focus on building a more complete DDM involving multiple sites and applications running across the DDM building a multi-domain overlay network. In addition, we investigate how dynamic programmable network architecture can be used to build a more secure system.

# Chapter 5

# Multi-domain Network Infrastructure based on P4 Programmable Devices for Digital Data Marketplaces

In Chapters 3 and 4 we presented how to build a single domain data sharing platform based on containers. In this chapter, we extend the model to a multi-domain DDM. In a multi-domain DDM the participating organizations have control over the enforcement of the sharing policies between the containers within their infrastructure, which allows them to control the accessibility of data and algorithms locally. In the proposed architecture in this chapter, the containers within a domain are connected to each other through a virtual switch and the configuration of each switch is managed by a domain administrator. We use P4 [26] for programming the switch and setting up the connections. The proposed method can handle the communication of multiple domains and guarantee that the operation of transactions is based on pre-defined policies. We assign a connection ID to every asset (algorithm and data) within a domain and the cooperation between domains is established based on these IDs. We study the security implications of the architecture considering P4 as the underlying network technology. In addition, we measure the setup time and discuss the overhead of using P4 switches. This chapter is related to RQ3: *How to build a containerized multi-domain DDM on a programmable network infrastructure to enforce sharing policies?*

This chapter is based on:

**Shakeri, S.**, Veen, L. and Grosso, P. "Multi-domain network infrastructure based on P4 programmable devices for Digital Data Marketplaces". Cluster Comput (2022). https://doi.org/10.1007/s10586-021-03501-2.

## 5.1 Introduction

A DDM needs to provide means for data exchange and/or processing while enforcing sharing policies in its infrastructure. In some cases, legal restrictions keep data from being exchanged at all, for example when GDPR rules apply to data. Still, even under these constraints, it is possible to combine data from different sources through techniques like distributed machine learning, secure multiparty computation, and homomorphic encryption. Initial implementations of such algorithms are becoming available in systems like Vantage6 [79], PySyft [80], MPyC [81], and IBM's Federated Learning Library [82]. DDM algorithms are communication intensive, and in order for a DDM to support them efficiently, processes running on different independent systems will need to be able to exchange data directly while still maintaining the security level specified by the policies governing the application. In this chapter, we refer to the shared algorithms and data as *assets*.

A DDM is a multi-domain environment. For preserving privacy and implementing sharing policies all the connections related to each participating party, i.e, the domain, have to be managed by its own domain administrator. In Chapters 3 and 4, we showed how we could build a single-domain data sharing platform by using containers. We used both Kubernetes [22] and Docker Swarm [70] as container network orchestrators for creating network overlays and setting up filtering rules between containers. However, these methods are not applicable in a multi-domain environment: in all these frameworks one node should be selected as a master node so it has access to all of the containers in the network and can manage their connections. In a multi-domain scenario, each participating party should be able to manage its own domain connectivity independently from other domains participating in the sharing platform. Multiple masters need to work together while maintaining the sharing policies. Building a container-based DDM network infrastructure that can be integrated with the per-domain orchestrators is still an open challenge, which we study in this chapter.

For building a multi-domain DDM we adopt P4 [26] as the network data plane technology. We study in deploying P4 programmable switches satisfies our requirements, i.e., per domain orchestration and managing container connections. All the connections between containers will be programmed in such switches and can be controlled by the administrator. In addition, the connections can be reconfigured when needed which leads to dynamic management of DDM containers connections and sharing policies.

We conduct a security evaluation showing that our P4-based network setup is secure against most types of attacks. Moreover, we introduce a model for measuring the network setup time in our system, which we demonstrate to follow the real measurements and can be used for further design or topology decision makings. In addition, we measure the network setup time of our proposed method and show the low overhead of using the P4 switch.

The main contributions of this chapter are:

- Presenting an architecture for a multi-domain container-based DDM with P4
- Describing the required steps for executing a sharing request in the proposed architecture
- Studying the security aspects of the proposed framework
- Modeling and evaluating the setup time of the request execution

## 5.2 Containerized P4-based DDM

For constructing a container-based DDM we follow three main goals:

- Constructing a ***multi-domain*** environment in which each domain can manage its related configuration. As a matter of fact, DDM consists of different, independently managed, and configured domains, with data sharing to be done across the domains.
- Improving security by providing higher ***isolation*** between containers, controlling their network connections, and providing more advanced filtering possibilities.
- Providing ***programmability*** in the network. Because of the dynamic behavior of a network of containers, it is important to provide the ability to program and change the network configurations when needed. Especially in a DDM where sharing policies may change at any moment.

In overlay technologies like Calico or Cilium [41, 42] filtering rules can be set between containers, so they can provide the required isolation. However, by using Calico or Cilium, all the hosts are under the control of one master node that makes a cluster. Therefore, these methods are single-domain and are not applicable in our case.

The simplest way of connecting containers in a network that can be used in a multi-domain environment is using the Docker bridge in each domain [83]. However, in this method, all the containers are connected to the same bridge and it is not possible to set the filtering rules between them in the host. Therefore, this method cannot provide the required network isolation between containers.

As another method, a user-defined network in which a separate bridge is defined for each group of containers can be deployed. In this case, the containers connected to the network bridges are separated from each other [84]. It can provide better isolation compared to the single Docker bridge method, however, still, the containers that are connected to the same bridge are connected to each

other and their traffic cannot be filtered so the containers are not fully isolated. As a result, we selected programming the switch to handle all the mentioned requirements in one setup.

Given the inflexibility of OpenFlow switches due to the limitations of the OpenFlow protocol and their significant overhead on the switch controller, we have opted to use P4 [25]. P4 offers a more flexible and programmable approach, allowing us to address these issues effectively. By using P4, we can offload dynamic functions from the switch controller to the data plane, reducing the burden on the controller [26].

In our architecture, in each domain, the P4 switch is in charge of routing the traffic based on match-action rules independently from other domains so it can be deployed in a *multi-domain* environment. In addition, there is no connectivity between containers before setting up the rules in the switch. Therefore, every single container's connectivity is controlled by the switch. This provides more *isolation* between containers compared to the available methods. Finally, any connection and filtering rule between containers can be *programmed* dynamically.

In addition to the aforementioned capabilities, P4 can provide more features to the DDM:

- As the packet's header can be parsed in the P4 switch, every single field of the header at different layers can be checked against matching rules. This provides high flexibility in deploying the filtering rules.

- As containers are individually connected to the P4 switch interfaces, different kinds of telemetry information, like the amount of traffic that is being transferred via a specific interface, or the time of entering a packet to that interface can be tracked in P4 [85,86]. This will help manage the traffic and prevent different kinds of attacks [87–89] to improve security.

- The P4 program can be run on SmartNICs [90, 91]. Therefore, the packet processing can be offloaded to the hardware, and the host's CPU will not be used for the processes. This leads to better performance.

- New protocols can be implemented using the P4 programming language and therefore, if needed, extra information between multiple domains can be transferred through packets.

An alternative approach to connecting containers is through the use of VPNs, where each pair of containers is connected together. VPNs are a reliable and well-established approach, providing encryption and establishing secure connections. They have been widely used for creating secure network connections across different locations or networks. We investigate how P4 capabilities allow administrators to have fine-grained control over network behavior. This level of control enables efficient management of connections within the DDM system, leveraging techniques such as utilizing telemetry information [85,86]. Additionally, similar to

VPNs, P4-based solutions can also incorporate encryption mechanisms to ensure data security [92, 93]. Therefore, the P4-based approach explores the potential of P4 programmability and its benefits in administering connections while offering encryption capabilities as well.

## 5.3 Architecture

Fig. 5.1 shows the architecture of the multi-domain containerized DDM with three distinctive blocks: the orchestration block in charge of the coordination of the operations in the DDM, the containerization block in charge of creating and management of containers, and the networking block in charge of setting up the P4 network and the required connection between containers.

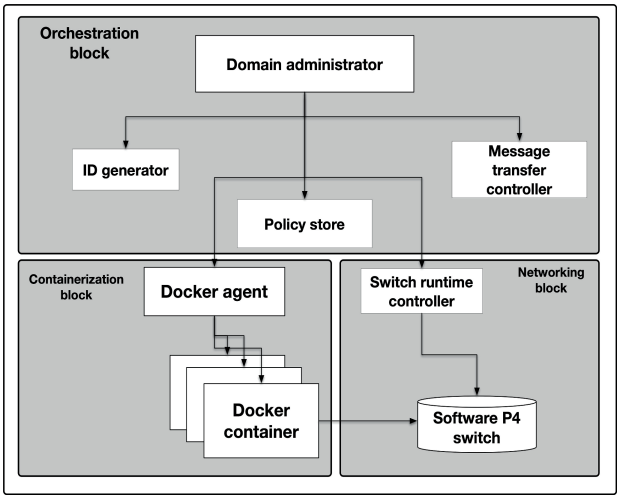



Figure 5.1: A Multi-domain containerized DDM architecture. It includes three main blocks: orchestration block, containerization block, and networking block

In each domain, all components are under the control of the domain administrator. The domain administrator can handle and manage all connections based on the policies and rules that have been established in the DDM. The role of each component in the architecture is as follows:

- Orchestration block
  - **Domain administrator:** Manages all components in its own domain and controls the sequence of steps for running the execution of a sharing request.

  - **Message transfer controller:** The domains need to exchange necessary information in order to make the connection between containers. The message controller is in charge of sending/receiving the required information to/from the other domains.
  - **ID generator:** In each domain, each asset within a container is identified by a unique number called the *connection ID*. This number identifies the destination port of the connection to the container that includes the asset. The ID generator generates this unique ID.
  - **Policy store:** All policies and rules about permissions to access specific data or algorithms in a domain are recorded in the policy store. Before running any request, the domain administrator checks in the policy store if the request is permitted or not. If it is not permitted the request will be rejected.

- Containerization block
  - **Docker agent:** This component is in charge of creating docker containers and setting up the required configurations for their network interfaces.

- Networking block
  - **Switch Runtime Management:** This component sets the rules in the switch. The rules are created by the domain administrator based on the information from the policy store, the Docker agent, and the ID generator. The rules allow traffic with a specific connection ID (port number) to a specific container.
  - **Software P4 switch:** This is the core networking element that will switch the traffic appropriately.

## 5.4 Workflow scenario

The operation of our proposed architecture can be better understood by looking at a concrete scenario. We consider two domains, domain A and domain B, in a DDM and assume the whole architecture described in the previous section is running on two servers, one in each domain (see Fig. 5.2). Therefore, Domain A is a Linux server that is connected to another server in Domain B.

Suppose that domain B requests access to an asset in domain A. We call domain A server-side and domain B client-side. In the following, we will guide the reader step by step through each of the operations that occur to set up the connection between containers in the two domains for running the sharing request.

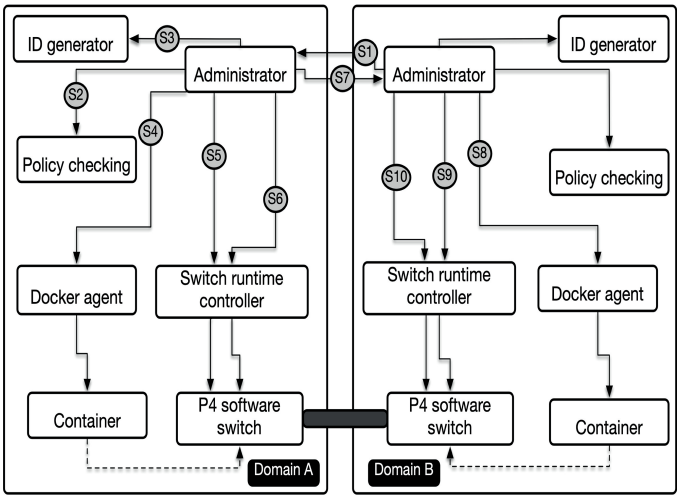



Figure 5.2: Steps for making the connection between containers of two different domains using a unique connection ID

- **step1:** The domain administrator of domain B sends a request to the administrator of domain A through the message controller. The request identifies the asset that it wants to access and asks for the unique ID that is needed for starting the connection between containers.

- **step2:** The domain administrator in domain A checks the request permission in the policy store and if the request is allowed it takes the next required steps for making the connection.

- **step3:** The ID generator in domain A creates the connection ID.

- **step4:** The domain administrator of domain A asks the Docker agent to create a container for the request. The container is initially created without any interfaces. Then, the Docker agent configures the network interface of the container with a specific IP address, and after that, it peers the container interface with one virtual interface (*veth*) of the host server. It must be noted that at this stage the container is still not connected to the switch.

- **step5:** After the container has been created and its IP address and interfaces are set, the domain administrator instructs the switch runtime controller to connect the container interface to one of the ports of the switch that is programmed with P4.

- **step6:** In this step the domain administrator generates the rules that allow establishing a connection between the containers of the two domains and set the rules via the switch controller in the switch. Each rule includes the matching phrase and the action on the packet if the match happens. On each side, two rules are needed. One for packets from the container to the other domain (outbound), and one for packets from the outside to the related container(inbound). In this step, we explain the required rules in domain A. The outbound rule matches the packets by the port that they entered the switch. The inbound rule matches the packets by their destination port (the connection ID). In the outbound rule, the action is setting the packet's source IP address to the host's public IP address and the source port number to the connection ID. In the inbound rule, the action is setting the destination IP address and port number of the packet to the IP address and port number of the destined container and sending out the packet to the container.

- **step7:** When all of the required configurations are set in Domain A, the domain administrator sends the unique ID out to Domain B. This also indicates that Domain A is ready for a connection with that specific ID.

- **step8:** Like domain A, domain B creates the container related to this request. The interface configuration is the same as domain A.

- **step9:** In this step the domain administrator connects the container to the switch port via the switch runtime controller.

- **step10:** Finally, in this step, domain B sets two rules that are needed for making the connection to the specific container based on the unique ID. On the client-side (that is domain B), the combination of the destination IP address and destination port is unique. Therefore, packets sent from a local container in domain B to the switch are matched with these two specifications and then sent out to the server-side. The second client-side rule is for packets coming from the server-side. These are matched by the same unique combination (which here is the source IP address and source port number) and then sent to the specific container.

By performing all of these steps the DDM creates a connection between two containers in a multi-domain environment. The connection between domains is isolated and based on a unique number that is specific to that request and is not repeated by any other connection, hence guaranteeing isolation between concurrent requests.

## 5.5 Security

In this section, we looked at possible types of attacks that can happen in a multi-domain containerized DDM and studied how the architecture proposed in this chapter is secure against these kinds of attacks.

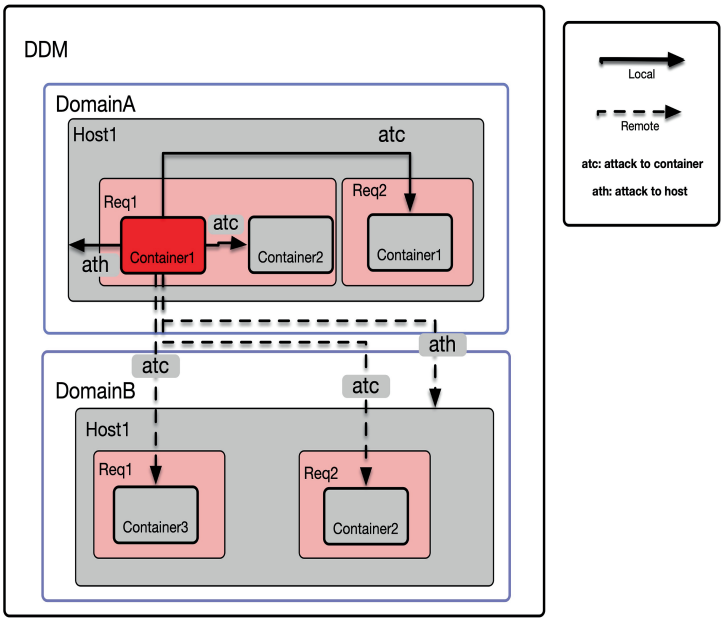



Figure 5.3: Threat model in a multi-domain DDM. Two types of attacks are possible: *local attacks* - Attacks from a container to host (*ath*) or to another container in the same domain. (*atc*); *remote attacks* - Attacks from a container to other container (*atc*) or host (*ath*) of the other domain

Fig. 5.3 shows the threat model of the architecture. In all attack scenarios, we assume that the attack originates from a malicious container. We considered below types of attacks:

1. container to container of the same request (*atc* in the figure);
2. container to container of different request (also called *atc* in the figure);
3. container to its host's service (called *ath*); in this case, a container attacks a service that is running in the host and listening on a specific port number.

In addition, in a multi-domain DDM, these types of attacks can be *local*, i.e. the attacking container is in the same domain as the victim, or *remote*, i.e. the attacker is in the other domain.

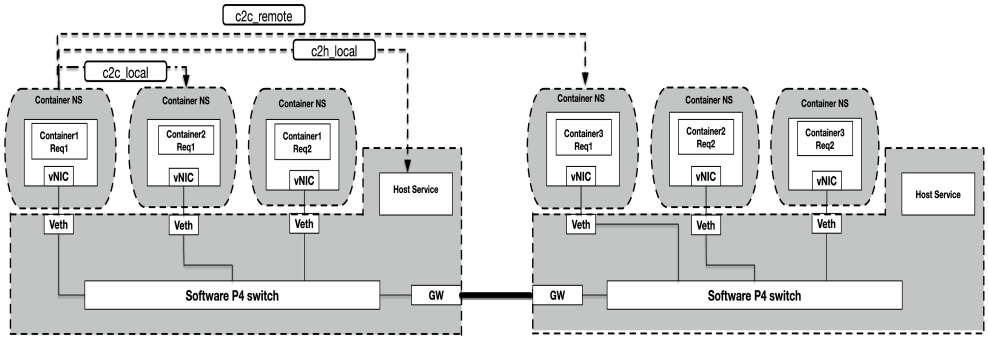



Figure 5.4: Container connections possible in a multi-domain DDM: connection to containers of the same request in the local domain (*c2c_local*); connection to containers of the same request in the remote domain (*c2c_remote*); connection to the local host (*c2h_local*).

Table 5.1: Possibility of attacks in a multi-domain containerized DDM: ○Attack is not possible ⊙Attack is possible but it can be mitigated ●Attack is possible and it cannot be mitigated

| Type of attack | Domain | Arp Spoofing | Mac Flooding | IP Spoofing | Syn Flooding | HTTP Flooding |
|---|---|---|---|---|---|---|
| Container to container of the same request | Single domain | ○ | ○ | ⊙ | ⊙ | ⊙ |
| | Multi-domain | ○ | ○ | ○ | ⊙ | ⊙ |
| Container to container of different request | Single domain | ○ | ○ | ⊙ | ○ | ○ |
| | Multi-domain | ○ | ○ | ○ | ○ | ○ |
| Container to host | Single domain | ○ | ○ | ○ | ● | ● |
| | Multi-domain | ○ | ○ | ⊙ | ○ | ○ |

In particular, we only consider networking attacks related to container connections. Based on the architecture and the possible connections that a malicious container can make, we discuss the feasibility of each attack.

Fig. 5.4 shows all possible connections that a container can make to any other services of other containers or hosts in our multi-domain environment. These include the local connection that happens in the same host with the container of the same request (*c2c_local*) or the host service (*c2h_local*); or the remote connections (*c2c_remote*) to the other domain.

In our architecture, any connection from a container to any other container is through the switch. However, for the connection between a container and the host, the packet does not need to go through the switch because of peering with the host virtual interfaces *veth* that are in the host network namespace.

Table 5.1 shows the possible attacks that can happen: ARP spoofing, MAC flooding, IP spoofing, Syn flooding, and HTTP flooding. In regard to the feasibility of each one of them, there are three possible outcomes: 1) the attack is not possible; 2) the attack is possible but it can be mitigated; 3) the attack is

possible and it cannot be mitigated.

The possibility of each kind of attack is explained in the following:

**ARP spoofing and MAC flooding**: For these kinds of attacks to happen there must be a connection between containers at layer 2 [94]. In our case, each container is in a different network subnet and the connections are at layer 3. Therefore, ARP spoofing and MAC flooding are not possible in any of the possible scenarios.

**IP spoofing**: In an IP spoofing attack scenario, an attacking container impersonates another container's IP address and sends packets with an incorrect source IP to another service that is running in the network. Therefore, the response of the packet will be sent back to the victim and not the actual source [95].

In our architecture, the feasibility of an IP spoofing attack depends on whether the attack is local or remote and on whether the attacked node is another container or a host.

More precisely we can observe that IP spoofing is not possible if the packet has to go through the switch, i.e. when the attacker wants to make a connection to other containers in the same host or to the outside world. As the routing is based on the unique ID number that is independent of the container's IP address, the packet will be dropped as it does not match any rule. This means that *c2c_remote* and *c2h_local* are not possible.

IP spoofing is possible if the attacker tries to make a connection to its host service and perform the attack via host service, like in the *c2c_local* and *c2h_remote* types of attacks. In these cases, packet does not go through the switch. However, its source IP address can be checked in the host's IPtables. Therefore, when the source IP address is not correct, the packet will be dropped.

**Syn and HTTP flooding:** In all cases where a container can make a connection, flooding attacks are possible. For cases in which the packet is going through the switch (*c2c_local* and *c2c_remote*) the attack can be mitigated by detection methods that can be implemented in the P4 program [87–89]. However, for the *c2h_local* type of attack, the attack cannot be mitigated as the packet does not go through the switch and the connection is directly between containers and the host service.

To conclude, as is shown in Table 5.1 most network attacks are either not possible in the proposed P4-based DDM or can be mitigated by customizing the P4 program. We must observe that security depends also on the availability of the P4 switch. As all containers are connected to the P4 switch and the whole setup relies on the rules in the P4 switch, this can become a failure point in the network. However, as there can be multiple servers in a DDM when the switch fails in a domain, only the containers that are connected to that specific switch will be affected.

### 5.5.1 Security considerations

Data encryption can happen in the P4 switch. To this purpose, there are multiple available algorithms that encrypt the connection in a P4 switch and can also be deployed in this work [92, 93, 96]. This can help mitigate the risk of Man in the Middle attacks.

The orchestration services at the two sites communicate with each other via the public Internet, which means they must be publicly accessible, and they also control the network plane, which requires at least some administrator-level privileges. This is clearly a potentially risky combination. Several steps must be taken to mitigate this risk. First, the connection between the orchestration services needs to be authenticated and encrypted, for example by using an HTTPS connection with client- and server-side certificates for authentication.

Next, to mitigate the risk of the orchestration application software itself being compromised, the network administration functionality must be separated out into a separate program, which is given the minimum necessary privileges (e.g. CAP_NET_ADMIN on Linux) and implements the bare minimum functionality needed to support the functioning of the system. The orchestration service itself can then be run as an unprivileged user so that if it is compromised the attacker will be limited to unprivileged operations plus a small number of very inflexible network administration functions.

Beyond these specific design features, all the usual general measures must be taken, including code quality assurance through testing, code reviews, and automated analysis.

## 5.6 Request setup time

In a containerized DDM, the setup time is from when a client issues a request until the moment that the network is ready for starting the data transfer between containers. In many cases, it is important and critical for a client to know the approximate setup time of the request; for example, for federated machine learning having the knowledge of resource availability is critical for running the model efficiently.

In a single domain DDM, as there is one centralized controller that handles all of the resources, the network setup is simple and straightforward. However, in a multi-domain DDM, as domain administrators have to communicate with each other, and all of the setup processes in one domain are separated from the other one, setup is lengthier. In fact, the client-side should be sure that the destination is ready for establishing the connection.

There are two approaches we can take to measure the setup time: one is a *Global view* that considers the time from when a request comes in, to when the network is ready for data transfer. It is measured by setting a timestamp from

Table 5.2: Theoretical request setup timetable based on the duration of every single step, $T_{s_n}$ is the time taken for running step $n$

| | **Domain A** | **Domain B** | **Parallel time** |
|---|---|---|---|
| **Total time** | $T_A$ | $T_B$ | $Max(T_A, T_B)$ |
| **Calculated time** | $T_A^{'}$ | $T_B^{'}$ | $Max(T_A^{'}, T_B^{'})$ |
| **Model error** | $(T_A - T_A^{'}/T_A) * 100$ | $(T_B - T_B^{'}/T_B) * 100$ | $(Max(T_A, T_B) - Max(T_A^{'}, T_B^{'})/Max(T_A, T_B)) * 100$ |
| **Communication delay** | $Ts_1$ | $Ts_1 + Ts_7$ | |
| **Creating container** | $Ts_4$ | $Ts_8$ | |
| **Adding interfaces** | $Ts_5$ | $Ts_9$ | |
| **Adding rules** | $Ts_6$ | $Ts_{10}$ | |

starting a request until it is ready to start transferring data.

The other one is a *Step view* which looks at the duration of each individual step. We measured the duration of each step by setting timestamps before and after running each step. For example, for measuring the duration of step 4, that is creating containers in domain A, we register the time stamp before running step 4: $C_A(start(S_4))$. It is a timestamp of the clock of domain A related to the start point of step 4. Likewise, we register the time stamp after finishing step 4 as $C_A(end(S_4))$. Therefore, the duration of step 4 is calculated based on Equation 5.1.

$$Ts_4 = C_A(end(S_4)) - C_A(start(S_4)) \tag{5.1}$$

However, we must observe that there will be a challenge in measuring the duration of $Ts_1$ and $Ts_7$. That is because the starting and ending of these steps are not in the same domain and the time is dependent on the clock time of both domains. To solve this problem, we assume that $Ts_1$ and $Ts_7$ are equal and eliminate clock differences by adding timestamps of different domains according to Equation 5.2.

$$\begin{aligned} Ts_1 = Ts_7 = (C_A(end(S_1)) - C_B(start(S_1)) + \\ C_B(end(S_7)) - C_A(start(S_4)))/2 \end{aligned} \tag{5.2}$$

We investigated two different approaches to set up the network: a sequential and a parallel mode.

### 5.6.1 Setup time in sequential mode

As explained in Sec. 5.4, the network is set up in 10 steps. If they are executed sequentially then the total time is the sum of the duration of each step. If we define $Ts_n$ as the time taken to run step $n$ then the total time can be calculated as in Equation 5.3.

$$Sequential\ setup\ time = \sum_{n=1}^{10} Ts_n \tag{5.3}$$

### 5.6.2 Setup time in parallel mode

To optimize the total setup time, it is possible to perform some of these steps in parallel. Looking back at Fig. 5.2, after the first three steps the other steps can be run simultaneously as they are in different domains and independent from each other. The parallel setup time is the sum of the duration of the three first steps and the maximum time of running the other steps in each domain. This is expressed in Equation 5.4.

$$\begin{aligned} Parallel\ setup\ time = \sum_{n=1}^{3} Ts_n + \\ Max\ (\sum_{n=4}^{6} Ts_n, \sum_{n=7}^{10} Ts_n) \end{aligned} \tag{5.4}$$

### 5.6.3 Global view and step view comparison

Table 5.2 shows the theoretical model for calculating the setup time. The total time in Table 5.2 represents the global view and the calculated time represents the step view. *Communication delay*, *Creating container*, *Adding interfaces*, and *Adding rules* are individual steps that are considered in the step view model. By comparing total time and calculated time (Model error in Table 5.2), we can determine the accuracy of the step view model. Concretely this would tell us if there is any overhead in running the steps that the model does not capture. If the overhead is negligible then we can conclude that the time to set up is predictable, which enables prediction of the setup time and other calculations for decision-making for making any further improvements.

$T_A$, $T_B$, $T_A^{'}$, and $T_B^{'}$ in Table 5.2 are calculated based on equations 5.5-5.8.

$$T_A = T_{s_1} + C_A(end(S_6)) - C_A(start(S_2)) \tag{5.5}$$

$$T_B = T_{s_1} + T_{s_7} + C_A(end(S_{10})) - C_A(start(S_8)) \tag{5.6}$$

$$T_A^{'} = \sum_{n=1}^{3} Ts_n + \sum_{n=4}^{6} Ts_n \tag{5.7}$$

$$T_B^{'} = \sum_{n=1}^{3} Ts_n + \sum_{n=7}^{10} Ts_n \tag{5.8}$$

## 5.7 Proof of concept

To test the operations of our architecture (see Sec. 5.3) we built a prototype DDM software suite. We implemented the connections between all the building blocks,

starting from the domain administrator at the higher level all the way down to managing the network configuration in the P4 switch. We then instantiated two DDM domains and connected them to validate the scenario in Fig. 5.2.

In our setup, we did not implement the policy checking part of the architecture and we assumed that all of the requests that come in are according to the agreed-upon rules.

For our implementation, we used Ubuntu 18.04 and Linux kernel 4.15.0 as the host OS, Docker Community Edition 18.09 for container management, and bmv2 P4 switch as the programmable software switch in each server [97].

The scenario in Fig. 5.2 is written in a *bash* script, which initiates each step by calling the programs to implement the functionalities of each block of architecture.

We have two servers representing the two different domains, each running the full software suite. Each server can be the requester server (*client-side*) or the one that is requested to share data (*server-side*). The servers are connected together through a switch, which serves as the physical underlay for our connectivity.

For message transfer between domain administrators, the script calls a message transferring program using RabbitMQ [98]. The receiver side of each server is always running and waiting for a new message. When a sharing request comes in from a client for access to data of the other domain, this program sends the access request information to the other domain. Likewise, the receiver side communicates back through the message bus.

After sending or receiving the required messages, the domain administrator starts to create containers and sets the interface configurations. This is the *containerization block* of the architecture. Listing 5.1 shows the procedure to create and connect containers to the network.

```
1  sudo docker run
2      -it -d --net=none "container_image"
3
4  container_pid =
5      $(sudo docker inspect --format '{{ .State
       .Pid }}' "container_name")
6
7  sudo ip link add
8      veth1 type veth peer name veth2
9
10 sudo ip link set
11     veth2 netns container_pid
12
13 sudo docker exec
14    "container_name" ifconfig veth2
15  "container_IP_address"
```

Listing 5.1: Container interface configuration procedure

At first (line 1-2) containers are created without any interface. Next (line 7-8) we create the virtual interfaces and these are then moved to the container's network

namespace (line 10-11). Finally (line 13-15) we configure the container interface. After the container configuration, the switch with the compiled P4 program can start running.

**P4 program:** Listing 5.2 shows the P4 program that was used for managing the connection between containers based on a unique ID.

```
1     table client_send_t {
2         key = {
3         hdr.tcp.dstPort :exact;
4         hdr.ipv4.dstAddr :exact;
5         }
6         actions = {client_send; }
7     }
8     table server_receive_t {
9         key = {
10        hdr.tcp.dstPort :exact;
11
12        }
13        actions = {server_receive; }
14    }
15    table server_send_t {
16        key = {
17    hdr.ipv4.srcAddr :exact;
18        }
19        actions = {server_send; }
20     }
21    table client_receive_t {
22        key = {
23    hdr.tcp.srcPort :exact;
24    hdr.ipv4.srcAddr: exact;
25    }
26    actions = {client_receive;}
27    }
```

Listing 5.2: List of tables used in the P4 program running in the switch defining the expected operations for packets sent or received by the server and client sides of the DDM

As each server can act as both server and client there are four tables defined in the P4 program:

- client_send_t (line 1-7): when a packet is sent from client-side to server-side;
- server_receive_t (line 8-14): when a packet is received on the server-side.
- server_send_t (line 15-20): when a packet is sent from server-side to client-side;
- client_receive_t (line 23-31): when a packet is received on the client-side;

A packet entering the switch is matched against the fields shown in Listing 5.2. If a packet matches any of the fields in one of the tables the specified action will be taken, else it will be dropped.

The actions for each table are shown in Listing 5.3. The action taken will depend on whether the packet is coming from the outside or from an internally connected container, and whether it is on the server-side or client-side.

```
1       action server_receive(bit<32> dst_ip,
    bit<9> port){
2           hdr.ipv4.dstAddr=dst_ip;
3           stdmeta.egress_spec = port;
4       }
5
6       action client_receive( bit<32> dst_ip,
    bit<9> port){
7           hdr.ipv4.dstAddr=dst_ip;
8           stdmeta.egress_spec = port;
9       }
10
11      action server_send(bit<32> src_ip, bit
    <9> port){
12          hdr.ipv4.srcAddr=src_ip;
13          stdmeta.egress_spec = port;
14      }
15
16      action client_send(bit<32> src_ip, bit
    <9> port){
17          hdr.ipv4.srcAddr=src_ip;
18          stdmeta.egress_spec = port;
19      }
```

Listing 5.3: P4 actions associated with the P4 program tables

When a packet comes into the P4 switch to be sent to a local container, the destination IP address is changed to the correct local (container) IP address, and the packet is sent to the destination: these are *server_receive* (line 1-5) or *client_receive* (line 6-10) actions.

When a packet leaves the P4 switch toward the other domain the source IP address of the packet is changed to the public address of the local server: these are the *server_send* (line 11-15) and *client_send* (line 16-19) actions.

The last call is related to adding new rules associated with containers connection. Adding these rules allows connection between two containers created in two domains; that is the only permitted connection between these two containers. Domain administrator uses the switch command line to insert required rules for making connection possible. For example, listing 5.4 shows these rules related to client-side. The first rule is when the packet is outgoing from the container to

server-side. The second rule is when the response from server-side arrives. Because filtering rules are a combination of IP address of the source or destination and of the connection ID, the connection ID needs only to be unique in each host but not between domains. Therefore, there are 64K values for every host to assign as connection ID and that is enough in practice.

```
1 table_add client_receive_t
    client_receive "Source_port" "
    Source_ip_address" => "
    Destination_ip" "Egress_port"
2
3 table_add client_send_t client_send "
    Destination_port" => "
    Source_ip_address" "Egress_port"
```

Listing 5.4: List of rules in the P4 program

## 5.8 Measured request setup time

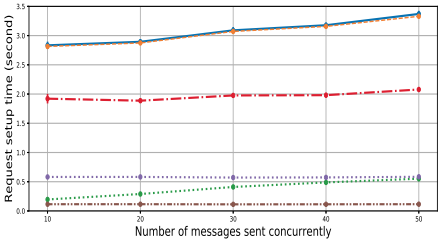
The next step for us was to measure the setup time in our experimental environment and try to numerically identify the possible overhead.

**Setup time of one request:** According to Table 5.2 and based on what is explained in Sec. 5.6 we measured both the total setup time based on Equation 5.4 as well as the duration of every single step (see Sec. 5.6.3). Table 5.3 shows the average value across 3 experiments. The results show that the model error is less than 3%. As the difference is negligible we can conclude that the stepwise model is reliable for estimating the setup time and that it can be used for further decision making and possible optimizations. The *creating container* step is the longest step for setting up the network ($\sim$ 1.8 seconds). The other steps related to the P4 switch are adding interfaces ($\sim$ 0.50 seconds) and adding rules ($\sim$ 0.110 seconds); they take much less time than creating the containers and this shows the low overhead of using a P4 switch in the setup process.

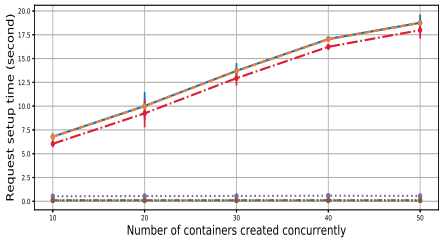
Table 5.3: Request setup time table in seconds, numbers are according to Table 5.2

| | **Domain A** | **Domain B** | **Parallel time** |
|---|---|---|---|
| **Total time(s)** | 2.561 | 2.721 | 2.721 |
| **Calculated time(s)** | 2.499 | 2.700 | 2.700 |
| **Model error(s)** | 0.025 | 0.007 | 0.007 |
| **Communication delay(s)** | 0.082 | 0.163 | |
| **Creating container(s)** | 1.765 | 1.850 | |
| **Adding interfaces(s)** | 0.540 | 0.571 | |
| **Adding rules(s)** | 0.112 | 0.116 | |

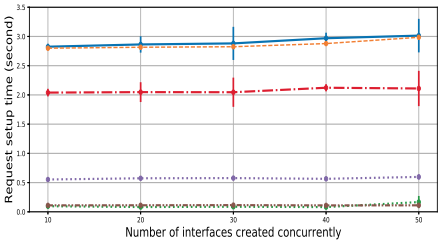
**Setup time as a function of increasing load:** For further investigation and to observe the individual impact of each step on the setup time, we explicitly



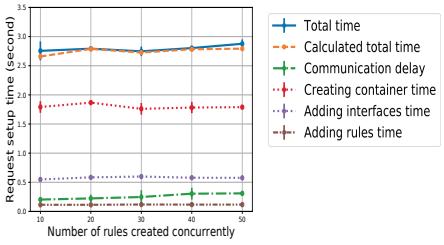

(a) Increasing load by increasing the number of messages being sent concurrently




(b) Increasing load by increasing the number of containers being created concurrently




(c) Increasing load by increasing the number of interfaces being created concurrently concurrently




(d) Increasing load by increasing the number of switch's rules being set in P4 switch concurrently

Figure 5.5: The impact of adding load on the system by increasing the number of operations related to each step separately

overloaded our system with concurrent operations and observed the change of setup time as a function of the increasing load. In each experiment, we measured each step's duration and also calculated the total time based on the stepwise model (see Sec. 5.6). We performed four different experiments (Fig. 5.5):

- Message transfer experiment: Fig. 5.5a shows the setup time of one sharing request, when the number of concurrent messages that are being sent from client-side to server-side is increasing. To create this additional load we sent a number of messages unrelated to the request from the client-side to the server-side via the message bus. We varied the number of concurrent messages from 10 to 50. The plot shows that the delay for message transferring between two domains is positively correlated with the number of concurrent messages on the bus. We observe a maximum increase of $\sim 50\%$. On the other hand, as this step takes much less time than creating containers, its dependency on the number of concurrent messages does not have a substantial effect on the overall setup time.

- Creating container experiment: To produce an additional load we created

a specific number of containers not related to the specific request we are measuring. We issued requests to the Docker engine for 10 to 50 containers in step 10. As shown in Fig. 5.5b, by increasing the number of containers that are being created at the same time, the creating container time of a single request also increases. The additional amount of time is substantial compared to the other steps. Additionally, this step is always the longest (see Table 5.3), and its dependence on the load will have the greatest impact on the variability of the total time.

- Adding interface experiment: Fig. 5.5c shows the effect of running concurrently adding interface operations on the setup time. We created containers' interfaces (varying the number from 10 to 50) and added them to the P4 switch at the same time. The figure shows that the time is lower than the container setup time and in addition, it does not change with an increasing number of concurrent operations.

- Adding rule experiment: In this experiment, we created an additional load by adding a varying number of rules, in the range of 10 to 50, into the P4 switch. Fig. 5.5d shows the same trend as for the adding interfaces experiment; in fact, the time for adding the rules to the switch does not increase with an increase in the number of rules.

As the results show, the creating container step is the step that is most affected by increasing load on the system, and more precisely when the system has to create many containers simultaneously. The steps that are related to the P4 switch, i.e., adding interfaces and adding rules do not change with increasing numbers of interfaces and rules. So we can conclude that the P4 switch is scalable enough for running multiple sharing requests. Also, in all experiments, the calculated setup time is close to the total time and this proves the accuracy of the stepwise model.

## 5.9 Discussion

We must note that in the scenarios in Sec. 5.4 we assume that all of the steps are run successfully. However, in reality, one or more of the steps may fail. This especially affects the parallel scenario (see Sec. 5.6), as the domain in which the error did not occur will continue to reserve resources that will never be used. In this case, running in sequential mode would be more efficient.

Our architecture is flexible and can easily scale up. What we have shown in this chapter is an example of an implementation of a containerized DDM running in one server per domain, e.g. all the containers in one domain reside in one physical node. However, the number of required containers may be more than the capacity of one device. In the case where there is an increasing number of

requests in DDM and more containers are needed, other servers can be easily added to the domain. Each server has its own software P4 switch. All the P4 switches would then be connected with each other and with the physical network infrastructure in the domain.

As we explained all connections between switches are based on a connection ID. Even if a malicious container can find other connection IDs, it cannot use them. As all the packets are going through the P4 software switch, the connection ID can be checked that it is from an eligible container.

## 5.10 Related Work

DDM prototypes are currently in development in a number of scientific and industrial contexts, including the Internet of Things, supply chain logistics, health care, and the exchange of personal information.

Datapace is a commercial blockchain-based DDM platform for trading IoT data streams [99]. It also has a blockchain-based trading infrastructure, in which URLs are traded and data may be retrieved. Datapace sells a curated collection of streams but also allows external sellers on its platform. Data is routed through its central infrastructure. The Ocean Protocol is similar but is inspired more by financial markets, with market makers and derivatives. Data exchange is done directly between buyers and sellers and is somewhat outside the scope of the platform [100]. Both of these systems list data processing as a possible future extension but do not currently support it.

International Data Spaces (IDS) is a DDM project addressing amongst others supply chain logistics. It defines data exchange protocols and provides central components including a data broker, clearing house, identity provider, app store, and vocabulary provider [16]. Data are requested from a data provider, optionally processed, and returned to the data consumer. An example use case is provided by the DL4LD project, which will apply this technology to enable sharing of potentially sensitive data regarding the transport of goods [2].

A science use case concerns personalized medicine: the EPI project will develop a secure and trustworthy platform to share patient data across medical institutions to help with diagnosis and decision making for patients and health providers; the sharing of information will still fully preserve patients' privacy. It also studies policy definitions and how to set up network infrastructure to enforce them [3].

None of these systems support data processing, nor is the implementation of data exchange described at the technical level. Our work shows how to realize the required network connectivity between DDM participating parties, e.g domains.

For building a containerized DDM within one domain, we have already proposed a number of solutions. In [101] we studied whether available container network overlay technologies are suitable for deployment in a DDM. We com-

pared the performance of each technology in terms of the network throughput with an increasing number of network policies and pods. In [102] we remained focused on single domain DDMs but we shifted our focus onto providing isolation between containers and improving the security. We studied three methods of container overlay implementations with particular attention devoted to the isolation between containers while enforcing the data sharing policies.

Xin et al. proposed a multi-domain distributed architecture for policy-driven data sharing applications [103]. The architecture includes components to manage policy auditing as well as to implement network connections. To do this, they use Docker containers and connect containers of each domain via VPN connections. Although this approach can secure the data by encryption, the ease of connection management method on larger scales and also the security aspects had not been studied.

In our current work, we are able to manage connections in a dynamic and straightforward programmable method. As we mentioned we did not cover encryption, but this can also be done in P4 switches. The load of this operation could, if needed, be offloaded to hardware by using SmartNics.

## 5.11 Conclusion

In this chapter, we proposed a multi-domain data sharing architecture that is constructed with containers and software P4 switches. Our architecture supports network connectivity between DDM domains. We described the required steps for setting up the network connections according to the architecture. We reasoned how the architecture is secure against a number of typical network attacks, considering the proposed approach of isolating every single connection via a unique ID number. However, to further validate the security of the architecture against potential attacks, future work may involve performing actual attack experiments and evaluating the efficacy of the proposed approach.

In addition, we studied the network setup time and security implications of adopting P4 programmable switches as underlying technologies. To support network setup and planning, we introduced a model for measuring the setup time and then showed that the model is reliable according to the experiments' results. We also determined that the overhead of using a P4 switch in the setup process is negligible, which makes it a promising technology to support the networking requirements of DDMs.

In Chapter 6, we focus on improving the P4-based network's functionality when it is under high load and enhancing its security by using the P4 program capabilities. To this end, we introduce methods to observe the status of the network based on real-time events and take the proper action accordingly. In addition, we introduce the federated policy-driven data exchange management system (Mahiru) and integrate it with the current work.

# Chapter 6

# Tracking container network connections in a DDM

In this chapter, we first introduce a federated data exchange management system (Mahiru) [104]. In the DDM framework, Mahiru handles the policy checking and manages the communication between participating parties to schedule the execution of the sharing requests.

We then demonstrate how a P4-based network infrastructure can assist Mahiru to acquire insight into the interactions between containers. We relate the incoming traffic to the shared assets by labeling the connections. In addition, we program the P4 switch [105] to gather and classify the information based on the labels. This information will be used to monitor the traffic behavior of each container and consequently the asset related to this container. Mahiru can analyze this information to make further decisions about scheduling the sharing requests and detecting potential anomalous behaviour in data transfer between containers. This chapter is related to RQ4: *How can P4-based network capabilities assist the data sharing management system in providing security and maintaining quality?*

This chapter is based on:

**Shakeri, S.**, Veen, L., and Grosso, P. "Tracking container network connections in a Digital Data Marketplace with P4". In 2022 International Conference on Computer, Information and Telecommunication Systems (CITS), Piraeus, Greece, 2022, pp. 1-8.
Veen, L., **Shakeri, S.**, and Grosso, P. "Mahiru: a federated, policy-driven data processing and exchange system". Submitted to arXiv:2210.17155.

## 6.1 Introduction

DDMs currently in development support access to data or algorithms from other parties, federated machine learning, and other forms of distributed data processing. DDMs that support distributed applications such as federated machine learning need to run software made by third parties or users of the system. This presents a security risk. Besides sandboxing and enforcing network connectivity policies, two other ways exist to mitigate this risk: code inspection and monitoring. Code inspections can detect bugs and malicious code, but they cannot do so perfectly; they are time-consuming, and sufficient expertise needs to be available. Furthermore, measures need to be taken to ensure that the code that runs matches exactly the code that was inspected, which entails repeatable builds and re-reviewing every new release. As a result, code inspections are expensive and cumbersome. Monitoring execution may provide an alternative or addition. Monitoring can be automated completely using AI and the potential for catching malicious behavior will deter would-be adversaries and reduce the risk in the system even if it is not guaranteed to catch all malicious behavior.

In addition, monitoring can help to improve performance. As the size of DDM increases, more data must be shared with more peers, and the scalability of the system becomes important. In a containerized DDM, where both data sets (data assets) and algorithms (compute assets) are containers, scalability can be achieved by adding more instances (horizontal scaling), if the network infrastructure is flexible enough to route incoming requests accordingly. To steer this process and select the method for scaling, the container's transactions must be tracked.

In this chapter, we first explain the implementation of the policy driven data exchange management system (Mahiru) that is in charge of checking the consistency of the user's sharing requests with the available policies and then planning the execution in the network infrastructure. Afterward, we present how Mahiru can be integrated with the DDM network infrastructure [104].

We then focus on monitoring the network traffic between containers. Since our infrastructure uses P4 switches, monitoring data can be gathered by P4 programs running on the switches. The sharing transactions can be logged in the switch and then sent to the controller or upper layers for further analysis. In our architecture, each container has only one asset. We define a ***sharing transaction*** as the action of transferring an asset from one container to the other one through an established connection. However, tracking the transactions related to the assets is not trivial. The traffic that is being logged must be related to each asset, i.e., what is understandable for the data exchange management system. Therefore, just the information about the traffic of different flows in the network is not helpful by itself. A mechanism has to connect the traffic flows to the asset and their corresponding policies.

We design a P4-based containerized network that can handle these challenges.

We specify a unique ID for each connection, and we use this ID in network connections for transferring the assets. By using this connection ID, we can track the transactions related to each asset in the DDM. We present sharing scenarios in a DDM to demonstrate how connection tracking can assist in providing security and improving performance.

The main contributions of this chapter are:

- Introducing Mahiru as a federated data exchange management system that can be deployed in a DDM
- Building a containerized DDM in which the flow of the traffic in the network infrastructure is associated with the shared assets in the DDM
- Demonstrating how the sharing transactions can be tracked in that infrastructure
- Presenting examples of the tracking scenarios to show how logging information by P4 can improve security and performance in DDM

## 6.2 Federated data exchange management system (Mahiru)

This section aims to give an introduction to Mahiru, a federated data exchange management system that is a work of collaboration with the eScience center of netherlands [1]. It is designed for federated data processing and exchange operations between multiple organizations in science that can be used in DDMs [106]. In designing the proposed architecture in this chapter, we considered the features and requirements of Mahiru. Mahiru and the P4-based container network introduced in Chapter 5 can be integrated to complete the process of execution of the sharing requests [104].

Different projects are currently in progress to create different kinds of cross organizational data sharing systems, including centralized download sites, peer-to-peer data exchange systems, and designs for federated learning and other kinds of distributed data processing. These data exchange and processing patterns can all be expressed as workflows in the Mahiru data exchange system. Mahiru runs the data sharing workflows and supports different models of distributed data exchange. It supports enforcing the policies that are established by the owners of data to determine how the data can be processed or transferred. The users of Mahiru submit their requests to their local setup on their own site. Mahiru plans and executes the steps for executing the requests according to the pre-established policies in collaboration with other involved sites. In Mahiru ***policies***

[1] https://www.esciencecenter.nl/projects/secconnet/

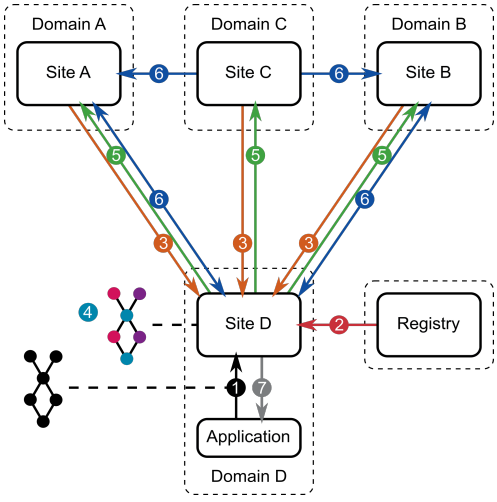



Figure 6.1: Mahiru's global architecture and operation. The system consists of any number of sites and a registry which records their identity and location. 1) Workflow submission, 2) registry update, 3) policy update, 4) planning, 5) execution request, 6) distributed execution, 7) result return.

are a collection of rules to permit sharing, processing, and delegation of control over assets. The rules are based on assets, parties, and sites. Assets are digital resources that are shared in DDM. Sites store assets, as well as their exchange policies. Parties are the owner of the assets that are participating in DDM. Each object in a DDM (Asset, Party, Site) has a unique identifier. **MayAccess**(asset, site) is an example of Mahiru's rules. This rule determines which data/algorithm asset can be presented/executed on which site. Mahiru also has rules for *data processing* and *organising objects*.

Requests for processing the data in Mahiru are ***workflows*** in the form of Directed Acyclic Graphs (DAG). Shared asset transfers along the edges, and each node represents a workflow step in which a compute asset processes one or more data assets.

The overview of Mahiru's global operation is shown in Figure 6.1. The user's application initiates the execution by submitting a workflow to the local Mahiru site (1). The site ensures the validity of its information about the parties and sites involved in the data sharing system. This information is stored in *Registry*(2); it then checks the current state of the policy replicas for the sites serving policies for the assets used in the workflow (3). After these checks, the site creates an execution plan (4) that determines the site at which each step in the workflow will be executed. The site then sends an execution request (5) containing the workflow and the plan to each of the involved sites. Upon receiving the request,

each site verifies that they have up-to-date policy replicas and that the requested actions are permitted before starting to execute the steps as their inputs become available.

## 6.3 Architecture

Fig. 6.2 represents the architecture of the containerized data sharing system. It is constructed of five main building blocks: the data exchange management system block, the administration block, the containerization, and networking block, and the monitoring block. In multi-domain DDM domain administrators communicate to build the connection according to the relative sharing policies. More details about the method of communication are represented in Sec. 6.4.

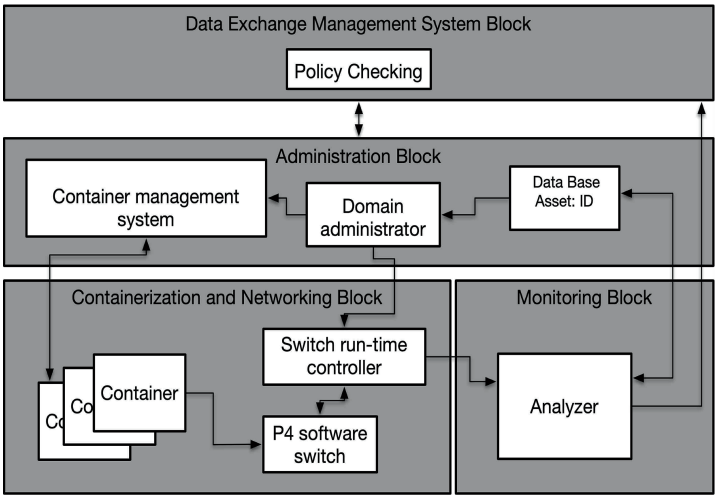



Figure 6.2: Architecture of the containerized DDM with tracking capability

**Data exchange management system block**: This component is in charge of checking the policies and collaborating with other domains for scheduling a request or a chain of requests (see Sec. 6.2).
**Administration block:** This block contains three components. The container management system creates the container and attaches the related asset. Containers containing a data asset are called *data containers*, and containers containing a compute asset are called *compute containers*. The containers are then connected to the switch.

The unique connection ID related to each connection is generated and then saved in a database. This connection ID will then be used for tracking the sharing transactions. Finally, there is a domain administrator. It takes the information

about the created container from the containers management system and the connection ID from the database. It then creates the appropriate P4 rules that make the allowed connections to this asset possible. These rules are passed to the controller.

**Containerization and networking block**: In this block, the P4 switch connects the containers. When the packets arrive with the specific connection ID, they are matched with the rules that are set by the controller and sent to the corresponding container. Importantly, the information of the traffic going through the switch will be saved in the switch's registers. Registers are stateful elements used to store values. They keep the state of the network between various network packets. The controller is in charge of setting up the rules on the switch. In addition, it reads the information from the registers of the switch when it is needed and sends it to the monitoring block.

**Monitoring block**: In this component, the monitoring information is translated from the networking traffic information to *asset* access information. Of course, it uses the connection ID mapping database to relate the traffic with specific connection IDs to assets. The information can be reported to the data exchange management system or domain administrator for further action.

## 6.4 Proof of Concept

Fig. 6.3 shows the steps for setting up the network for transferring a data asset from a data container in domain2 to a compute container in domain1. We call domain1 which is requesting the data the *client-side* and domain2 which is serving the data the *server-side*. Domains are distinguished by their IP addresses. The IP address is the IP of the server hosting the container. For our implementation, we used Ubuntu 18.04 and Linux kernel 5.4.0 as the host OS, Docker Community Edition 20.10 for container management, and bmv2 P4 switch as the programmable software switch [97].

In the following, we will guide the reader step by step through each of the operations that occur in a domain to set up the connection between containers:

**step1:** The request from the other domain will be checked against the available policies in the data exchange management system. If the request is allowed, the data exchange management system sends the request to the domain administrator to set up the network for the required connection.

**steps 2 and 3:** The domain administrator asks the ID generator to generate the ID for the connection for transferring the data asset from the data container to the compute container in step 2. The connection ID will be sent back to the administrator in step 3. The connection ID is a unique number for each Data asset ID, client-side IP address, and Compute asset ID:

$$Connection\ ID = Unique\ number\ of(DataAssetID, Client{-}sideIPaddress, ComputeAssetID)$$

**step4:** The domain administrator asks the container management system to

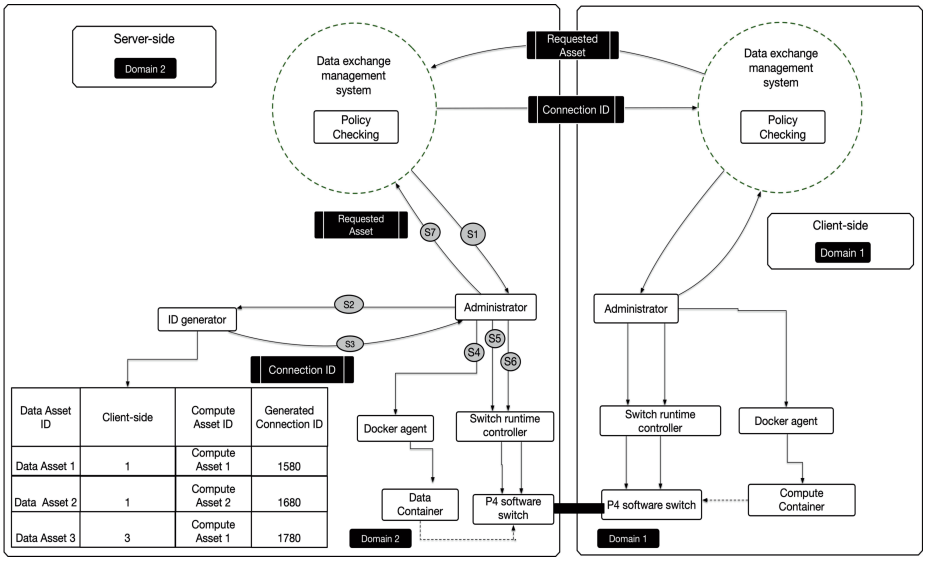



Figure 6.3: Steps of setting up the network for a request from client-side to server-side. (An extended version of Fig. 5.2. In this figure we added Mahiru and the table of generating the connection ID)

create a container for the Data Asset. The container management system then sends back the container specification to the domain administrator.

**step5:** The domain administrator connects the container to one of the ports of the switch.

**step6:** The domain administrator creates the appropriate rules and sends them to the controller of the switch for setup. In the connection from the client-side to the server-side, the connection ID will be used as the destination port of the connection.

In the following, we explain the rules of the server and client-side.

**server-side:** Listing 6.1 shows the rules and actions for incoming packets to the server-side in our setup. In this case, the packet's destination port number (the connection ID), and its source IP address - which is the client-side IP address - will be checked (line 4-5).

We first save the destination port number (connection ID) in a variable for further use (line 14). For sending the packet to the server-side container, we have to change the destination IP address to the server-side container's IP address (line 17) and the port number to the number that the server-side container is listening to (line 18). At this point, the packet is ready to be sent.

When a packet comes back, we have to be able to set the port number and IP address back to the original one. Therefore, before sending the packet to the

server-side container, we save the original source port in a register (line 15) and modify the source port number of the packet to the previously saved connection ID (line 16).

```
1
2     table server_receive_t {
3         key = {
4         hdr.tcp.dstPort :exact;
5         hdr.ipv4.srcAddr : exact;
6
7         }
8         actions = {server_receive; }
9     }
10
11
12      action server_receive(bit<32> dst_ip,
    bit<16> dst_port, bit<9> port){
13
14          Connection_ID_s = hdr.tcp.dstPort;
15          reg_srcport.write(Connection_ID_s ,
    hdr.tcp.srcPort);
16          hdr.tcp.srcPort = Connection_ID_s;
17          hdr.ipv4.dstAddr = dst_ip;
18          hdr.tcp.dstPort = dst_port;
19          stdmeta.egress_spec = port;
20
21      }
```

Listing 6.1: P4 table and actions related to the incoming packets in the server-side

Listing 6.2 shows the rules and actions for outgoing packets from the server-side to the client-side. In this case, the packet's destination port number (the connection ID) and its destination IP address, i.e., the client-side IP address, will be checked (line 4-5).

The destination port of the current packet is saved in a variable (line 12). When a packet is coming back from the server-side container, we have to change its source IP address to the IP address of the server-side (line 13). We then have to change the source and destination port numbers. We change the source port number to the connection ID that we saved in the variable (line 14) and the destination port to what we have saved in the register (line 15). Finally, the packet will be sent out to the related egress port of the switch (line 16).

```
1
2     table server_send_t {
3         key = {
4         hdr.tcp.dstPort :exact;
5         hdr.ipv4.dstAddr :exact;
6         }
7         actions = {server_send; }
8      }
9
10     action server_send(bit<32> src_ip, bit
    <9> port){
11
12         Connection_ID_c = hdr.tcp.dstPort;
13         hdr.ipv4.srcAddr = src_ip;
14         hdr.tcp.srcPort = Connection_ID_c;
15         reg_srcport.read(hdr.tcp.dstPort,
    Connection_ID_c);
16         stdmeta.egress_spec = port;
17     }
```

Listing 6.2: P4 table and actions related to the outgoing packets from the server-side

**Client-side:** On the client-side, the routing between containers is more straightforward. When a packet is sent out to the server-side, we change its source IP address to the host IP address. When a packet comes back from the server-side, it can be directed to the right container by looking at the connection ID (source port). However, as different domains may give the same connection ID for different requests related to the same host, the connection ID may be used by another server-side host. Therefore, what is unique is the combination of the source IP address and the connection ID that will be used for matching rules.

## 6.5 Tracking Scenarios

We evaluated the use of P4 switches in DDMs in two different cases corresponding to the two concerns raised in the introduction: scalability and security. If a DDM participant has a popular asset, or the DDM contains another participant with a large number of users that access many assets simultaneously, then the system of the data provider must scale to meet demand, and the network reconfigured accordingly. It is investigated in Sec. 6.5.1 as *Access Tracking*. For the security case, we focus on distributed data processing, in particular federated learning. In this use case, data transfer needs to be repeated multiple times. Therefore, the compute container keeps the connection open and asks for data whenever it is needed within the same session. In this type of connection, there will be an *active_time* where the packets are being transferred and an *idle_time* where

no packet is being transferred, but the connection is open. We observe it as a traffic pattern. The real network traffic is compared with this pattern to detect the possible anomalies in data transfer. It is investigated in Sec. 6.5.2 as *Pattern Tracking*.

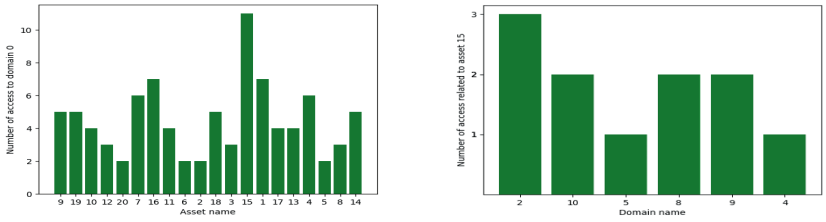
### 6.5.1 Access tracking

This case allows us to track the access behavior of the domains in a DDM to a specific data asset. This information can help in enhancing the performance by scaling the number of containers. When the load on the system is high, new containers are generated, and the load is distributed between these containers. In such a case, the containers that have the most effect on the load have to be selected to be duplicated and moved toanother server. After duplication, part of the load should be rerouted to these new containers.

We look at two values to select which containers have to be duplicated and run on another server when the network is under high load. The first is the number of access to each asset, and the second is the number of access from each domain. We calculate these numbers by identifying the client-side's domain, the client-side's compute container, and the server-side's data container using the connection ID.

Accordingly, there are two scenarios for tracking access. 1) asset access; 2) domain access; which are explained in the following. In addition, we explain a method for detecting the high load on the system at the end of this section.

1. **Tracking based on asset access:** In this case, we count the number of access to different data assets on the server-side to determine which asset or assets have the most access. To count the number of access to each data asset, we save the number of connections to data containers in the switch indexing by the connection ID. We then calculate the number of connections to the data containers from different domains. If the number of connections of a data container is more than a certain threshold, we select that data container for duplication. To illustrate with an example how this would work, we simulated a scenario of running sharing requests by considering 10 different client-side domains accessing 20 data assets in a server-side domain (Fig. 6.4). We set the average access to assets as the threshold. We counted the number of access to each asset by relating the connection ID to its asset (Fig. 6.4a). According to Fig. 6.4a, the average number of access from different domains to all data assets is 7. The number of access to asset 15 is more than the average. Therefore, it will be selected for duplication. We then have to select which domain's requests have to be rerouted to the new container. This can be done by looking at the number of access of each domain to the selected asset as shown in Fig. 6.4b. We divide the load related to asset 15 between two containers by sorting the domains based on



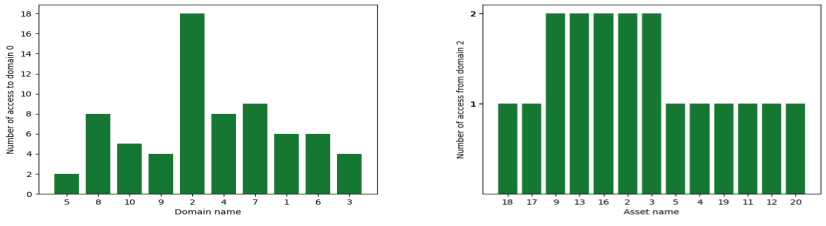

(a) Number of access to the data assets (b) Number of access to asset15

Figure 6.4: Asset access tracking in a DDM using connection ID

their number of access and then assigning each domain to one of the two containers from the beginning of the sorted list.

2. **Tracking based on domain access:** Another reason for a high load on the system can be lots of connections from client-side domain to server-side domain but not necessarily to one individual data container. A client-side domain may make connections to different data containers of the server-side domain so that the number of connections to a specific container is not high; however, the total number of connections related to this client-side domain is high, leading to increased load on the system.

   In this case, we simulated a scenario of running the sharing requests from 10 different client-side domains to server-side domain0 (Fig. 6.5).




(a) Number of access from the other domains to domain0 (b) Number of access to data assets from domain2 to domain0

Figure 6.5: Domain access tracking in a DDM using connection_ID

In Fig. 6.5a, the most number of access is from domain 2. For distributing the load, by detecting the most referring assets of this domain (Fig. 6.5b), we can duplicate the containers and reroute the connections of this domain to the new containers.

**Detecting high load:** To detect the high load on the system, we can consider the total time of a data transfer and the number of running connections as metrics. This method works when the connections that are made by the application are always active. In other words, the packets are being transferred continuously between SYN and FIN time. The thresholds in this method are application-specific and have to be set accordingly.

The total time of a sharing request can be measured in a P4 program by calculating the time difference between the SYN and the FIN flags of a connection. In P4, whenever an SYN packet of a connection is seen, its ingress time will be saved in a register with the index of its connection ID. The total time is then saved in another register when the FIN flag of that connection arrives. When the total time is calculated, it will be compared with the expected total time of the connection that is set by the controller. If it is more than what is expected, it will be counted as the connections that their total time is more than the threshold. When the number of requests with a total time of more than a threshold crosses a specific number (that is determined by the controller), the switch sends a notification to the controller as a situation where the load on the system is high.

When the controller receives a notification from the switch, it also reads the register that has the number of running connections. The switch keeps the number of running connections in the system by counting the connections that their SYN flag is seen, but their FIN is not. The combination of the high number of running connections and the long total time is a sign that the system is under a high load. The P4 code of this implementation is available on GitHub [107].

### 6.5.2 Pattern tracking

The last illustrative scenario is about detecting a specific pattern in the activities related to the connection from a compute asset to a data asset. There are lots of algorithms whose behavior in making the connections and transferring the data has a specific pattern. By detecting this pattern, the anomalies like system failure or malicious behaviors can be promptly identified to make the system more secure. In addition, the state of the system, like the number of incoming requests and their corresponding load, can be predicted. Our goal is to detect the pattern, and we perform it by programming the switch using the connection ID.

As an example, we consider federated machine learning algorithms' behavior. When the data has to stay in its location and cannot be moved, federated machine learning algorithms are deployed for performing the computation on data. Therefore, each part of data is in a location that is different from the other, and the model will be trained in a location where data resides [82, 108]. After training the model, the model parameters need to be transferred to where all the information is gathered, and the general model can be built [109]. This process is repeated multiple times so the training process is complete. In this case, the

compute container on client-side keeps the connection open for better optimization. Therefore, sometimes the connection is active and the data is transferred, and sometimes it is idle. If this process is run in our setup, what we observe on the server-side is a pattern of connections from the client-side to a specific data container. In these connections, the same amount of data is transferred every time, and they repeat with a specific time interval.

For detecting the pattern, we have to be able to detect the idle_time, the active_time, and the amount of data that is transferred between containers in active_time. We use a list of registers in the P4 program for finding these numbers. We define the registers and index them with the connection ID. The registers are:

1. Last_packet_ingress_time $< 48bit >$: This register always keeps the arrival time of the last packet.
2. Start_idle_time $< 48bit >$: This register keeps the start time of an idle_time.
3. End_idle_time $< 48bit >$: This register keeps the end time of an idle_time.
4. Total_data $< 48bit >$: This register keeps the total amount of data that is being transferred from the first packet to the last packet of the connection.
5. Data_before_max_idle_time $< 48bit >$: This register keeps the amount of data that is being transferred before the maximum idle time.
6. Time_interval $< 48bit >$: This register keeps the maximum duration of the idle_time.

**Algorithm 1** Finding the maximum idle_time of a TCP connection in P4

**Define** Last_time, Current_time, Difference
Last_time = Last_packet_ingress_time[Connection ID]
Current_time = packet.metadata.ingress_time
Difference = Current_time - Last_time
**if** *Difference > Time_interval[Connection ID]* **then**
    Update Time_interval[Connection ID] with Difference
    Update Start_idle_time[Connection ID] with Last_time
    Update End_idle_time[Connection ID] with Current_time
    Update Data_before_max_idle_time[ConnectionID]withTotal_data[Connection ID]
**end**
UpdateLast_packet_ingress_time[ConnectionID]with packet.metadata.ingresstime

The algorithm we implemented in the switch to detect the idle_time is shown in Algorithm 1. Whenever a packet is coming to the switch, the difference between the arrival time of the packet and the arrival time of the last packet will be saved

in a local variable in the P4 program. If the current difference is more than the previous difference that has been saved in the time_interval register, then the time_interval will be updated. The transferred data will be updated accordingly.

The registers are read by the controller repeatedly. However, in this method, only the maximum idle_time is saved. To avoid that, every time the controller reads the registers, it sets the time_interval to zero. As a result, different duration of idle_time can be detected.

We performed an experiment to illustrate the pattern tracking method. In our experiment, a compute container was running on the client-side in a physical machine. A data container was running on the server-side in another physical machine. The physical machines were connected via the internet. We generated traffic that follows a specific pattern. The same amount of data was transferred through the network every 5 seconds within an open connection.

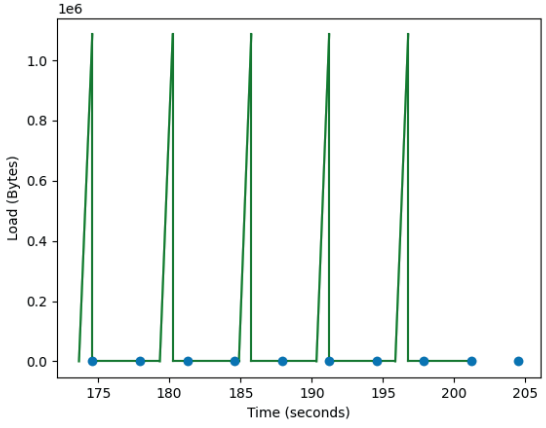



Figure 6.6: Pattern Tracking: active and idle_time of a connection from a compute container to a data container. The points show the time the information is read by the controller. The numbers of each reading time are shown in Table 6.1.

Table 6.1 shows the content of registers every time the controller reads the information from the switch. The polling time is when the controller reads the registers that is in every 3 seconds. Note that the time_interval was set to zero after the controller read the registers. The times in the table are the switch's time; that is, the time from when it started in seconds. According to Table 6.1, for finding the active_time and idle_time after reading the registers, if the Start_idle_time was equal to the previous one, we ignored that row of information. Among the remaining rows, we considered the time between Start_idle_time and End_idle_time of a row as idle_time and the End_idle_time of a row and Start_idle_time of the

Table 6.1: P4 register's content for finding the pattern of the connection from a compute container to a data container

| Polling_number | Polling_time | Start_idle_time | End_idle_time | Time_interval | Last_packet_ingress_time | Data_before_max_idle_time |
|---|---|---|---|---|---|---|
| 1 | 174,59 | 173,64 | 173,65 | 0,01 | 174,59 | 60 |
| 2 | 177,94 | 173,64 | 173,65 | 0,00 | 174,59 | 60 |
| 3 | 181,25 | 174,59 | 179,33 | 4,74 | 180,24 | 1086475 |
| 4 | 184,59 | 174,59 | 179,33 | 0,00 | 180,24 | 1086475 |
| 5 | 187,91 | 180,24 | 184,88 | 4,64 | 185,75 | 2172751 |
| 6 | 191,22 | 185,75 | 190,33 | 4,57 | 191,22 | 3259027 |

next row as the active_time. Fig. 6.6 shows the pattern based on the numbers in the table. The blue points in the figure are the polling time.

In the second row of the table, the time_interval is 0.0; that is because after the previous time that it was set to zero, no packet came, so it has not changed. This is the second point in Fig. 6.6. In the figure, we show the load that is transferred each time based on Table 6.1. The load that is shown in the table is the accumulative load that is read related to each connection ID.

The polling time has to be less than the sum of active_time and idle_time. Therefore, it has to be justified by the behavior of the compute asset. Polling time can be set with a small number at the start. After realizing the first active and idle_time, we update the polling time. We then adjust the pulling time every time we read the information.

## 6.6 Related Work

Multiple works have been done in the area of network flow monitoring in programmable data planes. For example, in [110] the authors design and implement a bandwidth-efficient in-band network telemetry system that can track the rules matched by the packets of a flow. They use a traffic reduction scheme in their INT system to reduce the rate of generated INT reports. They also store INT reports about the changes in the rule by using global unique IDs for every rule. Also, in [111] authors use hash tables to maintain the whole information about elephant flows but summarize records for mice flows, by applying a novel collision resolution and record promotion strategy to hash tables. We use the connection ID for saving the information and, in addition, set it as the destination port number of the connection to reduce the overhead. In our methodology, we show how using the connection ID as the destination port number of the connections is possible.

[112] proposes FlowLens. It is a system for traffic classification to support security in network applications based on machine learning algorithms. The authors propose a novel memory-efficient representation for features of flows called flow marker. A profiler running in the control plane automatically generates an application specific flow marker that optimizes the trade-off between resource consumption and classification accuracy according to a given criterion selected

by the operator. The authors in [113] try to design a low overhead system for monitoring and gathering information by deploying a P4 program. The monitoring phase in their setup includes a proactive phase that keeps the per-flow packet counter and a reactive phase that runs for large flows only and gathers metrics of the flow, e.g., packet counts and packet timestamps. Our proposed method is specifically designed for data sharing applications. It builds a multi-domain container-based DDM that provides the possibility of tracking the transactions related to the shared assets in the network layer.

In addition, there are some works that propose methods of monitoring in a DDM. For example, in [114] the authors propose an architecture to distinguish programs running inside containers by monitoring system calls. [115] offers an intrusion detection system based on OC-SVM that monitors and analyzes system calls of containers. [116] designs an auditor node in containerized DDM that is a node responsible for authorization through signatures application transactions. These methods focus on the operation of the containers in system calls and on the application layer. Our method focuses on monitoring the data exchange behavior of containers in the network layer through P4 programming. In this way, extra information like the amount of transferred data or the number of connections that have been made during a specific time can be extracted to make them available to the application layer.

## 6.7 Conclusion

In this chapter, we first showed how the data exchange management system can cooperate with a P4-based containerized DDM. We improved the architecture introduced in Chapter 5 by adding the capability of tracking the transactions related to the shared assets. We specified a unique connection ID for each connection and used this ID for transferring the asset in a sharing request from one container to the other. The connection ID is used as the destination port of the connection between containers. We used P4 to program the switch, make the connections based on the connection ID, and gather traffic information. We explained each step of setting up the network and presented the required configuration. We then demonstrated how we can use the connection ID for tracking the sharing transactions between containers and, accordingly, the transactions related to each shared asset. We presented asset access, domain access, and pattern tracking scenarios to show what kinds of information we can extract from the switch based on our setup and how to use this information to be able to provide better performance and security in a DDM.

# Chapter 7

# Conclusion

This Ph.D. thesis presented research on security measurements of the network infrastructure of containers for data exchange in a DDM. As the demand for data sharing increases, it becomes critical to make a secure data exchange infrastructure in a DDM. Part of this infrastructure is network connections between containers that make the data transfer from one participating party to the other one possible. Containers and the technologies that connect them together play an important role in security in a DDM. The connections between containers are based on the policies that are established between participating organizations in a DDM. They have to be described in a generic model in the whole DDM to make policy management more efficient. More importantly, the technology that connects containers has to support enforcing the agreed sharing policies. Sharing policies have to be translated to network policies, and then they have to be set between containers. In addition, for building a secure data sharing network infrastructure, providing the ability to define different levels of isolation between containers based on the application's needs is essential. On the other hand, as the containers' configuration and sharing policies may change, programmable network infrastructure is needed to support the changes in the network.

In this thesis, we studied how container and recent software developments, especially Kubernetes and P4, allow us to build a network of containers that can handle the abovementioned requirements. Through the experiences we gained from setting up such a network in both single and multi-domain environments, we identified the strong points and the limitations of the technologies used. In addition, we obtained insights to detect the essential elements to build the network infrastructure of the DDM using containers and presented different architectures of container networks, and reasoned how they are secure against different types of attacks.
The main contributions described in the work are:

- A generic model for describing DDM sharing policies

- Different architectural designs with different connectivity levels between containers using Kubernetes and Docker Swarm
- A programmable framework for building a multi-domain DDM and managing the containers connections using P4
- Tracking container connections in a DDM using P4

With these contributions and the insights we gathered during the experiments and the proof of concepts, we will now be able to answer the research questions.

## 7.1 Answers to the research questions

The aim of this thesis is to investigate the methods for improving the security of network connections between containers to build the network infrastructure of a DDM. Our main research question is formulated as:

> ***RQ: How does a container-based infrastructure guarantee secure and high-performance data sharing among organizations?***

To answer this question, we have defined a number of sub-questions that cover several aspects of our main research question more specifically:

- ***RQ1: How to describe high-level data sharing policies, and design a module for accepting or rejecting the sharing requests?***

  In order to be able to describe the sharing policies in a DDM we selected Open Digital Rights Language (ODRL). ODRL is a language for modeling the permissions and prohibitions regarding digital resources, i.e., assets. In Chapter 2, we defined ODRL policies as a set of rules. A rule includes the digital asset that is shared, the actions that can be done on the asset, and the location where the actions can happen. In addition, we defined a matching module to manage the user's requests automatically. The matching module uses SPARQL queries to verify if the user's requests are allowed or not. The user has to specify digital resources, the location of execution, and the location where the results have to be sent after the request execution.

- ***RQ2:How can overlay network technologies provide the required policy enforcement and isolation, while maintaining quality in a sharing environment?***

  To address RQ2, the following research questions were defined and addressed:

- RQ2.1: What are the functionalities of the available overlay technologies for managing container connectivity and enforcing sharing policies?

  The capabilities of various container overlay technologies were examined in Chapter 3, with a focus on how they establish connections between containers and enforce sharing policies. Two of the most popular technologies were selected and compared in terms of their ability to support the number of containers and policies required in a DDM.

- RQ2.2: Can different configurations of available container overlays meet the requirements of a DDM?

  In Chapter 4, different types of sharing requests were characterized based on varying isolation levels and mapped to container connectivity methods, including Overlay per DDM, Overlay per request, and Overlay per group. Using the results obtained in Chapter 3, each connectivity method was implemented with the appropriate technology and compared in terms of security and performance. The investigation revealed that the "Overlay per request" method was less susceptible to cross-container attacks than the other methods, and there was little difference between the methods in terms of the time taken to complete sharing requests.

- ***RQ3: How to build a containerized multi-domain DDM on a programmable network infrastructure to enforce sharing policies?***

  To answer RQ2 we used Kubernetes and Swarm as centralized orchestrators that can manage the connections. However, for building a multi-domain container-based network in which each domain can orchestrate its own resources, we could not rely on these technologies. To answer RQ3, we proposed using P4 software switches and presented an architecture that supports multi-domain data exchange in a DDM in Chapter 5. Each domain has one or more P4 switches to which the containers are connected. The switch is programmed by its domain administrator. In this way, the connections between the containers of the domain are managed independently from the other domains. In addition, the connections are programmable and can be modified when needed. To provide maximum isolation between containers and the sharing requests, we introduced the concept of the connection ID for each connection between two containers. The proposed method led to a multi-domain programmable network infrastructure that supports policy enforcement and can benefit from the advantages of the P4 programming language.

- ***RQ4: How can P4-based network capabilities assist the data***

***sharing management system in providing security and maintaining quality?***

We improved the proposed architecture in Chapter 5 to answer RQ4. In Chapter 6, we used the connection ID to relate the traffic passing through the switch to the assets shared in the DDM. We programmed the switch to track the behavior of containers in making connections and transferring data. We demonstrated how tracking could help in two cases: Scalability and Security. By looking at the tracking information, we realized which containers needed to be scaled up. By detecting the pattern of connections between containers, we assisted in detecting the malicious behavior of containers when there is an anomaly to improve security. In addition, we introduced a federated data exchange management system (Mahiru) that manages the execution of the sharing requests after checking them against DDM policies. We showed where it can be combined with the P4-based containerized infrastructure to complete the process of executing a sharing request in a DDM.

## 7.2 Future work

During our research and explorations, we identified topics for future research:

1. ***Supporting containerized network functions***

   In this thesis, we investigated the methods of connectivity between containers focusing on using the containers as *data containers* that include shared data, and *compute containers* that include shared algorithms. However, the containers can also be deployed as network functions to improve security and performance. The method of instantiating, bringing the connectivity, and the order of running such dynamic containerized network functions is a challenge that can be investigated further.

2. ***Supporting dynamic topology changes***

   In Chapter 4, we defined different container network typologies based on application needs. According to such needs of an application, these topologies may change, e.g., removing a container from a network or adding a new one. For example, in Chapter 6, we proposed instantiating new containers according to the results we got from tracking the connections. Creating new containers leads to changes in the network topology. Moreover, when including containerized networking functions, more complicated topologies have to be supported. We conclude that supporting a dynamic network topology in a container-based network is essential. Evaluating the performance of the architectures proposed in this thesis in supporting the dynamicity in the network is an interesting future work.

3. ***Improving tracking scenarios***

   We presented a pattern tracking method in Chapter 6 to show how P4 can assist in monitoring the network and improving security. The method can be improved by considering more metrics like the requests execution time, connections' endpoints, and communication protocols. In addition, we used federated machine learning algorithms' behavior as a use case. However, more complicated use cases with different patterns can be defined. In this case, the capability of P4 in detecting the patterns has to be evaluated.

4. ***Improving network resource management***

   The method of allocating network resources to containers is still a challenge. For example, it is still not possible to limit the bandwidth utilization of a container in its host. In Chapter 6, we demonstrated how each containers' connection has a specific connection ID. This allows us to be able to determine the resource utilization of each container in P4. We need to research if P4 can support running an algorithm that manages the utilization of resources like available bandwidth according to the application considerations.

5. ***Integrating of Mahiru and P4-based containerized network infrastructure*** In Chapter 6, we introduced Mahiru, a federated data exchange management system in which the owners have complete control over their shared data and users can submit a wide variety of requests. In designing the network infrastructure of DDM, we considered the characteristics of Mahiru to make the containers' network consistent with Mahiru's features. Although the container-based network infrastructure and Mahiru are already implemented, they are still not fully integrated. It is necessary to investigate the integration methods of the proposed containerized network with Mahiru.

# Bibliography


[1] "Fleet and MRO Forecast," https://www.planestats.com/about, 2023, [Online; accessed July 2023].

[2] "DL4LD," https://www.dl4ld.nl, 2023, [Online; accessed April 2023].

[3] "Enabling Personal Intervention," https://delaat.net/epi/, 2023, [Online; accessed April 2023].

[4] "What is Data Sharing," https://www.gensquared.com/what-is-data-sharing/#gref, 2023, [Online; accessed July 2023].

[5] K. Suo, Y. Zhao, W. Chen, and J. Rao, "An analysis and empirical study of container networks," in *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, April 2018, pp. 189–197.

[6] S. Sultan, I. Ahmad, and T. Dimitriou, "Container security: Issues, challenges, and the road ahead," *IEEE Access*, vol. 7, pp. 52 976–52 996, 2019.

[7] H. Zeng, B. Wang, W. Deng, and W. Zhang, "Measurement and evaluation for docker container networking," in *2017 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, Oct 2017, pp. 105–108.

[8] L. Makowski and P. Grosso, "Evaluation of virtualization and traffic filtering methods for container networks," *Future Generation Computer Systems*, vol. 93, pp. 345 – 357, 2019.

[9] J. Claassen, R. Koning, and P. Grosso, "Linux containers networking: Performance and scalability of kernel modules," in *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*, April 2016, pp. 713–717.

[10] D. Thilakanathan, S. Chen, S. Nepal, R. Calvo, and L. Alem, "A platform for secure monitoring and sharing of generic health data in the cloud," *Future Generation Computer Systems*, vol. 35, pp. 102 – 113, 2014, special Section: Integration of Cloud Computing and Body Sensor Networks; Guest Editors: Giancarlo Fortino and Mukaddim Pathan.

[11] M. Ali, R. Dhamotharan, E. Khan, S. U. Khan, A. V. Vasilakos, K. Li, and A. Y. Zomaya, "Sedasc: Secure data sharing in clouds," *IEEE Systems Journal*, vol. 11, no. 2, pp. 395–404, June 2017.

[12] D. Harris, L. Khan, R. Paul, and B. Thuraisingham, "Standards for secure data sharing across organizations," *Comput. Stand. Interfaces*, vol. 29, no. 1, pp. 86–96, Jan. 2007.

[13] L. Gommans, J. Vollbrecht, B. G. de Bruijn, and C. de Laat, "The service provider group framework: A framework for arranging trust and power to facilitate authorization of network services," *Future Generation Computer Systems*, vol. 45, pp. 176 – 192, 2015.

[14] R. Cushing, R. Koning, L. Zhang, C. d. Laat, and P. Grosso, "Auditable secure network overlays for multi-domain distributed applications," in *2020 IFIP Networking Conference (Networking)*, 2020, pp. 658–660.

[15] "International Data Spaces Association," https://www.internationaldataspaces.org, 2022, [Online; accessed July 2022].

[16] "International data spaces reference architecture model version 3.0," April 2019. [Online]. Available: https://internationaldataspaces.org/download/16630/

[17] "Smart Connected Supplier Network," https://smart-connected.nl/en, 2022, [Online; accessed July 2022].

[18] "Green Village Sharing Platform," https://bit.ly/2HQDUNu, 2022, [Online; accessed April 2022].

[19] "NLIP," https://www.nlip.org, 2019, [Online; accessed April 2019].

[20] "Docker," https://www.docker.com/, 2019, [Online; accessed June 2019].

[21] "What are namespaces and cgroups?" https://www.nginx.com/blog/what-are-namespaces-cgroups-how-do-they-work/#:~:text=Namespace, 2023, [Online; accessed July 2023].

[22] "Kubernetes," https://kubernetes.io/docs/tutorials/kubernetes-basics/, 2019, [Online; accessed June 2019].

[23] C. Boettiger, "An introduction to docker for reproducible research," *SIGOPS Oper. Syst. Rev.*, vol. 49, no. 1, pp. 71–79, Jan. 2015.

[24] "Software-Defined Networking (SDN) Definition," https://opennetworking.org/sdn-definition/, 2022, [Online; accessed July 2022].

[25] "ONF Specifications," https://opennetworking.org/sdn-resources/onf-specifications/onf-specifications/, 2022, [Online; accessed July 2022].

[26] "P4: Open Source Programming Language," https://p4.org/, 2022, [Online; accessed April 2022].

[27] M. Menth, H. Mostafaei, D. Merling, and M. Häberle, "Implementation and evaluation of activity-based congestion management using p4 (p4-abc)," *Future Internet*, vol. 11, p. 159, 07 2019.

[28] S. van den Braak, S. Choenni, R. Meijer, and A. Zuiderwijk, "Trusted third parties for secure and privacy-preserving data integration and sharing in the public sector," in *Proc. Conf. on Digital Government Research*. New York, NY, USA: ACM, 2012, pp. 135–144.

[29] L. Gommans, J. Vollbrecht, B. G. de Bruijn, and C. de Laat, "The service provider group framework: A framework for arranging trust and power to facilitate authorization of network services," *Future Generation Computer Systems*, vol. 45, pp. 176–192, 2015.

[30] M. M. Mello, J. K. Francer, M. Wilenzick, P. Teden, B. E. Bierer, and M. Barnes, "Preparing for responsible sharing of clinical trial data," *New England Journal of Medicine*, vol. 369, no. 17, pp. 1651–1658, 2013, pMID: 24144394. [Online]. Available: https://doi.org/10.1056/NEJMhle1309073

[31] R. Cushing, L. Zhang, Y. Demchenko, C. de Laat, and P. Grosso, "Data harbours: Computing archetypes for digital marketplaces," in *Proc. Conf. International Conference on High Performance Computing and Simulation (HPCS 2019)*, 2018.

[32] A. Uszok, J. M. Bradshaw, and R. Jeffers, "Kaos: A policy and domain services framework for grid computing and semantic web services," in *Trust Management*, C. Jensen, S. Poslad, and T. Dimitrakos, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 16–26.

[33] R. Hoekstra, J. Breuker, M. D. Bello, and A. Boer, "The lkif core ontology of basic legal concepts," in *In Legal Ontologies and Artificial Intelligence Techniques*. Ceur.org, 2007.

[34] S. V. Fabien Gandon, Guido Governatori, "Normative requirements as linked data," in *JURIX 2017 - The 30th international conference on Legal Knowledge and Information Systems*, Luxembourg, Luxembourg, 2017, pp. 1–10.

[35] H.-P. Lam, M. Hashmi, and B. Scofield, "Enabling reasoning with legal-ruleml," in *Rule Technologies. Research, Tools, and Applications*, J. J. Alferes, L. Bertossi, G. Governatori, P. Fodor, and D. Roman, Eds. Cham: Springer International Publishing, 2016, pp. 241–257.

[36] D. Ferraiolo, R. Chandramouli, R. Kuhn, and V. Hu, "Extensible access control markup language (xacml) and next generation access control (ngac)," in *Proc. ACM Workshop on Attribute Based Access Control*, ser. ABAC '16. New York, NY, USA: ACM, 2016, pp. 13–24. [Online]. Available: http://doi.acm.org.vu-nl.idm.oclc.org/10.1145/2875491.2875496

[37] X. Maroñas, E. Rodriguez, and J. Delgado, "An architecture for the interoperability between rights expression languages based on xacml," in *Proc. Workshop for technical, economic and legal aspects of business models for virtual goods incorporation the 5th international ODRL workshop*, Sep 2009, pp. 29–47. [Online]. Available: http://www.virtualgoods.org/2009/29_VirtualGoods2009Book.pdf

[38] F. G. Serena Villata, "L4LOD Vocabulary Specification 0.2," https://ns.inria.fr/l4lod/v2/l4lod_v2.html, 2013, [Online; accessed 4-June-2019].

[39] M. Palmirani, M. Martoni, A. Rossi, C. Bartolini, and L. Robaldo, "Pronto: Privacy ontology for legal reasoning," in *Electronic Government and the Information Systems Perspective*, A. Kő and E. Francesconi, Eds. Cham: Springer International Publishing, 2018, pp. 139–152.

[40] M. Li and R. Samavi, "Dsap: Data sharing agreement privacy ontology," in *Proc. Conf. Semantic Web Applications and Tools for Life Sciences (SWAT4HCLS 2018)*. CEUR.org, 2018.

[41] "Calico," https://docs.projectcalico.org/v2.0/introduction/, 2019, [Online; accessed June 2019].

[42] "Cilium," https://docs.cilium.io/en/v1.5/, 2019, [Online; accessed June 2019].

[43] "Weave Net," https://www.weave.works/oss/net/, 2019, [Online; accessed May 2019].

[44] "Flannel," https://github.com/coreos/flannel#flannel, 2019, [Online; accessed June 2019].

[45] D. T. Narten, E. Gray, D. L. Black, L. Fang, L. Kreeger, and M. Napierala, "Problem Statement: Overlays for Network Virtualization," RFC 7364, Oct. 2014. [Online]. Available: https://www.rfc-editor.org/info/rfc7364

[46] M. Lasserre, F. Balus, T. Morin, D. N. N. Bitar, and Y. Rekhter, "Framework for Data Center (DC) Network Virtualization," RFC 7365, Oct. 2014. [Online]. Available: https://www.rfc-editor.org/info/rfc7365

[47] "etcd," https://github.com/coreos/etcd, 2019, [Online; accessed June 2019].

[48] "Networking and Cryptography library," https://nacl.cr.yp.to/, 2022, [Online; accessed July 2022].

[49] "The Berkely Packet Filter," https://www.kernel.org/doc/html/latest/bpf/index.html, 2019, [Online; accessed June 2019].

[50] "Troubleshooting in Weave," https://www.weave.works/docs/net/latest/troubleshooting/, 2022, [Online; accessed July 2022].

[51] "Weave policy profile," https://docs.gitops.weave.works/docs/policy/weave-policy-profile/, 2022, [Online; accessed July 2022].

[52] "Policy audit in Cilium," https://docs.cilium.io/en/v1.8/gettingstarted/policy-creation/, 2022, [Online; accessed July 2022].

[53] "Cilium Troubleshooting," https://docs.cilium.io/en/v1.10/operations/troubleshooting/, 2022, [Online; accessed July 2022].

[54] "Policy monitoring in Calico," https://projectcalico.docs.tigera.io/security/calico-enterprise/compliance, 2022, [Online; accessed July 2022].

[55] "Defend against DoS attacks," https://projectcalico.docs.tigera.io/security/defend-dos-attack, 2022, [Online; accessed July 2022].

[56] "Troubleshooting in Calico," https://projectcalico.docs.tigera.io/maintenance/troubleshoot/troubleshooting, 2022, [Online; accessed July 2022].

[57] "Calico firewall integration," https://www.tigera.io/features/firewall-integration/, 2022, [Online; accessed July 2022].

[58] "Container Security by Adding Zero Trust to Calico Cloud," https://channelvisionmag.com/tigera-tightens-container-security-by-adding-zero-trust-to-calico-cloud/, 2022, [Online; accessed July 2022].

[59] "Tigera Releases Container Security Features on Calico Cloud," https://www.devopsdigest.com/tigera-releases-container-security-features-on-calico-cloud, 2022, [Online; accessed July 2022].

[60] "Layer 7 Protocol Visibility," https://docs.cilium.io/en/latest/observability/visibility/, 2022, [Online; accessed July 2022].

[61] "iPerf," https://iperf.fr/, 2019, [Online; accessed May 2019].

[62] J. Khalid, E. Rozner, W. Felter, C. Xu, K. Rajamani, A. Ferreira, and A. Akella, "Iron: Isolating network-based {CPU} in container environments," in *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, 2018, pp. 313–328.

[63] "ISHARE project," https://www.ishareworks.org/project, 2019, [Online; accessed April 2019].

[64] A. Buzachis, A. Galletta, L. Carnevale, A. Celesti, M. Fazio, and M. Villari, "Towards osmotic computing: Analyzing overlay network solutions to optimize the deployment of container-based microservices in fog, edge and iot environments," in *2018 IEEE 2nd International Conference on Fog and Edge Computing (ICFEC)*, May 2018, pp. 1–10.

[65] "AppArmor," https://wiki.archlinux.org/index.php/AppArmor, 2020, [Online; accessed May-2020].

[66] "SecurityEnhancedLinux," https://en.wikipedia.org/wiki/Security-Enhanced-Linux, 2020, [Online; accessed May 2020].

[67] S. Hosseinzadeh, S. Laurén, and V. Leppänen, "Security in container-based virtualization through vtpm," in *2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC)*, 2016, pp. 214–219.

[68] F. Loukidis-Andreou, I. Giannakopoulos, K. Doka, and N. Koziris, "Dockersec: A fully automated container security enhancement mechanism," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, 2018, pp. 1561–1564.

[69] J. Chelladhurai, P. R. Chelliah, and S. A. Kumar, "Securing docker containers from denial of service (dos) attacks," in *2016 IEEE International Conference on Services Computing (SCC)*, 2016, pp. 856–859.

[70] "Evaluating Container Platforms at Scale," https://medium.com/on-docker/evaluating-container-platforms-at-scale-5e7b44d93f2c, 2020, [Online; accessed May 2020].

[71] "ODRL Information Model 2.2," https://www.w3.org/TR/odrl-model/, 2019, [Online; accessed April 2019].

[72] "Calico Routing Modes," https://octetz.com/docs/2020/2020-10-01-calico-routing-modes/, 2020, [Online; accessed May 2020].

[73] "Docker overlay networks," https://docs.docker.com/network/overlay/, 2020, [Online; accessed May 2020].

[74] "Prevent DNS (and other) spoofing with Calico," https://www.tigera.io/blog/prevent-dns-and-other-spoofing-with-calico/, 2020, [Online; accessed May 2020].

[75] "OVN," https://github.com/ovn-org/ovn-kubernetes, 2019, [Online; accessed June 2019].

[76] S. Shakeri, N. van Noort, and P. Grosso, "Scalability of container overlays for policy enforcement in digital marketplaces," in *2019 IEEE 8th International Conference on Cloud Networking (CloudNet)*, 2019, pp. 1–4.

[77] E. Bacis, S. Mutti, S. Capelli, and S. Paraboschi, "Dockerpolicymodules: Mandatory access control for docker containers," in *2015 IEEE Conference on Communications and Network Security (CNS)*, 2015, pp. 749–750.

[78] A. Martin, S. Raponi, T. Combe, and R. D. Pietro], "Docker ecosystem – vulnerability analysis," *Computer Communications*, vol. 122, pp. 30 – 43, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0140366417300956

[79] "priVAcy preserviNg federaTed leArninG infrastructurE for Secure Insight eXchange," https://distributedlearning.ai/, 2021, [Online; accessed April 2021].

[80] "A library for computing on data you do not own and cannot see," https://github.com/OpenMined/PySyft, 2021, [Online; accessed April 2021].

[81] "MPyC: Secure Multiparty Computation in Python," https://www.win.tue.nl/~berry/mpyc/, 2021, [Online; accessed April 2021].

[82] "IBM Federated Learning," https://github.com/IBM/federated-learning-lib, 2022, [Online; accessed July 2022].

[83] "Default bridge network," https://docs.docker.com/network/network-tutorial-standalone/#use-the-default-bridge-network, 2021, [Online; accessed September 2021].

[84] "User-defined bridge networks," https://docs.docker.com/network/network-tutorial-standalone/#use-user-defined-bridge-networks, 2021, [Online; accessed September 2021].

[85] "Improving Network Monitoring and Management with Programmable Data Planes," https://opennetworking.org/news-and-events/blog/improving-network-monitoring-and-management-with-programmable-data-planes/, 2021, [Online; accessed September 2021].

[86] P. Manzanares-Lopez, J. P. Muñoz-Gea, and J. Malgosa-Sanahuja, "Passive in-band network telemetry systems: The potential of programmable data plane on network-wide telemetry," *IEEE Access*, vol. 9, pp. 20 391–20 409, 2021.

[87] A. C. Lapolli, J. Adilson Marques, and L. P. Gaspary, "Offloading real-time ddos attack detection to programmable data planes," in *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, 2019, pp. 19–27.

[88] A. Febro, H. Xiao, and J. Spring, "Distributed sip ddos defense with p4," in *2019 IEEE Wireless Communications and Networking Conference (WCNC)*, 2019, pp. 1–8.

[89] M. Dimolianis, A. Pavlidis, and V. Maglaris, "A multi-feature ddos detection schema on p4 network hardware," in *2020 23rd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, 2020, pp. 1–6.

[90] "About Agilio SmartNICs," https://www.netronome.com/products/smartnic/overview/, 2021, [Online; accessed September 2021].

[91] "P4SmartNics," https://opennetworking.org/wp-content/uploads/2020/12/p4_d2_2017_nfp_architecture.pdf, 2021, [Online; accessed September 2021].

[92] Y. Qin, W. Quan, F. Song, L. Zhang, G. Liu, M. Liu, and C. Yu, "Flexible encryption for reliable transmission based on the p4 programmable platform," in *2020 Information Communication Technologies Conference (ICTC)*, 2020, pp. 147–152.

[93] F. Hauser, M. Schmidt, M. Häberle, and M. Menth, "P4-macsec: Dynamic topology monitoring and data layer protection with macsec in p4-based sdn," *IEEE Access*, vol. 8, pp. 58 845–58 858, 2020.

[94] "ARP spoofing," https://www.veracode.com/security/arp-spoofing, 2021, [Online; accessed April 2021].

[95] "IP spoofing," https://www.oreilly.com/library/view/ccna-security-210-260/9781787128873/78f2bb48-0c68-452b-8edc-eb1482f7dbfc.xhtml, 2021, [Online; accessed April 2021].

[96] F. Hauser, M. Häberle, M. Schmidt, and M. Menth, "P4-ipsec: Site-to-site and host-to-site vpn with ipsec in p4-based sdn," *IEEE Access*, vol. 8, pp. 139 567–139 586, 2020.

[97] "BEHAVIORAL MODEL (bmv2)," https://github.com/p4lang/behavioral-model, 2021, [Online; accessed April 2021].

[98] "RabbitMQ," https://www.cloudamqp.com/blog/part1-rabbitmq-for-beginners-what-is-rabbitmq.html, 2021, [Online; accessed April 2021].

[99] D. Draskovic and G. Saleh, "Datapace," December 2017. [Online]. Available: https://datapace.io/datapace_whitepaper.pdf

[100] O. P. Foundation and B. GmbH, "Ocean protocol: Tools for the web3 data economy," December 2020. [Online]. Available: https://oceanprotocol.com/tech-whitepaper.pdf

[101] S. Shakeri, N. van Noort, and P. Grosso, "Scalability of container overlays for policy enforcement in digital marketplaces," in *2019 IEEE 8th International Conference on Cloud Networking (CloudNet)*, 2019, pp. 1–4.

[102] S. Shakeri, L. Veen, and P. Grosso, "Evaluation of container overlays for secure data sharing," in *2020 IEEE 45th LCN Symposium on Emerging Topics in Networking (LCN Symposium)*, 2020, pp. 99–108.

[103] X. Zhou, R. Cushing, R. Koning, A. Belloum, P. Grosso, S. Klous, T. van Engers, and C. de Laat, "Policy enforcement for secure and trustworthy data sharing in multi-domain infrastructures," in *2020 IEEE 14th International Conference on Big Data Science and Engineering (BigDataSE)*, 2020, pp. 104–113.

[104] L. E. Veen, S. Shakeri, and P. Grosso, "Mahiru: a federated, policy-driven data processing and exchange system," 2022. [Online]. Available: https://arxiv.org/abs/2210.17155

[105] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM*

*Comput. Commun. Rev.*, vol. 44, no. 3, p. 87–95, Jul. 2014. [Online]. Available: https://doi.org/10.1145/2656877.2656890

[106] L. Veen, S. Shakeri, and P. Grosso, "Secure data sharing and distributed processing with Mahiru," 2022, Poster presented at ICT.Open 2022. [Online]. Available: https://doi.org/10.5281/zenodo.6497704

[107] "Access tracking in P4," https://github.com/sarashakeri/SecConNet-tracking/, 2022, [Online; accessed April 2022].

[108] K. V. Sarma, S. Harmon, T. Sanford, H. R. Roth, Z. Xu, J. Tetreault, D. Xu, M. G. Flores, A. G. Raman, R. Kulkarni, B. J. Wood, P. L. Choyke, A. M. Priester, L. S. Marks, S. S. Raman, D. Enzmann, B. Turkbey, W. Speier, and C. W. Arnold, "Federated learning improves site performance in multicenter deep learning without data sharing," *Journal of the American Medical Informatics Association*, vol. 28, no. 6, pp. 1259–1264, 02 2021. [Online]. Available: https://doi.org/10.1093/jamia/ocaa341

[109] A. Durrant, M. Markovic, D. Matthews, D. May, J. A. Enright, and G. Leontidis, "The role of cross-silo federated learning in facilitating data sharing in the agri-food sector," *CoRR*, vol. abs/2104.07468, 2021. [Online]. Available: https://arxiv.org/abs/2104.07468

[110] S.-Y. Wang, Y.-R. Chen, J.-Y. Li, H.-W. Hu, J.-A. Tsai, and Y.-B. Lin, "A bandwidth-efficient int system for tracking the rules matched by the packets of a flow," in *2019 IEEE Global Communications Conference (GLOBECOM)*, 2019, pp. 1–6.

[111] Z. Zhao, X. Shi, X. Yin, and Z. Wang, "Hashflow for better flow record collection," 2018. [Online]. Available: https://arxiv.org/abs/1812.01846

[112] D. Barradas, N. Santos, L. Rodrigues, S. Signorello, F. M. V. Ramos, and A. Madeira, "Flowlens: Enabling efficient flow classification for ml-based network security applications," in *Proceedings 2021 Network and Distributed System Security Symposium*. Virtual: Internet Society, 2021.

[113] L. Castanheira, R. Parizotto, and A. E. Schaeffer-Filho, "Flowstalker: Comprehensive traffic flow monitoring on the data plane using p4," in *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*, 2019, pp. 1–6.

[114] L. Zhang, R. Cushing, R. Koning, C. de Laat, and P. Grosso, "Profiling and discriminating of containerized ml applications in digital data marketplaces (ddm)." in *ICISSP*, 2021, pp. 508–515.

[115] L. Zhang, R. Cushing, C. d. Laat, and P. Grosso, "A real-time intrusion detection system based on oc-svm for containerized applications," in *2021 IEEE 24th International Conference on Computational Science and Engineering (CSE)*, 2021, pp. 138–145.

[116] R. Cushing, R. Koning, L. Zhang, C. d. Laat, and P. Grosso, "Auditable secure network overlays for multi-domain distributed applications," in *2020 IFIP Networking Conference (Networking)*, 2020, pp. 658–660.

# Publications

- **Shakeri, S.**, Maccatrozzo, V., Veen, L., Bakhshi, R., Gommans, L., de Laat, C., and Grosso, P. "Modeling and Matching Digital Data Marketplace Policies". In 2019 15th International Conference on eScience (eScience), pp. 570–577.
- **Shakeri, S.**, van Noort, N., and Grosso, P. "Scalability of Container Overlays for Policy Enforcement in Digital Marketplaces". In 2019 IEEE 8th International Conference on Cloud Networking (CloudNet), pp. 1–4.
- **Shakeri, S.**, Veen, L., and Grosso, P. "Evaluation of Container Overlays for Secure Data Sharing". In: 2020 IEEE 45th LCN Symposium on Emerging Topics in Networking (LCN Symposium), pp. 99–108.
- **Shakeri, S.**, Veen, L. and Grosso, P. "Multi-domain network infrastructure based on P4 programmable devices for Digital Data Marketplaces". Cluster Computing (2022). https://doi.org/10.1007/s10586-021-03501-2.
- **Shakeri, S.**, Veen, L., and Grosso, P. "Tracking container network connections in a Digital Data Marketplace with P4". In 2022 International Conference on Computer, Information and Telecommunication Systems (CITS), Piraeus, Greece, 2022, pp. 1-8.
- Veen, L., **Shakeri, S.**, and Grosso, P. "Mahiru: a federated, policy-driven data processing and exchange system". Submitted to arXiv:2210.17155.

# Source Code

The source code for SecConNet project is published at https://github.com/SecConNet. Here we highlight the main components:

- Applicability of container overlays for data sharing
  https://github.com/SecConNet/Container-overlays-applicability
- Evaluation of Container Overlays for Secure Data Sharing
  https://github.com/SecConNet/Container-overlays-architecture
- Multi-domain Network Infrastructure based on P4 Programmable Devices for DDMs
  https://github.com/SecConNet/P4-based-containerized-DDM
- Tracking container network connections in a DDM
  https://github.com/SecConNet/Tracking-connections-in-a-DDM
- Mahiru Data Exchange
  https://github.com/SecConNet/mahiru

# Acknowledgement

First and foremost, I would like to show my gratitude to my partner, Morteza, for believing in me. Your support and patience throughout the ups and downs of the PhD process have been invaluable.

I extend my thanks to my supervisor, Paola, for always making time for me, teaching me how to look into the big picture, and directing me to the right approach. I also had so much fun when we had small chats over different topics such as our trips and travel plans. I had the privilege to learn many things from Cees and I am grateful for the invaluable guidance and input he provided during my research and writing journey. I also want to thank Leon for his valuable input about digital data marketplaces and the data sharing concepts.

I would like to thank Fernando, Rob, Zoltan, Chrysa, and Andy for their time and effort in reading my work and participating in my graduation committee.

To perform this research I had the opportunity to work with colleagues from the eScience Center. Lourens with whom I closely collaborated, taught me valuable lessons. I appreciate his input, contribution, and critical point of view to my work. I extend my thanks to those we work together during eScience meetings, namely Rena and Valentina.

Special thanks to my colleagues at the MNS lab, who made my Ph.D. truly memorable. Thanks, Lu, Ruyue, Jamilla, and Zeshun, I had many enjoyable moments with you. During lunch, coffee breaks, and group events with colleagues such as Joseph, Ralph, Misha, Ana, Dolly, Giovanni, Giulio, and many others I had a great time. I enjoyed very much spending my time with them.

I want to thank my long-time friends who provided moments of fun and relaxation, including Zeynab, Zahra, Mohammad, Arezoo, Mojdeh, Atie, Ali, and others. They were a great support during the Covid time and I appreciate it.

During my Ph.D. my family has been there for me in the ups and downs. I want to express my gratitude to my family, including Afagh, Abbas, Saeideh, Somayeh, Mohammad, and others.

I have tried to capture everyone who was with me during this journey. If

a name is not mentioned here, please forgive me, as the list is long, but your support has been appreciated.

# Summary

There are many organizations interested in sharing data with others. However, they can do this only if a secure platform is available. Digital Data Marketplaces (DDMs) are emerging as a framework for organizations to share their data. To increase trust among participating organizations, multiple agreements should be established to determine who has access to what. These sharing policies have to be described in a general model to be applicable in different DDMs. More importantly, translating these high-level sharing policies to actionable code and setting up an infrastructure that implements and enforces the policies is still a challenge.

In SecConNet, we use containers for building the sharing infrastructure. A container can operate as a secure, isolated, and individual entity that on behalf of its owner, manages and processes the data it is given. For exchanging data among multiple organizations, the containers need to be connected. Overlay networks connect containers and make a virtual network upon the physical network. The method of running the overlays plays a critical role in building a secure DDM.

In this thesis, the focus is particularly on the novel container overlay architectures, which utilize programmable infrastructures and virtualization technologies across multiple administrative domains whilst maintaining security and quality requirements.

The first part of the thesis presents a semantic model for describing the access policies by means of semantic web technologies. In particular, we used and extended the Open Digital Rights Language (ODRL) to describe the pre-established agreements in a DDM.

In the next part of the thesis, we evaluated the functionality of available technologies of container overlay networks in making the connection between containers in DDM and implementing the high level policies. We assessed the capability of Cilium and Calico, as they have the best support for enforcing network policies in providing security (policy scalability) and handling the multi-tenancy requirements (pod scalability) of DDMs. Both Calico and Cilium scaled

well in policy scalability, and in terms of pod scalability, Calico performed better.

In the next chapter of the thesis, we aimed our focus on providing the isolation between sharing requests based on the application needs. We defined three container connectivity types: Overlay per DDM, Overlay per request, and Overlay per group. Using the available technologies, we implemented the overlay setups and compared their performance and security.

DDM can operate in both single-domain and multi-domain environments. In the works mentioned above, we considered a DDM as a single-domain environment in which the overlay orchestrator acts on behalf of all participating parties, and all the resources and their connections are handled by a centralized controller. In a multi-domain DDM, each participating party (domain) can manage its own connectivity while all of the transactions follow the sharing agreements.

To build a multi-domain containerized DDM, we introduced a P4-based platform in which the connections of containers are controlled by the rules set in the P4 switch and handled by the domain administrator. The proposed method can handle the communication of multiple domains and guarantee that the operation of transactions is based on predefined policies. We also studied the network setup performance by defining a model which we demonstrated follows the real measurements, and we can use it for decision making.

On top of P4-based containerized networks, we built a distributed data exchange management system that handles the collaboration between data owners automatically based on the agreed policies. It checks if a sharing request from a specific organization is based on the DDM policies and then sends the required information to the administrator to connect containers for data sharing.

In the last part of the thesis, we extended the introduced P4-based platform in order to utilize the capabilities P4 programming language. Considering distributed machine learning applications as a use case, we measured and collected information like the number of access to different containers from multiple domains, the amount of data that was exchanged, and the time of data transferred between containers. In some cases, we reprogrammed the switch based on the situation of the network for the next references.

# Samenvatting

Er zijn veel organisaties die geïnteresseerd zijn in het delen van gegeven met anderen. Dit kunnen ze echter alleen doen als er een veilig platform beschikbaar is. Digitale data marktplaatsen (DDMs) ontstaan als een raamwerk voor organisaties om hun gegeven te delen. Om het vertrouwen tussen deelnemende organisaties te vergroten, moeten er meerdere overeenkomsten worden gesloten om te bepalen wie toegang heeft tot wat. Deze deelovereenkomsten moeten worden beschreven in een algemeen model om toepasbaar te zijn in verschillende DDM's. Belangrijker nog, het vertalen van deze hoog-niveau deelovereenkomsten naar uitvoerbare code en het opzetten van een infrastructuur die de overeenkomsten implementeert en handhaaft, blijft een uitdaging.

In SecConNet gebruiken we containers voor het opbouwen van de deel infrastructuur. Een container kan optreden als een veilige, geïsoleerde en individuele entiteit die namens zijn eigenaar de gegeven beheert en verwerkt die aan hem zijn gegeven. Voor het uitwisselen van gegeven tussen meerdere organisaties moeten de containers met elkaar worden verbonden. Overlay-netwerken verbinden containers en creëren een virtueel netwerk bovenop het fysieke netwerk. De methode om de overlays uit te voeren, speelt een cruciale rol bij het opbouwen van een veilige DDM.

In deze scriptie ligt de focus met name op de nieuwe container-overlay architecturen, die gebruikmaken van programmeerbare infrastructuren en virtualisatie technologieën over meerdere administratieve domeinen, met behoud van beveiligings- en kwaliteitseisen.

Het eerste deel van de scriptie presenteert een semantisch model voor het beschrijven van de toegang beleidsregels met behulp van semantische webtechnologieën. In het bijzonder hebben we de Open Digital Rights Language (ODRL) gebruikt en uitgebreid om de vooraf vastgestelde overeenkomsten in een DDM te beschrijven.

In het volgende deel van de scriptie hebben we de functionaliteit geevalueerd van beschikbare technologieen van container-overlay netwerken om de verbinding

tussen containers in DDM te maken en de hoog-niveau beleidsregels te implementeren. We hebben de capaciteit van Cilium en Calico beoordeeld, omdat ze de beste ondersteuning bieden voor het afdwingen van netwerkbeleid met betrekking tot beveiliging (beleid- schaalbaarheid) en het omgaan met de vereisten voor multi-tenancy (pod-schaalbaarheid) van DDM's. Zowel Calico als Cilium presteerden goed op het gebied van beleid- schaalbaarheid, en wat betreft pod-schaalbaarheid presteerde Calico beter.

In het volgende hoofdstuk van de scriptie richtten we ons op het bieden van isolatie tussen deelverzoeken op basis van de behoeften van de toepassing. We hebben drie soorten container connectiviteit gedefinieerd: Overlay per DDM, Overlay per verzoek en Overlay per groep. Met behulp van de beschikbare technologieën hebben we de overlay-configuraties geimplementeerd en hun prestaties en beveiliging vergeleken.

DDM kan zowel in single-domain als multi-domain omgevingen opereren. In de hierboven genoemde werken beschouwden we een DDM als een single-domain omgeving waarin de overlay-orchestrator namens alle deelnemende partijen optreedt, en alle middelen en hun verbindingen worden beheerd door een gecentraliseerde controller. In een multi-domain DDM kan elke deelnemende partij (domein) zijn eigen connectiviteit beheren, terwijl alle transacties de deelovereenkomsten volgen.

Om een multi-domain containerized DDM op te bouwen, introduceerden we een op P4 gebaseerd platform waarin de verbindingen van containers worden gecontroleerd door de regels die zijn ingesteld in de P4-switch en worden behandeld door de domeinbeheerder. De voorgestelde methode kan de communicatie van meerdere domeinen afhandelen en garanderen dat de werking van transacties gebaseerd is op vooraf gedefinieerde beleidsregels. We hebben ook de prestaties van de netwerk configuratie bestudeerd door een model te definiëren dat we hebben gedemonstreerd aan de hand van echte metingen, en dat we kunnen gebruiken voor besluitvorming.

Bovenop P4-gebaseerde containerized netwerken hebben we een gedistribueerd gegeven uitwisseling beheersysteem gebouwd dat de samen werking tussen gegeven eigenaren automatisch afhandelt op basis van de overeengekomen beleidsregels. Het controleert of een verzoek tot delen van een specifieke organisatie wordt gebaseerd op het DDM-beleid en verzendt vervolgens de vereiste informatie aan de beheerder om containers aan te sluiten voor het delen van gegevens.

In het laatste deel van het proefschrift hebben we het geïntroduceerde op P4 gebaseerde platform uitgebreid om de mogelijkheden van de programmeertaal P4 tebenutten. Door gedistribueerde machine learning-toepassingen als gebruiksscenario te beschouwen, hebben we informatie gemeten en verzameld, zoals het aantal toegangen tot verschillende containers vanuit meerdere domeinen, de hoeveelheid gegevens die werd uitgewisseld en het tijdstip waarop de gegevens werden overgedragen tussen containers.