# Choreography-Defined Networks: a Case Study on DoS Mitigation

Saverio Giallorenzo[1,2], Jacopo Mauro[3],

Andrea Melis[1], Fabrizio Montesi[3], Marco Peressotti[3], and Marco Prandini[1]

[1] Università di Bologna, Italy {saverio.giallorenzo2,a.melis,marco.prandini}@unibo.it
[2] INRIA, France
[3] University of Southern Denmark, Denmark {mauro,fmontesi,peressotti}@imada.sdu.dk

**Abstract.** Software-defined networking and network function virtualization have brought unparalleled flexibility in defining and managing network architectures. With the widespread diffusion of cloud platforms, more resources are available to execute virtual network functions concurrently, but the current approach to defining networks in the cloud development is held back by the lack of tools to manage the composition of more complex flows than simple sequential invocations.

In this paper, we advocate for the usage of choreographic programming for defining the multiparty workflows of a network. When applied to the composition of virtual network functions, this approach yields multiple advantages: a single program expresses the behavior of all components, in a way that is easier to understand and check; a compiler can produce the executable code for each component, guaranteeing correctness properties of their interactions such as deadlock freedom; and the bottleneck of a central orchestrator is removed. We describe the proposed approach and show its feasibility via a case study where different functions cooperatively solve a security monitoring task.

**Keywords:** Software-defined Networks, Virtual Network Functions, Choreographic Programming, Network Security, Denial-of-Service

## 1 Introduction

Software-Defined Networks (SDNs) [1] and Network Function Virtualization (NFV) [2] have revolutionised network architectures: SDNs enable the straightforward, dynamic management and configuration of network resources through a programmable software layer, while NFV replaces dedicated hardware appliances with software (Virtual Network Functions – VNFs) that runs on commodity hardware, promoting flexibility and scalability. Traditionally, VNFs are programmed in a way that recalls orchestrated sequential compositions from service-oriented computing. We attribute the choice of this programming approach to resource constraints, which make running multiple VNFs together unfeasible, even if more than half of real-world enterprise network functions could logically work in parallel [3], and to the complexity of distributed composition, which is a renowned problem of concurrent/distributed programming that can lead to inconsistent behavior and incorrect results [2].

Nowadays, VNFs could be replaced by Cloud-native Network Functions (CNFs), a specialisation of VNFs designed to run in cloud environments, leveraging containerization and microservice technologies. CNFs are modular, scalable, and dynamically deployable, thus overcoming the constraints on resources that hindered the parallel execution of monolithic VFNs. Yet, CNFs still struggle to become standard practice mainly because their execution and coordination essentially constitute distributed software based on message passing, whose correct implementation is notoriously challenging even for experts [4]. It is easy, for example, to write communication actions in different programs that fail at interacting because of wrong timing or mismatches in expected payload types. Avoiding these bugs with code analysis tools is often impractical because of the state explosion problem of concurrent software [5]. Furthermore, editing the code of one VNF might break compatibility with other VNFs, so their deployment needs to be coordinated carefully. The full potential of SDNs and VNFs deployment and management via CNFs remains therefore untapped due to the challenge of writing and managing distributed software.

We address the problem of *correctly implementing distributed CNF architectures for SDN systems* by connecting the fields of SDNs and programming languages. Specifically, we take a step towards taming the complexity of developing correct CNF architectures with a development process for SDNs based on Choreographic Programming [6] (CP), a recent programming paradigm for concurrent and distributed software. CP allows developers to write a coordination plan (a 'choreography') for a set of distributed roles (abstractions of communicating processes), which is then automatically translated by a compiler into an executable program for each role. This approach greatly reduces code complexity, because the planned communications become syntactically manifest and can be expressed succinctly. Furthermore, the compilation of choreographies is backed by well-understood mathematical theories that focus on the correct matching of message send and receive actions in the generated programs. As a result, CP can guarantee important safety and liveness properties like the absence of communication mismatches (messages have the expected type, are sent on the right channels, etc.) and deadlock-freedom [7].

To reify the approach we propose, we use the most advanced choreographic language to date: Choral [8]. Practically, Choral extends the Java language with locality information about data. In Choral, `T@A` denotes data of type `T` located at a participant, also called *role*, `A`. Given a collection of located data, we can move any of these values from a role to another with methods that take data at a role and return it at second one. Following this abstraction, we propose to model a CNF as a role and have multiple CNFs participate in a choreography to implement a desired distributed behaviour. Our main contribution is a prototype software development method for SDNs based on Choral, called *Choreography-Defined Network* (CDN), which we schematically represent as follows.

$$\begin{array}{c}\text{Choral}\\ \text{Choreography}\end{array} \xRightarrow{\substack{\text{Choral}\\ \text{Compiler}}} \begin{array}{c}\text{Java code}\\ \text{for each VNF}\\ +\text{ local code}\end{array} \xRightarrow{\substack{\text{Java Compiler}\\ \text{\& Containerisation}}} \begin{array}{c}\text{CNF}\\ \text{executables}\end{array}$$

In CDN, developers write a choreography that collectively defines the overall behaviour of multiple CNFs. Then, leveraging Choral's compiler, they obtain the implementation of each VNF for the target SDN as a Java program that they can link to local code (which can implement some private logic, e.g., traffic filtering), compile it to Java executables and containerise it to obtain a deployable CNF.

To the best of our knowledge, this is the first work that proposes such a connection, of which we provide a concrete instantiation, through the usage of Choral and the implementation of a timely, representative case study on network attack management and mitigation of Distributed Volumetric Attacks. We show that, by using a choreographic approach, it is possible to program networks going beyond the simple chaining of functions, allowing for more complex parallel patterns. Besides managing complexity, the choreographic approach provides a by-construction guarantee that removes problems such as deadlocks and race conditions.

In this paper, we provides the necessary background knowledge, and compares our approach with related work, in Section 2. Then, in Section 3, we present a case study focused on mitigating Distributed Denial of Service Attacks (DDoS) that we use to showcase the practical application of CDN. Section 4 describes the implementation of the use case illustrating both the ergonomics of the approach and how it naturally lends itself to translating workflow-like schemas into code artefacts that generate the implementation of the system. In Section 5, we discuss the advantages and challenges of the choreographic approach. In Section 6, we draw closing remarks and discuss future work.

## 2 Background and Related Work

### 2.1 Modern Networking

Modern architectures are the product of two (r)evolutionary waves of innovation. The first wave saw the advent of layering and "softwarisation" of network functions. It began with the separation of duties between the control plane (managing sessions and signalling) and user plane (handling data traffic)—as, e.g., adopted in the Software-Defined Networking (SDN) model, which places the burden of network programming fully on a controller that gives detailed forwarding instructions to devices via a dedicated protocol [9]—and proceeded with the introduction of Virtual Network Functions (VNFs), i.e., software-based network components such as routers and security gateways.

Network Function Virtualization (NFV), the process of replacing specialized devices with VNFs that can be deployed, e.g., on a virtual machine or a container, nicely integrates into the SDN paradigm [10]. Separating VNFs from their underlying hardware introduces various management challenges, such as mapping services to NFV networks, placing VNFs correctly to fulfil service objectives, and dynamically allocating and scaling hardware resources. It also involves monitoring the location of VNF instances and managing fault detection and recovery across the network.

To support the development of NFV components, the Linux Foundation, in cooperation with ETSI, launched an open-source reference platform called the Open Platform for Network Function Virtualization (OPNFV)[4] in 2014, and later expanded to include the Management and Orchestration (MANO) section.

[4] https://www.opnfv.org/

The second wave saw some intelligence put back on a *programmable* data plane (PDP) by leveraging devices that can execute code, e.g., P4-enabled switches [11, 12], to re-enable line-rate traffic analysis. To avoid losing the advantages of the SDN/NFV approach, these devices should be integrated in the management paradigm. To this end, various approaches have been proposed for runtime interaction with P4 devices [13], with Real Time Pipeline Reconfiguration being the latest frontier of network programmability [14]. Clearly, choosing *how* the data plane devices should behave in different conditions is a decision that needs to be orchestrated together with all the higher-level network management decisions.

### 2.2 Choreographic Programming and Choral

Choreographic Programming [7, 15–18] sinks its roots in service-oriented programming. Service-orientation distinguishes between two ways of implementing the logic of services that belong to a distributed system: *orchestration* and *choreography*.

In orchestration, one service, called the *orchestrator*, coordinates the actions of the other services involved in an architecture. The orchestrator encapsulates and executes the distributed system's logic, managing all interactions among the participating services. While orchestration simplifies implementation and verification against a reference specification, it has several drawbacks. The orchestrator acts as a single control point, thus it can become a twofold bottleneck: its computational resources may reduce the efficiency at which it dictates operations to other services, and it may add latency in scenarios with network limitations, since it must mediate all data exchanges. Furthermore, it is a potential single point of failure and a highly valuable target for cyberattacks, putting system resilience at risk.

As an alternative to orchestration, choreographies distribute the logic of the distributed system among the participants in the architecture. Like a choreographed performance, each service in a choreography plays a specific role and performs the corresponding actions, implementing its part in the architecture's overall interaction scheme. In this paper, we follow an interpretation of choreographies called *choreographic programming*, whereby developers specify the actions and interactions of all the involved services as a choreographic program. Then, given a source choreography, the developers use a compiler to automatically generate the correct code of all the services that participate therein.

CP differentiates itself from neighbouring approaches, such as using choreographies as specifications or as types [19], by the fact its artefacts are written in a fairly concrete language. For instance, a choreographic language usually allows programmers to specify the distribution of values among the participants, message exchanges, and distributed branching behaviours. The hallmark characteristic of choreographic programming is that programmers cannot express deadlocks on messages—thanks to the fact that interactions syntactically pair the sending and reception of messages. Then, compilers that support behaviour-preserving properties can generate the code of the participants from a given choreography, guaranteeing that their combined, distributed execution faithfully follows the semantics of the source, including the absence of message deadlocks.

Concretely, we focus on the usage of Choral [8], the first language that marries choreographic programming with object orientation. In Choral, `T@A` denotes data of type `T` at the role `A`, which one can move by applying methods that take data at a role and return it at another one. Objects that provide these methods are typically called channels. For example, the following two lines of Choral code produce some data at a role `A` and then use a channel to copy the data to another role `B`.

```Choral
PacketFeature@A x = analyser.extractFeatures();
PacketFeature@B y = channel.<PacketFeature>com(x);
```

Note how, in the second line, the implicit send action at `A` and receive action at `B` are safely abstracted away by the atomic invocation of method `com`. Using the Choral compiler, we obtain the Java code for `A` and `B`, shown below, where we find the above actions implemented by the respective participants: `A` generates the data (in `x`) and sends it via `channel`, which `B` uses to receive it (in `y`).

Given a Choral program, we can compile it into pure Java libraries, each implementing the behaviour of one of the participants. As an example, we report below the Java code for `A` (left) and `B` (right) compiled from Choral. Using their respective Java code, programmers can modularly compose the choreographic behaviour of a role with local libraries and correctly participate in their distributed architecture.

```Java
// Implementation of A
PacketFeature x = analyser.extractFeatures();
channel.<PacketFeature>com(x);
```

```Java
// Implementation of B
PacketFeature y = channel.<PacketFeature>com();
```

## 2.3 Related Work

Since the first release of the Open-Source MANO framework by the Etsi Foundation in 2016, the focus of most of the related research has been on enhancing the adaptability, efficiency, and security of VNF deployments in increasingly complex network environments. Contrarily, only a few works consider the communication logic between NFVs, mainly looking at standardising the identification and representation of an NFV through descriptors.

Nguyen et al. [20] introduced an AI-driven approach to VNF chain orchestration, which optimises resource allocation through predictive analysis of network demands and conditions. He et al. [21] expanded on the integration of VNFs with edge computing, proposing a decentralised orchestration model that enables more efficient data processing and reduces the strain on core network resources. This model leverages edge nodes to perform local data processing before transferring information to centralised servers, thereby enhancing the responsiveness of network services. He et al.'s approach improves the ability to automatise the process of NFV deployment via resource allocation analysis. However, there is no reference to the possibility of automating the generation of the VNFs themselves through a more structured, high-level language, which is instead one of the main advantages of our solution.

To the best of our knowledge, the only approach for VNF definition that can be considered at a similar level of abstraction as ours is Intent-Based Networking (IBN). IBN is a concept that aims to apply automation intelligence to devise network configuration plans, replacing the manual processes of initial set up and reaction to issues. Similar to the choreographic approach, IBN can abstract and define the behavior of the network functions at a higher level; yet, in its current state, it would require a different hardware technology, making its implementation not promptly feasible [22].

Focussing on security, Hasneen and Sadique [23] surveyed the security challenges 5G must face when implementing its slicing capabilities with SDN and VNF technologies. In particular, Lakshmanan et al. [24] and Sun et al. [3] provide deployment solutions that can prevent a chain misconfiguration or vulnerability by design, but they consider the simplified scenario in which functions are not invoked in parallel.

Considering multidomain VNF deployment, Huff et al. [25] address the challenge of the management of the reliability of the network deployment in different domains (e.g., cloud providers, on-premises servers, etc.) with an architecture that can connect to the different chains in the cloud through tunnels (VPN or VXLAN) and guarantee a certain level of reliability. With the choreographic approach, the reliability level can be natively introduced in the choreographic logic in a way that is both terse and reusable.

Regarding the case study we chose for the validation of our approach (cf. Sections 3 and 4), the closest related work is SDNShield [26], a network solution based on NFV technologies that enforces comprehensive defence against potential DDoS attacks on SDN control plane. The authors implemented their scheme by deploying VNFs, but differently from us, they rely on a centralised SDN controller that has to manage the flow on the chain; the controller is a unique point of failure that reduces the reliability of the network and possibly constitutes a bottleneck. In addition, the logic is hardcoded inside VNFs, which makes it not portable to local scenarios and difficult to adapt to other attacks. Our choreographic approach allows more flexibility and adaptability to the scenario, yet introducing better resilience by eliminating by construction problems like deadlocks and races.

## 3 A Case Study on DoS mitigation

To showcase the usage of choreographic programming for the development of an SDN, we consider the case of using traffic analysis to detect volumetric network attacks. In particular, we aim to detect attacks using anomaly detection techniques, such as flow asymmetry [27], characterised by the possibility of generating many false positives depending on the anomaly threshold that is set or the efficiency of the detection engine [28]. In these scenarios, an effective strategy is to combine multiple detection engines, with different sensitivities and thresholds [29] to have a deeper and more certain result rather than relying on a single oracle.

We consider a network topology in which traffic flows through a switch, programmed to mirror it towards virtual network functions deployed on the edge, to avoid sending huge amounts toward, e.g., a cloud computing centre. As visualised in Figure 1, the network relies on the following four VNFs:

- `Split&Agg (SA)`. This function sends the traffic to the VNFs in charge of analysing it, possibly selecting which ones to involve, and then deciding which packets should be forwarded depending on the received responses.
- `Volumetric Anomaly Traffic Inspection (VOL)`. This function is configured with a set of Anomaly detection rules[5] that try to identify the maliciousness of a

[5] https://www.ibm.com/docs/en/qradar-on-cloud?topic=rules-anomaly-detection

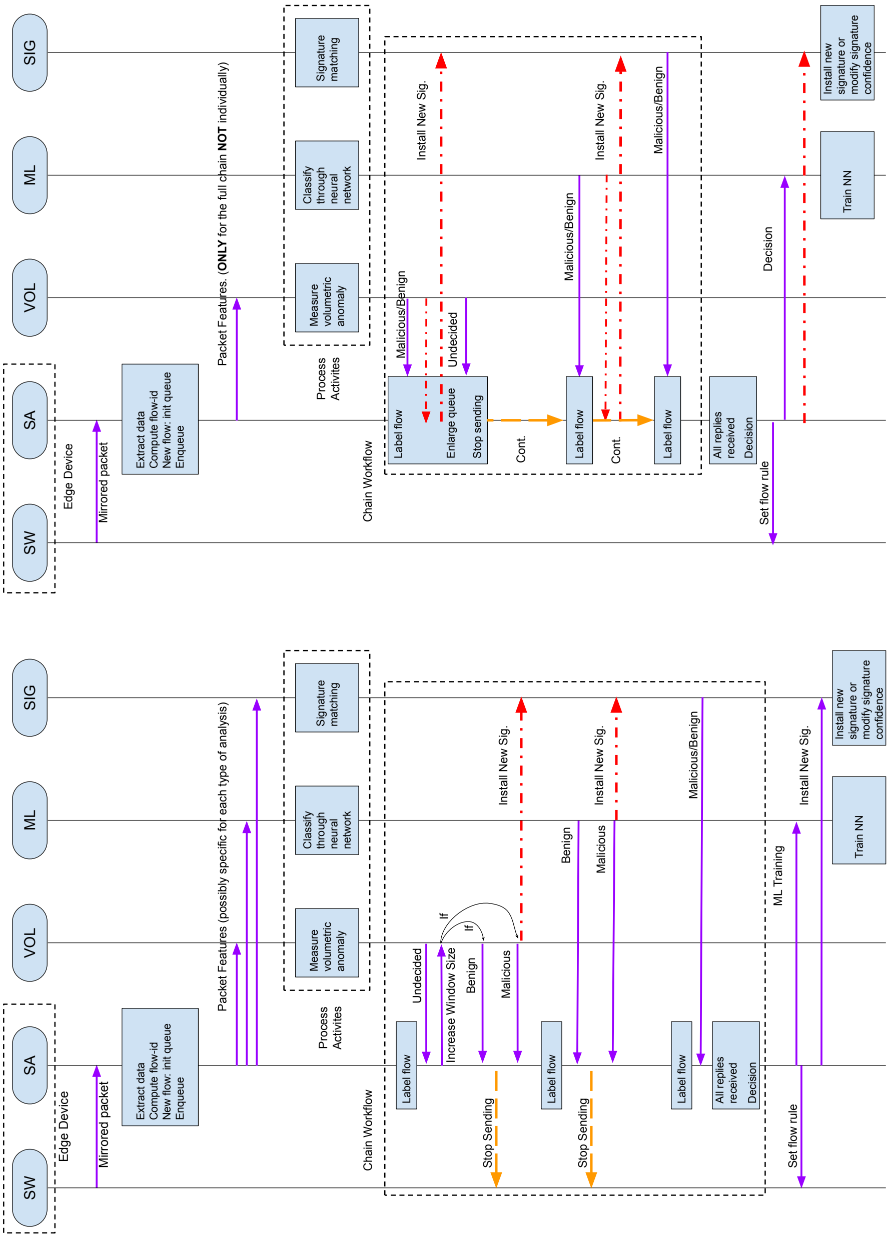

**Fig. 1.** A comparison of the case study workflow using the choreographic approach (left side) vs the classic SDN orchestrated one (right side). The red dotted arrows represent a new attack signature generated by each VNF, and the yellow ones are the workflow possibilities (when the VNF can stop the detection process) (cf. Section 4).

specific flow. The output is a "Benign/Malicious" answer to tell whether the flow must be deeper analysed or it is a legit flow.

- `ML Detection Engine (ML)`. This function uses ML techniques (e.g., a neural network [30]) to inspect and detect if a flow is malicious or not, with a known level of reliability. When a malicious flow is detected it reports the finding to the first VNF.
- `Signature Attack Detection (SIG)`. Every attack (e.g., DoS, Spoofing) can be represented with a signature (e.g., a hash of the payload). A commonly done by antivirus software, this function checks the flows against a database of signatures to find indications of known attacks. This detection mechanism is the fastest and less prone to false positives, but it cannot detect attacks that have not been previously classified.

As presented in Figure 1, the traffic is first mirrored by the SW to SA that filters the packets to forward to all the other three VNF. The VOL, ML, and SIG functions receive the filtered flow from SA and perform their analysis in parallel independently. When they reach a decision and classify the flow either as malign or benign, they independently inform the SA about the decision. Notice that the interaction could also be more complex since VOL may not be able to make a decision due to insufficient data and, as a consequence, SA can instruct to increase the amount of traffic to collect (Increase Window Size arrow).

To further illustrate the advantage of the choreographed approach w.r.t. the orchestrated one, on the right side of Figure 1 we represented the same workflow but implemented with a classic SDN NVF chain with a centralised controller. The red dotted arrows on the left side show two more inter-VNF communications that can happen without the mediation of the controller, which in these cases would be useless in principle since the interactions do not belong to the process of attack detection. These messages are used to update SIG when a new attack is confidently detected, and its signature can be added to the database. These actions improve the system without overloading the controller. The same interactions are represented with the same red dotted arrows on the right side, where is it possible to note the difference; they must always go back to the centralized controller flow, and they cannot act independently. The yellow dashed arrow otherwise indicates the direction of the chain flow. As can be seen in the choreographic approach (left side) any NFV can independently interrupt the analysis, if appropriate (e.g., high-confidence attack detection), while the classical approach (right side) each time needs to pass the whole chain before producing a result.

Finally, also the reaction to the attack can bypass the controller in the interest of timeliness (leftmost left-pointing arrows). SA can use P4Runtime[6] to instruct the switch to stop monitoring benign flows, or to implement a mitigation action (e.g., packet filtering) against a malicious flow.

## 4 Implementation

We now report salient remarks on the implementation of the case study from Section 3: its Choral implementation and its deployment as a system of VNFs.

[6] `https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html`

## 4.1 Choral Implementation

We illustrate the experience of programming the scenario from Section 3 using Choral by focusing on the multiparty interaction between the volumetric anomaly traffic inspection function `VOL`, the ML detection engine `MLE`, and the signature attack detection function `SIG` for updating the attack signature (the red, dot-dashed arrows within the chain workflow in Figure 1). The interested reader can find the full code that implements the case study at `https://anonymous.4open.science/r/chorSDN-676D`.

Recalling the relevant exchanges in Figure 1, ML and VOL send to SIG their analysed data signatures, which then SIG processes to label the flow. A possible Choral implementation of said exchange is the following.

```
Optional@SIG<DataSignature> s1 = ch_ml_sig.<>com( ml_analyser.genSignature() );
Optional@SIG<DataSignature> s2 = ch_vol_sig.<>com( vol_analyser.genSignature() );
sig_analyser.labelFlow( s1, s2 );
```

*Choral*

In the first line, on the right of the assignment, we write that ML sends to SIG the result of the analysis of the data it previously processed, found in the object (located at ML) `ml_analyser` and obtained through the invocation of the method `genSignature`. The communication happens by passing to the method `com` of the object `ch_ml_sig` the result of `genSignature`. Like `channel` in Section 2.2, `ch_ml_sig` is a (symmetric) channel shared between ML and SIG, which transmits the data returned by `genSignature` —an `Optional` that can contain the `DataSignature` of the attack, if any—to `SIG`. At the left of the assignment, we find the variable `s1`, local to `SIG`, where it stores the data sent from ML. Similarly, in the second line, we find that VOL sends a possible attack signature to `SIG`, which stores said data in `s2`. At the third line, SIG invokes the method `labelFlow` of its analyzer (`sig_analyser`) to update its set of attack signatures.

The Choral code above is compiled into separate Java implementations for VOL, ML, and SIG, as shown below.

```
// Implementation for SIG
Optional<DataSignature> s1
        = ch_ml_sig.<>com();
Optional<DataSignature> s2
        = ch_vol_sig.<>com();
sig_analyser.labelFlow( s1, s2 );
```

```
// Implementation for ML
ch_ml_sig.<>com( ml_analyser.genSignature() );
```

```
// Implementation for VOL
ch_vol_sig.<>com( vol_analyser.genSignature() );
```

We conclude our example by contrasting the distributed implementation above with the one below, which implements the same logic in the traditional orchestrated way, where SA is the orchestrator. The main takeaway is that the orchestrator needs to mediate the interactions between VOL, ML, and SIG, both imposing an unnecessary bottleneck and increasing the total number of communications (wasting time and bandwidth and exposing the system to increased risk of communication failures).

```
// orchestration at SA
Optional@SA<DataSignature>  t1 = ch_ml_sa.<>com( ml_analyser.genSignature() );
Optional@SA<DataSignature>  t2 = ch_vol_sa.<>com( vol_analyser.genSignature() );
Optional@SIG<DataSignature> s1 = ch_sa_sig.<>com( t1 );
Optional@SIG<DataSignature> s2 = ch_sa_sig.<>com( t2 );
sig_analyser.labelFlow( s1, s2 );
```

*Choral*

### 4.2 Deployment of the Network

For the creation of the SDN, as practised in cloud network development, each VNF was instantiated within a container. The infrastructure was created according to custom docker Linux-like family images, connected via a local docker network capable of handling up to 14MB/s bandwidth. All code was executed on a PC with Ubuntu 22.04 with 16GB RAM and an i7 core processor. For the creation and management of the infrastructure, we used the Kathara tool[7], i.e., an open-source container-based network emulation system for testing production networks in a sandbox environment. We therefore created an architecture composed of 5 containers, one for each of the 4 VNFs and the switch. The virtual switch is developed in the P4 language and contains flow rules to monitor anomalies. The P4 switch was emulated using the v1 model architecture for P4 and its virtualized version BMv2.[8] The VNF were instead deployed creating a docker image for each of the VNFs and installing the specific tools and settings for each one.

For implementing `VOL` we used a modified version of a symmetric Count-min Sketch[31] designed by observing the behavior of volumetric DDoS attacks. This unexpected value is represented by the traffic volume between the compromised client and the victim which, the more restricted the flow surface is, the more is expected to be much larger than the traffic volume in the opposite direction.

`ML` uses a standard Random Forest Classifier implemented using the scikit-learn python library,[9] with the ability to read a process real-time traffic with the scapy library.[10] For the training set we used a custom dataset composed of 10% of benign traffic (taken from the CIC-IDS2017 dataset[32]) and 90% of DDoS traffic (generated with the hping3[11] Linux utility).

`SIG` has instead been implemented with a light version of Suricata[12] threat detection software and a database of default rules taken from the nuclei-discover repository [33].
A new database entry, in the form of a new rule/signature, can be added if the overall system identifies a new malicious flow. In this way, either the choreography in the final stage, ML or VOL can add new signatures to the database of the SIG.

The overall workflow of the deployment of the scenario is shown in Figure 2. The initial choreography written in Choral is compiled and the result is a set of java files, one for each VNF. The Choral compiled code and the code to implement the local functions are then compiled to obtain a JAR application ready to be deployed in a container. Kathara then creates the final infrastructure, loading the JAR application and installing the necessary tools to execute the VNF at runtime. Once the containers are created, Kathara creates the network scenarios and deploys the infrastructure using a cloud provider, which in our case has been a local deployment with both docker containers and docker network.

[7] `https://www.kathara.org/`

[8] `https://github.com/p4lang/behavioral-model`

[9] `https://scikit-learn.org/stable/`

[10] `https://scapy.net/`

[11] `http://wiki.hping.org/`

[12] https://suricata.io/

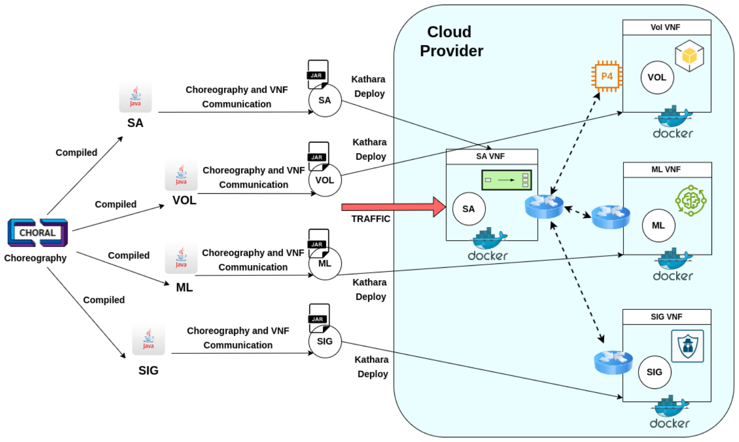

**Fig. 2.** The case study workflow, from the Choral code to the infrastructure deployment

**Test** To test our network, we generated traffic with up to 5 malicious flows. VOL has been set with a low detection threshold, useful for promptly identifying a volumetric attack and submitting it for analysis to MLE. We ran the attack traffic for 5 minutes and considered the amount of traffic management generated from all the REST API and inter-process communication calls. As expected, even in the worst scenario of 5 malicious flows generated (which represented 90% of the total traffic) the amount of management traffic generated was proportional to the number of malicious flows identified, and reached a maximum level of 15MB that was manageable by our infrastructure.

## 5 Discussion: Advantages and Limitations

We conclude by discussing the advantages and open challenges of using choreographic programming and looking at future work. We structure the discussion by comparing our proposal against the traditional SDN implementation with a centralized controller orchestrating all the function chains.

*Advantage: Direct Intra VNF Communications* The choreographic approach allows direct communication between VNFs. In classical SDN architectures, such communication is not possible unless hardcoded directly into the VNFs, which is discouraged since hardcoding communication makes the component difficult to port and extend. The best practice chosen by ETSI is instead to run all requests through the controller, following a star-like architecture where the controller mediates all communications. Using direct communication between the VNFs can save traffic (e.g., no need to have two communications with the orchestrator if a VNF has to send data to another one).

*Advantage: No SDN Controller/Orchestrator* The orchestration of VNF chains is typically implemented within the controller itself or as an application layer. The controller (often seen as a Network Operation System) is designed to interact with applications through a so-called Northbound Interface, similar to an operating system kernel that accesses device drivers. Traditionally, to create a VNF chain, it is necessary to create a new northbound application, that implements the communication logic between VNFs. This requires implementing the communication logic and adapting it to the proper controller like ONOS [34]. With the choreographic approach, we are not tied to any particular type of controller and we are not required to follow any specific design pattern. We are not bound to use libraries and controller code that must be compatible with the rest of the environment and applications. Since choreographic programming allows independent communication among VNFs, the choreographic solution thus avoids the need to create bottlenecks typical of controller-based SDN architecture.

*Advantage: Security by Design* In a classical SDN approach, performing verification on the validity of network policies is done with various formal model techniques such as reachability graphs [35] or by using atomic predicates [36] at the data plane level. With choreographic programming, we use a security-by-design approach for network development that avoids the typical communication problems of distributed systems (e.g., deadlocks, race conditions, etc). Moreover, with a choreographic approach, the availability of a global overview of the entire system eases the task of verifying the global properties of the system at the application/logic level.

*Advantage: Parallel VNF execution* In the traditional approach, the VNFs workflow is often rigid and sequential: each VNF is executed one after another, leading to a linear progression of tasks. While this method is effective in ensuring that each function is processed in a controlled manner, it may also introduce bottlenecks and inefficiencies, especially when dealing with complex network architectures or high volumes of data. Choreographic programming removes the constraint of executing VNFs sequentially. Multiple VNFs can be initiated and processed simultaneously, without the need to wait for the completion of preceding tasks, unlocking new possibilities for optimizing network performance and resource utilization.

*Challenge: failure handling* Choreographic languages assume reliable communications. The only exception is the language theory presented in [37], which shows that one can relax this assumption, by allowing the choreographic language to handle local exceptions. Choral follows this strategy relying on the exception mechanism of Java and local failure recovery code [8, Sec. 2.5] which results in codebases that mix high-level choreographic interactions and low-level recovery strategies. Although Choral's object-orientation allows programmers to encapsulate the latter into high-level APIs, its type system can offer limited support to reason about the robustness of recovery strategies. Indeed, supporting programmers in writing robust and effective choreographies is still an open issue beyond Choral or even choreographic programming.

*Challenge: knowledge of choice* When a choreography describes a choice between two possible branches, all affected participants must be (made) aware of the outcome to ensure that their local implementations agree on which branch to execute. In choreographies, this is called knowledge of choice (KoC). The standard solution for achieving KoC is communicating the choice outcome to the affected participants using special messages used by choreographic compilers to check that KoC is indeed achieved. Because of limitations in the current compilers, some of these communications might be redundant and there is ongoing work to address this issue: [38] proposed a more flexible analysis for the Choral compiler that allows piggybacking of these special messages batching them with other communications in the protocol; [39, 40] detail automatic procedures for inserting these communications for programmers; [41] proposed an analysis in an abstract choreographic language that dispenses from many of the communications needed by current analyses.

## 6 Conclusion

We presented a novel methodology for service composition in Software-Defined Networks and Network Function Virtualization, specifically tailored for cloud environments. Departing from conventional sequential service chaining, the approach uses choreographies to model Virtual Network Functions' (VNFs) roles and interactions. We showcase several advantages of the proposed approach, such as a holistic view of interactions and automatic code generation for each VNF, which eliminates the need for a centralised control node, reducing concurrency issues and communication overhead with the controller.

The validity of the proposed approach derives from the CP paradigm, which guarantees the implementation of a correct-by-construction VNF architecture given a choreography. We demonstrated the feasibility of the proposed approach via a practical case study where we used the state-of-the-art choreographic language Choral to develop a distributed composition of several VNFs collaborating to analyse network traffic and detect security threats. Qualitatively, we show that our approach decreases the number of communications happening in the system w.r.t. the traditional SDN implementation as an orchestrated system. Providing a quantitative validation is a necessary future step of this research direction, considering representative, real scenarios and metrics that demonstrate the efficiency of the proposed solution.

We envision two further future directions. The first one encompasses the challenges presented above, related to error handling and recovery. The second one envisages the definition of a meta-choreography that could define the infrastructural interactions needed to deploy the VNFs, by interacting with the SDN controller, and the dynamic and flexible forwarding of traffic by means of programmable data plane devices. A further extension of this undertaking could be the development of a new compiler for Choral that, instead of generating Java, could output P4 code, so that some parts of the distributed application may be executed on programmable switches instead of containers or virtual machines.

**Acknowledgements.** Partially supported by Villum Fonden (grant no. 29518). Co-funded by the European Union (ERC, CHORDS, 101124225). Partially supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU. Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

## References

[1] L. Yang *et al.*, "Forwarding and control element separation (forces) framework,"

Tech. Rep., 2004.

[2] R. Mijumbi *et al.*, "Network function virtualization: State-of-the-art and research challenges," *IEEE Communications Surveys & Tutorials*, 2015.

[3] C. Sun *et al.*, "Nfp: Enabling network function parallelism in nfv," in *Proc. Conference of the ACM Special Interest Group on Data Communication*, 2017.

[4] T. Leesatapornwongsa *et al.*, "Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems," in *ASPLOS*, T. Conte and Y. Zhou, Eds., ACM, 2016.

[5] E. M. Clarke *et al.*, "Model checking and the state explosion problem," in *Tools for Practical Software Verification, LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*, B. Meyer and M. Nordio, Eds., ser. Lecture Notes in Computer Science, Springer, 2011.

[6] F. Montesi, "Choreographic programming," Ph.D. Thesis, IT University of Copenhagen, 2013.

[7] F. Montesi, *Introduction to Choreographies.* Cambridge University Press, 2023.

[8] S. Giallorenzo *et al.*, "Choral: Object-oriented choreographic programming," *ACM Trans. Program. Lang. Syst.*, 2024.

[9] F. Hu *et al.*, "A survey on software-defined network and openflow: From concept to implementation," *IEEE Communications Surveys & Tutorials*, 2014.

[10] F. Callegati *et al.*, "Sdn for dynamic nfv deployment," *IEEE Communications Magazine*, 2016.

[11] A. Liatifis *et al.*, "Advancing sdn from openflow to p4: A survey," *ACM Computing Surveys*, 2023.

[12] E. F. Kfoury *et al.*, "An exhaustive survey on p4 programmable data plane switches: Taxonomy, applications, challenges, and future trends," *IEEE Access*, 2021.

[13] J. Xing *et al.*, "Runtime programmable switches," in *USENIX*, 2022.

[14] A. Al Sadi *et al.*, "Unleashing dynamic pipeline reconfiguration of p4 switches for efficient network monitoring," *IEEE Transactions on Network and Service Management*, 2024.

[15] M. Carbone and F. Montesi, "Deadlock-freedom-by-design: Multiparty asynchronous global programming," in *POPL*, R. Giacobazzi and R. Cousot, Eds.

[16] M. Dalla Preda *et al.*, "Dynamic choreographies: Theory and implementation," *Logical Methods in Computer Science*, 2017.

[17] A. K. Hirsch and D. Garg, "Pirouette: Higher-order typed functional choreographies," *Proc. ACM Program. Lang.*, 2022.

[18] G. Shen *et al.*, "Haschor: Functional choreographic programming for all (functional pearl)," *Proc. ACM Program. Lang.*, 2023.

[19] H. Hüttel *et al.*, "Foundations of session types and behavioural contracts," *ACM Comput. Surv.*, 2016.

[20] T.-M. Nguyen *et al.*, "Optimizing resource utilization in nfv dynamic systems: New exact and heuristic approaches," *Computer Networks*, 2019.

[21] L. He *et al.*, "Towards chain-aware scaling detection in nfv with reinforcement learning," in *2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS)*, IEEE, 2021.

[22] A. Leivadeas and M. Falkner, "A survey on intent-based networking," *IEEE Communications Surveys & Tutorials*, 2023.

[23] J. Hasneen and K. M. Sadique, "A survey on 5g architecture and security scopes in sdn and nfv," in *ICCET*, Springer, 2022.

[24] S. Lakshmanan Thirunavukkarasu *et al.*, "Modeling nfv deployment to identify the cross-level inconsistency vulnerabilities," in *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2019.

[25] A. Huff *et al.*, "Building multi-domain service function chains based on multiple nfv orchestrators," in *IEEE NFV-SDN*, 2020.

[26] K.-Y. Chen *et al.*, "Sdnshield: Nfv-based defense framework against ddos attacks on sdn control plane," *IEEE/ACM Transactions on Networking*, 2021.

[27] A. Melis *et al.*, "P-scor: Integration of constraint programming orchestration and programmable data plane," *IEEE Transactions on Network and Service Management*, 2020.

[28] D. Arp *et al.*, "Dos and don'ts of machine learning in computer security," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

[29] A. A. Sadi *et al.*, "Unleashing dynamic pipeline reconfiguration of p4 switches for efficient network monitoring," *IEEE Transactions on Network and Service Management*, 2024.

[30] R. Doriguzzi-Corin *et al.*, "Lucid: A practical, lightweight deep learning solution for ddos attack detection," *IEEE TNSM*, 2020.

[31] A. Al Sadi *et al.*, "Real-time pipeline reconfiguration of p4 programmable switches to efficiently detect and mitigate ddos attacks," in *ICIN*, IEEE, 2023.

[32] "Cicids2017 dataset," *https://www.unb.ca/cic/datasets/ids-2017.html*,

[33] "Nuclei discovery project," *https://github.com/projectdiscovery/nuclei-templates/*,

[34] P. Berde *et al.*, "Onos: Towards an open, distributed sdn os," in *Proceedings of the third workshop on Hot topics in software defined networking*, 2014.

[35] D. Berardi *et al.*, "Technetium: Atomic predicates and model driven development to verify security network policies," in *2020 IEEE 17th Annual Consumer Communications & Networking Conference (CCNC)*, IEEE, 2020.

[36] H. Yang and S. S. Lam, "Real-time verification of network properties using atomic predicates," *IEEE/ACM Transactions on Networking*, 2015.

[37] F. Montesi and M. Peressotti, "Choreographies meet communication failures," *CoRR*, 2017. arXiv: `1712.05465`.

[38] L. Lugovic and F. Montesi, "Real-world choreographic programming: Full-duplex asynchrony and interoperability," *Art Sci. Eng. Program.*, 2023.

[39] S. Basu and T. Bultan, "Automated choreography repair," in *FASE*, ser. Lecture Notes in Computer Science, Springer, 2016.

[40] L. Cruz-Filipe and F. Montesi, "A core model for choreographic programming," *TCS*, 2020.

[41] S. Jongmans and P. van den Bos, "A predicate transformer for choreographies - computing preconditions in choreographic programming," in *ESOP*, I. Sergey, Ed., ser. Lecture Notes in Computer Science, Springer, 2022.