D4.4
Implementation of Self-Management of Network Security and Resilience

Project number:	643964
Project acronym:	**SUPERCLOUD**
Project title:	User-centric management of security and dependability in clouds of clouds
Project Start Date:	1st of February, 2015
Duration:	36 months
Programme:	H2020-ICT-2014-1
Deliverable Type:	Report
Reference Number:	ICT-643964-D4.4/ 1.0
Work Package:	WP 4
Due Date:	Jan 2018 - M36
Actual Submission Date:	31st of January, 2018
Responsible Organisation:	FCiencias.ID
Editor:	Fernando M. V. Ramos, Nuno Neves
Dissemination Level:	PU
Revision:	1.0
Abstract:	This deliverable presents the overall architecture of the network virtualization platform, including the final version of the description, implementation and evaluation of the services and protocols that were developed.
Keywords:	Network virtualization, multi-cloud, software-defined network, security

This project has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No 643964.

This work was supported (in part) by the Swiss State Secretariat for Education, Research and Innovation (SERI) under contract number 15.0091.

Editor

Fernando M. V. Ramos, Nuno Neves (FCiencias.ID)

Contributors (ordered according to beneficiary numbers)

Ruan He, Pascal Legouge, Marc Lacoste (ORANGE)
Khalifa Toumi (IMT)

Disclaimer

Executive Summary

In this deliverable we describe the final version of the SUPERCLOUD network virtualization platform. We make an overview of its architecture, following a Software-Defined Networking (SDN) design, present its main components, and the techniques used to improve the dependability and security of the solution. The document is focused on the three main challenges involved in the conception and development of the platform: how to offer virtual networks to SUPERCLOUD users with enhanced security services; how to ensure the security and dependability of the infrastructure; and how to leverage it to offer autonomic security services to users' virtual networks. Towards this goal, we first present the solutions developed in SUPERCLOUD for secure and scalable virtual network embedding, the core component of the network hypervisor, consisting of algorithms that map virtual network requests to physical multi-cloud resources, taking into account the security and dependability demands by users. Second, we present the design, implementation, and evaluation of our proposal of a logically-centralised security architecture for the platform, and its use to enable efficient and secure control plane communications. For dependability, we present our design of a novel fault-tolerant SDN controller. Finally, we describe the autonomic security management framework of the network virtualization platform, a component that offers data plane monitoring and proactive attack detection across the different providers of the SUPERCLOUD. The integration of the network hypervisor with other core components of the SUPERCLOUD was successfully demonstrated, namely with the authentication service (Work Package 2), the storage service Janus (Work Package 3), and the Maxdata and Phillips use cases (Work Package 5). As for future work, the network hypervisor could be enhanced with the capability to scale virtual networks up and down, offering elasticity to tenants' infrastructures; to allow for migration of network and compute resources, for improved efficiency of the substrate infrastructure; and to integrate programmability to virtual data plane elements, for enhanced virtual network services, including advanced network functions for security.

Contents

List of Figures

List of Tables

Chapter 1 Introduction

The objective of SUPERCLOUD Work Package 4 (WP4) was to develop a platform that creates a virtual network abstraction to the SUPERCLOUD user, spanning multiple heterogeneous Cloud Service Providers (CSPs). For this purpose we followed a Software-Defined Networking [88] approach, and proposed innovative solutions, including the enhancement of network virtualization with security services, the design and implementation of resilient SDN infrastructures, and the introduction of autonomic security management in virtual networks. Previously, we have focused on the design of the SUPERCLOUD network virtualisation architecture and on prototyping its various components. The preliminary architecture was presented in Deliverable D4.1. In Deliverable D4.2, we presented its evolution alongside a detailed description of its main components and the techniques used to improve the dependability, scalability, and security of the platform. In Deliverable D4.3, we presented the proof-of-concept prototype of the multi-cloud network virtualisation infrastructure, and the APIs of its main components.

1.1 Objective of the document

In this deliverable we present the overall architecture of the network virtualization platform, including the final version of the description, implementation and evaluation of the core services and protocols that were developed.

1.2 Outline

The rest of this document is organized as follows. In Chapter 2 we start with an overview of the network virtualisation architecture. Then, in Chapter 3 we present the core component of the network hypervisor: the embedding module. In particular, we address the problem of enhancing virtual networks with security, and achivieng it in a way that scales to large networks. Afterwards, we focus on the security and dependability of the infrastructure, in Chapter 4. We describe our proposal of a logically centralized security architecture, its use in enabling secure and efficient control plane communications, and the design of a novel fault-tolerant SDN controller. In Chapter 5 we describe the autonomic security management framework of the SUPERCLOUD network hypervisor. The deliverable closes with a discussion on integration aspects and conclusions, in Chapter 6.

Chapter 2 Multi-Cloud Network Virtualization Architecture

Current multi-tenant network hypervisors target single-provider deployments and traditional services, such as flat L2 or L3 routing, as their goal is to enable tenants to use their existing cloud infrastructures. Such single-cloud paradigm has inherent limitations in terms of scalability, security, and dependability, which may potentially dissuade critical systems to be migrated to the cloud. For instance, a tenant may want to outsource part of its compute and network infrastructure to a public cloud, but may not be willing to trust the same provider to store its confidential business data or to run sensitive services, which should stay in a more trusted environment (e.g., a private datacenter). To avoid cloud outages disrupting its services – a type of incident increasingly common [61, 151] – the tenant may also wish to spread its services across clouds, to avoid Internet-scale single points of failures.
To address this challenge, in SUPERCLOUD, we have developed Sirius, a multi-cloud network virtualisation platform. Contrary to previous approaches, Sirius leverages a substrate infrastructure that entails both public clouds and private datacenters. This brings several important benefits. First, it increases resilience. Replicating services across providers avoids single points of failure and therefore makes a tenant immune to any datacenter outage. Secondly, it can improve security, for instance by exploring the interaction between public and private clouds. A tenant that needs to comply with privacy legislation may demand certain data or specific services to be placed in trusted locations. In addition, it can improve performance and efficiency. For example, the placement of virtual machines may consider service affinity to reduce latencies. Dynamic pricing plans from multiple cloud providers can also be explored to improve cost-efficiency [173]. The multi-cloud model has been successfully applied in the context of computation [160] and storage [28] recently. To the best of our knowledge, this is the first time the model is applied for network virtualisation.
In our platform, users can define virtual networks with arbitrary topologies, while making use of the full address space. Sirius further improves over existing network virtualisation solutions by allowing users to specify security and dependability requirements for all virtual resources. In this chapter, we present the Sirius architecture.

2.1 General design and operation

Sirius allows an organization to manage resources belonging to multiple clouds, which can then be transparently shared by various users (or *tenants*). Resources are organized as a single substrate infrastructure, effectively creating the abstraction of a cloud that spreads over several clouds, i.e., a *cloud-of-clouds* [91]. In this chapter, the considered resources are interconnected *virtual machines* (VM) that are either acquired from public cloud providers or are placed in local facilities (i.e., private clouds). Envisioned extensions include other cloud resources, such as storage services.
Users can define virtual networks composed of a number of *containers* interconnected according to an arbitrary topology. Sirius deploys these virtual networks on the substrate infrastructure, ensuring isolation of the traffic by setting up separated datapaths (or *flows*). While specifying the virtual network, it is possible to indicate several requirements for the nodes and links, for example with respect to the needed bandwidth, security properties, and fault tolerance guarantees. These requirements are enforced during embedding by laying out the containers at the appropriate locations, where the

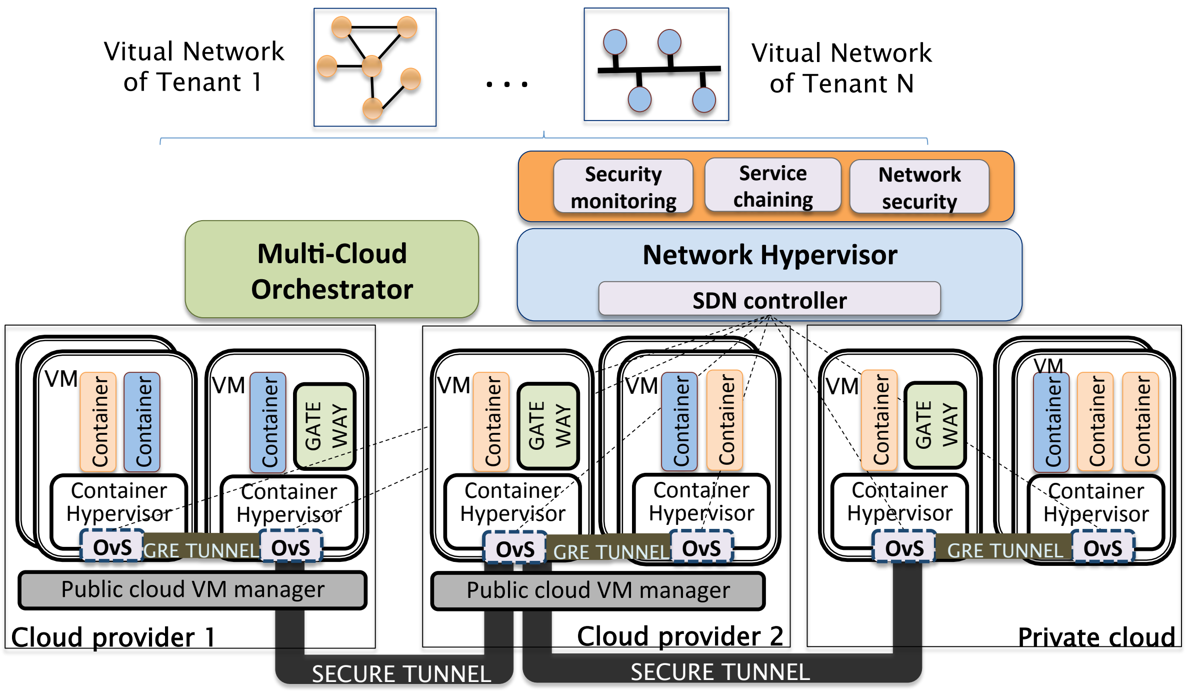


Figure 2.1: Sirius architecture.

substrate infrastructure still has enough resources to satisfy the particular demands. In addition, the datapaths are configured to follow adequate routes through the network.
In the rest of this section, we present the design of Sirius. First, we describe the architecture of the platform and give a step-by-step overview of its operation while creating a virtual network. Next, we elaborate on the two main components of Sirius, the network hypervisor and the cloud orchestrator.[1]

2.1.1 Architecture

The architecture of Sirius is displayed in Figure 2.1. The *cloud orchestrator* (Section 2.2.1) is responsible for the dynamic creation of the substrate infrastructure by deploying the VMs and containers. It also configures secure tunnels between gateway modules, normally building a fully connected topology among the participating clouds. A gateway acts like an edge router, receiving local packets whose destination is in another cloud and then forwarding them to its peer gateways, allowing data to be sent securely to any container in the infrastructure. Intra-cloud communications between tenant containers use GRE (Generic Routing Encapsulation) tunnels setup between the local VMs, to ensure isolation.
The *network hypervisor* (Section 2.2.2) runs as an application on top of a Software-Defined Networking (SDN [88]) controller. It takes all decisions related to the placement of the virtual networks, and setups the network paths by configuring software switches (Open vSwitch [120]) that are installed in all VMs (along with OpenFlow [102] hardware switches that may exist in private clouds, not shown in the figure). The hypervisor intercepts the control messages between the substrate infrastructure and the users' virtual networks, and vice-versa, thus enabling full network virtualisation.
The hypervisor was developed using a shared controller approach (the solution also adopted in [81]). Alternative solutions, including OVX [8], assume one controller per tenant. Ours is a more lightweight solution, as only one logically-centralized component is needed for all tenants. It is also simpler to implement as it can take advantage of the high-level APIs offered by the SDN controller, instead of having to deal with "raw" Openflow messages when interacting with the switches. Finally, this architecture follows a fate-sharing design as the controller and the network hypervisor reside in the same host. This facilitates replication for fault-tolerance.

[1]Like in the Sirius star system, our platform also has two companion components.

The *self-management security services* run on top of the network hypervisor. They include a *security monitor* to detect security incidents, a *network security service* that responds to these incidents, and a *service chaining component* that allows users to compose their own security service chains. In Section 2.3, we give an overview of each of these services.

2.1.2 Overview of Sirius operation

The deployment of a virtual network in the platform involves the execution of a few tasks.
Building the substrate infrastructure. The first task is to assemble the substrate infrastructure. The administrator of Sirius within the organization needs to indicate the resources that are available to build the infrastructure. She interacts with a graphical interface[2] offered by the cloud orchestrator that allows the selection of the cloud providers, the type and number of VMs that should be created, and the provision of the necessary access credentials. The network topology is also specified, pinpointing for instance the connections between clouds. For each provider, it is possible to specify a few attributes, such as the associated trust level.
Based on such data, the orchestrator constructs the substrate infrastructure by interacting with the cloud providers and by setting up the VMs. In each VM, a few skeleton containers are started with minimal functionality. The gateways are also interconnected with the secure tunnels. The next step is for the hypervisor to be initialized by obtaining, from the orchestrator, information about the infrastructure. Then, it contacts each network switch to obtain data about the existing interfaces, port numbers and connected containers. After populating the hypervisor's internal data structures, Sirius is ready to start serving the users' virtual network (VN) requests.
Running virtual networks on-demand. The second task starts when a user of the organization needs to run an application in the cloud. The user employs a graphical interface of the orchestrator to represent a virtual network with the various containers that implement the application. Containers are then interconnected with the desired (virtual) switches and links. Complete flexibility is given on the choice of the network topology and addressing schemes. Attributes may be associated with the containers and links, specifying particular requirements with respect to security and dependability. For example, certain links may need to have backup paths to allow for fast fail-over, while certain containers may only be deployed in clouds with the highest trust levels.
The orchestrator receives the VN request and forwards it to the hypervisor to perform the virtual network embedding. The *embedding algorithm* decides on the location of the containers and network paths considering all constraints, namely the available resources in the substrate infrastructure and the security requirements. The computed mapping is transmitted to the orchestrator so that it can be displayed upon request of the Sirius administrator. Hereafter, the orchestrator and the hypervisor work in parallel to start the VN. The orchestrator downloads and initializes the containers images in the chosen VMs, and configures the IP and MAC addresses based on the tenant's request. The hypervisor enables connectivity by configuring the necessary routes by setting up the flows in the switches, while enforcing isolation between tenants.

2.2 Network virtualisation core components

2.2.1 Multi-cloud orchestrator

The main modules of the multi-cloud orchestrator are detailed in Figure 2.2. The multi-cloud orchestrator combines three main features. First, it manages interactions with users through a web-based graphical interface. Users with administrator privileges can design the substrate infrastructure topology (Admin GUI), indicating the kind of VMs that should be deployed in each cloud provider. Similarly, normal users can represent virtual networks of virtual hosts (e.g., containers), and later

[2]The same sort of information can also be provided through configuration files, to simplify the use of scripts.

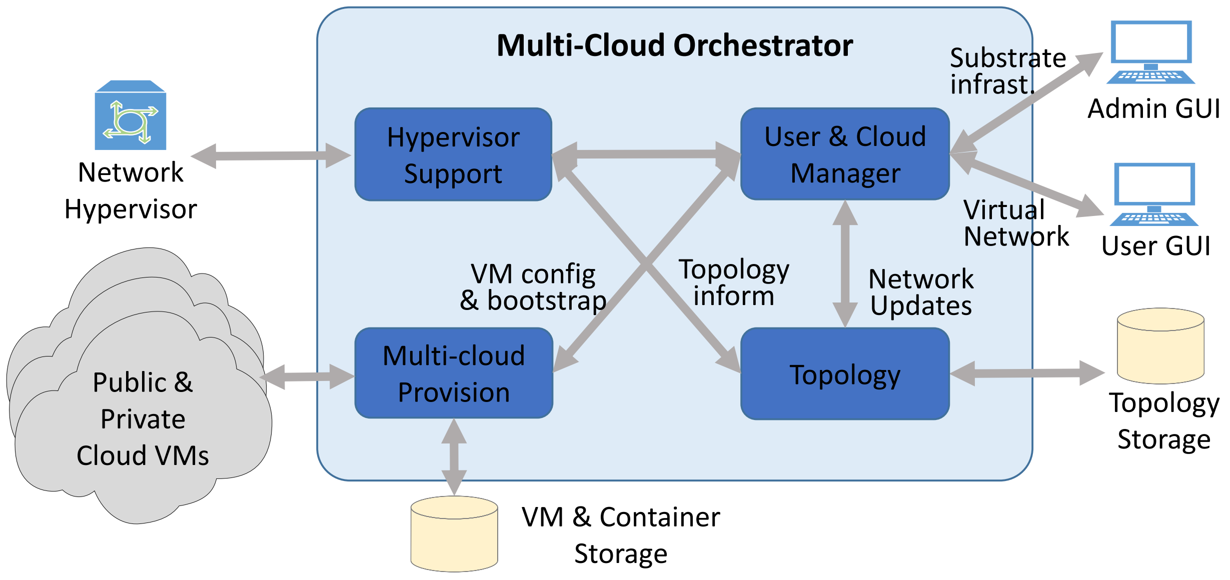


Figure 2.2: Orchestrator's main modules.

request their deployment (User GUI). The graphical interface also displays the mappings between the containers and links in the substrate infrastructure and the status of the various components.
Second, it keeps information about the topologies of the substrate and virtual networks and their mappings. This information is kept updated, as virtual networks are created and destroyed, thus offering a complete view of how the infrastructure is currently organized. In addition, it maintains in external storage a representation of the different networks that were specified, allowing their re-utilization when users want to run similar deployments.
Third, it configures and bootstraps VMs in the clouds in cooperation with the network hypervisor and setups the tunnels for the inter-cloud connections. Apart from that, when a virtual network is started, it also initiates the containers in the VMs selected by the hypervisor. A storage of VM and containers is kept locally, in case the users prefer to work and save the images within the organization.

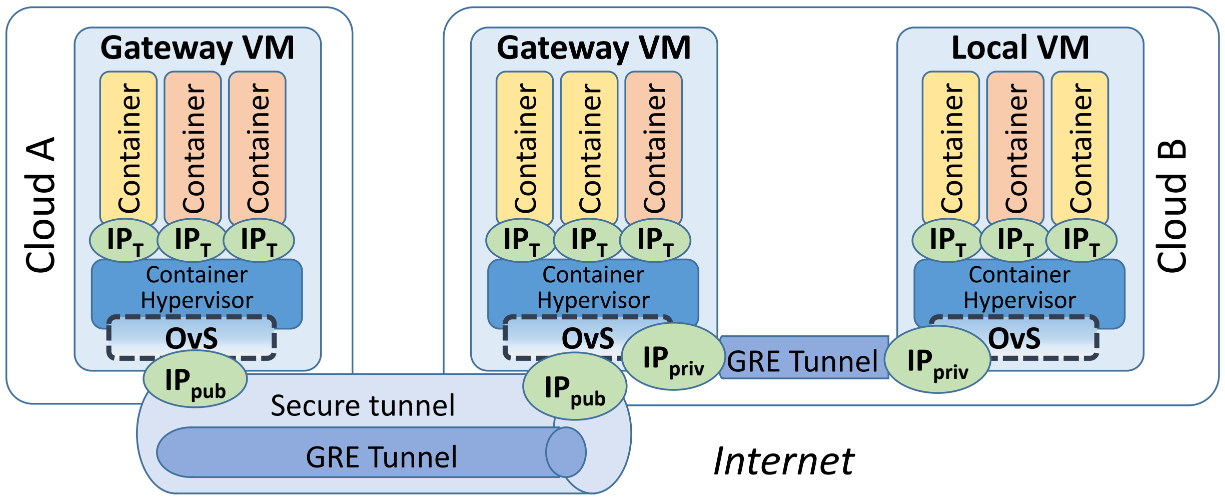


Figure 2.3: Intra- and inter-clouds connections.

Figure 2.3 shows the main connections that are managed within the infrastructure. Gateways have public IPs that work as endpoints of secure tunnels between the clouds. In our current implementation, OpenVPN with asymmetric key authentication is employed as the default solution as it presents the advantage of being generic and independent from the provider's gateway service (e.g. VPC service for Amazon EC2). Links between VMs rely on GRE tunnels. We chose this simple approach as intra-cloud communications are expected to run within a controlled environment and inter-cloud traffic is protected by the secure tunnel. The containers use the IP addresses defined by the tenants (without restrictions), and isolation is achieved by the network hypervisor properly configuring the switches' flow tables (an aspect to be detailed in Section 2.2.2).

2.2.2 Hypervisor

The design of the hypervisor software follows a modular approach. We present its building blocks in Figure 2.4.
The **Embedder** addresses the problem of mapping the virtual networks specified by the tenants into the substrate infrastructure [60]. As soon as a virtual network request arrives, the *secure Virtual Network Embedding* (VNE) module finds an effective and efficient mapping of the virtual nodes and links onto the substrate network, with the objectives of minimizing the cost of the provider and maximizing its revenue.
This objective takes into account, firstly, constraints about the available processing capacity of the substrate nodes and of the available bandwidth resources on the links. Moreover, we consider security and dependability constraints based on the requirements specified by the tenants to each virtual resource. These constraints address, for instance, concerns about attacks on virtual machines or on substrate links (e.g., replay/eavesdropping). As such, each particular node may have different security levels, to guarantee for instance that sensitive resources are not co-hosted on the same substrate resource as potentially malicious virtual resources. In addition, we consider the coexistence of resources (nodes/links) in multiple clouds, both public and private, and assume that each individual cloud may have distinct levels of trust from a user standpoint.

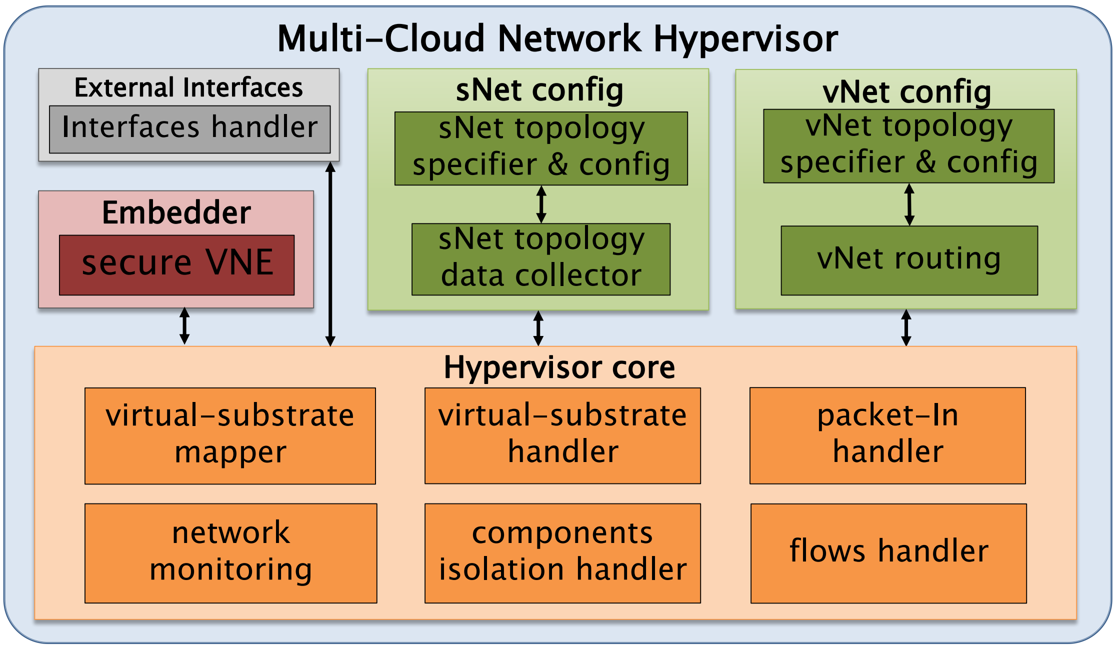


Figure 2.4: Modular architecture of the network hypervisor.

The **Substrate Network (sNet) Configuration** module is responsible for maintaining information about the substrate topology. It performs two main functions. First, it retrieves information from the orchestrator about the substrate nodes and links, alongside their security and dependability characteristics. Second, it interacts with each switch to set itself as its master controller, and to collect more

detailed information, including switch identifiers, port information (e.g., which ports are connected to which containers), etc. This information is maintained in efficient data structures to speed up data access.
The **Virtual Network (vNet) Configuration** module is responsible for maintaining information about the virtual network topologies. This includes both storing tenant requests and the mapping that results from the embedding phase. As the embedding module outputs only the substrate topology that maps to the virtual network request, this module runs a routing algorithm to define the necessary flow rules to install in the switches (without populating them, which is left for the next module).
The **Hypervisor core** module is configured as a controller module (in our case, Floodlight). Its first component is the *virtual-substrate mapper* that, after interacting with the substrate topology and virtual topology modules, requests a specific mapping to the embedder. When the VNE returns successfully, the mapping is stored in specific data structures of the core module and this information is shared with other interested modules (namely, the vNet configuration module).
The *network monitoring* component is responsible to detect changes in the substrate topology when a reconfiguration occurs (e.g., due to failures in the substrate network). This module then sends requests to the virtual-substrate handler to update its data structures accordingly. As ongoing work, we are implementing mechanisms to respond to network changes.
Isolation is handled by several sub-modules, including the *isolation handler*, the *packet-in handler* and the *flows handler*. These components' goal is to guarantee that each tenant perceives itself as the only user of the infrastructure. We currently use four main techniques for this purpose.

- First, as we have control over the entire infrastructure, from the network core to the edge, we uniquely identify each tenants' host by its precise location.
- Second, based on this unique identification and on the tenant ID we perform address translation at the network edge from the tenant's MAC to an ephemeral MAC address (eMAC) and install the required flows based on the eMAC. For communication between all virtual nodes, a set of flows is initially installed pro-actively by the flow handler module in such a way as to guarantee isolation between tenants' traffic. For efficiency reasons, flows are installed with predefined timeouts. When a timeout expires [3] the flow is removed from the switches to save flow table resources. If communication ensues between those nodes afterwards, the first packet of the flow generates a packet-in that is sent to the hypervisor, triggering the packet-in handler to install the required flows in switches.
- Third, we perform traffic isolation during the initial steps of communication, namely, by treating ARP requests and replies.
- Finally, flow table isolation is guaranteed by each virtual switch having its own virtual flow tables, with predefined size limits.

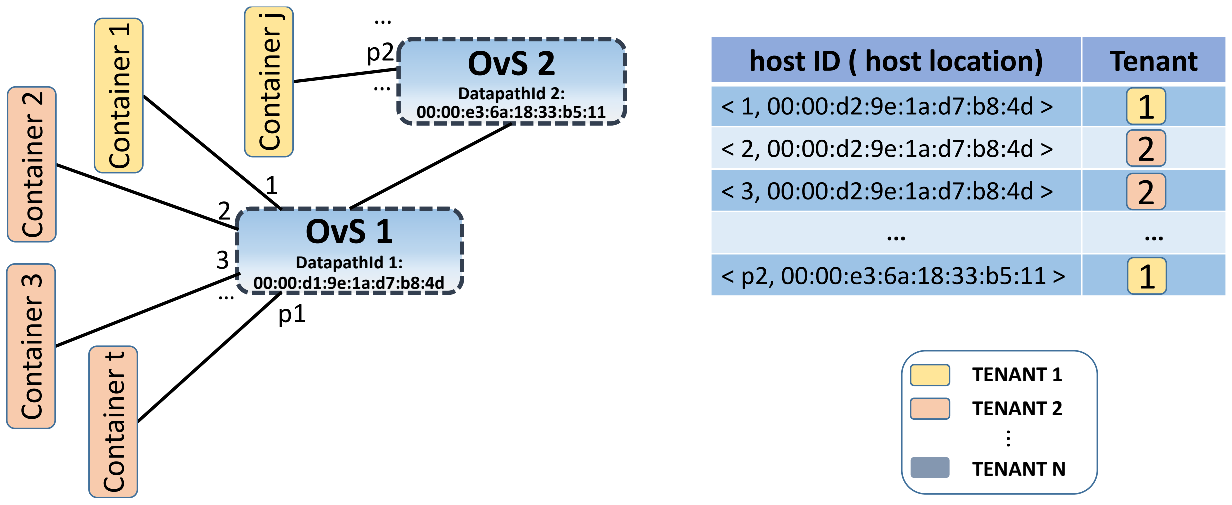
We detail these techniques further in the next section.

2.2.3 Virtualisation runtime: achieving isolation

The main requirement of our multi-tenant platform is to provide full network virtualisation. To achieve this goal it is necessary to virtualize the topology and addressing, and to guarantee isolation between tenants' networks. Topology virtualisation is achieved in our system by means of the embedding procedure already described. In this section, we focus on the other aspects.
Sirius allows tenants to configure their VMs with any L2 (Ethernet) and L3 (IP) addresses. Tenants thus have complete autonomy to manage their address space. To achieve this goal and guarantee isolation, we create a unique identifier for each tenant's hosts based on their location. We then perform edge-based translation of the host MAC address to an ephemeral MAC address that includes

[3]Which means a particular pair of nodes has not communicated during that period.

this ID. Finally, we setup tunnels between every Open vSwitch (i.e., between every VM of the substrate infrastructure).
An alternative solution that would also fulfill our requirements would be to setup tunnels between all tenant's hosts (in our solution this would mean setting up tunnels between containers). This would avoid the need to maintain host location information and of edge-based translation. The problem with this option is scalability. The number of tunnels would grow with the number of containers (i.e., with the number of tenant's hosts), whereas our solution scales much better, as it grows with the number of provider VMs (in a production setting, each VM is expected to run hundreds or even thousands of containers).
Uniquely identifying hosts. As explained, the tenant's hosts of our solution are containers. We opted for this operating system virtualisation technology as it provides functionality similar to a VM but with a lighter footprint. Each container (i.e., each tenant's host) has its own namespace (IP and MAC addresses, name, etc.) and its own resources (processing capacity, memory), and as such can be seen as a lightweight VM.



host ID (host location)	Tenant
< 1, 00:00:d2:9e:1a:d7:b8:4d >	1
< 2, 00:00:d2:9e:1a:d7:b8:4d >	2
< 3, 00:00:d2:9e:1a:d7:b8:4d >	2
...	...
< p2, 00:00:e3:6a:18:33:b5:11 >	1

Figure 2.5: Tenant host identification: host ID = <Switch port, DatapathId>

To uniquely identify a tenant's host, we use its network location. Each container is connected to a specific software switch (identified by a *DatapathID*), being attached to a unique port. As such, we use as *hostID* the tuple $\langle switch\ port, DatapathId\rangle$. Figure 2.5 shows an example.
Edge address translation. Packets generated in a virtual network cannot be transmitted unmodified in the substrate network. As different tenants can use the same addresses, collisions could occur. For this reason, we perform edge-based address translation to ensure isolation. We assign an *ephemeral* MAC address – eMAC – at the network edge, to replace the host's MAC address. The translation occurs at the edge switch. Every time traffic originates from a container, its host MAC is converted to the eMAC. Before the traffic arrives at the receiving container, the reverse operation occurs at the edge switch. The eMAC is composed of a tenant ID and a shortened version of the *hostID*, unique per tenant.
This mechanism guarantees isolation in the data plane. The control plane guarantees are provided by the hypervisor, as it has network-wide control and visibility. For this purpose the hypervisor populates the flow tables with two types of rules: translation rules in the edge switches, as just explained; and forwarding rules that enable communication between all hosts from a single tenant.
ARP handling. Hosts use the ARP protocol to map an IP address to an Ethernet address. As we want unmodified hosts to run in our platform, Sirius emulates the behavior of this protocol. When an ARP message arrives at a switch, it is forwarded directly to the destination host. Flooding is never needed as the switches are configured by the hypervisor. Even in those cases where the packet arriving at the switch does not match any flow rule – because it has expired – a packet-in is sent to the hypervisor, which populates the required tables with the necessary flow rules for the packet to be

forwarded to the destination.
Flow table virtualisation. As forwarding tables have limited capacity, in terms of TCAM (Ternary Content Addressable Memory) entries (hardware switches) or memory (software switches), in Sirius, each tenant has a finite quota of forwarding rules in each switch. This is important because the failure to isolate forwarding entries between users might allow one tenant to overflow the number of forwarding rules in a switch and prevent others from inserting their flows. Our hypervisor maintains a counter of the number of flow entries used per tenant switch, and ensures that a preset limit is not exceeded.
The hypervisor controls the maximum number of flows allowed per tenant, in both physical and virtual switches. This control is performed using the OpenFlow field *cookie* (an opaque data value that allows flows to be identified). When the hypervisor inserts a new flow in a switch (which only occurs if the limit was not exceeded), the cookie field is properly set to identify its tenant owner, and the counter for the number of flows in this switch that belong to this particular tenant is incremented. When a flow is removed, the hypervisor is informed, extracts from the cookie the tenant owner of the flow just removed, and decrements the corresponding counter.

2.2.4 Additional implementation details

The Sirius network hypervisor is implemented in Java as a Floodlight controller module. The orchestrator runs in an Apache Tomcat server. The client GUI is written in *Javascript/ JQuery* and uses *vis.js* [3], an open-source library for network visualization. Communication between the HTTP client and server is performed using *Servlet* technology.
We have deployed the Linux VMs and the Docker containers in three cloud infrastructures: Amazon EC2 and Google Cloud Platform as public clouds, and a private platform based on a set of VMs running in VirtualBox. In order to interconnect clouds, we use openvpn tunnels installed in each gateway VM (as illustrated in Figure 2.1). To interconnect the OVS of each VM, we use GRE tunnels. We manage the public cloud using *Apache jclouds*, a library that offers a simple interface to manage VMs running in public clouds. More importantly, it supports a large number of cloud providers and its generic API assures higher portability, which will facilitate future integration of other public clouds into the substrate infrastructure.

2.3 Self-management network security

To enhance the security of tenants' virtual networks, the hypervisor includes a self-management security module, composed of three components: security monitoring, network security, and service chaining. The *security monitoring component* allows the detection of security incidents in a tenant network hosted over a multi-cloud. For this purpose, it collects and processes information from these infrastructures to have a complete view of the state of the network, enabling automatic response to security incidents. These responses can be materialised by means of the *network security component*, that manages and deploys network security policies automatically by interacting with the security monitoring module. Finally, the *service chaining component* can be used as a response, and also to allow users to customize the composition of security services.

Given this overview, we detail in the next chapters the main components of the SUPERCLOUD network virtualization platform.

Chapter 3 Network Virtualization

In this chapter we present the core component of the network hypervisor: the *embedding module*. In Section 3.1, we present our solution for the *secure virtual network embedding* problem: mapping virtual network requests into the substrate infrastructure considering the security constraints required by SUPERCLOUD tenants. The solution is formulated as a Mixed Integer Linear Program, which due to its computational intractability does not scale to large networks. To address this challenge, in Section 3.2 we propose a heuristic solution that scales well and yet achieves performance close to the optimal, making it practical for large-scale deployments.

3.1 Secure Virtual Network Embedding

Network virtualization has emerged as a powerful technique to allow multiple heterogeneous virtual networks to run over a shared infrastructure. Nowadays, a number of production-level platforms have been proposed [81, 8], already achieving the necessary scale, performance, and required level of service. This has allowed cloud operators to start extending their service offerings of virtual storage and compute with network virtualization [81].
So far, these modern platforms have been confined to a datacenter, controlled by a single cloud operator. This constraint can be an important barrier as more critical applications start shifting to the cloud. To overcome this problem, the SUPERCLOUD network virtualization platform aims to extend network virtualization across multiple cloud providers [9, 91], bringing a number of benefits in terms of cost, performance, and versatility. In particular, a multi-cloud solution may contribute to security from several perspectives. For example, a tenant[1] that needs to comply with privacy legislation can demand a certain container (or virtual machine) to remain at a specific place while the rest can go to other facilities (*e.g.*, some services of a healthcare application, such as the analysis of patient medical images, can only be performed in pre-approved clouds). An application can also be made immune to any single datacenter (or cloud availability zone) outage by spreading its services across providers. Several incidents in cloud facilities are evidence of this increasingly acute risk [61, 151], motivating the exploration of availability-enhancing alternatives (*e.g.*, through replication over two providers).
This section tackles a fundamental component in our network virtualization solution – the *Virtual Network Embedding (VNE)* – from this new perspective. VNE addresses the problem of provisioning the virtual networks specified by the tenants [60]. When a virtual network request arrives, the goal is to find an effective mapping of the virtual nodes and links onto the substrate network, while maximizing the revenue of the virtualization operator. This objective is subject to various constraints, such as the processing capacity on the substrate nodes and bandwidth of the links.
A mostly unexplored perspective on this problem is providing security assurances. We propose a VNE solution that considers security constraints based on indications from the tenants. These constraints address, for instance, concerns about attacks on containers (*e.g.*, covert channels) or on physical links (*e.g.*, replay/eavesdropping). To further extend the resiliency properties of our solution, we support the coexistence of resources (nodes/links) in multiple clouds, both public and private, and assume that each individual cloud may have distinct levels of trust from a user standpoint.

[1]We employ the terms user and tenant interchangeably.

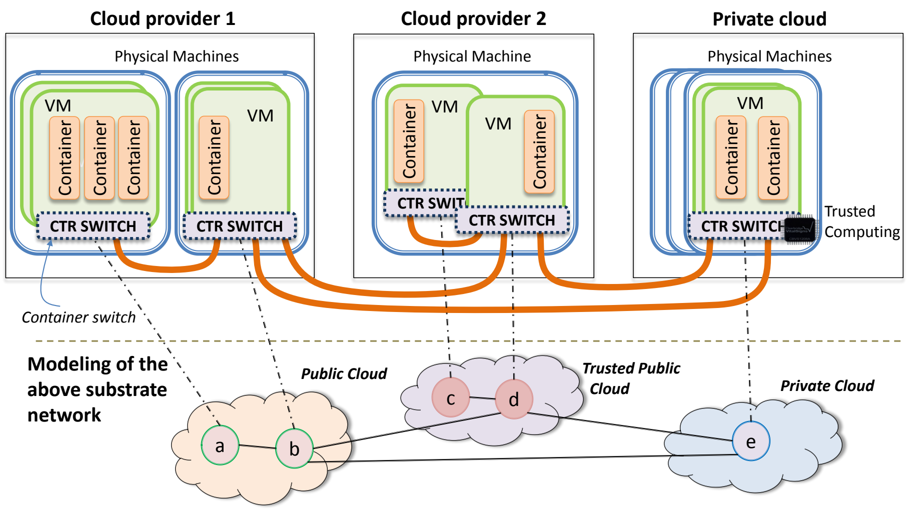


Figure 3.1: Example substrate network encompassing resources from multiple clouds.

We evaluate our proposal against the most commonly used VNE alternative [96]. The results show a better behavior of our approach in terms of acceptance rate of requests. Even when a reasonable number of requests imposes security constraints, the performance decrease is limited. This is a direct consequence of being harder to fulfill the requirements of the requests. With regard to resource utilization, the approach makes an efficient use of the links and nodes made available in the substrate, keeping a high average utilization.
The contributions of our work can be summarized as: (i) We formulate the SecVNE model and solve it as a Mixed Integer Linear Program (MILP). The novelty of our approach is in considering comprehensive security aspects over a multi-cloud deployment; (ii) We propose a new policy language to specify the characteristics of the substrate network, and to allow the expression of user requirements; (iii) We compare our formulation with the most commonly used VNE alternative [96], and analyze various trade-offs related to embedding efficiency, costs and revenues.

3.1.1 Network model

Our multi-cloud network virtualization platform, Sirius, leverages Software Defined Networking (SDN) to build a substrate infrastructure that spreads through both public clouds and private datacenters [9]. These resources can then be transparently shared by various users (or tenants) by allowing the definition and deployment of virtual networks (VN) composed of a number of containers arranged in an arbitrary network topology. While specifying the virtual network, it is possible to indicate several requirements for the switches and links, for example with respect to the needed bandwidth, CPU capacity, and security guarantees. These requirements are enforced during embedding by laying out the containers at the appropriate locations, where the substrate infrastructure still has enough resources to satisfy the particular demands. In addition, the datapaths are configured by the SDN controller by configuring the forwarding rules in the switches.
For example, as illustrated in Figure 3.1, virtual machines (VM) might be acquired at specific cloud providers to run tenants' containers implementing distributed services. In this scenario, the most relevant security aspects that may need to be assessed are the following:

- First, the trust level associated to a cloud provider is influenced by various factors, which may have to be taken into consideration with more critical applications. Providers are normally better regarded if they show a good past track record on breaches and failures, have been on the market for a while, and advertise Service Level Agreements (SLAs) with stronger assurances for the users. Moreover, as the virtualization operator has full control over its own data centers, he might employ protection features and procedures to make them compliant with regulations that have to be fulfilled by tenants (e.g., the EU GDPR that enters in force in 2018).

- Second, VMs can be configured with a mix of defense mechanisms, e.g., firewalls and antivirus, to build execution environments with stronger degrees of security at a premium price. These mechanisms can be selected by the operator when setting up the VMs, eventually based on the particular requirements of a group of tenants, or they could be sold ready to use by the cloud providers (e.g., like in Amazon[2] or Azure[3] offerings). Highly protected VMs arguably give more trustworthy conditions for the execution of the switch employed by the container manager, ensuring correct packet forwarding among the containers and the external network (e.g., without being eavesdropped or tampered with by malicious co-located containers).

- Third, the switches can also be configured with various defenses to protect the message traffic. In particular, it is possible to setup tunnels between switches implementing alternative security measures. For instance, if confidentiality is not a concern, then it is possible to add message authentication codes (MAC) to packets to afford integrity but without paying the full performance cost of encryption. Further countermeasures could also be added, such as denial of service detection and deep packet inspection to selected flows. In some cases, if trusted hardware is accessible (such as Intel CPUs with SGX extensions in the private cloud), one could leverage from it to enforce greater isolation while performing the cryptographic operations, guaranteeing that keys would never be exposed.

The reader should notice that the above discussion would also apply to other deployment scenarios. For example, the virtualization operator could offer VNs of virtual machines in distinct cloud providers, which would mean that the relevant network appliances would be the switch utilized by the virtual machine manager and the corresponding links interconnecting them.

Substrate Network Modeling. Given the envisioned scenarios, the substrate network is modeled as a weighted undirected graph, composed of a set of nodes N^S (e.g., switches/routers) and edges E^S connecting them, $G^S = (N^S, E^S, A^S_N, A^S_E)$. Both the nodes and edges have attributes that reflect their particular characteristics. The current collection of attributes resulted from conversations with several companies from the healthcare and energy sectors that are moving their critical services to the cloud, and they represent a balance among three goals: they should be (i) expressive enough to represent the main security requirements when deploying virtual networks; (ii) easy to specify when configuring a network, requiring a limited number of options; (iii) implementable with readily available technologies.
The following attributes are considered for substrate nodes:

$$A^S_N = \{\{cpu^S(n),\ sec^S(n),\ cloud^S(n)\} \mid n \in N^S\}$$

The total amount of CPU that can be allocated for the switching operations of node n is given by $cpu^S(n) > 0$. Depending on the underlying machine capacity and the division of CPU cycles among the various tasks (e.g., tenant jobs, storage, network), $cpu^S(n)$ can take a greater or smaller value. The security level associated to the node is $sec^S(n) \geq 0$. Nodes that run in an environment that implements stronger protections will have a greater value for $sec^S(n)$. The trustworthiness degree associated with a cloud provider is indicated with $cloud^S(n) \geq 0$.
The substrate edges have the following attributes:

$$A^S_E = \{\{bw^S(l),\ sec^S(l)\} \mid l \in E^S\}$$

[2]For example: Trend Micro Deep Security at aws.amazon.com/marketplace.
[3]For example: Check Point vSEC at azuremarketplace.microsoft.com.

The first attribute, $bw^S(l) \geq 0$, corresponds to the total amount of bandwidth capacity of the substrate link l. The security measures enforced by the link are reflected in $sec^S(l) \geq 0$. If the link implements tunnels that ensure integrity and confidentiality (by resorting to MACs and encryption) then it will have a higher $sec^S(l)$ than a default edge that simply forwards packets.

Virtual Network Modeling. VNs have an arbitrary topology and are composed of a number of nodes and the edges that connect them. When a tenant wants to instantiate a VN, besides indicating the nodes' required processing capacity and bandwidth for the links, she/he may also include as requirements security demands. These demands are defined by specifying security attributes values associated with the resources.
In terms of modeling, a VN is also modeled as a weighted undirected graph, $G^V = (N^V,\ E^V,\ A^V_N,\ A^V_E)$, composed by a set of nodes N^V and edges (or links) E^V. Both the nodes and edges have attributes that portray characteristics that need to be fulfilled when embedding is performed. Both A^V_N and A^V_E mimic the attributes presented for the substrate network. The only exception is an extra attribute that allows for the specification of security requirements related to availability.
The attribute $avail^V(n)$ indicates that a particular node should have a backup replica to be used as a cold spare. This causes the embedding to allocate an additional node and the necessary links to connect it to the other nodes. These resources will only be used in case the virtualization platform detects a failures in the primary (or working) node / links. $avail^V(n)$ defines where the backup of virtual node n should be mapped. Typically, it would take value 0 if no backup is necessary. If virtual node n should have the backup placed in the same cloud then $avail^V(n) = 1$. If n should have a backup in another cloud (e.g., to survive cloud outages), then $avail^V(n) = 2$.

Virtual Network Request. VNRs are defined by the tenants of the system. They are modeled as a VN with two additional parameters, $VNR^V = (N^V,\ E^V,\ A^V_N,\ A^V_E,\ Time^V,\ Dur^V)$, where $Time^V$ is the arrival time of the VNR and Dur^V is the interval of time during which the VN is valid.

3.1.2 Secure Virtual Network Embedding Problem

Our approach to VNE enables the specification of VNs to be mapped over a multi-cloud substrate, enhancing the security and flexibility of network virtualization. More precisely, we can define the Secure Virtual Network Embedding (SecVNE) problem as follows:

SecVNE problem: *Given a virtual network request with resources and security demands, G^V, and a substrate network G^S with the resources to serve incoming VNRs, can G^V be mapped to G^S ensuring an efficient use of resources while satisfying the following constraints? (i) Each virtual edge is mapped to the substrate network meeting the bandwidth and security constraints; (ii) Each virtual node is mapped to the substrate network meeting the CPU capacity and security constraints (including node availability and cloud trust domain requirements).*

Our approach handles the SecVNE problem, mapping a VN onto a substrate network respecting all constraints. When a VNR arrives, the optimal embedding is searched for to decrease the costs, *i.e.*, reduce the total quantity of substrate resources allocated to it. It can happen that there are not enough substrate resources available at a certain instant, and in that case the incoming request has to be rejected. In order to increase the acceptance rate, we will allocate resources that are at least as secure as the ones specified in the VNR. This means that a VN might end up being mapped onto substrate resources deemed "more secure" than what is required. We find this option an acceptable trade-off as the alternative would cause resources to be under utilized if for instance during a period tenants only had VNRs with weak security demands. If it is possible to solve SecVNE for a request, then the asked resources will be consumed from the substrate for the period Dur^V defined in the VNR.
Figure 3.2 illustrates the result of embedding a VNR (displayed on top) onto the substrate of Figure 3.1 (represented at the bottom). The VNR has node 1 that requires a medium level of security ($sec^V(1) =$

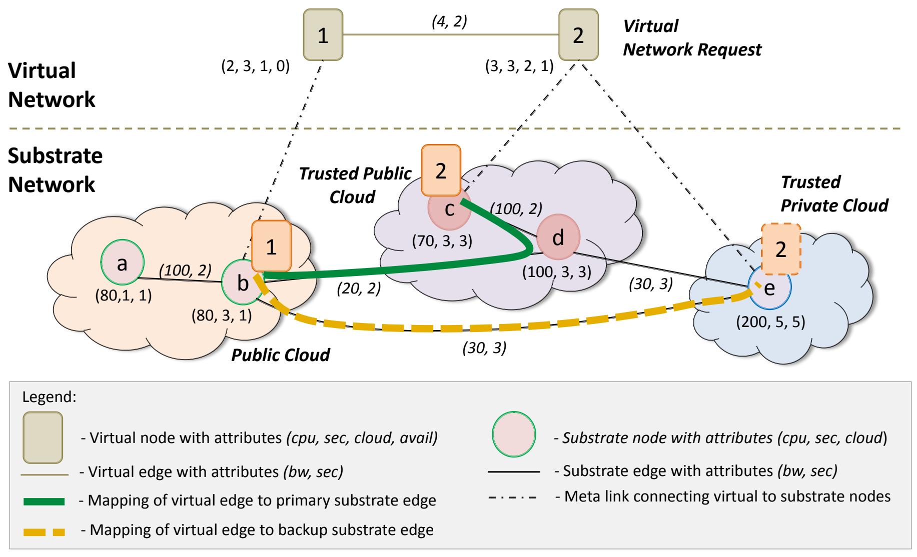

1
(4, 2)
2
Virtual Network Request
(2, 3, 1, 0)
(3, 3, 2, 1)
Virtual Network
Substrate Network
Trusted Public Cloud
2
c
(100, 2)
d
(70, 3, 3)
(100, 3, 3)
Trusted Private Cloud
2
e
(200, 5, 5)
(30, 3)
1
a
(100, 2)
b
(80,1, 1)
(80, 3, 1)
(20, 2)
Public Cloud
(30, 3)
Legend:
- Virtual node with attributes (cpu, sec, cloud, avail)
- Virtual edge with attributes (bw, sec)
- Mapping of virtual edge to primary substrate edge
- Mapping of virtual edge to backup substrate edge
- Substrate node with attributes (cpu, sec, cloud)
- Substrate edge with attributes (bw, sec)
- Meta link connecting virtual to substrate nodes

Figure 3.2: Example of the embedding of a virtual network request (top) onto a multi-cloud substrate network (bottom). The figure also illustrates the various constraints and the resulting mapping after the execution of our MILP formulation.

3) on a default trust cloud ($cloud^V(1) = 1$). The other node needs to be replicated ($avail^V(2) = 1$), in such a way that the primary and backup are in different clouds. It has a similar security demand ($sec^V(2) = 3$) but asks for more trusted clouds ($cloud^V(2) = 2$).
The chosen embedding guarantees that all requirements are satisfied. Node 1 is mapped on the left public cloud to a substrate node with a security level equal to the one requested ($sec^S(b) = 3$ and $cloud^S(b) = 1$). The other virtual node is embedded on more trustworthy clouds (with respectively $cloud^S(c) = 3$ and $cloud^S(e) = 5$). It is also possible to observe that one of the substrate paths (the primary/working) corresponds to more than one substrate edge (*e.g.*, edge (b, d) plus (d, c)), but all with the necessary security level (2 in this case). The figure also displays meta-links that connect the virtual nodes to the substrate nodes where they are mapped (e.g., line between 1 and b). This is an artifact in our modeling that is going to be explored in the MILP formulation.

3.1.3 A Policy Language to Specify SecVNE

Currently, we support two alternative ways for a user to indicate the information necessary to solve the SecVNE problem, namely give a description of the substrate network and the VNRs. The first is based on a *graphical interface* where the user can draw arbitrary substrates, with nodes and links and the associated attributes. The tenants can then depict the VNRs, which are then embedded into the substrate by our solution.
The other approach is based on a *policy language* that lets the user describe both the substrate and VNRs in a computer friendly manner, allowing scripts and tools to process them. The production rules of the grammar were kept relatively simple, but the achieved level of expressiveness is greater than what is attained with the graphical interface. As the characteristics of the substrate and VNRs are distinct, we explain them separately.
The substrate part of SecVNE policy grammar (top rows of Table 3.1) enables the listing of resources that compose the substrate. There are only functions and values to represent the current status of the network. For example, the leftmost cloud of the Figure 3.2 is specified as:

$substrate \rightarrow cpu^S(a) = 80 \ \& \ sec^S(a) = 1 \ \& \ cloud^S(a) = 1 \ \&$
$cpu^S(b) = 80 \ \& \ sec^S(b) = 3 \ \& \ cloud^S(b) = 1 \ \&$
$bw^S(a, b) = 100 \ \& \ sec^S(a, b) = 2 \ \& \ ...$

Substrate Specification
$S \rightarrow func^S(parameter) = value_{num}$
$S \rightarrow S \ \& \ S$
Virtual Network Specification
$V \rightarrow func^V(parameter) = value_{num}$
$V \rightarrow func^V(parameter) \geq value_{num}$
$V \rightarrow !V; \ (V); \ V \ \& \ V; \ V \mid V$

Table 3.1: Policy grammar to define SecVNE parameters.

In the virtual part of SecVNE policy grammar (bottom rows of Table 3.1), the relations dictate the requirements for each node and link of the VNR. As the grammar supports boolean operations, such as *or* ("|"), *and* ("&"), and *not* ("!"), it is possible to express alternative constraints for the resources. When processing a VNR containing a VN with several optional demands, we generate all possible requests that would satisfy the tenant. Then, we evaluate each one and select the solution with lowest cost. There are two main benefits of this approach: (i) the acceptance rate grows because a request may be mapped in more ways; (ii) the tenants can explore different trade offs with respect to security (e.g., replication is only necessary if clouds are not highly trusted).
As an example, consider the following VN with two nodes and one edge:

$VN \rightarrow (CPU^V(1) = 2 \ \& \ sec^V(1) = 3 \ \& \ cloud^V(1) \geqslant 1 \ \&$
$avail^V(1) = 0 \ ) \ \& \ (CPU^V(2) = 3 \ \& \ avail^V(2) = 1 \ \&$
$((sec^V(2) \geqslant 1 \ \& \ cloud^V(2) \geqslant 4) \mid (sec^V(2) \geqslant 4 \ \& \ cloud^V(2) \geqslant 1))) \ \&$
$bw^V(1,2) = 4 \ \& \ sec^V(1,2) = 2$

Node 1 needs to have security degree o 3 but the cloud trustworthiness can be 1 or more. For node 2 there is a compromise between node security and the degree of cloud trust, establishing two acceptable options where either of the attributes needs to have a higher value ($sec^V(2)$ or $cloud^V(2)$ should be larger or equal to 4). In this case, the VNR would be converted into two requests, the first with the constraint ($sec^V(2) \geqslant 1 \ \& \ cloud^V(2) \geqslant 4$) (plus others attributes) and the other with ($sec^V(2) \geqslant 4 \ \& \ cloud^V(2) \geqslant 1$) (plus other attributes).

3.1.4 MILP formulation

We have developed a MILP formulation to solve the SecVNE problem. The section starts by explaining the decision variables used in the formulation, the objective function, and finally the constraints required to model the problem.

3.1.4.1 Decision variables

Table 3.2 presents the variables that are used in our MILP formulation. Briefly, $wf^{i,j}_{p,q}$, $bf^{i,j}_{p,q}$, $wl^{i,j}_{p,q}$ and $bl^{i,j}_{p,q}$ are related to working and backup links; $wn_{i,p}$ and $bn_{i,p}$ are associated with the working and backup nodes; $wc_{i,c}$ and $bc_{i,c}$ are related to the location of virtual node embedding in clouds.
The formulation also employs a few auxiliary sets whose value depends on the VNR, as shown in Table 3.3. For example, $\overline{N}^V = \emptyset$ means that no virtual node requires a backup. When this happens, we only model a working network in the substrate, making every backup related decision variable ($bf^{i,j}_{p,q}$, $bl^{i,j}_{p,q}$, $bn_{i,p}$, $bc_{i,c}$) become 0. On the other hand, if $\overline{N}^V \neq \emptyset$, then we model both a working and a backup network in the substrate. In this case, the decision variables take different values depending on the tenant request. If the virtual node i has $avail^V(i) = 0$, indicating that there is no need to

Symbol	Meaning
$wf_{p,q}^{i,j} \geqslant 0$	The amount of working flow, i.e., bandwidth, on the physical link (p,q) for the virtual link (i,j)
$bf_{p,q}^{i,j} \geqslant 0$	The amount of backup flow, i.e., backup bandwidth, on the physical link (p,q) for the virtual link (i,j)
$wl_{p,q}^{i,j} \in \{0,1\}$	Denotes whether the virtual link (i,j) is mapped onto the physical link (p,q). (1 if (i,j) is mapped on (p,q), 0 otherwise)
$bl_{p,q}^{i,j} \in \{0,1\}$	Denotes whether the backup of virtual link (i,j) is mapped onto the physical link (p,q). (1 if backup of (i,j) is mapped on (p,q), 0 otherwise)
$wn_{i,p} \in \{0,1\}$	Denotes whether virtual node i is mapped onto the physical node p. (1 if i is mapped on p, 0 otherwise)
$bn_{i,p} \in \{0,1\}$	Denotes whether virtual node i's backup is mapped onto the physical node p. (1 if i's backup is mapped on p, 0 otherwise)
$wc_{i,c} \in \{0,1\}$	Denotes whether virtual node i is mapped on cloud c. (1 if i is mapped on c, 0 otherwise)
$bc_{i,c} \in \{0,1\}$	Denotes whether virtual node i's backup is mapped on cloud c. (1 if i's backup is mapped on c, 0 otherwise)

Table 3.2: Domain constraints (decision variables) used in the MILP formulation.

$$\begin{aligned}
\mathring{N}^V &= \{\ i \in N^V \ :\ avail^V(i) = 0\ \} \\
\overline{N}^V &= N^V \setminus \mathring{N}^V \\
\mathring{E}^V &= \{\ (i,j) \in E^V \ :\ avail^V(i) = 0 \text{ and } avail^V(j) = 0\ \} \\
\overline{E}^V &= E^V \setminus \mathring{E}^V
\end{aligned}$$

Table 3.3: Auxiliary sets to facilitate the description of the constraints.

replicate, then both the working and the backup nodes of i are placed in the same substrate node p (i.e., $wn_{i,p} = bn_{i,p} = 1$), but the backup does not consume resources (e.g., CPU). When a virtual node j has $avail^V(j) > 0$, it is necessary to locate the working and backup in different substrate nodes, belonging eventually to distinct clouds. Here, the backup will reserve the resources to be able to substitute the primary in case of failure.

3.1.4.2 Objective Function

The objective function wants to minimize three aspects (see Eq. 3.1): 1) the sum of all computing costs, 2) the sum of all communication costs, and 3) the overall number of hops of the substrate paths for the virtual links. Since these objectives are measured in different units, we resort to a composite function, which can be parametrized and used to compute different solutions (others approaches could be used, see Steuer [149]). Thus, the formulation is based on a weighted-sum function with three different coefficients, β_1, β_2, and β_3, which should be reasonably parameterized for each objective.

$$
\begin{aligned}
min \quad & \beta_1 \Big[\sum_{i \in N^V} \sum_{p \in N^S} cpu^V(i) \quad sec^S(p) \quad cloud^S(p) \quad wn_{i,p} \\
& + \sum_{i \in \overline{N}^V} \sum_{p \in N^S} cpu^V(i) \quad sec^S(p) \quad cloud^S(p) \quad bn_{i,p} \Big] \\
& + \beta_2 \Big[\sum_{(i,j) \in E^V} \sum_{(p,q) \in E^S} \alpha_{p,q} \quad sec^S(p,q) \quad wf^{i,j}_{p,q} \\
& + \sum_{(i,j) \in \overline{E}^V} \sum_{(p,q) \in E^S} \alpha_{p,q} \quad sec^S(p,q) \quad bf^{i,j}_{p,q} \Big] \\
& + \beta_3 \Big[\sum_{(i,j) \in E^V} \sum_{(p,q) \in E^S} wl^{i,j}_{p,q} + \sum_{(i,j) \in \overline{E}^V} \sum_{(p,q) \in E^S} bl^{i,j}_{p,q} \Big]
\end{aligned}
\tag{3.1}
$$

The first part of Eq. 3.1 covers the computing costs, including both the working and backup nodes (top 2 lines). The second part is the sum of all working and backup link bandwidth costs (lines 3-4). The last part of the objective function achieves the third goal presented above. The equation considers the level of security of the substrate resources, where the selection of higher security incurs in increased costs. Likewise, the costs are proportional to the trust associated with the cloud where the resource is located. To address the possibility that substrate edges connecting two distinct clouds might have a different cost (monetary, delay, or other) than links inside a cloud, we have added a multiplicative parameter $\alpha_{p,q}$. This parameter is a weight for each physical link that may assume a different value depending on whether (p,q) is a inter-cloud edge (connection between two clouds) or an intra-domain link (connection inside a cloud).
Intuitively, this objective function attempts to economize the most "powerful" resources (e.g., those with higher security levels) for VNRs that explicitly require them. Therefore, for instance, virtual nodes with $sec^V = 1$ will be mapped onto substrate nodes with $sec^S = 2$ if and only if there are no other substrate nodes with $sec^S = 1$ available.

3.1.4.3 Security Constraints

Below are enumerated the constraints related to the security of nodes, edges, and clouds:

$$wn_{i,p} \quad sec^V(i) \leqslant sec^S(p), \ \forall i \in N^V, \ p \in N^S \tag{3.2}$$
$$bn_{i,p} \quad sec^V(i) \leqslant sec^S(p), \ \forall i \in N^V, \ p \in N^S \tag{3.3}$$
$$wl^{i,j}_{p,q} \quad sec^V(i,j) \leqslant sec^S(p,q), \ \forall (i,j) \in E^V, \ (p,q) \in E^S \tag{3.4}$$
$$bl^{i,j}_{p,q} \quad sec^V(i,j) \leqslant sec^S(p,q), \ \forall (i,j) \in E^V, \ (p,q) \in E^S \tag{3.5}$$
$$wn_{i,p} \quad cloud^V(i) \leqslant cloud^S(p), \ \forall i \in N^V, \ p \in N^S \tag{3.6}$$
$$bn_{i,p} \quad cloud^V(i) \leqslant cloud^S(p), \ \forall i \in N^V, \ p \in N^S \tag{3.7}$$

These constraints guarantee that a virtual node is only mapped to a substrate node that has a security level equal or greater than its demand (Eq. 3.2). They also ensure the same for backup nodes (Eq. 3.3). The following two equations force each virtual edge to be mapped to (one or more) physical links that provide a larger or equivalent security level as the request. This is true for links connecting the primary nodes and the backups. The last constraints ensure that a virtual node i is mapped to a substrate node p only if the cloud where p is located has a trust level equal or greater than the cloud demanded by i (both for working and backups — Eq. 3.6 and 3.7).

3.1.4.4 Mapping Constraints

Node Embedding: We force each virtual node to be mapped to exactly one working substrate node, and if requested, to a single backup substrate node (Eq. 3.8 - 3.9). We also have to guarantee that *(i)* a substrate node only receives at most a virtual node (Eq. 3.10 - 3.12); *(ii)* however, as explained in Section 3.1.4.1, if the virtual node requires no replicas then its working and backup must be mapped onto the same substrate node (Eq. 3.13 - 3.14).

$$\sum_{p \in N^S} wn_{i,p} = 1, \ \forall i \in N^V \tag{3.8}$$

$$\sum_{p \in N^S} bn_{i,p} = 1, \ \forall i \in \overline{N}^V \tag{3.9}$$

$$\sum_{i \in N^V} wn_{i,p} \leqslant 1, \ \forall p \in N^S \tag{3.10}$$

$$\sum_{i \in N^V} wn_{i,p} + bn_{j,p} \leqslant 1, \ \forall j \in \overline{N}^V, \ p \in N^S \tag{3.11}$$

$$\sum_{i \in N^V \setminus \{j\}} bn_{i,p} + bn_{j,p} \leqslant 1, \ \forall j \in N^V, \ p \in N^S \tag{3.12}$$

$$bn_{i,p} \leqslant wn_{i,p}, \ \forall i \in \mathring{N}^V, \ p \in N^S \tag{3.13}$$

$$\sum_{i \in N^V} wn_{i,p} + bn_{j,p} \leqslant 2, \ \forall j \in \mathring{N}^V, \ p \in N^S \tag{3.14}$$

The next constraints create relationships among the nodes and flows.

$$wl^{i,j}_{p,q} \ \ bw^V(i,j) \geqslant wf^{i,j}_{p,q}, \ \forall (i,j) \in E^V, \ (p,q) \in E^S \tag{3.15}$$

$$bl^{i,j}_{p,q} \ \ bw^V(i,j) \geqslant bf^{i,j}_{p,q}, \ \forall (i,j) \in E^V, \ (p,q) \in E^S \tag{3.16}$$

$$wl^{i,j}_{p,q} = wl^{i,j}_{q,p}, \ \forall (i,j) \in E^V, \ p,q \in N^S \cup N^V \tag{3.17}$$

$$bl^{i,j}_{p,q} = bl^{i,j}_{q,p}, \ \forall (i,j) \in E^V, \ p,q \in N^S \cup N^V \tag{3.18}$$

$$\sum_{p \in N^S} (wn_{i,p} \ \ doesItBelong_{p,c}) \geqslant wc_{i,c}, \ \forall i \in N^V, \ c \in C \tag{3.19}$$

$$\sum_{p \in N_S} (bn_{i,p} \ \ doesItBelong_{p,c}) \geqslant bc_{i,c}, \ \forall i \in N^V, \ c \in C \tag{3.20}$$

Eq. 3.15 ensures that if there is a flow between nodes p and q for a virtual edge (i,j), then this means that (i,j) is mapped to the substrate link whose end-points are p and q. For example, if $wf^{i,j}_{p,q} \neq 0$ then $wl^{i,j}_{p,q} = 1$. The next equation achieves the same goal but for the backup. We also include two binary constraints to force the symmetric property for the binary variables related with links (Eq. 3.17 - 3.18). In a similar fashion, we also need to establish a relation between the virtual nodes and the clouds where they are embedded (both for working and backups) (Eq. 3.19 - 3.20). Namely, if virtual node i is mapped onto a substrate node p and p belongs to cloud c, then i is mapped on cloud c. Parameter $doesItBelong_{p,c}$ is 1 if substrate node p belongs to cloud c, and 0 otherwise.
Since we allow the tenant to choose between having no replication, or replication in one cloud or across different clouds, it is necessary to specify these constraints. First, we require each virtual node to be mapped to exactly one cloud (working or backup, Eq. 3.21 - 3.22). Parameter *wantBackup* assumes value 1 if a backup is needed for at least one of the nodes of a VNR or value 0 otherwise. Second, we must restrict the location of the working and backup nodes to the same or distinct clouds, depending

on the value of the availability attribute ($avail^V(i)$) (Eq. 3.23[4]).

$$\sum_{c \in C} wc_{i,c} = 1, \ \forall i \in N^V \tag{3.21}$$

$$\sum_{c \in C} bc_{i,c} = wantBackup, \ \forall i \in N^V \tag{3.22}$$

$$|wc_{i,c} \ \ wantBackup \ - \ bc_{i,c}| = (avail^V(i) - 1) \times$$

$$(wc_{i,c} \ \ wantBackup + bc_{i,c}), \ \forall i \in N^V, \ c \in C \tag{3.23}$$

Link Embedding: These constraints are related to the mapping of virtual links into the substrate. They take advantage of the meta link artifact (recall Figure 3.2), which connects a virtual node i to the substrate node p where it is mapped, to enforce a few restrictions.

$$wn_{i,p} \quad bw^V(i,j) = wf^{i,j}_{i,p}, \ \forall \ (i,j) \in E^V, \ p \in N^S \tag{3.24}$$

$$wn_{j,q} \quad bw^V(i,j) = wf^{i,j}_{q,j}, \ \forall (i,j) \in E^V, \ p \in N^S \tag{3.25}$$

$$bn_{i,p} \ \ bw^V(i,j) = bf^{i,j}_{i,p} \ \ wantBackup, \ \forall (i,j) \in E^V, \ p \in N^S \tag{3.26}$$

$$bn_{j,q} \ \ bw^V(i,j) = bf^{i,j}_{q,j} \ \ wantBackup, \ \forall (i,j) \in E^V, \ q \in N^S \tag{3.27}$$

$$\sum_{j,k!=i \ \ j,k \in N^V} wf^{j,k}_{i,p} + wf^{j,k}_{p,i} + bf^{j,k}_{i,p} + bf^{j,k}_{p,i} = 0, \ \forall i \in N^V, \ p \in N^S \tag{3.28}$$

These constraints guarantee that the working flow of a virtual link (i, j) always departs from i and arrives to j, passing through the corresponding substrate nodes (p and q) (Eq. 3.24 and 3.25). The next two equations compel the same requirement for the backup nodes. Notice that even though the backup path is only used if the working substrate path fails, we reserve the necessary resources during embedding to make sure they are available when needed. Eq. 3.28 forces meta-links to carry only working or backup traffic to their correspondent virtual nodes. This means that, if a virtual node 1 needs to send information to virtual node 2, the data does not need to pass through the meta-links of a virtual node 3.
The next equations specify flow conservation restrictions at the nodes.

$$\sum_{p \in N^S} wf^{i,j}_{i,p} \ - \sum_{p \in N^S} wf^{i,j}_{p,i} = bw^V(i,j), \ \forall (i,j) \in E^V \tag{3.29}$$

$$\sum_{p \in N^S} wf^{i,j}_{j,p} \ - \sum_{p \in N^S} wf^{i,j}_{p,j} = -bw^V(i,j), \ \forall (i,j) \in E^V \tag{3.30}$$

$$\sum_{p \in N^S \cup N^V} wf^{i,j}_{q,p} \ - \sum_{p \in N^S \cup N^V} wf^{i,j}_{p,q} = 0, \ \forall (i,j) \in E^V, \ q \in N^S \tag{3.31}$$

$$\sum_{p \in N^S} bf^{i,j}_{i,p} \ - \sum_{p \in N^S} bf^{i,j}_{p,i} = bw^V(i,j) \ \ wantBackup, \ \forall (i,j) \in E^V \tag{3.32}$$

$$\sum_{q \in N^S} bf^{i,j}_{j,q} \ - \sum_{q \in N^S} bf^{i,j}_{q,j} = -bw^V(i,j) \ \ wantBackup, \ \forall (i,j) \in E^V \tag{3.33}$$

$$\sum_{p \in N^S \cup N^V} bf^{i,j}_{q,p} \ - \sum_{p \in N^S \cup N^V} bf^{i,j}_{p,q} = 0, \ \forall (i,j) \in E^V, \ q \in N^S \tag{3.34}$$

Eq. 3.29, 3.30 and 3.31 refer to the working flow conservation conditions, which denote that the network flow to a node is zero, except for the source and the sink nodes, respectively. In an analogous way, the following three equations refer to the backup flow conservation conditions (Eq. 3.32, 3.33 and 3.34). The next constraints guarantee the same bandwidth in both directions. The first two equations

[4]Notice that in the implementation, the *modulus* function had to be linearized because it is not allowed with variables as parameters.

for working and backups separately (Eq. 3.35 - 3.36) and the others in case there are relations between them (Eq. 3.37 - 3.38).

$$wf_{p,q}^{i,j} = wf_{q,p}^{j,i},\ \forall (i,j) \in E^V,\ p,q \in N^S \cup N^V \tag{3.35}$$

$$bf_{p,q}^{i,j} = bf_{q,p}^{j,i},\ \forall (i,j) \in E^V,\ p,q \in N^S \cup N^V \tag{3.36}$$

$$wf_{p,q}^{i,j} = bf_{p,q}^{i,j},\ \forall (i,j) \in \mathring{E}^V,\ p,q \in N^S \tag{3.37}$$

$$wf_{p,q}^{j,i} = bf_{p,q}^{j,i},\ \forall (i,j) \in \mathring{E}^V,\ p,q \in N^S \tag{3.38}$$

Nodes and Links Disjointness: Since any substrate node or link of a working path can fail, we have to ensure that paths connecting the backups of the virtual nodes are disjoint from the substrate resources that are being used for the working part (otherwise a single failure could compromise both paths). The auxiliary binary variables $working_{p,q}$ and $backup_{p,q}$ define if a physical link (p,q) belongs to the working or backup networks in the substrate.

$$working_{p,q} \leqslant 1 - backup_{p,q},\ \forall (p,q) \in E^S \tag{3.39}$$

$$wl_{p,q}^{i,j} \leqslant working_{p,q},\ \forall (i,j) \in E^V,\ (p,q) \in E^S \tag{3.40}$$

$$bl_{p,q}^{i,j} \leqslant backup_{p,q},\ \forall (i,j) \in \overline{E}^V,\ (p,q) \in E^S \tag{3.41}$$

First, we require disjointness between the working and backup parts (Eq. 3.39). Second, we guarantee that if the working path of a virtual edge (i,j) is mapped onto a substrate link (p,q), then (p,q) needs to be in the working part. Similarly, we constraint the backups (Eq. 3.40 - 3.41).

3.1.4.5 Capacity Constraints

Node Capacity Constraints: Virtual nodes from different VNRs can be mapped to the same substrate node. For instance, a substrate node can receive both a working node of a virtual node i from a VNR x and a backup node of a virtual node j from a VNR y. Lets call $\mathbb{N}^V$ the set of all virtual nodes belonging to every VNR that is at this moment mapped onto the substrate and $i \uparrow p$ to indicate that virtual node i is hosted on the substrate node p. Then, the residual capacity of a substrate node, $R_N(p)$, is defined as the currently available CPU capacity of the substrate node $p \in N^S$.

$$R_N(p) = cpu^S(p) - \sum_{\forall i \uparrow p} cpu^V(i),\ i \in \mathbb{N}^V$$

For a substrate node, it is necessary to ensure that we never allocate more than the residual capacity when carrying out a new embedding. This needs to take into consideration both the resources consumed by the working and backups (Eq. 3.42).

$$\sum_{i \in N^V} wn_{i,p}\ \ cpu^V(i)\ + \sum_{j \in \overline{N}^V} bn_{j,p}\ \ cpu^V(j) \leqslant R_N(p),\ \forall p \in N^S \tag{3.42}$$

Link Capacity Constraints: Similarly, substrate links can also map virtual edges from different VNRs. Lets define $\mathbb{E}^V$ as the set of all virtual edges of every VNR currently mapped onto the substrate and $(i,j) \uparrow (p,q)$ denote that the flow of the virtual link (i,j) traverses the substrate link (p,q). The residual capacity of a substrate link, $R_E(p,q)$, is defined as the total amount of bandwidth available on the substrate link $(p,q) \in E^S$.

$$R_E(p,q) = bw^S(p,q) - \sum_{\forall (i,j) \uparrow (p,q)} bw^V(i,j),\ (i,j) \in \mathbb{E}^V$$

The following constraint ensures that the allocated capacity of a substrate link should be less than the residual capacity of that physical link, taking into consideration both the working and backup parts.

$$\sum_{(i,j) \in E^V} wf_{p,q}^{i,j} + \sum_{(i,j) \in \overline{E}^V} bf_{p,q}^{i,j} \leqslant R_E(p,q),\ \forall (p,q) \in E^S \tag{3.43}$$

Notation	Algorithm description
NS+NA	SecVNE with no security or availability requirements for VNs
10S+NA	SecVNE with VNRs having 10% of their resources (nodes and links) with security requirements (excluding availability)
20S+NA	Similar to *10S+NA*, but with security requirements (excluding availability) for 20% of the resources
NS+10A	SecVNE with no security requirements, but with 10% of the nodes requesting replication for increased availability
NS+20A	Similar to *NS+10A*, but with 20% of the nodes asking for replication
20S+20A	SecVNE with 20% of the resources (nodes and links) with security requirements and 20% of the nodes with replication
D-ViNE	VNE MILP model presented in [96]

Table 3.4: VNR configurations that were evaluated in the experiments.

3.1.5 Evaluation

This section presents performance results of our solution in random and Waxman network topologies [5] and in diverse VNR settings. The simulations show promising results as our solution was able to show high acceptance rates and substrate resource utilizations across the various experiments.

3.1.5.1 Experimental Setup

We have extended a simulator [1] to evaluate the embedding when processing the dynamic arrival of VNRs to a system. To create the substrate networks we resorted to the GT-ITM tool [170]. Two kinds of networks were utilized: one based on random topologies, where every pair of nodes is randomly connected with a probability between 25% and 30%; and the other employing the Waxman model to link the nodes with a probability 50% [105].
Substrate networks have a total of 25 nodes. CPU and bandwidth (cpu^S and bw^S) of nodes and links is uniformly distributed between 50 and 100. These resources are also uniformly associated with one of three levels of security ($sec^S \in \{1.0, 1.2, 5.0\}$). The rationale for these values is to achieve a good balance between the diversity of security levels and their monetary cost. We performed an analysis of the pricing schemes of Amazon EC2 and Microsoft Azure for plain and secure VMs. It was possible to observe a wide range of values depending on the included defenses. For example, while an EC2 instance that has container protection is around 20% more expensive than a normal instance (hence our choice of 1.2 for the intermediate level of security), the cost of instances that offer threat prevention or encryption is at least 5 times greater (our choice for the highest level of security).
The substrate nodes are also uniformly divided among three clouds, each one with a different security level ($cloud^S \in \{1.0, 1.2, 5.0\}$), which are justified along the same line of reasoning. The goal is to represent a setup that includes a public cloud (lowest level), a trusted public cloud, and a private datacenter (assumed to offer the highest security).
VNRs have a number of virtual nodes uniformly distributed between 2 and 4[6]. Pairs of virtual nodes are connected with a Waxman topology with probability 50%. The CPU and bandwidth of the virtual nodes and links are uniformly distributed between 10 and 20. Several alternative security and availability requirements are evaluated, as shown in Table 3.4. We assume that VNRs arrivals ($Time^V$) are modeled as a Poisson process with an average rate of 4 VNRs per 100 time units. Each VNR has an exponentially distributed lifetime (Dur^V) with an average of 1000 time units.

[5]A Waxman topology is a variant of the classical Erdos-Renyi random graph that includes network-specific characteristics, such as placing the nodes on a plane and using a probability function to interconnect two nodes which is parametrized by their distance.

[6]Notice that a node corresponds to a switch, which can connect many hundreds of containers in a large VM (recall Figure 3.1). Therefore, a VNR with 4 nodes can easily link together in the order of a thousand of containers.

The MILPs are solved using the open source library GLPK [63]. In the objective function, we set $\beta_1 = \beta_2 = \beta_3 = 1$ to balance evenly the cost components (Eq. 3.1). Parameter α was also set to 1 because our pricing analysis showed negligible differences in cost between intra- and inter-cloud links in most of the relevant scenarios. We setup 20 experiments, each with a different substrate topology (10 random and 10 Waxman). Every experiment ran for 50 000 time units, during which embedding is attempted for a group VNRs (10 sets of 2 000 VNRs were tested). The order of arrival and the capacity requirements of each VNR are kept the same in each of the configurations of Table 3.4, ensuring that they solve equivalent problems.
In the evaluation, we compared our approach with the algorithm D-ViNE [96]. D-ViNE was chosen because it has been considered as the baseline for many VNE works and due to the availability of its implementation as open-source software. D-ViNE requirements are only based on CPU and bandwidth capacities, while our algorithm adds to these requirements also security demands, including availability needs, and cloud preferences.

3.1.5.2 Metrics

We used several performance metrics for the evaluation:
– *VNR acceptance ratio:* the percentage of accepted requests (i.e., the number of accepted VNRs divided by the total number of VNRs);
– *Node stress ratio:* average load on the substrate nodes (i.e., average over all nodes of the percentage of CPU that is in use);
– *Link stress ratio:* average load on the substrate links (i.e., average over all edges of the percentage of bandwidth that is in use);
– *Average revenue by accepting VNRs:* One of the main goals of VNE is to maximize the profit of the virtualization provider. For this purpose, and similar to [96, 169], the revenue generated by accepting a VNR is proportional to the value of the acquired resources. As such, in our case, we take into consideration that stronger security defenses will be charged at a higher (monetary) value. Therefore, the revenue associated with a VNR is:

$$\begin{aligned}\mathbb{R}(\text{VNR}) = \lambda_1 \sum_{i \in N^V} [1 + \varphi_1(i)]\ cpu^V(i)\ \ sec^V(i)\ \ cloud^V(i)\ + \\ \lambda_2 \sum_{(i,j) \in E^V} [1 + \varphi_2(i,j)]\ bw^V(i,j)\ \ sec^V(i,j),\end{aligned}$$

where λ_1 and λ_2 are scaling coefficients that denote the relative proportion of each revenue component to the total revenue. These parameters offer providers the flexibility required to price differently the different resources. Variables φ account for the need to have backups, either in the nodes $\varphi_1(i)$ or in the edges $\varphi_2(i,j)$ ($\varphi_1(i) = 1$ if a backup is required or 0 otherwise; $\varphi_1(i,j) = 1$, in case of at least one node needs a backup or 0 otherwise).
This metric accounts for the average revenue obtained by embedding a VNR (i.e., the total revenue generated by accepting the VNRs divided by the number of accepted VNRs). In the experiments, we set $\lambda_1 = \lambda_2 = 1$.
– *Average cost of accepting a VNR:* The cost of embedding a VNR is proportional to the total sum of substrate resources allocated to that VN. In particular, this cost has to take into consideration that certain virtual edges may end up being embedded in more than one physical link (as in the substrate edge between nodes b, d and c, in Figure 3.2). The cost may also increase if the VNR requires higher security for its virtual nodes and links. Thus, we define the cost of embedding a VNR as:

$$\begin{aligned}\mathbb{C}(\text{VNR}) = \lambda_1 \sum_{i \in N^V} \sum_{p \in N^S} cpu_p^i\ \ sec^S(p)\ \ cloud^S(p)\ + \\ \lambda_2 \sum_{(i,j) \in E^V} \sum_{(p,q) \in E^S} f_{p,q}^{i,j}\ \ sec^S(p,q),\end{aligned}$$

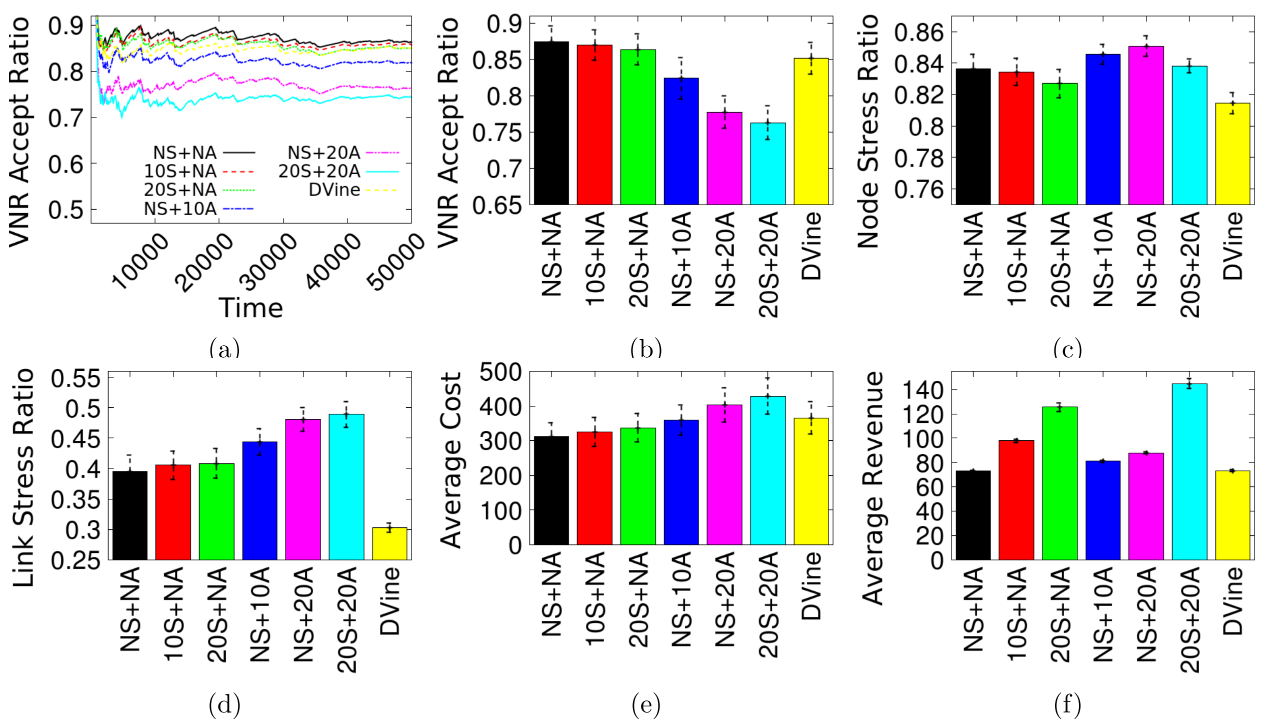


Figure 3.3: Average results and standard deviation (except first graph): (a) VNR acceptance ratio over time; (b) VNR acceptance ratio; (c) Node utilization; (d) Link utilization; (e) Cost for accepting VNRs; (f) Revenue for accepting VNRs.

where cpu_p^i corresponds to the total amount of CPU allocated on the substrate node p for virtual node i (either working or backup). Similarly, $f_{p,q}^{i,j}$ denotes the total amount of bandwidth allocated on the substrate link (p, q) for virtual link (i, j). λ_1 and λ_2 are the same weights introduced in the revenue formula to denote the relative proportion of each cost component to the total cost.

3.1.5.3 Evaluation Results

Figure 3.3a displays the acceptance ratio over time for one particular experiment with a random topology substrate. We can observe that after the first few thousand time units, the acceptance ratio tends to stabilize. A similar trend also occurs with the other experiments, and for this reason the rest of the results are taken at the end of each simulation. Due to space constraints, the results in the following graphs are from the Waxman topologies only (Figure 3.3b - 3.3f). We note however that the conclusions to be drawn are exactly the same as for the random topologies. The main conclusions are:

SecVNE exhibits a higher average acceptance ratio when compared to D-ViNE, not only for the baseline case, but also when including security requirements: Figure 3.3b indicates that SecVNE can make better use of the available substrate resources to embed the arriving VNRs when compared to the most commonly employed VNE algorithm. It is interesting to note that SecVNE is better than D-Vine even when 20% of the VNRs include security requirements, which are harder to fulfill. This does not mean D-Vine is a poor solution – it merely shows that its model is not the best fit for our particular problem. In particular, D-Vine uses geographical distance of substrate nodes as one of the variables to consider in node assignment. This parameter is less relevant in our virtualized environment but constrains D-Vine options. In any case, notice that the results for D-Vine represent its best configuration with respect to geographical location – we have tested D-Vine with the entire range of options for this parameter.

A richer set of demands decreases the acceptance ratio, but only slightly: VNRs with stronger requirements have a greater number of constraints that need to be satisfied, and therefore it becomes more difficult to find the necessary substrate resources to embed them. However, a surprising result is the small penalty in terms of acceptance rate in the presence of security demands (see Figure 3.3b again). For instance, an increase of 20 percentage points (pp) in the resources with security needs results in a penalty of only around 1 pp in the acceptance ratio. Also interesting is the fact that the reduction in acceptance ratio is more pronounced when VNRs have availability requirements, when compared to security. In this case, an increase of 20 pp in the number of nodes with replication results in a penalty of around 10 pp. This is because of the higher use of substrate resources due to the reservation of backup nodes/links.

Security demands only cause a small decrease on substrate resources utilization. Figures 3.3c and 3.3d show the substrate node and link stress ratio, respectively. We observe that the utilization of node resources is very high in all cases (over 80%), meaning the mapping to be effective. It is also possible to see that slightly more resources are allocated in the substrate network with SecVNE than with D-ViNE, which justifies the higher acceptance ratio achieved. If the existing resources are used more extensively to be able to serve more virtual network requests, then the assignment of virtual requests is being more effective. As the link stress ratio is lower (again, for all cases), this means the bottleneck is the node CPU. Finally, the link stress ratio of D-ViNE is lower than in our solution. This is due to D-ViNE incorporating load balancing into the formulation.

Security and availability requirements increases costs and revenues. Figures 3.3e and 3.3f display the average cost and revenue for each VNR embedding, respectively. The results show that reasonable increases in the security requirements (10% and 20%) only cause a slight impact on the costs. However, higher costs are incurred to fulfill availability needs due to the extra reservation of resources (nodes and links). Since D-vine does not consider security and availability aspects, it ends up choosing embeddings that are more expensive (e.g., with respect to “NS+NA”). In terms of revenue, it can be observed that by charging higher prices for security services, virtualization providers can significantly enhance their income — average revenue almost doubles for a 20% increase in security needs.

3.1.6 Related work

There is already a wide literature on this problem [60]. Yu et al. [169] were the first to solve it efficiently, by assuming the capability of path splitting (multi-path) in the substrate network, which enable the computationally harder part of the problem to be solved as a multicommodity flow (MCF), for which efficient algorithms exist. The authors solve the problem considering two independent phases – an approach commonly used by most algorithms. In the first phase, a greedy algorithm is used for virtual node embedding. Then, to map the virtual links, either efficient MCF solutions or k-shortest path algorithms can be used. In [96], Chowdhury et al. proposed two algorithms for VNE that introduce coordination between the node and link mapping phases. The main technique proposed in this work is to augment the substrate graph with meta-nodes and meta-links that allow the two phases to be well correlated, achieving more efficient solutions. Neither of these works considers security.
As failures in networks are inevitable, the issue of failure recovery and survavibility in VNE has gained attention recently. H. Yu et al. [164] have focused on the failure recovery of nodes. They proposed to extend the basic VNE mapping with the inclusion of redundant nodes. Rahman et al. [124] formulated the survivable virtual network embedding (SVNE) problem to incorporate single substrate link failures. Another problem in the same area is to ensure virtual network connectivity in the presence of multiple substrate link failures. This is explored by Shahriar et al. [136], with the authors proposing two heuristic solutions. Contrary to our work, these proposals target only availability.
A mostly unexplored perspective on the VNE problem is providing security guarantees. Fischer et al. [59] have introduced this problem with a position paper where the authors proposed the assignment of security levels in the physical resources and virtual network requests. No algorithms were presented. Liu et al. [94] have afterwards proposed a VNE algorithm based on this idea. Their simple model does not support the detailed specification of security we propose, and does not consider availability nor a

user-centric cloud setting with different trust domains.
The majority of the works in VNE field only consider a single infrastructure provider (InP). The only exception is the work by M. Chowdhury et al. [44], that addresses the conflicts of interest between Services Providers, SPs (interested in satisfying their demands while minimizing their expenditure) and InPs (that strives to optimize the allocation of their resources by preferring requests with higher revenue while offloading unprofitable work onto their competitors). They present PolyVINE – a policy-based end-to-end VNE framework – that, in short, partitions a VN request into k subgraphs to be embedded onto k SNs, establishes inter-connections between the k subgraphs using inter-domain paths, and embeds each subgraph in each InP SN using an intra-domain algorithm.

3.1.7 Conclusions

In this section we presented our solution to the secure VNE problem. Our solution addresses a diverse set of security requirements, applied both to communication links and virtual nodes. These requirements enable several trade-offs to be explored with regard to the selected defenses. In addition, multiple clouds are considered with distinct levels of trust. By not relying on a single cloud provider we avoid internet-scale single points of failures (with the support of backups), and privacy issues can be accommodated by constraining the mapping of certain virtual nodes to specific classes of clouds (e.g., private). The experimental results show that our solution leads to a high request acceptance rate and an efficient use of substrate resources. The inclusion of requests with stronger security demands can cause an increase on the average revenue for the provider if appropriate pricing schemes are employed.

3.2 Scalable Virtual Network Embedding

The main idea of our multi-cloud network virtualization solution, Sirius, is to enrich the substrate network with resources from public and private clouds in order to enhance the networking services of users, namely with respect to security. As explained, three important benefits are: first, it allows the provider to scale out by outsourcing resources, enabling elasticity of the infrastructure. This can be used not only to extend the substrate but also to save costs (e.g., by optimizing pricing schemes [173, 138]). Second, it improves performance and dependability. If resources are spread (and replicated) across clouds it becomes possible to explore locality to decrease delays and to tolerate operator-wide failure [150, 121]. Third, it enhances the security options for the virtual networks. For instance, the user may consider a subset of her virtual network to be more sensitive, restricting it to a specific location considered more trustful (e.g., a cloud with stricter SLA). These advantages result in the importance of this type of deployments to be trending upwards, with 85% of enterprises reporting to already have a multi-cloud strategy for their business [127].
However, exploring these benefits is challenging. As the multi-cloud provider has no control over the public cloud resources (e.g., the VM hypervisor), it is necessary to employ some form of nested virtualization [18], and this may impact performance. In addition, the problem of embedding virtual resources in this setting is highly complicated. In particular, it is necessary to deal with an hybrid substrate, as private data center topologies (typically a Clos variant) differ greatly from the network offered by a public cloud (a full mesh or "big switch"). Since most network embedding algorithms [60] target wide-area networks and mesh topologies, they perform poorly when directly applied to this context. They are also unsuitable for any realistic deployment, as they often recur to solving the Multi-Commodity Flow (MCF) problem for link mapping [169]. This approach is slow to be applied to large infrastructures. It is also impractical, as it would be necessary to adjust the resulting path splits to the granularity supported by the underlying cloud, a problem not addressed in the embedding literature. Aside from this, as our goal is to leverage from an enriched substrate to enhance the security and availability of virtual networks, it is necessary to further extend the embedding algorithms with these new requirements. So far, little attention has been paid to security. A recent exception is the solution we presented in the previous section. However, that MILP solution scales very poorly.
To address these challenges we propose a novel embedding algorithm that achieves five goals:

- First, it makes efficient use of the substrate resources, achieving a very high acceptance rate of virtual network requests, consequently increasing provider profit.
- Second, it is topology-agnostic, allowing it to achieve good results for radically different topologies.
- Third, it allows users to specify the level of security (including availability) of each element of their virtual networks, and guarantees their requests are fulfilled.
- Fourth, it improves application performance by significantly reducing the average path length.
- Finally, it scales well, making it practical for large-scale deployments.

This algorithm has been implemented in Sirius, supporting the management of multi-cloud substrate infrastructures, and the provisioning of user-defined virtual network requests. We analyze the behavior and performance of the algorithm with large-scale simulations that consider a private data center (following the Google Jupiter topology design [145]), extended with hundreds of cloud resources spread across two public clouds. In addition, we evaluate our prototype in a substrate composed of one private datacenter and two public clouds (Amazon EC2 and Google Cloud Platform). The main conclusion is that our system fulfills all goals set. Namely, it allows virtual networks to extend across multiple clouds, without significant loss of performance compared to a non-virtualized substrate. When compared with alternative approaches, our novel embedding algorithm enables multi-cloud providers to increase the virtual network acceptance rate, reduce path lengths and grow provider revenue. Our evaluation also shows that virtual networks with several thousands of virtual hosts can be provisioned in short intervals, even if spread over several clouds.

3.2.1 Design requirements

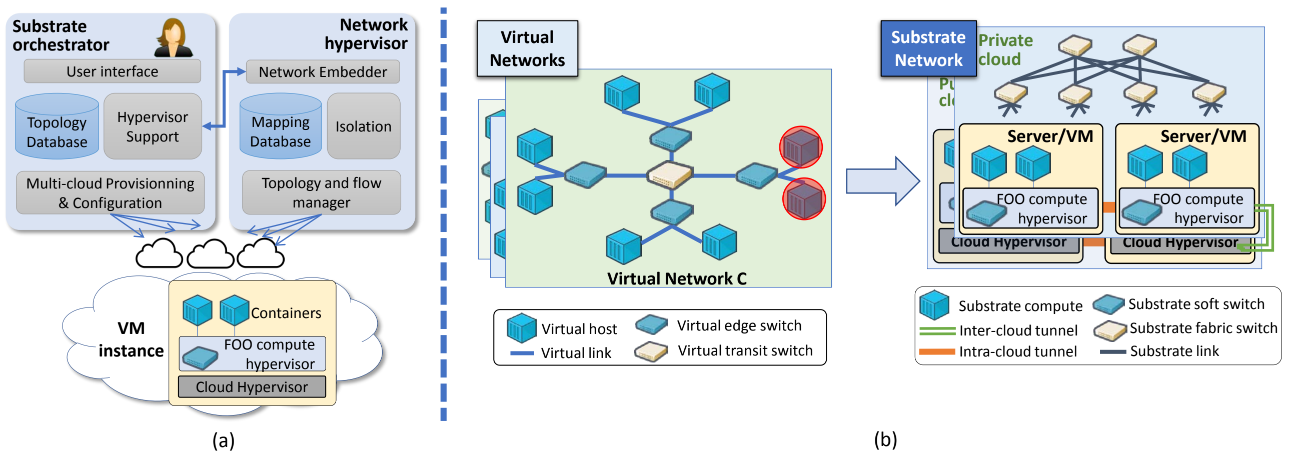


Figure 3.4: (a) System architecture; (b) Virtual networks and substrate.

Recall that our solution allows users to define their virtual networks (VN) with arbitrary topologies and addressing schemes, and guarantee isolation of all tenants that share the substrate (see Figure 3.4). On top of this foundation, we set a couple of ambitious goals to our platform, including the improvement of its scalability properties, alongside the enhancement of the networking services available to users. Towards these objectives we introduce an additional set of requirements – the technical innovations that materialize these requirements are the core of this work. These are:

- *Substrate scalability:* Allow the network substrate to scale out, by extending it with public cloud resources.
- *System scalability:* Handles on the order of the hundreds of requests per second.

- *Enhanced virtual network services:* Allow users to define the security and availability requirements of every compute element of their virtual networks, increasing their dependability.
- *Provider profit:* The mapping of virtual to substrate resources should maximize provider profit, by means of high acceptance rates and efficient utilization of resources.
- *Fit for a hybrid multi-cloud:* The solution should perform well in a diverse substrate network with different topologies, including public (e.g., Amazon EC2), private (e.g., a modern data center), and hybrid clouds (e.g., a private DC extended with public cloud resources).
- *Practicality:* The constraints of the network substrate should be taken into account.
- *Performance:* The virtualization layer should not introduce significant overhead and application performance should not degrade.

These requirements are not met by any existing solution, which at most address a subset of these criteria.

3.2.2 Virtual and Substrate Networks

Virtual networks. In our platform, the user can define her virtual network (Figure 3.4(b), left) by means of a graphical user interface or through a configuration file. She can define any arbitrary topology composed of a set of *virtual hosts* and a group of *virtual switches*, interconnected by *virtual links*. The virtual switches can be of one of two types: *virtual edge switches*, in case they have virtual hosts attached, or *virtual transit switches*, in case they do not. The virtual hosts can be configured with any addresses from the entire L2 and L3 address space[7]. The virtual links are configured with the bandwidths and latencies required.

A core contribution of our solution arises from allowing users to further enhance their networks by setting the specific security requirements of each virtual host, switch, and link. For instance, specific virtual hosts may be considered sensitive, leading to restrictions about their location (e.g., to be hosted in a trusted facility) or of its type (e.g., to be hosted in a secure substrate, such as one offering threat prevention or encryption). For this purpose, our solution enables the network provider to define the *security level* of each cloud and of each data plane element of the substrate, allowing users to specify the security requirements of all hosts, switches, and links of the virtual network. It is also possible to define the *level of availability* of virtual hosts. In this case, our system enforces these elements to be replicated, accordingly with their level. For instance, if the level of availability required is high, it may be replicated in a different cloud, to tolerate large-scale cloud outages.

Figure 3.4 illustrates a virtual network in our system, with a simplified set of requirements. In this example, the nodes within a red circle represent sensitive elements, and therefore need to be located in a trusted cloud.

Substrate network. Our system allows a network provider to extend its infrastructure by enriching its substrate with resources from public cloud providers, allowing it to be shared (transparently) by various users (Figure 3.4(b), right). The resources are organized in such a way to create a single multi-cloud abstraction. Our substrate is composed of one or several private infrastructures, and one or several public clouds. In this infrastructure, the *substrate compute* elements run on top of the compute hypervisor, and are inter-connected by *substrate links* and *substrate switches*. Every hypervisor runs one *substrate software switch* to allow communication between the substrate compute elements and between these and the outside world.

We consider a second type of substrate switches: the *substrate fabric switch*. This corresponds to a physical switch under our control – typically part of a private datacenter (DC) fabric. Note, however, that this does not mean the solution to require full control of all DC switches. If the provider does not have centralized control over the fabric (as it uses traditional L2/L3 networking), then the substrate

[7]With the obvious restriction that they are not allowed to use the same address in different hosts of their own network.

topology will not include the fabric switches. In this case, the only switching elements are the software switches that run at the edge. The *substrate links* include tunnels (both intra-cloud and inter-cloud) and physical links. Again, physical links are only included in the substrate topology if we have control over the physical switches they connect to. Finally, every cloud includes one gateway that acts as an edge router. The inter-cloud tunnels are set up and maintained in this node (as such, this is the only element that requires a public IP address).

3.2.3 Virtual Network Embedding

The next section abstracts the elements of the infrastructure in a model that captures the fundamental characteristics of the substrate and user demands for the virtual networks. Our solution then uses the models to optimize the embedding of user requests in the clouds.

3.2.3.1 Network model

Substrate Network:

The substrate network is modeled as a weighted undirected graph $G^S = (N^S, E^S, A^S_N, A^S_E)$, where N^S is a set of nodes, E^S is the group of links (or edges) and A^S_N / A^S_E are the attributes of the nodes and edges. A node n^S is a network element capable of forwarding packets. It can be either a software switch or a fabric switch, which is modeled by an attribute $type(n^S)$ with values 0 or 1 respectively.
A software switch connects a number of local compute elements (e.g., containers) to the infrastructure. All run in the same physical machine and share the local resources. Therefore, we employ attribute $cpu(n^S)$ to aggregate the total CPU capacity available for network tasks and processing of user applications. Similarly, these components have an equivalent set of protections and are located in the same cloud. We use attributes $sec(n^S) \geq 0$ and $cloud(n^S) \geq 0$ to represent the security level of the ensemble and the trust associated with the cloud, with higher values associated to stronger safeguards.
Fabric switches are internal routing devices, utilized for example in the access or aggregation layers of a datacenter. They are optional elements because our solution is able to enforce all necessary traffic forwarding decisions by configuring only the software switches at the edge. However, they allow for additional flexibility when computing the paths, often leading to more efficient embeddings. Since public cloud providers disallow the configuration of internal network devices, it is expected that fabric switches will only be modeled in private datacenters. These switches have the same attributes as above, where the values for $sec(n^S)$ and $cloud(n^S)$ are dictated by the risk appetite of the owner organization. Overall, the attributes of a node are $A^S_N = \{\{type(n^S), cpu(n^S), sec(n^S), cloud(n^S)\} \mid n^S \in N^S\}$.
The edges are characterized by the total bandwidth capacity (attribute $bw(e^S) > 0$) and the average latency ($lat(e^S) > 0$). They also have an associated security level ($sec(e^S) > 0$), where for example inter-cloud links may be perceived less secure than edges of a private datacenter, as the first have to be routed over the Internet. Overall, the edge attributes are $A^S_E = \{\{bw(e^S), lat(e^S), sec(e^S)\} \mid e^S \in E^S\}$.
As explained, in our system users have two means to specify the substrate network. We have a graphical tool to enable the drawing of the network topology and set the attributes. In alternative, the policy grammar described in the previous section can be employed to describe the substrate in a text file. In both cases, some aspects of our infrastructure can be explored to ease the specification. For example, default security levels can be inherited by all internal links and nodes of a cloud. In addition, instead of relying on the user to provide the links latencies, they can be calculated using measurements performed on the network (which are periodically updated).

Virtual Networks:

VNs are also modeled as weighted undirected graphs $G^V = (N^V, E^V, A^V_N, A^V_E)$, where N^V is the set of virtual nodes, E^V is the set of virtual links (or edges), and A^V_N / A^V_E are the node and edge attributes. These attributes are much alike the substrate network attributes. For example, $type()$ classifies whether a node is a virtual edge or a virtual transit switch. In this case, and to reduce the

specification effort, $type(n^V)$ is actually inferred by the system using a straightforward rule – *a switch with no virtual host attached is considered a virtual transit switch; otherwise, it is a virtual edge switch.* A node n^V that corresponds to a virtual edge switch models the requirements of the switch and the locally connected hosts. In terms of demanded CPU, the attribute $cpu(n^V)$ is the sum of all requested processing capacity (for network tasks and applications). With the other attributes, we take the most strict requirement of all elements (e.g., if hosts need a security level of 4 but the switch only asks for 1, then $sec(n^V)$ is 4). For virtual transit switches, there is a direct relation between the requirements and the attributes of the matching node.
VNs only have an extra attribute in A_N^V to support enhanced availability. In many scenarios, hosts should be replicated so that backups can take over the computations after a failure. This means that during embedding additional substrate resources need to be allocated for the backups. The attribute $avail(n^V)$ can take three pre-defined values: $avail(n^V) = 0$ means no replication; $avail(n^V) = 1$ requests backups in the same cloud; for replication in another cloud then $avail(n^V) = 2$ (e.g., to survive from a cloud outage).
Overall, the two sets of attributes are:

$$A_N^V = \{\{type(n^V), cpu(n^V), sec(n^V), cloud(n^V), avail(n^V)\} \mid n^V \in N^V\} \text{ and } A_E^V = \{\{bw(e^V), lat(e^V), sec(e^V)\} \mid e^V \in E^V\}.$$

Virtual Network Requests:

VNRs are composed by the requested virtual network plus two extra parameters:

$$VNR = (N^V, E^V, A_N^V, A_E^V, Time^V, Dur^V).$$

The first corresponds to the instant when the VNR arrived to the system ($Time^V$), while the second to the period during which the virtual network should remain active in the substrate (Dur^V). If necessary, at a later moment, the user may extend the duration to avoid eviction from the substrate.

3.2.4 Scalable VNE

Sirius instantiates the users requests by mapping the associated VNs onto the substrate while respecting all declared attributes. Our solution handles the on-line VNE problem, where every time a VNR arrives there is an attempt to find an appropriate provisioning in the substrate. While deciding on the location of the resources, an heuristic mapping is performed with the purpose of increasing the overall acceptance ratio, reducing the usage of substrate resources, and fulfilling the security needs. If a solution is found, the residual capacity of the substrate resources is decreased by the amount that is going to be consumed. Otherwise, the request is rejected.
While experimenting with existing embedding algorithms, we observed some limitations when applied to our multi-cloud environment, namely related with the inability to address the security requirements, the incapacity to support large numbers of switches, and inefficiencies on the use of resources (e.g., because the assumed network model is not a good match for our setting). Therefore, we designed a new algorithm built around the following ideas: (i) Optimal embedding solutions, for instance, based on linear program optimization do not scale to our envisioned scenarios, and therefore we employ a greedy approach based on two utility functions to guide the selection of the resources; (ii) The mapping of the virtual resources to the substrate is carried out in two phases, where in the first the nodes are chosen and then the links. While ensuring the security constraints, we give priority to the security level over the cloud trust; (iii) The backup resources necessary to fulfil availability requirements are only reserved after the normal resources have been completely mapped, giving precedence to the common case where no failures occur. The next subsections detail the various parts of our solution.

3.2.4.1 Utility Functions

The process for selecting the substrate nodes uses two utility functions to prioritize the nodes that should be picked earlier. The first function, $UResSec()$, utilizes only information about the current resource consumption and the node security characteristics. In particular, it values more the nodes that: (i) have the greatest percentage of available resources, as this contributes for an increase in the VNR acceptance ratio; and (ii) provide the lowest security assurances (but still larger than the demanded ones), to reduce the embedding costs (recall that highly secure cloud instances are normally substantially more expensive[8]). The utility of a substrate node n^S is:

$$UResSec(n^S) \;=\; \frac{\%R_N(n^S) \times \sum\limits_{\forall e^S \to n^S} \%R_E(e^S)}{sec(n^S) \times cloud(n^S)} \tag{3.44}$$

where the $\%R_N(n^S) = R_N(n^S)/cpu(n^S)$ is the percentage of residual CPU capacity (or available capacity) of a substrate node. The residual capacity is computed with Equation 3.45, with $n^V \in N^V$ and $x \uparrow y$ denoting that the virtual node x is hosted on the substrate node y:

$$R_N(n^S) = cpu(n^S) \;-\; \sum_{\forall n^V \uparrow n^S} cpu(n^V) \tag{3.45}$$

and, where the sum element of Equation 3.44 corresponds to the overall available bandwidth of the edges connected to n^S ($e^S \to n^S$ means n^S is an endpoint of e^S). The value of $\%R_E(e^S) = R_E(e^S)/bw(e^S)$ is the percentage of the residual capacity of a substrate link. The residual capacity is calculated with Equation 3.46, with $e^V \in E^V$ and $x \uparrow y$ denoting that the flow of the virtual link x traverses the substrate link y:

$$R_E(e^S) = bw(e^S) \;-\; \sum_{\forall e^V \uparrow e^S} bw(e^V) \tag{3.46}$$

The second utility function, $UPath()$, contributes to decrease the distance (in number of hops) among the substrate nodes of a VN embedding (we also call it Path Contraction), improving the QoS and decreasing further the provider costs. When a virtual node n^V is being mapped, the utility of selecting a substrate node n^S is computed with two factors: (i) the $UResSec()$ of n^S; and (ii) the average distance between n^S and the substrate nodes that have already been used to place the neighbors of n^V (given by function $avgDist2Neighbors()$). The substrate node utility is:

$$UPath(n^S, n^V) \;=\; \frac{UResSec(n^S)}{avgDist2Neighbors(n^S, n^V)} \tag{3.47}$$

Notice that the division by the average distance will diminish the utility of substrate nodes located further away from the already mapped nodes. The effect is a lower communication delay (average path length is shorter) and, simultaneously, a decrease on bandwidth costs.

3.2.4.2 Scalable and Secure VNE Algorithm

Our *Scalable and Secure Virtual Network Embedding (SSecVNE)* algorithm is based on three procedures: (i) SecVNE, that receives and processes the tenants VN request; (ii) node and edge primary mapping, employing the utility functions to steer the substrate resource selection; and (iii) backup mapping allocates disjoint resources to avoid common mode failures.

[8] For example, compare the cost of a normal instance and the Trend Micro Deep Security instance at aws.amazon.com/marketplace.

Algorithm 1: SecVNE Procedure

```
Input: $G^V$, $G^S$, $R_N$, $R_E$
Output: $PMap$      // mapping for primary network
Output: $BMap$      // mapping for backup network
1  $PMap.N \leftarrow NodeMapping(G^V, G^S, R_N, R_E)$;
2  $PMap.L \leftarrow LinkMapping(G^V, G^S, R_N, R_E, PMap.N)$;
3  if ($(PMap.N \neq \emptyset) \wedge (PMap.L \neq \emptyset)$) then
4      $Rtemp_N \leftarrow R_N$;
5      $Rtemp_E \leftarrow R_E$;
6      $UpdateResources(R_N, R_E, PMap)$;
7      if (at least one virtual node needs backup) then
8          $BMap.N \leftarrow BNodeMap(G^V, G^S, R_N, PMap)$;
9          $BMap.L \leftarrow BLinkMap(G^V, G^S, R_N, R_E, PMap)$;
10         if ($(BMap.N \neq \emptyset) \wedge (BMap.L \neq \emptyset)$) then
11             $UpdateResources(R^S_N, R^S_E, BMap)$;
12             return ($PMap$, $BMap$);
13         else
14             $R_N \leftarrow Rtemp_N$;
15             $R_E \leftarrow Rtemp_E$;
16             return ($\emptyset$, $\emptyset$);
17     else
18         return ($PMap$, $\emptyset$);
19 else
20     return ($\emptyset$,$\emptyset$);
```

SecVNE:

VN requests are embedded into the substrate with Algorithm 1. Two kinds of inputs are expected: (1) the VN model (G^V) including all attributes; and (2) the substrate description, with the model (G^S) and current nodes and edges residual capacities (R_N, R_E). At system initialization, the residual capacities are assigned the values of the resource capacities in the substrate model, but as VNR are serviced, they are updated using Equations 3.45 and 3.46. In addition, when a VNR execution ends, the residual capacities are increased to reflect the release of the associated resources.
Two mappings are potentially produced by the algorithm. One is for the network that should be used in normal operation, called the $PMap$ (from *primary mapping*), and the other that contains the backup nodes and edges to be employed in case of failure, called the $BMap$ (from *backup mapping*). A mapping is a set of tuples, each with a virtual resource identifier and the corresponding substrate resource(s) where it will placed (and some additional information).
The algorithm logic is relatively simple. It starts by seeking for a $PMap$ (Lines 1-2). Then, it changes the residual capacities based on what is going to be consumed after deployment (Line 6). If at least one of the virtual nodes requires availability support ($aval(n^V_i) > 0$), then a backup mapping is also obtained (Lines 8-9) and the capacities are updated (Line 11).

Node Mapping:

Algorithm 2 implements the $NodeMapping()$ procedure. It is responsible for finding a valid embedding for the virtual nodes of the VN, ensuring that all attributes are taken into consideration.
The procedure starts by creating a table where the virtual nodes are ordered by a score value (Line 2, and top of Figure 3.5). The virtual node score is calculated using

$$NScore(n^V) = \frac{cpu(n^V) \times \sum_{\forall e^V \to n^V} bw(e^V)}{sec(n^V) \times cloud(n^V)} \quad (3.48)$$

```
Algorithm 2: NodeMapping()
    Input: $G^V$, $G^S$, $R_N$, $R_E$
    Output: nodeMap        // mapping for the nodes
 1  $nodeMap \leftarrow \emptyset$;
 2  $scoreT \leftarrow getScore(G^V)$;
 3  $utilT \leftarrow getUtil(G^S, R_N, R_E)$;
 4  forall ($n_i^V \in G^V$) do
 5      $virtualNodeMapped \leftarrow false$;
 6      $candidT = getCandidates(n_i^V, G^V, G^S, utilT)$;
 7      forall ($n_j^S \in candidT$) do
 8          if ($cpu(n_i^V) \leqslant R_N(n_j^S)$) then
 9              $nodeMap \leftarrow nodeMap \cup (n_i^V, n_j^S)$;
10              $delUtil(n_j^S, utilT)$;
11              $virtualNodeMapped \leftarrow true$;
12              $break$;
13      if ($virtualNodeMapped = false$) then
14          return $\emptyset$;
15  return $nodeMap$;
```

where $e^V \rightarrow n^V$ means n^V is an endpoint of link e^V. Next, the $UResSec$ utility is calculated for all substrate nodes (Line 3, and middle of the figure). Table *utilT* is an hashMap indexed by the security level and cloud trust of the node, to optimize accesses by security demand.
Virtual nodes n_i^V are processed one at a time. For each, we select all acceptable candidate substrate nodes, i.e., the switches n_j^S that provide stronger security assurances than what is requested ($sec(n_j^S) \geq sec(n_i^V)$ and $cloud(n_j^S) \geq cloud(n_i^V)$). In addition, candidates are chosen based on the type of virtual node: if n_i^V is a virtual edge switch ($type(n_i^V) = 0$) then only substrate software switches are acceptable ($type(n_j^S) = 0$); otherwise, for virtual transit switches ($type(n_i^V) = 1$), we allow either software or fabric substrate switches. These candidates are placed in a table called *candidT* using the $UPath$ utility value to order them (Line 6, and bottom of the figure). Next, the table is searched for a candidate that has enough residual CPU capacity (Line 8). The first to be found will be used in the embedding (Line 9). We also remove this node from *utilT* to prevent further mappings from this VNR, thus avoiding situations where a single failure would compromise a significant part of the primary virtual network (Line 10).

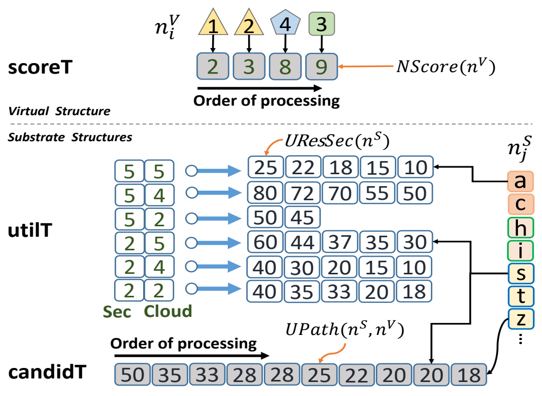


Figure 3.5: Data structures used in node mapping.

The ordering of table search has a strong impact on performance, namely in terms of request acceptance and costs — *scoreT* should be processed from lower-to-higher score, and *candidT* from higher-to-lower utilities. The intuition, which was confirmed by our simulations, is that this leads to embeddings where:

(i) virtual nodes with modest security demands go to the substrate nodes that give less assurances; (ii) nodes end up being physically located near to each other; (iii) there is an even distribution of the residual capacities in most scenarios.

Algorithm 3: LinkMapping()

```
Input: $G^V$,$G^S$,$R_N$, $R_E$, $nodeMap$
Output: $linkMap$    // link mappings
1  $linkMap \leftarrow \emptyset$;
2  $Rtemp_N \leftarrow R_N$;
3  $Rtemp_E \leftarrow R_E$;
4  forall ($e_i^V \in E^V$) do
5      $totalBw \leftarrow 0$;
6      $Rloop_E \leftarrow R_E$;
7      forall ($e_j^S \in E^S$) do
8          if ($sec(e_i^V) > sec(e_j^S)$) then
9              $Rloop_E(e_j^S) \leftarrow 0$;
10     $Paths \leftarrow getPaths(e_i^V, G^V, G^S, Rloop_E, nodeMap)$;
11     foreach ($p \in Paths$) do
12         if ($lat(e_i^V) \geq getLatency(p, G^S)$) then
13             $bwp \leftarrow getMinBandwidth(p, Rloop_E)$;
14             $totalBw \leftarrow totalBw + bwp$;
15             $candP \leftarrow candP \cup (e_i^V, bwp, p)$;
16         if ($|candP| = MaxPaths$) then
17             $break$;
18     if ($totalBw \geq bw(e_i^V)$) then
19         forall ($mp \in candP$) do
20             $bw(mp) \leftarrow \lceil (bw(mp)/totalBw) * bw(e_i^V) \rceil$;
21         $UpdateLinkResources(R_E, candP)$;
22         $linkMap \leftarrow linkMap \cup candP$;
23     else
24         $R_N \leftarrow Rtemp_N$;
25         $R_E \leftarrow Rtemp_E$;
26         return $\emptyset$;
27 return $linkMap$;
```

Link Mapping:

Algorithm 3 finds a mapping between the virtual edges and the substrate network. Each edge is processed individually, searching for a suitable network connection between the two substrate nodes where its virtual endpoints will be embedded. The approach is flexible, allowing the use of a single or multiple paths.

The algorithm works in a few steps. First, the substrate edges that do not provide the necessary security guarantees are excluded (Lines 6-9). This is achieved by setting to null the residual bandwidth capacity of those less secure edges on an auxiliary variable $Rloop_E$ (Line 9), thus preventing their selection at later steps.

Second, we obtain a set of paths that could be employed to connect the two substrate nodes where the virtual edge endpoints will be embedded (Line 10). In our implementation, we resort to the k-edge disjoint shortest path algorithm to find these paths, using as edge weights the inverse of the residual bandwidths. This ensures that when “distance” is minimized, the algorithm picks the paths that have the most available bandwidth.

Third, the connections are chosen (Lines 11-17). Prospect paths p are an ordered sequence of substrate

edges ($p = (e_1^S, e_2^S, ...)$), which have a certain latency (equal to the sum of the link latencies, and calculated by $getLatency()$) and a maximum bandwidth (given by the smallest residual bandwidth of all the edges, and computed by $getMinBandwidth()$). Eligible paths need to have a latency less than the requested (Line 12). These paths are stored in a candidate set $candP$ together with the corresponding virtual edge and available bandwidth (Line 15). The set will have at most $MaxPaths$, a constant that defines the degree of multipathing (when set to 1, a single path is used) (Lines 16-17). This constant can be used to prevent an excessive level of traffic fragmentation, which is important when managing the number of entries in the packet forwarding tables of the switches. On the other hand, it improves dependability because localized link failures can be automatically tolerated with multi-path data forwarding (if enough residual capacity exists in the surviving paths).
The last step is to define how much traffic goes through each path, ensuring that together they provide the requested edge bandwidth (Lines 18-22). An edge can only be mapped if enough bandwidth is available in the paths (Line 18). In this case, we update the bandwidth in every path to an amount proportional to their maximum capacity, therefore distributing the load (Lines 19-20). Then, the residual capacities of the substrate edges are decreased to represent the future embedding (Line 21) and the set of paths is saved (Line 22).

Backup Mapping:

We aim to provide more available virtual networks through the use of replication. For this purpose, the algorithm creates a backup embedding that satisfies the same requested attributes as the primary mapping. The backup functions, $BNodeMap()$ and $BLinkMap()$ called in Algorithm 1, operate in a similar manner as their normal counterparts, with the exception that all resources used in the primary mappings are excluded from the embedding. There are a few cases, however, where it is impossible to enforce this objective. For instance, inside the same rack typically there is only one ToR switch, so in this case there is some level of sharing.

3.2.5 Evaluation

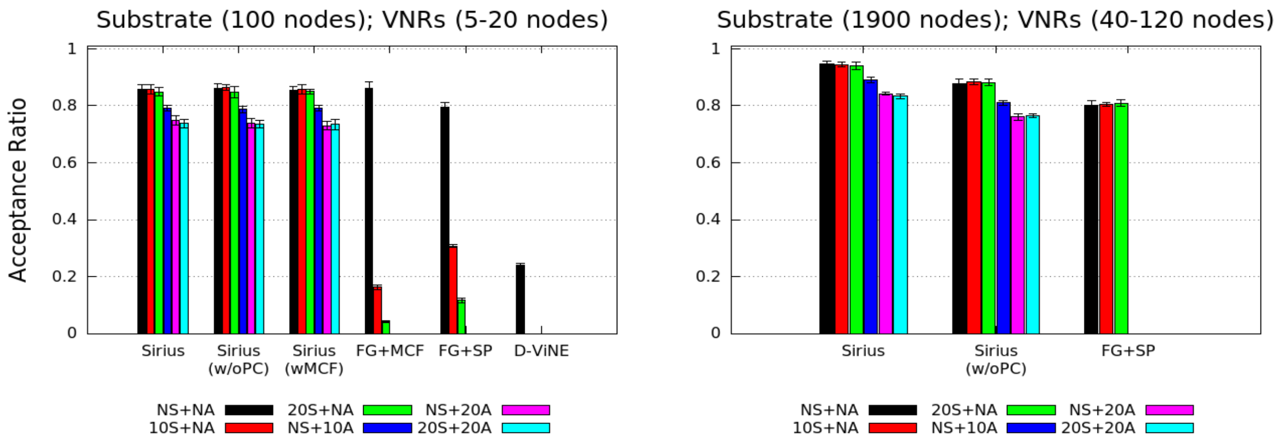
The evaluation aims to answer several questions. First, we want to determine if our solution is efficient in using the substrate resources. Namely, in terms of the acceptance ratio of virtual network requests, which will translate into profit for the multi-cloud provider. Second, we want to understand how the system scales, both with respect to the enrichment of the substrate with cloud resources and the rate of arrival of VNRs. Additionally, we need to find out if Sirius handles well different kinds of topologies, including private, public, and hybrid clouds. Finally, we would like to measure the overhead introduced by the virtualization layer, and how it affects application performance.
We evaluate the solution using large-scale simulations, comparing it with the two most commonly used heuristics: `D-ViNE`, a solution that uses a relaxation of a MIP solution for node mapping and MCF for link mapping [45, 96], and the heuristics proposed by Yu et al. [169], that follow a greedy approach for node mapping and use MCF for link mapping (we label this solution `FG+MCF`). As MCF has scalability limitations, for this second approach we have also used the shortest path algorithm (`FG+SP`).
In addition, we evaluate the performance of our prototype over a multi-cloud substrate composed of a private data center and two public clouds (Amazon and Google), measuring the elapsed time to create various networks.

3.2.5.1 Testing environment

We extended an existing simulator [1] to collect various metrics about the embedding while a VNR workload arrives to the system. Two types of substrate network models were employed: for public clouds we utilized Waxman, where nodes are linked with a probability of 50% [105] (using the GT-ITM tool [170]); for the private data center we created networks following the Google Jupiter topology design [145]. Three substrate networks were considered: `pub_substrate` - 100 nodes spread evenly in three clouds; `pvt_substrate` - 1900 nodes in one private datacenter; and `multi_substrate` -

Notation	Description
NS+NA	no security or availability demands on the VNRs
10S+NA	VNRs with 10% of resources (nodes and links) with security demands (excluding availability)
20S+NA	like *10S+NA*, but with security demands for 20% of the resources
NS+10A	VNRs with no security demands, except for 10% of the nodes requesting replication
NS+20A	like *NS+10A*, but for 20% of the nodes
20S+20A	20% of the resources (nodes and links) with security demands and 20% of the nodes with replication

Table 3.5: VNR configurations that were evaluated.

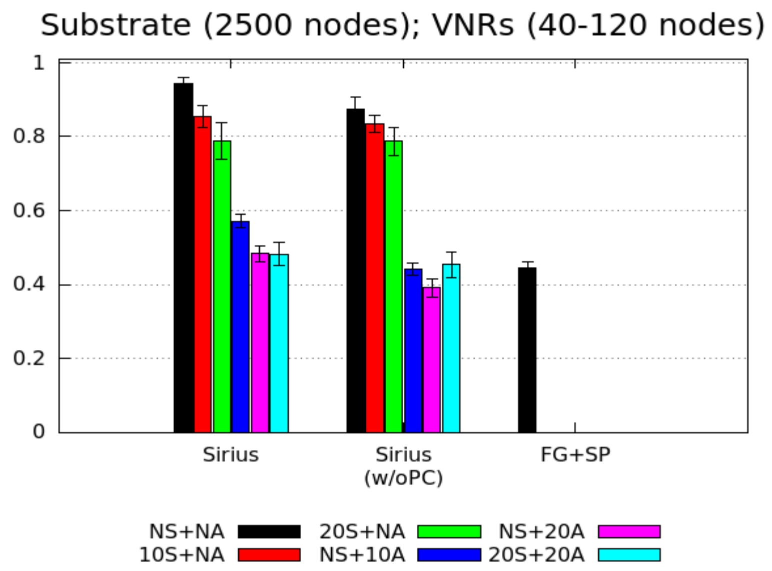


(a) Public cloud (100 nodes)

(b) Private DC (1900 nodes)

Figure 3.6: Acceptance ratio: percentage VNRs for which it was possible to find an embedding.

2500 nodes spread in three clouds and a private datacenter. The CPU and bandwidth (cpu^S and bw^S) of nodes and links is uniformly distributed between 50 and 100 and 500 and 1000 respectively. Latencies inside a data center were small ($lat^S \in \{1.0\}$), but among clouds were much larger ($lat^S \in \{50.0\}$). These resources are also uniformly associated with one of three levels of security and trust ($sec^S \wedge cloud^S \in \{1.0, 1.2, 5.0\}$) in the public clouds and one level ($sec^S \wedge cloudS \in \{6.0\}$) in the private cloud. These values were chosen to achieve a good balance between the diversity of security levels and their monetary cost. An analysis of the cost of Amazon instances with normal and secure VM configurations shows a wide range of values, which are related to the implemented defenses. For example, while an EC2 instance with container protection is around 20% more expensive than a normal instance (hence our choice of 1.2 for the intermediate level of security), the cost of instances with more sophisticated defenses are at least 5 times greater (our choice for the highest level of security).
VNRs have a number of virtual nodes uniformly distributed between 5 and 20 nodes for a first setup and between 40 and 120 nodes for the others. Pairs of virtual nodes are connected with a Waxman topology with probability 50%. The CPU and bandwidth of the virtual nodes and links are uniformly distributed between 10 and 20, and 100 and 200 respectively. Several alternative security and availability requirements are evaluated, as shown in Table 3.5. We assume that VNRs arrivals ($Time^V$) are modeled as a Poisson process with an average rate of 4 VNRs per 100 time units for the first setup and 8 VNRs for the others. Each VNR has an exponentially distributed lifetime (Dur^V) with an average of 1000 time units.



(a) Multi-cloud (2500 nodes)

Figure 3.7: Acceptance ratio: percentage VNRs for which it was possible to find an embedding.

3.2.5.2 Simulations

Figures 3.6 and 3.7 present the results for the acceptance ratio of the three networks. We consider two variants of our approach: `Sirius(w/oPC)` only employs the $UResSec()$, and therefore does not take into consideration the lengths of the paths offered by $UPath()$ (i.e., without Path Contraction); and `Sirius(wMCF)` using MCF for link mapping.
In the case of VNRs with no security demands (`NS+NA`) in the smaller network (Figure 3.6a), our approaches behave similarly to `FG+MCF`, but improve by 6pp (percentual points) over `FG+SP` and by over 50pp over `D-ViNE`. The poor performance of `D-ViNE` is a result of its underlying model not fitting our specific multi-cloud environment. For instance, this solution considers geographical distance, which is not as relevant in a virtualized environment. As first conclusion, in the network topology offered by a public cloud (a full mesh), both `FG+MCF` and `Sirius` achieve very good results.
As security demands are introduced, Sirius acceptance ratio decreases but only slightly. For instance, when 20% of all virtual elements request a level of security above the baseline, the reduction of the acceptance ratio is only 1pp. The same is true with requests that include availability, although the decrease is a bit more pronounced (up to 10pp). This was expected, as replication needs not only to double node resources, but also leads to an increase of the number of substrate paths (to maintain replica connectivity). The alternative solutions perform poorly with security requirements, as they do not consider these additional services. Therefore, most of the produced embeddings had to be rejected because they would violate at least one of the demands. In the case of `D-VINE` no results are shown because after more than one week of running this experiment, the algorithm had not yet finished. Availability bars are not displayed because the algorithms do not consider the possibility of replication.
When observing Figures 3.6b and 3.7a, the advantage of our approach is made clear. No results are included for algorithms with MCF for link mapping, as they take an extremely long time to complete. `FOO` has an acceptance ratio 15pp above `FG+SP` in the `pvt_substrate`, and of over 25pp in the `multi_substrate`. These results demonstrate the effectiveness of our solution in improving the acceptance ratio over the alternatives for both virtualized datacenters and, even more strikingly, for a multi-cloud scenario. The main reason is our more detailed model, which incorporates different types of nodes (software and fabric switches), increasing the options available to map virtual nodes. The conclusions with respect to security are similar to above. One note, however, to explain why the

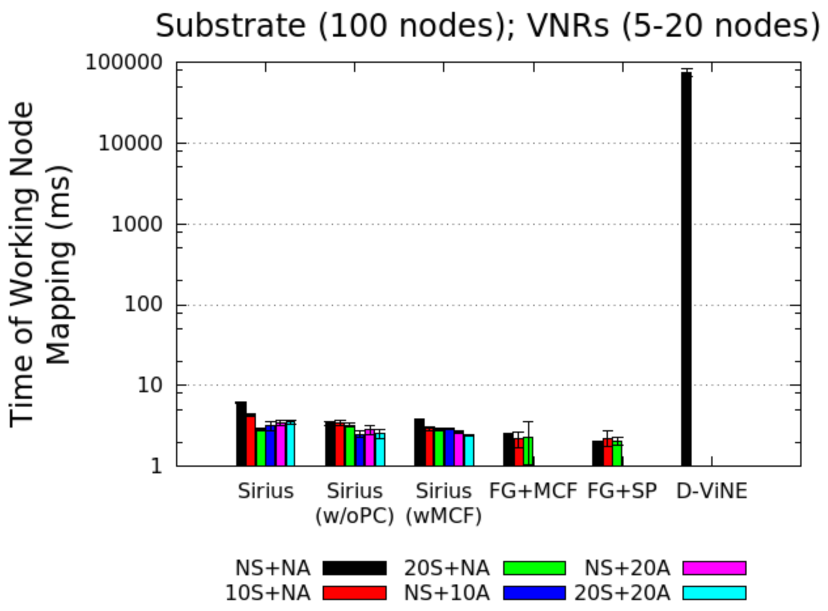
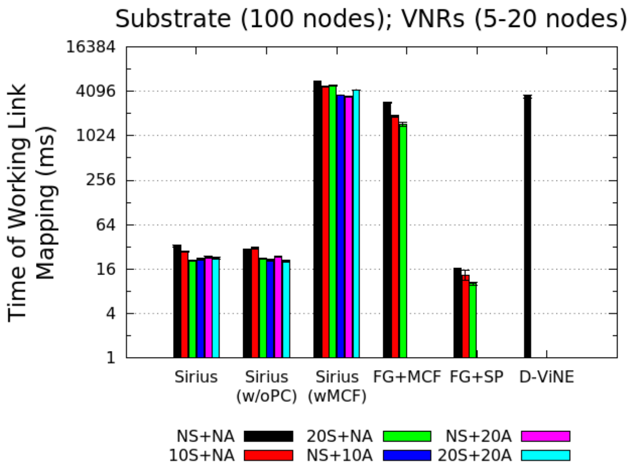


Figure 3.8: Embedding time taken by node mapping (100 nodes)

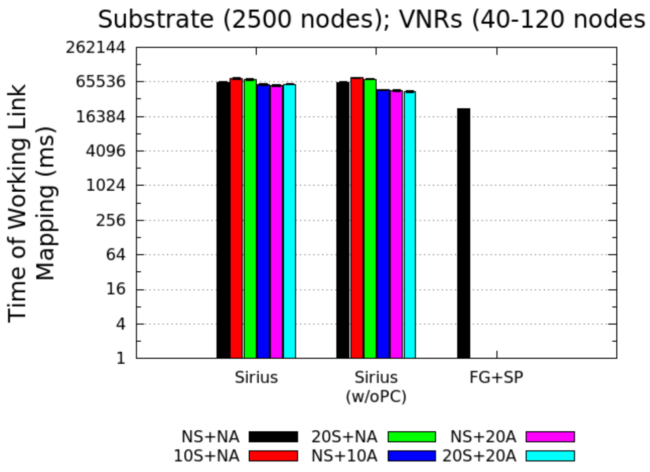
results do not degrade with security in the `pvt_substrate` case, compared to the others. The reason is that in this experiment all nodes are considered of the highest security level, as they are inside the private data center. Another observation is that in some cases the average acceptance ratio is higher (despite still inside the confidence interval) with security demands, which can be counter-intuitive. The reason is that in some cases fulfilling security requirements tends to slightly better balance the substrate load.
Next, we turn to system scalability. Here we focus on embedding latency, as this metric translates into the attainable service rate for virtual network requests. The measurements are taken with code that is equivalent to the one used in our network hypervisor (for our approach, it is actually the same Java implementation). Figures 3.8 and 3.9a present the time to map nodes and links. As can be seen, `D-ViNE` scales very poorly in both phases, while `Sirius`, `Sirius(w/oPC)`, and `FG+SP` behave better. With mappings taking in the order of tens of ms, these solutions enable hundreds to thousands of virtual elements to be embedded per second. The time to embed backup elements is of the same order of magnitude (we omit the graph for space reasons). Finally, as the substrate and the size of the virtual networks grow, the embedding latency increases accordingly. Figure 3.9b displays the worst case of our experiments: the time for link mapping in the `multi_substrate`. For such large-scale network, embedding increases to around 60 seconds per virtual network. In summary, `Sirius`, `Sirius(w/oPC)`, and `FG+SP` are the only embedding solutions that scale to reasonable numbers in the context of a realistic network virtualization system.
Next, we focus on the profit of the multi-cloud provider. As in previous work [169, 96], we assume the revenue of accepting a VNR is proportional to the acquired resources. However, in our case, we assume that security is charged at a higher premium value (inline with public cloud services). We calculate the revenue per VN as:

$$\mathbb{R}(\text{VNR}) = \lambda_1 \sum_{i \in N^V} [1 + \varphi_1(i)]\ cpu^V(i)\ \ sec^V(i)\ \ cloud^V(i)\ +$$
$$\lambda_2 \sum_{(i,j) \in E^V} [1 + \varphi_2(i,j)]\ bw^V(i,j)\ \ sec^V(i,j),$$

where λ_1 and λ_2 are scaling coefficients that denote the relative proportion of each revenue component to the total revenue. These parameters offer providers the flexibility required to price differently the



(a) Link embedding time (100 nodes)



(b) Link embedding time (2500 nodes)

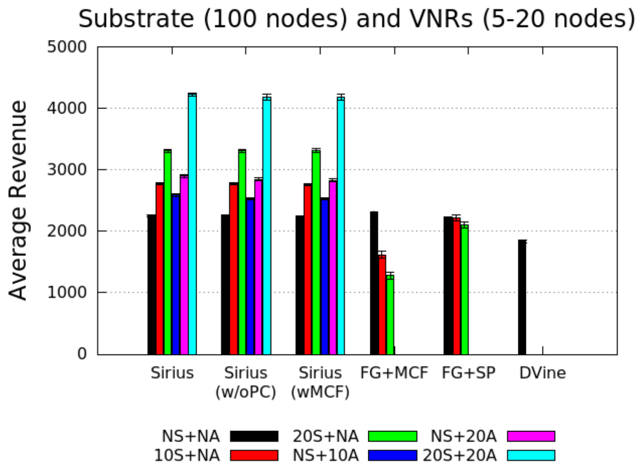
Figure 3.9: Embedding time taken by link mapping.

resources. Variables φ account for the need to have backups, either in the nodes $\varphi_1(i)$ or in the edges $\varphi_2(i,j)$ [9]. In the experiments, we set $\lambda_1 = \lambda_2 = 1$.
Figure 3.10a presents the average revenue generated by embedding VNRs in `pub_substrate`. The main conclusion is that Sirius generally improves the profit of the multi-cloud provider. First, revenue is enhanced when security is included, which gives incentives for providers to offer value-added services. Second, availability can even have a stronger impact because more resources are used to satisfy VNRs.
Figure 3.10b shows the total number of allocated links during the whole experiment. The main conclusion is that by considering the path length in the utility function, Sirius is able to allocate significantly less edges. This is true even when comparing with `FG+SP` that uses shortest path. The reason is that our heuristic is able to bring neighboring nodes closer to each other. Reducing path lengths is important to decrease costs and to improve application performance, as long paths impose a latency penalty.

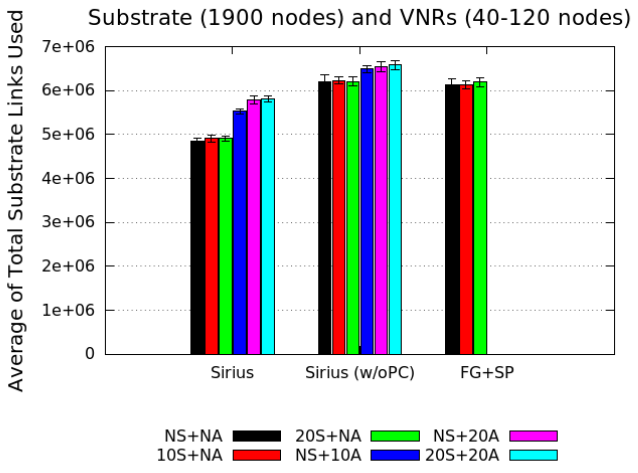
3.2.5.3 Prototype performance

In this section we evaluate the performance of our prototype. Figure 3.11a shows the time to setup a substrate with VMs distributed through the 3 clouds. Since almost all operations are performed in

[9] $\varphi_1(i) = 1$ if a backup is required or 0 otherwise; $\varphi_1(i,j) = 1$, in case of at least one node needs a backup or 0 otherwise.



(a) Average revenue (100 nodes)



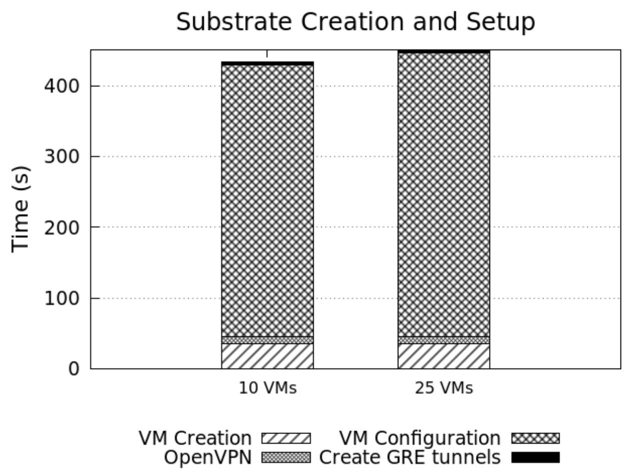
(b) Total embedded links (1900 nodes)

Figure 3.10: Simulations: average VNR revenue & total number of embedded links

parallel, it is possible to observe only a small increase in time when the number of VMs doubled. The slowest operation is VM configuration, which includes the time for software installation (e.g., Docker), getting a basic container image and creation of some of the tunnels. After that, the relevant delay is due to VM provision by the cloud provider. Therefore, overall the added cost of our solution is small. Figure 3.12a shows the setup time for VNs of different sizes in number of compute nodes (i.e., containers). Over 99% of the cost is related to container creation and configuration, with the embedding being insignificant. The jump in the elapsed time between 1k to 4k containers is due to the rise in number of containers deployed per VM (it goes from 100 to 400).
Figure 3.13 further presents the cost of virtualization, in terms of increased latencies and decreased throughputs, using as baseline a VM configuration that accesses directly the network. Inter-cloud RTTs grow around 30%, between 5 and 10ms, and intra-cloud RTTs increase less than 400us. As inter-cloud applications typically assume latencies of this magnitude in their design, and the added intra-cloud cost is small, this overhead is deemed arguably acceptable. Throughput decreases further, with a higher cost being experienced when the baseline is high, as expected [81]. We are currently investigating networking-enhanced VM instances to reduce this overhead.

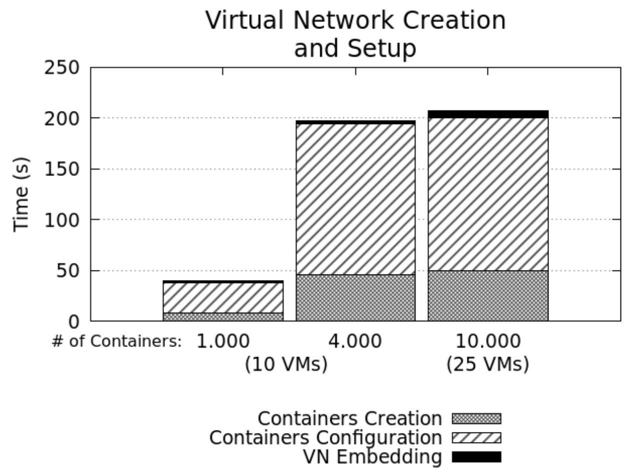
3.2.6 Related work

Cloud networking. As cloud application performance critically depends on the network, Balani et al. [16] have recently proposed the extension of cloud services with an abstraction, the virtual cluster (VC), which offers bandwidth guarantees over existing VM-based abstractions [16]. Follow-up work extended this model with dynamic traffic patterns [162], improved embedding algorithms [129], and scaling [62, 168]. This line of work extends the cloud model with network guarantees, but none offers network virtualization.
Network virtualization. With the emergence of Software-Defined Networking (SDN) [88], it became feasible to fully decouple virtual networks from the substrate on which they run. The first SDN-based system was FlowVisor [141], which allowed different tenants to own a slice of the network. OVX [7] improves over FlowVisor to offer topology and address virtualization. While these solutions assume an SDN infrastructure, VMware's NVP [81] is edge-based, with control at the hypervisor-level. Contrary to our work, these modern platforms do not consider a multi-cloud substrate or the integration of security and availability services.
Virtual network embedding. While efficient greedy heuristics already existed for the node mapping phase of the Virtual Network Embedding (VNE) problem, Yu et al. [169] where the first to solve the link mapping problem efficiently. The authors assumed a substrate that is path splitting-capable, enabling



(a) Substrate creation

Figure 3.11: Substrate creation time.



(a) Virtual network provisioning time

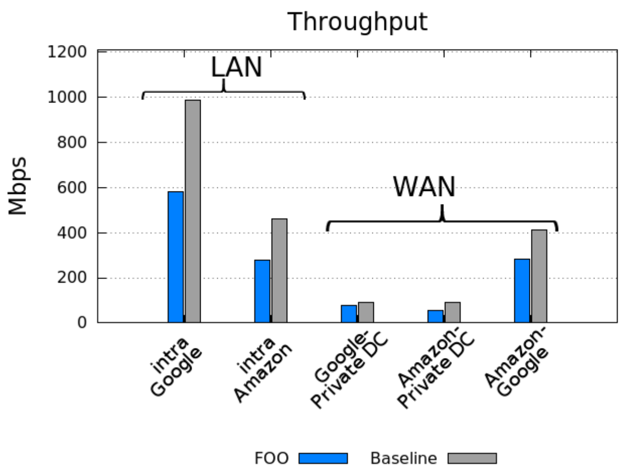
Figure 3.12: Substrate and virtual network creation times.

the problem to be solved as a multicommodity flow (MCF). MCF improved the situation but, as we've argued, is not a good fit for production-quality systems. D-VINE [96] achieves higher acceptance rates by coordinating node and link mapping. The drawback is that it scales very poorly. Recent work [60] has extended these proposals with redundancy for node [164] and link recovery [124, 136]. None of these works considers security. PolyVINE [44] is a framework that considers several infrastructure providers, with the goal of coordinating policies between inter-domains, a different problem from ours. One common limitation of all these works is that they focus solely on the embedding problem, and no system is built.

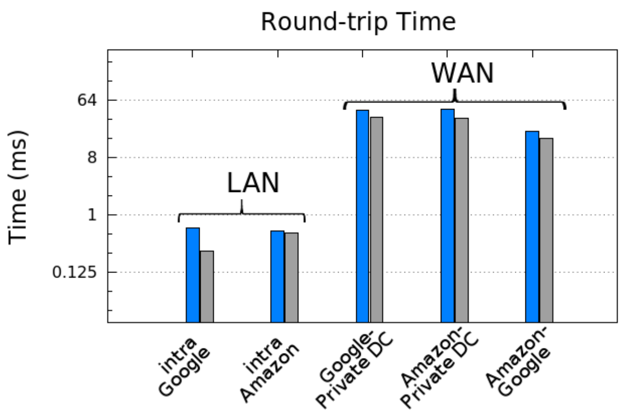
Multi-cloud systems. The multi-cloud model has been successfully applied in the context of computation [160] and storage [28]. One common goal is to improve system dependability. Examples include MapReduce [47], coordination [27], and file systems [28]. To the best of our knowledge, Sirius is the first system to apply this model for network virtualization.

3.2.7 Conclusions

In this section we presented our proposal of a secure virtual network embedding algorithm that scales to large networks. Our solution improves the state-of-the-art by extending the substrate network with



(a) Throughput



(b) Latency

Figure 3.13: Intra- and inter-cloud throughput and latencies.

cloud services and enhancing the virtual networks with security and dependability. Evaluations of our prototype in large scale simulations reveal that, compared with the common alternatives, our solution scales well, increases the acceptance ratio and the provider profit for diverse topologies, maintaining short path lengths to guarantee application performance.

In the next chapter we discuss the techniques employed to improve the security and dependability of Sirius, our network virtualization platform.

Chapter 4 Infrastructure

In this chapter we describe, in Section 4.1, ANCHOR, a logically-centralized security architecture for Software-Defined Networks we propose for the SUPERCLOUD network virtualization infrastructure. As use case of the ANCHOR architecture, in Section 4.2 we present a secure and efficient control plane communications protocol for SDN. To improve the dependability of the infrastructure, in Section 4.3 we present the design and implementation of Rama, the SUPERCLOUD fault-tolerant SDN controller.

4.1 Logically-centralized security

Software-defined networking (SDN) has moved the control function out of the forwarding devices, leading to a logical centralization of functional properties. This decoupling between control and data plane leads to higher flexibility and programmability of network control, enabling fast innovation. Network applications are now deployed on a software-based logically centralized controller, providing the agility of software evolution rather than hardware one. Moreover, as the forwarding devices are now directly controlled by a centralized entity, it is straightforward to provide a global network view to network applications. In spite of all these benefits, this decoupling, associated with a common southbound API (e.g., OpenFlow), has removed an important natural protection of traditional networks. Namely, the heterogeneity of different solutions, the diversity of configuration protocols and operations, among others. For instance, an attack on traditional forwarding devices would need to compromise different protocol interfaces. Hence, from a security perspective, SDN introduces new attack vectors and radically changes the threat surface [90, 134, 49].
So far, the SDN literature has been mostly concerned with functional properties, such as improved routing and traffic engineering [75, 12]. However, gaps in the enforcement of non-functional properties are critical to the deployment of SDN, especially at infrastructure/enterprise scale. For instance: insecure control plane associations or communications, network information disclosure, spoofing attacks, and hijacking of devices, can easily compromise the network operation; performance crises can escalate to globally affect QoS; unavailability and lack or reliability of controllers, forwarding devices, or clock synchronization parameters, can considerably degrade network operation [80, 6, 134].
Addressing these problems in an ad-hoc, piecemeal way, may work, but will inevitably lead to efficiency and effectiveness problems. Although several specific works concerning non-functional properties have recently seen the light e.g., in dependability [36, 128, 76, 85, 21] or security [122, 144, 142, 134], enforcement of non-functional properties as a pillar of SDN robustness calls, in our opinion, for a systemic approach. As such, in this section we claim for a re-iteration of the successful formula behind SDN – 'logical centralization' – for its materialization.
In fact, the problematic scenarios exemplified above can be best avoided by the logical centralization of the system-wide enforcement of non-functional properties, increasing the chances that the whole architecture inherits them in a more balanced and coherent way. The steps to achieve such goal are to: (a) select the crucial properties to enforce (dependability, security, quality-of-service, etc.); (b) identify the current gaps that stand in the way of achieving such properties in SDNs; (c) design a logically-centralized subsystem architecture and middleware, with hooks to the main SDN architectural components, in a way that they can inherit the desired properties; (d) populate the middleware with the appropriate mechanisms and protocols to enforce the desired properties/predicates, across controllers and devices, in a global and consistent manner.

Generically speaking, it is worth emphasizing that centralization has been proposed as a means to address different problems of current networks. For instance, the use of centralized cryptography schemes and centralized sources of trust to authenticate and authorize known entities has been pointed out as a solution for improving the security of Ethernet networks [79]. Similarly, recent research has suggested network security as a service as a means to provide the required security of enterprise networks [134]. However, centralization has its drawbacks, so let us explain why centralization of non-functional property enforcement brings important gains to software-defined networking. We claim, and justify ahead, that it allows to define and enforce global policies for those properties, reduce the complexity of networking devices, ensure higher levels of robustness for critical services, foster interoperability of the non-functional enforcement mechanisms, and better promote the resilience of the architecture itself.
The reader will note that this design philosophy concerns non-functional properties in abstract. To prove our point, in this work, we have chosen *security* as our use case and identified at least four gaps that stand in the way of achieving the former in current SDN systems: (i) *security-performance* gap; (ii) *complexity-robustness* gap; (iii) global security policies gap; and (iv) *resilient roots-of-trust* gap. The security-performance gap comes from the frequent conflict between mechanisms enforcing those two properties. The complexity-robustness gap represents the conflict between the current complexity of security and cryptographic implementations, and the negative impact this has on robustness and hence correctness. The lack of global security policies leads to ad-hoc and discretionary solutions creating weak spots in architectures. The lack of a resilient root-of-trust burdens controllers and devices with trust enforcement mechanisms that are ad-hoc, have limited reach and are often sub-optimal. We further elaborate in the following on the reasons behind these gaps, their negative effects in SDN architectures, and how they can possibly be mitigated through a logically-centralized security enforcement architecture.
To achieve our goals, we propose ANCHOR, a subsystem architecture that does not modify the essence of the current SDN architecture with its payload controllers and devices, but rather stands aside, 'anchors' (logically-centralizes) crucial functionality and properties, and 'hooks' to the latter components, in order to secure the desired properties. In this particular case study, the architecture middleware is populated with specific functionality whose main aim is to ensure the 'security' of control plane associations and of communication amongst controllers and devices.
In addition, we give first steps in addressing a long-standing problem, the fact that a single root-of trust — like ANCHOR, but also like any other standard trusted-third-party, like e.g., CAs in X.509 PKI or the KDC in Kerberos — is a *single point failure* (SPoF). There is nothing wrong with SPoFs, as long as they do not fail often, and/or the consequences of failure can be mitigated, which is unfortunately not the common case. As such, we start by carefully promoting reliability in the design of ANCHOR, endowing it with robust functions in the different modules, in order to reduce the probability of failure/compromise. Moreover, the proposed architecture only requires symmetric key cryptography. This not only ensures a very high performance, but also makes the system secure against attacks by a quantum computer. Thus, the system is also *post-quantum secure* [22]. Second, we mitigate the consequences of successful attacks, by protecting past, pre-compromise communication, and ensuring the quasi-automatic recovery of ANCHOR after detection, even in the face of total control by an adversary, achieving respectively, *perfect forward secrecy (PFS)* and *post-compromise security (PCS)*. Third, our architecture promotes resilience, or the continued prevention of failure/compromise by automatic means such as fault and intrusion tolerance. Though out of the scope of this document, this avenue is part of our plans for future work, and the door is open by our design, since it definitely plugs the SPoF problem, as is well known from the literature, which we debate in Section 4.1.6.
To summarize, the key contributions of our work include the following:

1. The concept of logical centralization of SDN non-functional properties provision.
2. The blueprint of an architectural framework based on middleware composed of a central 'anchor', and local 'hooks' in controllers and devices, hosting whatever functionality needed to enforce these properties.

3. A gap analysis concerning barriers in the achievement of non-functional properties in the security domain, as a proof-of-concept case study.

4. Definition, design and implementation of the mechanisms and algorithms to populate the middleware in order to fill those gaps, and achieve a logically-centralized security architecture that is reliable and highly efficient, post-quantum secure, and provides perfect forward secrecy and post-compromise security.

5. Evaluation of the architecture.

We show that, compared to the state-of-the-art in SDN security, our solution preserves at least the same security functionality, but achieves higher levels of implementation robustness, by vulnerability reduction, while providing high performance. Whilst we try to prove our point with security, our contribution is generic enough to inspire further research concerning other non-functional properties (such as dependability or quality-of-service). It is also worth emphasizing that the architectural concept that we propose in this work would require a greater effort to be deployed in traditional networks, due to the heterogeneity of the infrastructure and its vertical integration. This will be made clear throughout.
We have structured this section as follows. Section 4.1.1 gives the rationale and presents the generic logically-centralized architecture for the system-wide enforcement of non-functional properties, and explains its benefits and limitations. In Section 4.1.2, we discuss the challenges and requirements brought by the current gaps in security-related non-functional properties. Section 4.1.3 describes the logically-centralized security architecture that we propose, along with its mechanisms and algorithms. Then, in Sections 4.1.4 and 4.1.5, we discuss design and implementation aspects of the architecture, and present evaluation results. In Sections 4.1.6 and 4.1.7, we give a brief overview of related work, discuss some challenges and justify some design options of our architecture. Finally, in Section 4.1.8, we conclude.

4.1.1 Architecture

In this section we introduce ANCHOR, a general architecture for logically-centralized enforcement of non-functional properties, such as 'security', 'dependability', or 'quality-of-service' (Figure 4.1). The logical centralization of the provision of non-functional properties allows us to: (1) define and enforce global policies for those properties; (2) reduce the complexity of controllers and forwarding devices; (3) ensure higher levels of robustness for critical services; (4) foster interoperability of the non-functional property enforcement mechanisms; and finally (5) better promote the resilience of the architecture itself. Let us explain the rationale for these claims.
Define and enforce global policies for non-functional properties. One can enforce non-functional properties through piece-wise, partial policies. But it is easier and less error-prone, as attested by SDN architectures with respect to the functional properties, to enforce e.g., security or dependability policies, from a central trust point, in a globally consistent way. Especially when one considers changing policies during system lifetime.
Reduce the complexity of controllers and forwarding devices. One of the most powerful ideas of SDN was exactly to simplify the construction of devices, by stripping them of functionality, centralized on controllers. We are extending the scope of the concept, by relieving both controllers and devices from ad-hoc and redundant implementations of sophisticated mechanisms that are bound to have a critical impact on the whole network.
Ensure higher levels of robustness for critical services. Enforcing non-functional properties like dependability or security has a critical scope, as it potentially affects the entire network. Unfortunately, the robustness of devices and controllers is still a concern, as they are becoming rather complex, which leads to several critical vulnerabilities, as amply exemplified in [134]. For these reasons, a single device or controller may become a single point of failure for the network. A centralized concept as

we advocate might considerably improve on the situation, exactly because the enforcement of non-functional properties would be achieved through a specialized susbsystem, minimally interfering with the SDN payload architecture. A dedicated implementation, carefully designed and verified, would be re-usable, not re-implemented, by the payload components.

Foster interoperability of the non-functional property enforcement mechanisms. Different controllers require different configurations today, and a potential lack of interoperability in terms of non-functional properties arises. Global policies and mechanisms for non-functional property enforcement would also mean an easy path to foster controller and device interoperability (e.g., East and Westbound APIs) in what concerns the former. This way, mechanisms can be modified or added, and have a global repercussion, without the challenge of having to implement such services in each component.

Better promote the resilience of the architecture itself. Having a specialized subsystem architecture already helps for a start, since for example, its operation is not affected by latency and throughput fluctuations of the (payload) control platforms themselves. However, the considerable advantage of both the decoupling and the centralization, is that it becomes straightforward to design in security and dependability measures for the architecture itself, such as advanced techniques and mechanisms to tolerate faults and intrusions (and in essence overcome the main disadvantage of centralization, the potential single-point-of-failure risk).

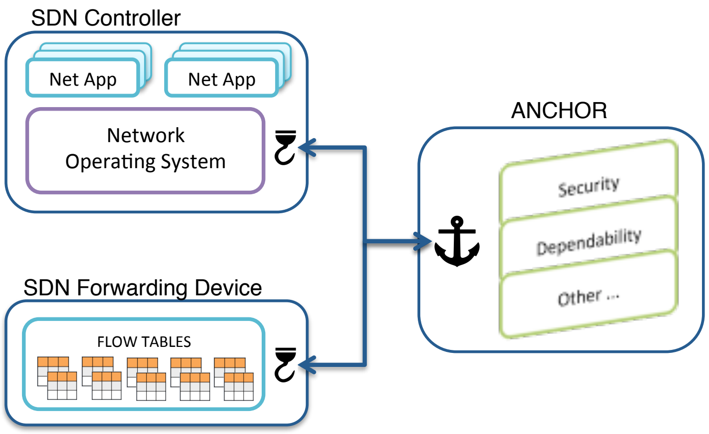


Figure 4.1: ANCHOR general architecture

The general outline of our reference architecture is depicted in Figure 4.1. The "logically-centralized" perspective of non-functional property enforcement is materialized through a subsystem architecture relying on a centralized anchor of trust, a specific middleware whose main aim is to ensure that certain properties – for example, the security of control plane associations and of communication amongst controllers and devices – are met throughout the architecture.

ANCHOR stands aside the payload SDN architecture, with its payload controllers and devices, not modifying but rather adding to it. It 'anchors' crucial functionality and properties, and 'hooks' to the former components, in order to secure the desired properties. So, on the devices, we just need the local counterparts to the ANCHOR middleware mechanisms and protocols, or HOOKS, to interpret and follow the ANCHOR's instructions.

After having made the case for logically-centralized non-functional property enforcement in software-defined networking, and presenting the outline of our general architecture, in the next two sections we introduce the use case we elected to show in this work, i.e., *logically-centralized security*. We start with a gap analysis that establishes the requirements for the architecture functionality in Section 4.1.2, and then, in Section 4.1.3, we show how to populate ANCHOR with the necessary mechanisms and protocols to meet those requirements.

4.1.2 Challenges

To elaborate on our 'security' case study, in this section we discuss, with more detail, the challenges brought in by the previously mentioned gaps — (i) security-performance; (ii) complexity-robustness; (iii) global security policies; and (iv) resilient roots-of-trust — as well as the requirements they put on a logically-centralized approach to enforcing security, as a non-functional system property.

4.1.2.1 Security *vs* performance

The security-performance gap comes from the conflict between ensuring high performance and using secure primitives. This gap affects directly the control plane communication, which is the crucial link between controllers and forwarding devices, allowing remote configuration of the data plane at runtime. Control channels need to provide high performance (high throughput and low latency) while keeping the communication secure.

The latency experienced by control plane communication is particularly critical for SDN operation. The increased latency is a problem *per se*, in terms of reduced responsiveness, but may also limit control plane scalability, which can be particularly problematic in large datacenters [19]. Most of the existing commercial switches already have low control plane performance on TCP (e.g., a few hundred flows/s [85], see Section V.A.). Adding cryptography worsens the problem: previous works have demonstrated that the use of cryptographic primitives has a perceivable impact on the latency of sensitive communication, such as VoIP [140] (e.g., TLS incurs in 166% of additional CPU cycles compared to TCP), network operations protocols such as SNMP [133], NTP [53], OpenFlow-enabled networks [84, 86], and HTTPS connections [107]. Perhaps not surprisingly, the number of SDN controllers and switching hardware supporting TLS (the protocol recommended by ONF to address security of control plane communication [111, 112]) is still reduced [4, 134]. Recent research has indeed suggested that one of the reasons for the slow adoption to be related with the security-performance trade-off [84].

Ideally, we would have both security robustness and performance on control plane channels. Considering the current scenario of SDN, it therefore seems clear the need to investigate lightweight alternatives for securing control plane communication. In the context of the security-performance gap, some directions that we point to in our architectural proposal ahead are, for instance, the careful selection of cryptographic primitives [84], and the adoption of cryptographic libraries exhibiting a good performance-security tradeoff, such as NaCl [24], or of mechanisms allowing per-message one-time-key distribution, such as iDVV [84, 86]. We return to these mechanisms later.

4.1.2.2 Complexity *vs* robustness

The complexity-robustness gap represents the conflict between the current complexity of security and cryptographic implementations, and the negative impact this has on robustness and hence correctness, hindering the ultimate goal.

In the past few years, studies have recurrently shown several critical misuse issues of cryptographic APIs of different TLS implementations [55, 40, 125]. One of the main root causes of these misuse issues is the inherent complexity of traditional cryptographic APIs and the knowledge required to use them without compromising security. For instance, more than 80% of the Android mobile applications make at least one mistake related to cryptographic APIs. Recent studies have also found different vulnerabilities in TLS implementations and have shown that longstanding implementations, such as OpenSSL[1], including its extensive cryptography, is unlikely to be completely verified in a near future [29, 56]. To address this issue, a few projects, such as miTLS [31] and Everest [30], propose new and verified implementations of TLS. However, several challenges remain to be addressed before having a solution ready for wide use [30].

[1]OpenSSL suffers from different fundamental issues such as too many legacy features accumulated over time, too many alternative modes as result of tradeoffs made in the standardization, and too much focus on the web and DNS names.

While the problem persists, the number of alarming occurrences proliferates. Recent examples include vulnerabilities that allow to recover the secret key of OpenSSL at a low cost [163], and timing attacks that explore vulnerabilities in both PolarSSL and OpenSSL [14, 39]. On the other hand, failures in classical PKI-based authentication and authorisation subsystems have been persistently happening [48, 123, 69], with the sheer complexity of those systems being considered one of the root causes behind these problems.
Considering the widely acknowledged principle that simplicity is key to robustness, especially for secure systems, we advocate and try to demonstrate in this work, that the complexity-robustness gap can be significantly closed through a methodic approach toward less complex but equally secure alternative solutions. NaCl [24], which we mentioned in the previous section, can be rightly called again in this context: it is one of the first attempts to provide a less complex, efficient, yet secure alternative to OpenSSL-like implementations. Mechanisms simplifying key distribution, authentication and authorization, such as iDVVs [84], could help mitigate PKIs' problems. By following this direction, we are applying the same principle of vulnerability reduction used in other systems, such as unikernels, where the idea is to reduce the attack surface by generating a smaller overall footprint of the operating system and applications [161].

4.1.2.3 Global security policies

The impact of the lack of global security policies can be illustrated with different examples. Although ONF describes data authenticity, confidentiality, integrity, and freshness as fundamental requirements to ensure the security of control plane communication, it does so in an abstract way, and these measures are often ignored, or implemented in an ad-hoc manner [134]. Another example is the lack of strong authentication and authorisation in the control plane. Recent reports show that widely used controllers, such as Floodlight and OpenDaylight, employ weak network authentication mechanisms [157, 134]. This leads to any forwarding device being able to connect to any controller. However, fake or hostile controllers or forwarding devices should not be allowed to become part of the network, in order to keep the network in healthy operation.
From a security perspective, it is non-controversial that device identification, authentication and authorization should be among the forefront requirements of any network. All data plane devices should be appropriately registered and authenticated within the network domain, with each association request between any two devices (e.g., between a switch and a controller) being strictly authorized by a security policy enforcement point. In addition, control traffic should be secured, since it is the fundamental vehicle for network control programmability. This begs the question: why aren't these mechanisms employed in most deployments?
A strong reason for the current state of affairs is the lack of global guiding and enforcement policies. It is necessary to define and establish global policies, and design, or adopt, the necessary mechanisms to enforce them and meet the essential requirements in order to fill the policy gap. With policies put in place, it becomes easier to manage all network elements, with respect to registration, authentication, authorization, and secure communication.

4.1.2.4 Resilient roots-of-trust

A globally recognized, resilient root-of-trust, could dramatically improve the global security of SDN, since current approaches to achieve trust are ad-hoc and partial [4]. Solving that gap would assist in fostering global mechanisms to ensure trustworthy registration and association between devices, as discussed previously, but the benefits would be ampler. For instance, a root-of-trust can be used to provide fundamental mechanisms (e.g., sources of strong entropy or pseudo-random generators), which would serve as building blocks for specific security functions.
As a first example, modern cryptography relies heavily on strong keys and the ability to keep them secret. The core feature that defines a strong key is its randomness. However, the randomness of keys is still a widely neglected issue [154], and not surprisingly, weak entropy, and weak random number

generation have been the cause of several significant vulnerabilities in software and devices [66, 10, 78, 67]. Recent research has shown that there are still non-negligible problems for hosts and networking devices [67, 10, 66]. For instance, a common pattern found in low-resource devices, such as switches, is that the random number generator of the operating system may lack the input of external sources of entropy to generate reliable cryptographic keys. Even long-standing cryptographic libraries such as OpenSSL have been recurrently affected by this problem [78, 115].
Similarly, as a second example, sources of accurate time, such as the local clock and the network time protocol, have to be secured to avoid attacks that can compromise network operation, since time manipulation attacks (e.g., NTP attack [98, 148]) can affect the operation of controllers and applications. For instance, a controller can be led to deliberately disconnect forwarding devices if it wrongly perceives the expiration of heartbeat message timeouts.
It is worth emphasizing that the resilient roots-of-trust gap lies exactly in the relative trust that can be put in partial, local, ad-hoc implementations of critical functions by controller developers and manufacturers of forwarding devices, in contrast to a careful, once-and-for-all architectural approach that can be reinstantiated in different SDN deployments. The list not being exhaustive, we claim that strong sources of entropy, resilient, indistinguishable-from-random number generators, and accurate, non-forgeable global time services, are fitting examples of such critical functions to be provided by logically-centralized roots-of-trust, helping close the former gap.

4.1.3 Security architecture

In this section we introduce the specialization of the ANCHOR architecture for logically-centralized security properties enforcement (Figure 4.2), guided by the conclusions from the previous section. Our main goal is to provide security properties such as authenticity, integrity, and confidentiality for control plane communication. To achieve this goal, the ANCHOR provides mechanisms (e.g., registration, authentication, a source of strong entropy, a resilient pseudo random number generator) required to fulfill some of the major security requirements of SDNs.
As illustrated in Figure 4.2, we "anchor" the enforcement of security properties on ANCHOR, which provides all the necessary mechanisms and protocols to achieve the goal. It is also a central point for enforcing security policies by means of services such as device registration, device association, controller recommendation, or global time, thereby reducing the burden on controllers and forwarding devices, which just need the local HOOKs, protocol elements that interpret and follow the ANCHOR's instructions.

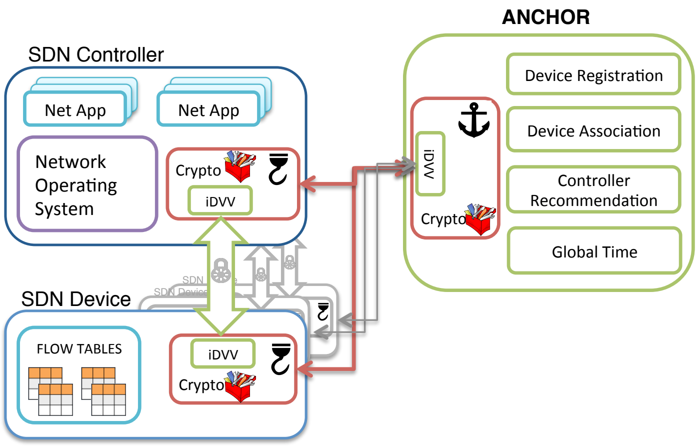


Figure 4.2: Logically-centralized Security

Next, we review the components and essential security services provided by ANCHOR. We first illustrate, in Section 4.1.3.1, how we implement our strategy of improving the robustness of ANCHOR as a single root-of-trust, by hardening ANCHOR in the face of failures. Next, we propose a source of

strong entropy (Section 4.1.3.2) and a resilient pseudo random generator (PRG) (Section 4.1.3.3) for generating security-sensitive materials. These are crucial components, as attested by the impact of vulnerabilities discovered in the recent past, in sub-optimal implementations of the former in several software packages [23, 104, 171, 131]. We implement and evaluate the robustness of these mechanisms. We also leverage on a mechanism, the integrated device verification value (iDVV), to simplify authentication, authorization and key generation amongst SDN components [84], which we review and put in the context of ANCHOR (Section 4.1.3.4). This particular mechanism will be detailed later in Section 4.2, but we give a brief overview here. The iDVV protocol runs between the ANCHOR, and the HOOKs in controllers and switching devices. We implement and evaluate iDVV generators for OpenFlow-enabled control plane communication. Next, we present three essential services for secure network operation — device registration (Section 4.1.3.6), device association (Section 4.1.3.7), and controller recommendation (Section 4.1.3.8) — and we describe how the above mechanisms interplay with our secure device-to-device communication approach (Section 4.1.3.9).
Concerning the mitigation of possible (though expectedly infrequent) security failures, across the explanation of the algorithms we observe how several robustness measures work, like for example the achievement of PFS, protecting pre-compromise communications in the presence of successful attacks. Finally, in Section 4.1.3.10, we see the capstone of these measures, explaining how to re-establish secure communication channels in a semi-automatic way, after ANCHOR has been reinstated in the sequel of compromise.
The roster of services of ANCHOR is not closed, and one can think of other functionalities, not described here, including keeping track of forwarding devices association, generating alerts in case of strange behaviors (e.g., recurrent reconnections, connections with multiple controllers), and so forth. These ancillary management tasks are important to keep track of the network operation status. In what follows, we describe the above components in detail.

4.1.3.1 Hardening anchor

The compromise of a root-of-trust is of great concern, since crucial services normally depend on it being secure and dependable. As we stated in the introduction, we have a long-term strategy towards the resilience of ANCHOR, which starts by improving the inherent reliability of its simplex (non-replicated) version, by hardening it in the face of failures, namely, by still providing some security guarantees even when ANCHOR has been compromised. In particular, we propose protocols to achieve two security properties guaranteeing respectively, the security of past (pre-compromise) communications, and of future (post-recovery) communications. This provides a significant improvement over other existing root-of-trust infrastructures.
The first security property is *perfect forward secrecy (PFS)*, namely, the assurance that the compromise of all secrets in a current session does not compromise the confidentiality of the communications of the past sessions. The enforcement of PFS is systematically approached in the algorithms we present next.
The second property is *post-compromise security (PCS)*. While PFS considers how to protect the past communications, PCS considers how to automatically reinstate and re-establish the secure communication channels, for future communications. This security property has so far been considered only in the specific scenario of secure messaging [167], and only limited works [165, 166] are available. In particular, we consider that when ANCHOR has been compromised by an attacker (e.g., through the exploitation of software vulnerabilities), and has been reinstated by the operator (e.g., by applying software patches and rebuilding servers), the system should have a way to automatically re-establish secure communications between ANCHOR and all other participants, without having to reinstate these components (controllers and forwarding devices in this case, whose shared secrets became compromised).
In summary, even though ANCHOR is a single root-of-trust in our system, we mitigate the associated risks by guaranteeing:

- PFS: the compromise of ANCHOR in the current session does not expose *past communications*;
- PCS: when ANCHOR is compromised and reinstated, ANCHOR can automatically re-establish secure communication channels with all other participants in the system to protect the security of *future communications*.

As a side note, since our system only uses symmetric key cryptography, it will stand up even against an attacker with quantum computers. In other words, our infrastructure will be *post-quantum secure (PQS)*.

4.1.3.2 A source of strong entropy

Entropy still represents a challenge for modern computers because they have been designed to behave deterministically [154]. Sources of true randomness can be difficult to use because they work differently from a typical computer.
To avoid the pitfalls of weak sources of entropy, in particular in networking devices, ANCHOR provides a source of strong entropy to ensure the randomness required to generate seeds, pseudorandom values, secrets, among other cryptographic material. The strong source of entropy, implemented by Algorithm 4, has the following property:
Strong Entropy - Every value *entropy* returned by the function *entropy_get* is indistinguishable-from-random.

Algorithm 4: Source of strong entropy

```
 1: entropy_setup(data)
 2:    e_entropy ← rand_bytes() $\oplus$ H(data)
 3:    i_entropy ← rand_bytes() $\oplus$ e_entropy

 4: entropy_update()
 5:    e_entropy ← H($P_i||P_j$) $\oplus$ i_entropy
 6:    E_counter ← 0

 7: entropy_get()
 8:    if E_counter >= MAX_LONG call entropy_update()
 9:    i_entropy ← H(rand_bytes() || E_counter)
10:    entropy ← e_entropy $\oplus$ i_entropy
```

Algorithm 4 shows how the external (from other devices) and internal (from the local operating system) sources of entropy are kept updated and used to generate random bytes per function call (*entropy_get()*). The state of the internal and external entropy is initially set by calling the *entropy_setup(data)*. This function requires an input data, which can be a combination of current system time, process number, bytes from special devices, among other things, and random bytes (*rand_bytes()*) from a local (deterministic) source of entropy (e.g., */dev/urandom*) to initialize the state of the entropy generator. As we cannot assume anything regarding the predictability of the input data, we use it in conjunction with a *rand_bytes()* function call (line 2). A call to *rand_bytes()* is assumed to return (by default) 64 bytes of random data.
Function *entropy_update()* uses as input the statistics of external sources and the ANCHOR's own packet arrival rate to update the external entropy. The noise (events) of the external sources of entropy is stored in 32 pools (P_0, P_1, P_2, P_3, ..., P_{31}), as suggested by previous work [58]. Each pool has an event counter, which is reset to zero once the pool is used to update the external entropy. At every update, two different pools of noise (P_i and P_j) are used as input of a hashing function H. The two pools of noise can be randomly selected, for instance. The output of this function is XORed with the internal entropy to generate the new state of the external entropy. It is worth emphasizing that *entropy_update()* is automatically called when *E_counter* (the global event counter) reaches its maximum value and whenever needed, i.e., the user can define when to do the function call.
The resulting 64 bytes of entropy, indistinguishable-from-random bytes (*entropy_get()*), are the outcome of an XOR operation between the external and internal entropy. While the external entropy

provides the unpredictability required by strong entropy, the internal source provides a good, yet predictable [154], continuous source of entropy. At each time the *entropy_get()* function is called, the internal entropy is updated by using a local source of random data, which is typically provided by a library or by the operating system itself, and the global number of events currently in the 32 pools of noise ($E\_counter$). These two values are used as input of a hashing function H.
Such sources of strong entropy can be achieved in practice by combining different sources of noise, such as the unpredictability of network traffic [65], the unpredictability of idleness of links [20], packet arrival rate of network controllers, and sources of entropy provided by operating systems. We provide implementation details in Section 4.1.4.1. A discussion about the correctness of Algorithm 4 can be found in appendix A of [87].

4.1.3.3 Pseudorandom generator (PRG)

A source of entropy is necessary but not sufficient. Most cryptographic algorithms are highly vulnerable to the weaknesses of random generators [52]. For instance, nonces generated with weak pseudo-random generators can lead to attacks capable of recovering secret keys. Different security properties need to be ensured when building strong pseudo-random number generators (PRG), such as resilience, forward security, backward security and recovering security. In particular, the latter was recently proposed as a measure to recover the internal state of a PRG [52]. We propose a PRG that uses our source of strong entropy and implements a refresh function to increase its resilience and recovering capability.
The pseudo-random number generator, implemented by Algorithm 5, has the following property:
Robust PRG - Every value *nprd* returned by the function *PRG_next* is indistinguishable-from-random.
A robust PRG needs three well-defined constructions, namely *setup()*, *refresh()* (or re-seed), and *next()*, as described in Algorithm 5. The internal state of our PRG is represented by three variables, the *seed*, the *counter* and the next pseudo-random data *nprd*. The setup process generates a new seed, by using our strong source of entropy, which is used to update the internal state. In line 3, we initialize the *counter* by calling the $long\_uint()$ function, which returns a long unsigned int value that will be used to re-seed and to generate the next pseudorandom value. In line 4, we call $entropy\_update()$ to make sure that the external entropy gets updated before calling one more time the $entropy\_get()$ function. The first *nprd* is the outcome of an XOR operation between the newly generated seed and a second call to our source of entropy. It is worth emphasizing that the set up of the initial state of the PRG does not require any intervention or interaction with the end user. We provide strong and reliable entropy to set up the initial values of all three variables. This ensures that our PRG is non-sensitive to the initial state. For instance, in a tradicional PRG the user could provide an initial seed, or other setup values, that could compromise the quality of the generator's output. The *counter*, which is concatenated with the *nprd* (lines 9 and 13), gives the idea of an unbounded state space [147]. This is possible because we are using cryptographically strong primitives such as a hash function H and the MAC function HMAC. Thus, in theory, we have unbounded state spaces, i.e., we can keep concatenating values to the input of these primitives.
The *PRG_refresh()* function updates the internal state, i.e., the *seed*, the *counter* and the *nprd*. It uses H to update the state of the *nprd*. Finally, the *PRG_next()* function outputs a new, indistinguishable-from-random stream of bytes, applying HMAC on the internal state. In this function, the *counter* is decremented by one. The idea is for it to start with a very large unsigned 8-bytes value, which is used until it reaches zero. At this point, the *PRG_refresh()* function will be called to update the internal state of the generator. The newly generated *nprd* is the outcome of an HMAC function with a dimension of 128 bits.
The main motivation for having a PRG along with a strong source of entropy is speed. Studies have shown that entropy generation for local use can be rather slow, such as 1.5 seconds to 2 minutes for generating 128 bits of entropy [97]. Our source of entropy uses external entropy and random bytes from special devices, whereas the PRG uses an HMAC function, in order to have a fast and reliable generation of pseudo-random values.

Algorithm 5: Pseudo-random number generator

```
1: PRG_setup()
2:    seed ← entropy_get()
3:    counter ← long_uint(entropy_get())
4:    call entropy_update()
5:    nprd ← seed ⊕ entropy_get()

6: PRG_refresh()
7:    seed ← entropy_get()
8:    counter ← long_uint(entropy_get())
9:    nprd ← H(seed ‖ nprd ‖ counter)

10: PRG_next()
11:    counter ← counter - 1
12:    if counter <= 0 call PRG_refresh()
13:    nprd ← HMAC(seed, nprd ‖ counter)
```

In spite of the fact that we could use any good PRG to generate cryptographic material (e.g. keys, nonce), it is worth emphasizing that we introduce a PRG that works in a seamless way with our strong source of entropy, improving its quality. In Section 4.1.4.2, we discuss the specifics of the implementation. We also evaluate the robustness and level of confidence of our algorithms in Section 4.1.5.1. A discussion about the correctness of Algorithm 5 can be found in appendix B of [87].

4.1.3.4 Integrated device verification value

The design of our logically-centralized security architecture also includes the integrated device verification value (iDVV) component [84]. The iDVV idea was inspired by the iCVVs (integrated card verification values) used in credit cards to authenticate and authorize transactions in a secure and inexpensive way. In Section 4.2 we show how we applied this concept to SDN, proposing a flexible method of generating iDVVs that can be safely used to secure communication between any two devices, but we leave a short description here. iDVVs can be used to partially address two gaps of non-functional properties, security-performance and complexity-robustness.
An iDVV is a unique value generated by device A (e.g., forwarding device) which can be verified by device B (e.g., controller). An iDVV generator has essentially two interfaces. First, *idvv_setup (seed, secret)*, which is used to set up the generator. It receives as input two secret, random and unique values, the seed and the (higher-level protocol dependent) secret. The source of strong entropy and the robust PRG are, amongst other things, used to bootstrap and keep the iDVV generators fresh. Second, the *idvv_next()* interface returns the next iDVV. This interface can be called as many times as needed.
So, iDVVs are sequentially generated to authenticate and authorize requests between two networking devices, and/or protect communication. Starting with the same seed and secret, the iDVV generator will generate, for example, at both ends of a controller-device association, the exact same sequence of values. In other words, it is a deterministic generator of authentication or authorization codes, or one-time keys, which are, however, indistinguishable from random. The main advantages of iDVVs are their low cost, which makes them even usable on a per-message basis, and the fact that they can be generated off-line, i.e., without having to establish any previous agreement.
Correctness. The randomness and performance of the iDVV algorithm as deterministic generator of authentication or authorization codes, or one-time keys which are however indistinguishable from random, are analyzed, and its properties proved, in Section 4.2 and [84]. The performance study is also complemented in that section. Overall, these analyses show that iDVVs are robust, achieve a high level of confidence and outperform traditional key generation and derivation functions without compromising the security.

4.1.3.5 System roles and setup

Let us assume the roles of *system administrator*, controlling the operation of central services such as ANCHOR, and *network administrator*, controlling the operation of network devices. Each time a new network device (a forwarding device or a controller) is added to the network, it must first be registered, before being able to operate.
In the current practice, the device registration is a manual process triggered by a network administrator through an out-of-band channel. This process would involve manual work from both the system and the network administrators. Given the potentially large number of network devices in SDN, such a manual process is unsatisfactory.
Thus, we propose a protocol, described below, to fulfill the desire of a semi-automated device registration process, which is efficient, secure, and requires the least involvement of ANCHOR. The ANCHOR is first setup by the system admin. Next, each network device is set up by ANCHOR. Devices just need that a key shared with their overseeing network admin is set up initially, at first use. The set up of this key and the registration of devices is described in Section 4.1.3.6. Then, devices can be registered automatically.
For simplicity and without loss of generality, in what follows we denote $E_{XY}()$ an encryption using encryption key Ke_{XY}, and we denote [],$HMAC_{XY}$, respectively a message field inside [], followed by an HMAC over the whole material within [], using MAC key Kh_{XY}, where $X, Y \in \{A, D_i, M, C, F\}$ (*Notation:* Anchor, Device i, Manager, Controller, Forwarding device). When $X = A$, we omit X for simplicity. For example, we use E_M(msg) to refer E_{AM}(msg), and they both denote the ciphertext of encrypting *msg* under key Ke_{AM}. In what follows, ANCHOR can generate strong keys using a suitable key derivation function (KDF) based on the high entropy random material described in the previous sections.
Now we present the set up required for ANCHOR, network admin, and device. After that, we describe the device registration and association algorithms, respectively Algorithms 6 and 7.
ANCHOR setup. The ANCHOR needs two master recovery keys, namely the master recovery encryption key Ke_{rec} and master recovery MAC key Kh_{rec}, fundamental for the post-compromise recovery steps described ahead. However, these two master recovery keys, in possession of the authority overseeing ANCHOR (the system administrator), must never appear in the ANCHOR server (if they are to recover from a possible full server compromise), being securely stored and used only in an offline manner [2].
As we will present later, the master recovery keys are only used in two cases, namely (a) when a new network admin is registered with ANCHOR (i.e. the network admin setup process); and (b) when ANCHOR was compromised and is reinstated into a trustworthy state (i.e. the post-compromise recovery process presented in §4.1.3.10). When either case occurs, the ANCHOR authority only needs to use the master recovery keys once, to recursively compute the recovery keys of all devices and network admins. The output of the calculation will be imported into the ANCHOR server through an out-of-band channel (e.g. by using a USB).
Network admin setup. Each network administrator (or manager, denoted M) with identity M_ID is registered with ANCHOR manually. This is the *only manual process* to initialize a new network administrator. Afterwards all devices managed by this administrator can be registered with ANCHOR through our device registration protocol.
During the network admin registration phase, ANCHOR locally generates encryption key Ke_{AM} and MAC key Kh_{AM} to be shared with M, and they are manually imported into M through an out-of-band channel (again, by using a USB, for example).
Further, M recovery keys $Kre_{AM} = \mathrm{H}(Ke_{rec}||\mathrm{M\_ID})$ and $Krh_{AM} = \mathrm{H}(Kh_{rec}||\mathrm{M\_ID})$ are also computed by ANCHOR offline. M recovery keys live essentially offline, since M needs to perform only infrequent operations with these keys (e.g. upon device registration). Note that ANCHOR does not

[2] Just to give a real feel, one possible implementation of this principle is: a pristine ANCHOR server image is created; it boots offline in single user mode; it generates Ke_{rec} and Kh_{rec} through a strong KDF as discussed above; keys are written into a USB device, and then deleted; first online boot proceeds.

store Kre_{AM} or Krh_{AM} as well, but can recompute them offline when the post-compromise recovery process is triggered, as we detail in Section 4.1.3.10.

Device setup. A device with identity D_i is either a forwarding device (F) or a controller (C), but we do not differentiate them during the set up and registration processes. The first operation to be made after a device is first brought to the system is the setup, which concerns the establishment of credentials, for secure management access by the network administrator.

Upon request from M, ANCHOR locally generates a pair of keys for each device D_i being set up, Ke_{MD_i} and Kh_{MD_i}, to be respectively the encryption and MAC key to be shared between M and D_i, for management. They are sent to M under the protection of Ke_{AM} and Kh_{AM}. Then, they are manually imported by the network admin into each D_i through an out-of-band channel.

4.1.3.6 Device registration

The device registration protocol is presented in Algorithm 6. We assume that Ke_{MD_i} and Kh_{MD_i} described above are in place.

Algorithm 6: Device registration

```
                 {Bootstrap for devices $D_1 - D_n$ }
1.  M → A        [Reg, M_ID, $E_M(\{(D_i, x_m^i)\}_{i=1}^n)$], $HMAC_M$
2.  A            for each $D_i$, generate $Ke_{AD_i}, Kh_{AD_i}, x_a^i$
3.  A → M        [Reg, M_ID, $E_M(\{(D_i, x_m^i, x_a^i, Ke_{AD_i},$
                 $Kh_{AD_i})\}_{i=1}^n)$], $HMAC_M$
                 {For each device $D_i$}
4.  M            $Kr_{AD_i} \leftarrow$ H($Kr_{AM}||D_i$)
5.  M → $D_i$    [Reg, $E_{MD_i}(x_a^i, Ke_{AD_i}, Kh_{AD_i}, Kr_{AD_i})$],
                 $HMAC_{MD_i}$
6.  $D_i$ → A    [M_ID, $D_i, E_{D_i}(x_a^i)$], $HMAC_{D_i}$
7.  A → M        [M_ID, $D_i, E_M(x_a^i)$], $HMAC_M$
8.  A            tag($D_i$) = registered;
9.               for $t \in \{C, F\}$, if Type($D_i$)==t, then $tList =$
                 $tList \cup \{Di\}$
10.              $\forall i \in [1, n]$, if tag($D_i$) == registered is True
11.                  $Ke_{AM}$ = H($Ke_{AM}$); $Kh_{AM}$ = H($Kh_{AM}$).
12. M → $D_i$    [$D_i, E_{MD_i}(x_a^i)$], $HMAC_{MD_i}$
13. M            tag($D_i$) = registered;
14.              destroys ($Ke_{AD_i}, Kh_{AD_i}, Kr_{AD_i}$);
15.              $Ke_{MD_i}$ = H($Ke_{MD_i}$); $Kh_{MD_i}$ = H($Kh_{MD_i}$);
16.              $\forall i \in [1, n]$, if tag($D_i$) == registered is True
17.                  $Ke_{AM}$ = H($Ke_{AM}$); $Kh_{AM}$ = H($Kh_{AM}$).
18. $D_i$        $Ke_{MD_i}$ = H($Ke_{MD_i}$); $Kh_{MD_i}$ = H($Kh_{MD_i}$).
```

The first part concerns the bootstrap of the registration of a batch of devices with ANCHOR (A), by a

network admin M. Let $\{D_i\}_{i=1}^n$ be the set of n device identities that the admin wants to register. M requests (line 1) the registration to A, accompanying each D_i with a nonce x_m^i. A computes its own nonce x_a^i, and keys Ke_{AD_i}, Kh_{AD_i}, for each D_i, and returns them encrypted to M (lines 2,3). The random nonces x_m^i and x_a^i are used to prevent replay attacks.
The process then follows for each device D_i. First, the device recovery key is created (line 4), using M's recovery key Kr_{AM}. Then M sends D_i the relevant crypto keys (line 5). Device D_i follows-up confirmation to A, which closes the loop with M, using the original nonce from A (lines 6,7). A then performs a set of operations (lines 8-11) to commit the registration of D_i, namely by inserting it into the controller or forwarding device list, respectively CList or FList, and updating several keys.
Note that in Algorithm 6, the update of several shared keys (i.e., lines 11, 15, 17, 18) at the end of the registration steps at A, M, and D_i, is used to provide PFS. When a key is updated, the old one is destroyed. Continuing, in line 12 M closes the loop with D_i, using the original nonce from A, finally confirming D_i's registration. Upon this step, both M and D_i perform the key update just mentioned. Note that the generation process of the recovery key Kr_{AD_i} lies with M (line 4), though using its recovery key shared with ANCHOR, Kr_{AM}. This reduces the number of uses of the master recovery key. However, as we will see, albeit not knowing Kr_{AD_i} and Kr_{AM}, ANCHOR can easily compute them offline, if needed. Second, Kr_{AM} possessed by the network admin is only used when new devices need to be registered. So, Kr_{AM} can be usually stored offline. This provides an extra layer of security.

4.1.3.7 Device association

The association service is required for authorizing control plane channels between any two devices, such as a forwarding device and a controller. A forwarding device has to request an association with a controller it wishes to communicate with. This association is mediated by the ANCHOR.
The association process between two devices is performed by the sequence steps detailed in Algorithm 7. Registered controllers and forwarding devices are inserted in *CList* and *FList*, respectively. *Notation:* As explained above, the registration process set in place shared secret keys between ANCHOR (A) and any controller C or forwarding device F.

Algorithm 7: Device association

1

		{Of forwarding device F with controller C}
1.	F → A	[x_g, F, GetCList],$HMAC_F$
2.	A → F	[x_g, F, E_F(CList(F), x_g)],$HMAC_F$
3.	F → C	x_g, GetAiD, F, C, E_F(GetAiD, F, C, x_f, x_g)
4.	C → A	[x_g, GetAiD, F, C, E_F(GetAiD, F, C, x_f, x_g), E_C(GetAiD, F, C, x_c, x_g)],$HMAC_C$
5.	A → C	[x_g, E_F(x_f, AiD), E_C(x_c, AiD)],$HMAC_C$
6.	A	destroys (AiD)
7.	C → F	x_g, E_F(x_f, AiD), E_{AiD}(SEED, x_g)
8.	F → C	x_g, E_{AiD}(SEED $\oplus$ x_g)
9.	A, F	Ke_{AF} = H(Ke_{AF}); Kh_{AF} = H(Kh_{AF})
10.	A, C	Ke_{AC} = H(Ke_{AC}); Kh_{AC} = H(Kh_{AC})

The device association implemented by Algorithm 7, has the following properties:
Controller Authorization - Any device F can only associate to a controller C authorized by the ANCHOR.
Device Authorization - Any device F can associate to some controller, only if F is authorized by the ANCHOR.
Association ID Secrecy - After termination of the algorithm, the association ID (AiD) is only known to F and C.
Seed Secrecy - After termination of the algorithm, the seed ($SEED$) is only known to F and C.

The algorithm coarse structure follows the line of the Needham-Schroeder (NS) original authentication and key distribution algorithm [108], but contemplates anti-replay measures such as including participant IDs, and a global initial nonce as suggested in [116]. Unlike NS, it uses encrypt-then-MAC to further prevent impersonation. Furthermore, it is specialized for device association, managing authorization lists, and distributing a double secret in the end (association ID and seed). Secure communication protocols running after association can, as explained below in Section 4.1.3.9, use iDVVs on a key-per-message or key-per-session basis, rolling from the initial seed and secret association ID.
The association process starts with a forwarding device (F) sending an association request to the ANCHOR (A) (line 1 in Algorithm 7). This request contains a nonce x_g, the identification of the device and the operation request *GetCList* (get list of controllers). The request also contains an HMAC to avoid device impersonation attacks. The ANCHOR checks if F is in FList (registered devices), and if so, it replies (line 2) with a list of controllers (CList(F)) which F is authorized to associate with. The list of controllers (and the nonce x_g) is encrypted using a key (set up during registration) shared between A and F. This protects the confidentiality of the list of controllers, and x_g ensures that the message is fresh, providing protection against replay attacks. A message authentication code also protects the integrity of the ANCHOR's reply, avoiding impersonation attacks.
Next, F sends an association request to the chosen controller C (line 3). The request contains a message that is encrypted using a key shared between F and A. This message contains the get association id (*GetAiD*) request, the identity of the principals involved (F,C), a nonce x_f, and binds to the nonce x_g. The controller forwards this message to A (line 4), adding its own encrypted association request field, similar to F's, but containing C's own noncex_c instead. This prevents the impersonation of the controller since only it would be able to encrypt the freshly generated x_g.
In line 5, C trusts that A's reply is fresh because it contains x_g. The controller also trusts that it is genuine (from A) because it contains x_c. As such, C endorses F as an authorized device and AiD as the association key for F. Future compromise of A should not represent any threat to established communication between C and F. To achieve this goal, A immediately destroys the AiD (line 6) and C and F further share a seed not known by A (line 7).
C forwards both the encrypted AiD message and its seed to F (line 7). The forwarding device trusts that this message is fresh and correct because it contains x_g, and x_f under encryption, together with the AiD, only know to F and C, which it endorses then as the association key. F trusts that C is the correct correspondent, otherwise A would not have advanced to step 5. That being true, future interactions will use AiD. F believes that the $SEED$ is genuine, as random entropy for future interactions, because it is encapsulated by AiD, known only to C and F. The forwarding device also trusts that the message is fresh because it contains x_g.
Finally (line 8), C trusts it is associated with F (as identified in step 3 and confirmed by A in step 5), when F replies showing it knows both the AiD and the $SEED$, by encrypting the $SEED$ XOR'ed with the current nonce x_g, with AiD. Replay and impersonation attacks are avoided because all encrypted interactions are dependent on nonces, so will become void in the future.
At the end of each device association protocol, all keys shared between a device (F or C) and ANCHOR will be updated to the hash value of this key (lines 9, 10). Again, this is used to provide perfect forward secrecy. All nonces are random, i.e., not predictable.
A discussion of the correctness of Algorithm 7 can be found in Appendix C of [87].

4.1.3.8 Controller recommendation

Similarly to moving target defense strategies [158], devices (e.g., controllers) are hidden by default, i.e., only registered and authenticated devices can get information about other devices. Even if a forwarding device finds out the IP of a controller, it will be able to establish a connection with the controller only after being registered and authorized by the ANCHOR.
Controllers can be recommended to forwarding devices using different parameters, such as latency, load, or trustworthiness. When a forwarding device requests an association with one or more controllers, the ANCHOR sends back a list of authorized controllers to connect with. The forwarding device will be

restricted to associate itself with any of the controllers on the list. In other words, forwarding devices will not be allowed to establish connections with other (e.g., hostile or fake) controllers. Similarly, fake forwarding devices will be, by default, forbidden to set up communication channels with non-compromised controllers.

4.1.3.9 Device-to-device communication

Communication between any two devices happens only after a successful association. Consider the end of an association establishment, as per Algorithm 7, e.g. between a controller C and a forwarding device F: at this point, both sides, and only them, have the secret and unique material $(SEED, AiD)$ (as proved in Appendix C of [87]). Using them, they can bootstrap the iDVV protocol (see Section 4.1.3.4 above), which from now on can be used at will by any secure communication primitives. As explained earlier, iDVV generation is flexible and low cost, to allow the generation: (a) on a per message basis; (b) for a sequence of messages; (c) for a specific interval of time; or (d) for one communication session. NaCl [24], as mentioned in previous sections, is a simple, efficient, and provably secure alternative to OpenSSL-like implementations, and is thus our choice for secure communication amongst controllers and devices.
Researchers have shown that NaCL is resistant to side-channel attacks [11] and that its implementation is robust [24]. Different from other cryptographic libraries, NaCL's API and implementation is kept very simple, justifying its robustness. Through ANCHOR, the SDN communication channels are securely encrypted using symmetric key ciphers provided by NaCl, with the strong cryptographic material required by the ciphers generated by our mechanisms, allowing secret codes per packet, session, time interval, or pre-defined ranges of packets.

4.1.3.10 Post-compromise recovery

As previously mentioned, after ANCHOR has been reinstated in the sequel of a compromise, it is crucial to have a way to automatically re-establish the secure communication channels between ANCHOR and all participants.
Algorithm 8 presents how to re-establish the secure communication channels when ANCHOR is compromised. Intuitively, since ANCHOR's master recovery keys Ke_{rec} and Kh_{rec} are stored securely offline, these keys are unknown to the attacker who has stolen all secrets from the ANCHOR server. As described before, all M and all D_i recovery keys can be recursively computed from the master keys, offline (line 1). Once this done, the operator imports those keys into the ANCHOR server. To continue the recovery process, ANCHOR generates new random keys to be shared with all Ms, and all D_i (line 2).
Now ANCHOR can send to each M (line 3) a recovery message to re-share keys (contained in M_k) both between that manager and the devices controlled by it. The messages are secured by using the according recovery keys. The new shared keys will be used to protect future communications. Each M implements the operation with each of the devices it manages (line 4).
The new keys replace the possibly compromised keys at M and each D_i (lines 5-6, and 9). Likewise, when the recovery process has been completed, the recovery keys will be updated to their hash value (lines 7-8, and 10-11). As mentioned previously, this key update is used to provide perfect forward secrecy (PFS).
Note that at line 3, an additional MAC value on the entire message under the current MAC key Kh_{AM} is created. Since the recovery keys are stored offline, without having this additional MAC value, the manager will have to perform the verification offline manually. This MAC value prevents possible DoS attacks where an attacker creates and sends fake recovery messages to network managers, as this additional MAC value can be verified online efficiently, and it cannot be created without having access to the current MAC key Kh_{AM}.
In case of compromise of a manager M, once it has recovered, it can also re-establish its shared secrets with ANCHOR and associated devices in a similar way as described above. A recovery is excluded,

Algorithm 8: ANCHOR recovery.

```
                 {For each manager $M$ and its associated devices $\{D_i\}_{i=1}^{n}\}$}
   1.   A              computes $Kre_{AM}$, $Krh_{AM}$, $Kre_{AD_i}$, $Krh_{AD_i}$;
   2.                  generates $M_k = (Ke'_{AM}, Kh'_{AM}, \{Ke'_{AD_i}, Kh'_{AD_i}\}_{i=1}^{n})$.
   3.   A → M          $[Recovery, A, M\_ID, [E_{Kre_{AM}}(M_k)], HMAC_{Krh_{AM}}], HMAC_M$.
                 {For each device $D_i$}
   4.   M → $D_i$      $[Recovery, A, M\_ID, D_i, E_{Kre_{AD_i}}(Ke'_{AD_i}, Kh'_{AD_i})], HMAC_{Krh_{AD_i}}$.
1  5.   M              destroys $Ke'_{AD_i}, Kh'_{AD_i}$;
   6.                  $Ke_{AM} = Ke'_{AM}$; $Kh_{AM} = Kh'_{AM}$;
   7.                  $Kre_{AM} = H(Kre_{AM})$; $Krh_{AM} = H(Krh_{AM})$;
   8.                  $Ke_{MD_i} = H(Ke_{MD_i})$; $Kh_{MD_i} = H(Kh_{MD_i})$.
   9.   $D_i$          $Ke_{AD_i} = Ke'_{AD_i}$; $Kh_{AD_i} = Kh'_{AD_i}$;
   10.                 $Kre_{AD_i} = H(Kre_{AD_i})$; $Krh_{AD_i} = H(Krh_{AD_i})$;
   11.                 $Ke_{MD_i} = H(Ke_{MD_i})$; $Kh_{MD_i} = H(Kh_{MD_i})$.
```

and the next steps are made only for a single M instead of all M, and with some differences: M gets the recovery keys (line 1) from ANCHOR through an out-of-band channel, Kre_{AM}, Krh_{AM}, and all Kre_{AD_i}, Krh_{AD_i} from $i = 1ton$. These keys remain the same, but M had lost them, having been rebuilt from scratch. Then, in lines 2-3, M will get (generated by A) the Ke'_{MD_i}, Kh'_{MD_i} keys for managing all devices, instead of Ke'_{AD_i}, Kh'_{AD_i}, which do not need to be changed. In line 4, the former are sent to each D_i, instead of the latter.

4.1.4 Implementation

A prototype of ANCHOR has been implemented as envisioned in Figure 4.2. Our implementation, using the NOX controller and CBench[3] (OpenFlow switches emulator), has approximately 2k lines of Python code and 700 lines of C code (integration with CBench). It uses Google's protobuf [64] for defining the communication protocols and efficiently serializing the data. In this section we give an overview of some important system implementation details.

4.1.4.1 A source of strong entropy

Each external source of noise (e.g., forwarding device, controller) sends heartbeats to the ANCHOR. Each heartbeat carries statistics of the current network traffic, idleness of links, and number of packets received by a controller within a specific time frame.
Recall from Algorithm 4 that for setting up the external entropy, the bytes read from the local source are combined (through an XOR operation) with the output of hashing function H(*data*). We have chosen SHA512 as our strong hashing function H [50]. After that, a second read of local random bytes is XORed with the external entropy to setup the internal entropy.
For implementing the *entropy_update()*, one can use the pools of noise in a circular approach (e.g., P_0 and P_1, P_2 and P_3, and so forth), in a combined circular and random way (P_0 and P_7, P_1 and P_{31}, and so forth), or in a completely random fashion, for instance. Using several pools of events, we create enough data to make it nearly impractical for an attacker to enumerate the possible values for the events used to update the generator's internal state [58]. In other words, the attacker will arguably be unable to rebuild the internal state of the source of strong entropy.
Even if an attacker is controlling two or more external sources in a timely manner, it will be hard to guess the new state of the external entropy. First, the attacker needs to enumerate the events of the pools being used on each update. This, by itself, is something hard to achieve since the attacker does not know the update sequence of these pools, i.e., which external sources are being used, in which sequence, to update each pool. In other words, he/she would have to know all sources of noise,

[3]CBench is the default and most widely used tool for benchmarking control plane performance [137, 77, 172].

and the sequence in which they are being used to update the pools. It is also worth emphasizing that the external sources need to have a pre-defined maximum rate for sending the heartbeats, i.e., compromised sources cannot send data at a higher frequency to influence subsequent updates of the external entropy. Second, the attacker would need to have additional knowledge regarding the internal entropy, which is a result of two combined values, as explained in the following paragraph.
Pools of noise. The 32 pools of events are feed by four different sources, (1) incoming packet rate sent by controllers; (2) incoming packet rate of ANCHOR; (3) network statistics of forwarding devices; and (4) random bytes from local systems. Each of the source feeds the pools in its own way. For instance, sources (1) and (3) use round-robin, while sources (2) and (4) use a random approach to select the next pool to put the new event in. In this way, we have a diversity of approaches for feeding the pools of noise, making the "guessing task" of an attacker even harder. Each pool needs to store only a single value, the digest of a hashing function (e.g., SHA512). The current digest and the newly arrived events are used as input of the hashing function. Lastly, once the pool has been used by the source of strong entropy, it is reset to a new initial state, which consists of the digest of a hash function using as input random bytes of a local entropy source such as */dev/urandom*.

4.1.4.2 Pseudorandom generator (PRG)

Our PRG, as was shown in Algorithm 5, combines the strengths of different solutions such as the PRF of SPINS [118] (which is based on an HMAC function), provably secure constructions for building robust PRGs [52, 58], and unbounded state spaces through cryptographic primitives [147].
As HASH function we have chosen SHA512. As HMAC function, we have chosen the one time authentication function *crypto_onetimeauth()* from NaCl [24]. It ensures security and performance while generating outputs of 16 bytes that are indistinguishable from random.
PRG at the controller. As the controller might not have a source of strong entropy, we implemented a slightly modified version of Algorithm 5. Essentially, we replace the *entropy_get()* function by *entropy_remote()*. This function makes an entropy request to the ANCHOR. This means that the recovering security by refreshing, which makes a PRG more resilient, is using our source of strong entropy. With this approach, we are strengthening the controller's PRG.

4.1.4.3 iDVV generators

Based on the algorithm we will detail in Section 4.2, we have implemented an iDVV generator that supports seven different cryptographic primitives. In this case, the *idvv_next(primitive_id)* also has an input, which is the id of the primitive to be used. In the implementation, we used the following primitives to generate the iDVVs: *MD5*, *SHA1*, *SHA512*, *SHA256*, *poly1305aes_ authenticate*, *crypto_onetimeauth*, and *crypto_hash*. While the first four functions are provided by OpenSSL, the last three are provided by an independent implementation of Poly1305-AES and NaCl. As MD5 and SHA1 are deprecated, we use them only for performance comparison purposes.

4.1.5 Evaluation

In this section we evaluate the essential security mechanisms and services of our architecture.
For the performance measurements, we used machines with two quad-core Intel Xeon E5620 2.4GHz, with 2x4x256KB L2 / 2x12MB L3 cache, 32GB SDIMM at 1066MHz, with hyper-threading enabled. These machines were interconnected by a Gigabit Ethernet switch and ran Ubuntu Server 14.04 LTS.

4.1.5.1 Source of entropy and PRGs

We empirically evaluate both the source of strong entropy and PRGs through statistical methods and tools, following state of the art recommendations [17]. To achieve our goal, we used NIST's test suite [109]. We generated one file containing 50MB of random bits per generator. These files were used as input for the test suite tool STS [109]. In the end, our generators passed the absolute majority of tests and sub-tests: they failed only 2 sub-tests out of 188 (passed 146 out of 148 non-overlapping template matching), as summarized in Table 4.1. This gives a very high level of confidence to our generators.

Test	Result
Frequency	✓
Block Frequency	✓
Cumulative Sums (forward)	✓
Cumulative Sums (backward)	✓
Runs	✓
Longest Run of Ones	✓
Binary Matrix Rank	✓
Discrete Fourier Transform	✓
Non-overlapping Template Matching	146/148
Approximate Entropy	✓
Random Excursions	8/8
Random Excursions Variant	18/18
Serial (first)	✓
Serial (second)	✓
Linear Complexity	✓

Table 4.1: STS: results of the single tests

4.1.5.2 Device-to-device communication performance

Connection establishment. While a TLS connection takes around 19 ms to be established, a device association using the ANCHOR takes less than 0.06 ms. In other words, our connection setup process outperforms the TLS handshake because we have only half the number of steps, namely, we do not incur the cost of exchanging data to generate the session keys, and we use NaCl for secure and fast ciphering.
Communications overhead. Figure 4.3 shows the results of communication between OpenSSL, TCP, and our proposal. For communication of up to 128 forwarding devices, sending 10k control messages each, our solution requires (while offering stronger security guarantees - see below) only half of the resources and time of an OpenSSL-based implementation using AES256-SHA, the most widely available cipher suite — adopted by most IT providers.
In Figure 4.3, we can also observe the overhead of confidentiality (TCP-iDVV-EMAC). In comparison with providing only authenticity and integrity (TCP-iDVV-MAC), confidentiality incurs in an overhead of 15%. Out-of-band control channels are one example scenario where confidentiality of control plane communications might not be always required.
It is worth emphasizing that we achieved these results by ensuring also much stronger security, as we generated one iDVV (i.e., one secret) *per packet*. On the other hand, the OpenSSL-based implementation used a single key (for the symmetric ciphering) for the entire communication session.

4.1.5.3 Traditional solutions *versus* anchor

In Table 4.2 we provide a summarised comparison between a traditional solution and our ANCHOR. As traditional solutions we considered the EJBCA (`http://www.ejbca.org/`) and OpenSSL, two popular

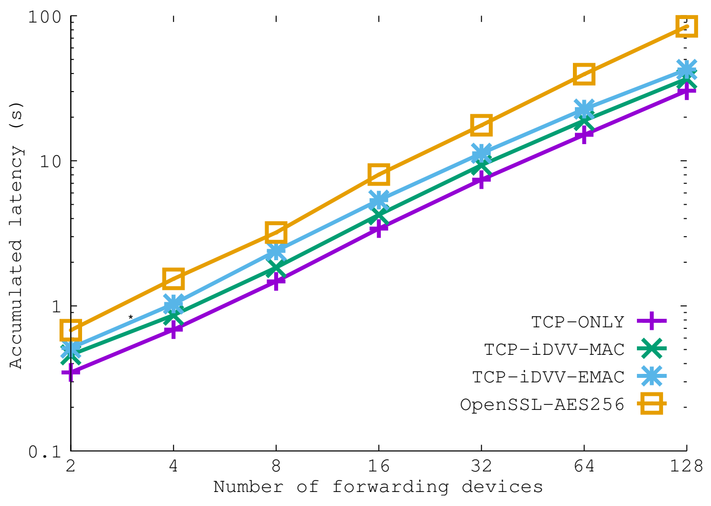


Figure 4.3: Control plane communication costs

implementations of PKI and TLS, respectively. As we have shown before, our bootstrap process (device registration and association) is much faster and our connection latency is also significantly lower. An interesting take away is that our solution has nearly one order of magnitude less LoC and uses four times less external libraries and only four programming languages. This makes a difference from a security and dependability perspective. For instance, to formally prove more than 717k LoC (EJBCA + OpenSSL) is by itself a tremendous challenge. And it gets considerably worse if we take into account eighty external libraries and eleven programming languages.
In conclusion, our proposed architecture offers a functionally equivalent level of security (with respect to security properties such as authenticity, integrity and confidentiality) to traditional alternatives by combining NaCl, our ANCHOR, and the iDVV mechanism. Additionally, our ANCHOR offers a higher level of security by providing post-compromise security (PCS) and post-quantum security (PQS). While the former is ensured through post-compromise recovery (see Section 4.1.3.10), the latter is a consequence of using only symmetric cryptography. Further, the lightweight nature of our mechanisms, such as the iDVV, make them amenable to be used on a per message basis to secure communication, increasing cryptographic robustness. Moreover, by having less LoC, we significantly reduce the threat surface.
Finally, it is worth emphasizing that the perfect forward secrecy of traditional solutions, such as those provided by the different implementations of TLS, is not easy or simple to enforce. First, in spite of TLS providing ciphers that offer PFS, in practice, different cipher suites do not feature it [139]. This means that not all implementations and deployments of TLS offer PFS or provide it with very low encryption grade [71, 51, 106]. To give an example, widely deployed web servers, such as Apache and Nginx, may suffer from weak PFS configuration [51]. Second, research results have recurrently shown that most DHE- and ECDHE-enabled servers use weak DH parameters or practices that greatly reduce the protection afforded by PFS, such as private value reuse, TLS session resumption, and TLS session tickets, i.e., provide a false sense of security [71, 5, 146].

4.1.6 Related work

Most attacks to SDN exploit different vulnerabilities of the control plane, such as the lack of authentication, authorization and other essential security properties (e.g., integrity, confidentiality and data freshness) [134, 13, 90]. As a consequence, the absolute majority of the works on security of SDN are related to the controller, applications, forwarding devices, or a set of specific attacks (e.g. DDoS, IP and MAC spoofing, eavesdropping on the data plane) [143, 135, 122, 142, 144, 157, 15]. However, not much attention has been paid to the security requirements of control plane associations and communication between devices (see [134, 85, 49] for broad surveys), one of the aspects we address in this

Table 4.2: Traditional solutions *versus* ANCHOR

Functionality		**Traditional solutions**	**ANCHOR**
Typical Software		EJBCA (PKI) + OpenSSL (TLS)	ANCHOR + iDVV + NaCl
Device Identification		based on certificates; costs = issue a certificate	based on unique IDs controlled by the ANCHOR; costs = register the device (assign a unique ID)
Device Registration		based on certificates; costs = certificate installation + security control policy/service	registration protocol; costs = register the device + iDVV bootstrap
Device Association & KeyGen		12 step mutual handshake + DH for session keys (incl. certificate validation - any two device can establish an association)	6 step trust establishment with ANCHOR + iDVVs per message, session, interval of time, ... (ANCHOR has to authorize association)
Security Properties	Authenticity	✓	✓
	Integrity	✓	✓
	Confidentiality	✓	✓
	PFS	✓(*)	✓
	PCS	✗	✓
	PQS	✗	✓
Communications		symmetric cryptography (cipher: AES256-SHA)	symmetric cryptography (cipher: Salsa20)
TLS stack		highly configurable and complex (717k LOC)	easy to use, simple (85k LOC), and efficient

work.
TLS and IPSec are examples of protocols that can be used to secure the communication between forwarding devices and controllers. While TLS is the one recommended by ONF, recent research discusses the strengths and weaknesses of these protocols as a mean to provide authenticated and encrypted control channels [130]. While the of use these protocols gives important security properties, they have an impact on control plane performance. Additionally, the complexity of existing software has been recurrently pointed out as one of the main cause for a high number of reported vulnerabilities, that in many cases have led to security attacks [175, 100, 101, 70]. By logically-centralizing crucial security mechanisms, our ANCHOR removes complexity from both controllers and switches, enhancing the robustness of the infrastructure, without significant compromise in performance.
Recently, the use of lightweight information hiding based authentication (by means of secrecy through obscurity) has been proposed as one way of protecting SDN controllers from DoS attacks [4]. The idea is to use a specific field in the IP protocol to hide the switch authentication ID. In order for the scheme to be workable, it is assumed that a look-up table and unique IDs are shared among devices through existing key distribution protocols. While such lightweight technique can indeed be used to mitigate DoS attacks, it does not address the security issues of control plane communications – such as authenticity, integrity, confidentiality, and data freshness – we address here.
To our knowledge, an architectural approach as the one we propose here (which ultimately led to following the SDN philosophy of "logical centralization") was lacking. Importantly, this approach allowed us to gain a global perspective of the relevant gaps in SDN and the limitations of existing solutions to the problem. This first step gave insight into one of the most relevant problems of SDN (as noted by the ONF or MEF security groups [113, 103]): the security of the associations and communications between devices— which jointly with the architecture itself, is one of the main contributions of this work.

4.1.7 Discussion

We briefly discuss how we filled the gaps identified in Section 4.1.2, with our specialization of the logically centralized ANCHOR architecture for 'security'. Incidentally, we also show, in Appendix D of [87], to which extent these solutions cover ONF's security requirements. We conclude the section with a critique of our choices and results.

4.1.7.1 Meeting the challenges

Security vs performance? Control channels need to provide high performance (high throughput and low latency) while keeping the communication secure. However, as it has been shown, security primitives have a non-negligible impact on performance. To mitigate this problem, we selected the most appropriate cryptographic primitives (SHA512) and libraries (NaCl) to ensure the security of control plane communications. Additionally, the proposed integrated device verification values (iDVVs) allow systematic refreshing of cryptographic material with high performance, while further improving cryptographic robustness. By logically centralizing the fundamental aspects of these mechanisms in the ANCHOR, the performance overhead introduced in forwarding devices and controllers is limited, as they require only minimal functionality to 'hook' to the ANCHOR instructions.
Complexity vs robustness? Traditional implementations of SSL/TLS, such as OpenSSL, have a large, complex code base, that recurrently leads to vulnerabilities been discovered. Similar problems are observed in PKI subsystems. It is well know that an effective means to achieve robustness is by reducing complexity. Hence our choice for the NaCl and iDVV mechanisms to help filling this gap, since they are respectively lightweight (small code base), efficient, yet secure alternatives to OpenSSL-like implementations. As such, they are a robust solution to provide authentication and authorisation material for the secure communications protocols we propose. They are also amenable to verification mechanisms aimed to assure correctness, which are much harder to employ in very large code bases. Again, the centralization of the nuclear parts of the non-functional mechanisms introduced in our solution is the key to reduce complexity of networking devices, improving their robustness.
Global security policies? We have argued that controllers and network devices often employ suboptimal network authentication and secure communication mechanisms, despite recommendations from ONF and other such organizations for the opposite. This problem is very similar in nature to the state of affairs in networking before SDN. In traditional networks, enforcing relatively "simple" policies such as access control rules [43] or traffic engineering mechanisms [75] was either very hard or even impossible in practice. Given the current undesirable situation, we believe the same to be true to non-functional properties, with security as a prominent example. Our logically centralized ANCHOR architecture addresses this gap by providing a means for making centralized policy rules (e.g., about registration, authentication and association of network devices) and the necessary mechanisms to enforce them, permeating the SDN architecture in a global and coherent way.
Resilient roots-of-trust? We debated that there is a (probably reduced) number of functions which should not be left to ad-hoc implementations, due to their criticality on system correctness. The list is not closed, but we hope to have shown that strong sources of entropy, resilient indistinguishable-from-random number generators, and accurate, non-forgeable global time services, are clear examples of difficult-to-get-right mechanisms that benefit from such logically centralized approach. ANCHOR addresses this issue, by providing the motivation to design and verify careful and resilient once-and-for-all implementations of such root-of-trust mechanisms, which can then be reinstantiated in different SDN deployments.

4.1.7.2 Devil's advocate analysis

Doesn't the logical centralization of non-functional properties create a single point of failure?
As mentioned in the introduction, we have a long-term strategy towards this problem. The results from this work already go a long way improving robustness of a single root-of-trust, compared to the state of the art: lowering failure probability; mitigating and recovering from the consequences of

failure. The logical next step would be to try and prevent failures in the first place. However, the failure of a simplex system of reasonable complexity cannot be prevented. Nevertheless, note that logical centralization is not necessarily physical centralization.
Our plan for future work is to leverage state-of-the-art security and dependability mechanisms using replication. For instance, to achieve tolerance of crash and Byzantine faults and attacks, we can readily enhance ANCHOR by replication, taking advantage of state machine replication libraries such as BFT-SMaRt [26], replicating and diversifying components to prevent failure of this logically central point, with the desired confidence. These concepts have been applied to root-of-trust like configurations similar to ANCHOR [174, 41, 83]. Furthermore, systems designed with state machine replication in mind can also handle different types of threats, such as DoS and resource exhaustion, without compromising the operation of the service [89].

Won't the natural hardware evolution be by itself enough to reduce the penalty imposed by cryptographic primitives? The recent reality seems to contradict this assertion – hardware evolution does not seem enough, for several reasons. First, new hardware architectures can (potentially) benefit different existing software-based solutions. For instance, both NaCl and OpenSSL take advantage of hardware-based AES accelerators. Second, and as is well known, the fixed price of advancements in hardware seems to be coming to an end [73]. Third, most of the major IT companies, such as Google and Microsoft, have been redesigning existing software to make it more usable, efficient, secure, and robust [95]. In short, hardware will not be the panacea.

Aren't traditional PKI and TLS implementations enough? Following what is becoming recurrently advocated by many in the industry and in the security community, we have tried to argue that the simplicity and size of software and IT infrastructure matters [46, 156]. Higher complexity has been shown to lead inevitably to an increased likelihood of bugs and security incidents in software. Indeed, different implementations of PKI and TLS have been recently used as powerful "weapons" for cyber-attacks and cyber-espionage [123, 32], leading to concerns about their robustness. Contrary to what this argument may suggest, that does not mean PKI and TLS are "broken". We believe they remain fundamental to various IT infrastructures. However, as the main challenges of securing SDN are usually relatively constrained to within a network domain, we have come to understand that simpler, domain-specific solutions seem to be preferable in this environment when compared to complex infrastructures such as the PKI, and large code bases as OpenSSL.

Wouldn't the use of out-of-band control channels solve most problems? Out-of-band channels may be useful in some contexts, but they are not "intrinsically" secure. It is a recurrent mistake to consider physical isolation, *per se*, as a form of security. Several studies have indeed argued the opposite: that out-of-band channels worsen the problem, by making control plane management more complex and less flexible, endangering control plane communications [54, 99]. We do not take a stance in this discussion, but the fact is that real incidents, such as NSA sniffing of Google's cables between data centers [132], seem clear examples that out-of-band channels are not, *per se*, enough.

4.1.8 Conclusions

In this section, we proposed a solution to the problem of enforcing non-functional properties in SDN, such as security or dependability. Re-iterating the successful philosophy behind the inception of SDN itself, we advocate the concept of logical centralization of SDN non-functional properties provision, which we materialize in terms of the blueprint of an architectural framework, ANCHOR.
Taking 'security' as a proof-of-concept use case, we have shown the effectiveness of our proposal. We made a gap analysis of security in SDN, and populated the ANCHOR middleware with crucial mechanisms and services to fill those gaps and enhance the security of SDN.
We evaluated the architecture, especially focusing on the security-performance analysis tradeoff, giving proofs of the algorithms, cryptographic robustness analyses, and experimental performance evalua-

tions. By resorting to recent primitives, lightweight albeit secure, like NaCl and iDVV, we outperform the most widely used encryption of OpenSSL by 50%, with a higher level of security. Our solution also fulfills eleven of the security requirements recommended by ONF.
The mechanisms we propose are certainly not the final answer to SDN security problems. That is not our claim. We however believe, and hope to have justified here, that an architecture that logically centralizes the non-functional properties of an SDN to have the potential to address some of the most prement unsolved problems regarding the robustness of the infrastructure. We thus hope our work to trigger an important discussion on these fundamental architectural aspects of SDN.

4.2 Secure and efficient control plane communications

As explained above, in Software-Defined Networking (SDN), network control is separated from the forwarding devices and logically centralised in a controller. This separation is achieved by means of a protocol (typically, OpenFlow) that enables the SDN controller to remotely populate the forwarding tables of network switches. The OpenFlow standard includes Transport Layer Security (TLS) (see IETF RFC 5246) as an *optional* security feature for authenticating forwarding devices and controllers and for encrypting the communication channel. However, to date most reported deployments still use TCP for control traffic, and SDN controllers and switching hardware with TLS support are still rare [85]. This makes the control plane communication vulnerable to different attacks [85, 134].
Four fundamental issues can slow down the rate of adoption of secure mechanisms in SDN:

- First, securing communications has a non-negligible cost in terms of increased communication latency and reduced performance. Several recent studies have analysed this overhead in various contexts [107].
- Second, the computing capabilities of commodity switches are typically weak. The typical SDN switch is equipped with a single or dual-core CPU running at approximately 1GHz, which compares unfavourably with the multi-core CPUs found in typical commodity servers. Imposing the additional cost of TLS to these computing-constrained networking devices is a problem.
- Third, poor choice of cryptographic primitive implementations can also have a significant impact on the performance of the control plane communications handled by the controller.
- Finally, the Public Key Infrastructure (PKI) on which TLS relies is complex and thus vulnerability-prone [159], opening a large surface for successful attacks [153].

In order to meet these challenges, we propose a modular secure SDN control plane communications solution, KISS[4] SDN (Section 4.2.1), which aims to increase the robustness of control communications whilst enhancing their performance, by decreasing the complexity of the support infrastructure, as an alternative to current approaches based on classic configurations of TLS and PKI.
A core novel component of our architecture is the integrated device verification value (iDVV), a deterministic but indistinguishable-from-random secret code generation protocol (Section 4.2.2). As explained in the previous section, the concept was inspired by the iCVVs (integrated card verification values) used in credit cards to authenticate and authorize transactions in a secure and inexpensive way. We develop and extend the idea for SDN, proposing a flexible method of generating iDVVs by adapting proven one-time password-like techniques. iDVV codes allow the safe decentralized generation/verification of keys at both ends of the channel, at will, even on a per-message basis.
To understand and minimize the cost of security, we quantify (Section 4.2.3) the impact of secure primitives on the performance and scalability of control plane communications, through a performance study of different implementations of TCP vs. TLS, complemented by a deeper study of underlying hashing and message authentication code (MAC) primitives. Those experiments confirm our intuition

[4]Keep It Simple and Secure

that the choice of protocols and primitives used in secure communication may well be one strong reason behind the slow adoption of these mechanisms in SDN. This in-depth study leads to the selection of the NaCl cryptographic library [24], and the best performing MAC and strong hash primitives — Poly1305 and SHA512 OpenSSL – as the baseline secure channel technologies for KISS.
iDVVs team-up with NaCl, in order to safely replace the cryptographic primitives and key-exchange protocols and key derivation functions commonly used in TLS. As a result, the NaCl-iDVV compound, while achieving the same functional level of security, is simpler, potentially leading to a higher level of implementation robustness by vulnerability reduction. In fact, we estimate the proposed security architecture footprint to be smaller than TLS-PKI alternatives with traditional protocols, by an order of magnitude, in terms of the number of lines of code (LOC). Such a differential also points to reducing the cyclomatic complexity. These metrics are typically used to assess the robustness and estimate verifiability of software systems.
Finally, in Section 4.2.4 we discuss aspects related to the security and robustness of our solution. We close this section with related work and some directions to further work.

4.2.1 KISS SDN

In this section we present KISS SDN, a secure and efficient control plane communications solution for SDN offering alternatives to classic configurations of secure channel and authentication protocols and subsystems followed in TLS and PKI. We assume a typical SDN architecture, as illustrated in Figure 4.4, composed of controllers and forwarding devices. It also includes an instance of ANCHOR: 'KDC', standing for key distribution center. We further assume that device registration and association services are in place. As they were presented in the section 4.1, we do not discuss them in detail but, for self-containment, we discuss some properties and their interface below.
The two components encapsulated by the KISS boxes (the "hooks" of this solution) are the crucial components of the architecture, and the main subject of our study: a secure channel protocol suite, composed of a judicious choice of state-of-the-art mechanisms and protocols, which we dub SC for convenience of description, and a novel deterministic but indistinguishable-from-random secret code generation protocol, which we call iDVV.
We have considered using TLS implementations (e.g. OpenSSL) as the baseline protocol for SC. However, the experiments in Section 4.2.3 have alerted us to: the sheer performance cost of cryptographic communication; and the further impact of sub-optimal choices of cryptographic primitives. This motivated us to adopt NaCl [24], a high performance yet secure cryptographic library, as the substrate of SC, complemented by the MAC and strong hash primitives with best performance according to our experiments – Poly1305 and SHA512 OpenSSL. SHA-512 is used by the iDVV generator while Poly1305 is a fast MAC algorithm.
The iDVV, a novel component we propose, helps to further enhance the security of SC, through strong crypto material generated at a low cost (e.g. one-time keys, per-message authentication and authorization codes) to be used by NaCl ciphers. The indistinguishability-from-random allied to the determinism allow the safe decentralized generation/verification of per-message keys at both ends of the channel.

4.2.1.1 System and threat model

For simplicity and without loss of generality, we assume that the controllers and forwarding devices are registered and associated through a secure and robust key distribution service provided by a key distribution center (KDC), an instance of the ANCHOR architecture.
The device registration process is by default invoked by network administrators to the KDC, to register new devices. As the result of device registration, the device and the KDC securely share a symmetric key. We denote K_{kc} the shared key between the KDC authority and a registered controller, and K_{kf} the shared key between the KDC authority and a registered forwarding device.

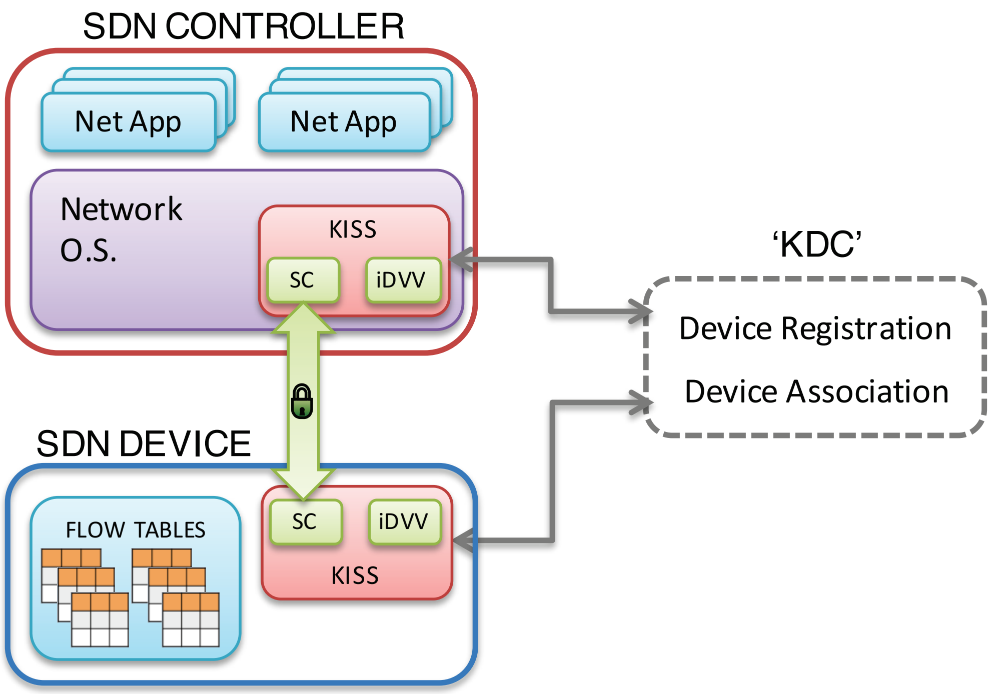


Figure 4.4: General architecture

Registered controllers and forwarding devices must be securely associated, also through the KDC authority, as a pre-condition to communicate securely. The most common case is a forwarding device f_i requesting an association to a controller c_j, through the KDC. After associating, a controller and a forwarding device share two symmetric secrets (of size 256 bits), namely a $seed_{ij}$ and a key_{ij}. The key is generated by the KDC and the seed is generated by the KDC in cooperation with the controller. These secrets will be used to bootstrap the iDVV module (see Section 4.2.2.1).
As threat model, we consider a Dolev-Yao style attacker, who has complete control of the network, namely the attacker logs all messages, and can arbitrarily delay, drop, re-order, insert, or modify messages. In addition, this strong attacker is able to compromise any network device (e.g. a controller or a forwarding device) at any time. We assume the security of the used cryptographic primitives, including MAC (i.e. Poly1305), hash function (i.e. SHA-512), and symmetric encryption algorithm (e.g. AES).

4.2.1.2 Security goals

The main goal of KISS is to provide security properties including authenticity, integrity, and confidentiality for control plane communications, while minimizing cost and complexity.
The secure communication between participants can be easily guaranteed when a secure encryption algorithm is used, as long as the shared secret key is kept secure. To provide a robust SDN system, we focus on advanced security guarantees for the situation when the shared key is exposed to an attacker, as this might happen in practice. In particular, if an attacker has compromised a device and learnt its shared keys, then we are aiming at providing "perfect forward secrecy" (PFS) of communications. That is, the secrecy of a device's past communications should be protected when the device is compromised and its shared keys are exposed to an attacker. It is important to emphasize again that PFS is an essential requirement for SDN. The lack of it can lead to information disclosure, i.e., reveal different aspects of the network's state and the controller's strategy (e.g., proactive or reactive flow setup).
Established KDC technologies like Kerberos have robust implementations and are intensely used by industry, which makes us consider the logical single-point-of-failure they present as moderate, and an acceptable option for the current state of the art. We present mitigation measures to achieve PFS

in case of compromise of the KDC. We also plan, as future work, to investigate the development of SDN KDCs resilient to accidental and malicious faults, drawing from fault and intrusion tolerance techniques [155].
On the devices side, we make no claim about their sheer resilience, since this is largely dependent on vendors. More precisely, when a controller and/or a forwarding device is compromised, we consider that the attacker is able to obtain all knowledge of the victim device(s), including all stored secrets and the session status. However, it is our goal to guarantee the confidentiality of all past communications through measures that allow us to achieve perfect forward secrecy.

4.2.2 iDVV: Keep It Simple and Secure

Integrated device verification values (iDVVs) are sequentially generated to protect and authenticate requests between two networking devices. The generator is conceived so that its output sequence has the indistinguishability-from-random and determinism properties. In consequence, the same sequence of random-looking secret values is generated on both ends of the channel, allowing the safe decentralized generation/verification of per-message keys at both ends. However, if the seed and key initial values and the state of the generator are kept secret, there is no way an adversary can know, predict or generate an iDVV.
In other words, an iDVV is a unique secret value generated by a device A (e.g. a forwarding device), which can be locally verified by another device B (e.g. a controller). The iDVV generation is made flexible to serve the needs of SDN. iDVVs can therefore be generated: (a) on a per message basis; (b) for a sequence of messages; (c) for a specific interval of time; and (d) for one communication session. The main advantages of iDVVs are their low cost and the fact that they can be generated locally, i.e., without having to establish any previous agreement.
Different from standard KDF algorithms such as HKDF, which assumes that keying material is not uniformly random or pseudorandom, our keying material (i.e. seed and key) are random symmetric secrets (each of size 256 bits), generated by the KDC, with high entropy. In such cases, a strong hash function can be safely used to derive a key (RFC 4880). As shown by the results in Section 4.2.3, the iDVV generation is simpler and faster than standard KDF algorithm such as HKDF (RFC 5869) and similar solutions.

4.2.2.1 iDVV bootstrap

As discussed before, the association between two SDN devices, e.g., forwarding device f_i and controller c_j, happens through the help of KDC, under the protection of the long-term secret keys obtained from registration (K_{kf}, resp. K_{kc}). The outcome of the association protocol is the distribution of two random secrets to both devices: a seed $seed_{ij}$, and an association key key_{ij}. The iDVV mechanism is bootstrapped by installing these two secret values in both the controller and the switch, to animate the iDVV generation algorithms, which we describe next.
Note that the set-up and generation of the iDVV values are performed in a deterministic way, so that they can be done locally at both ends. However, as iDVVs will be used as keys by cryptographic primitives such as MAC or encryption functions, they have to be indistinguishable from random. Hashing primitives are natural choices for our algorithms, since they provide indistinguishable-from-random values if one or more of the input values are known only by the sender and the receiver. This explains why it is crucial that seed and association key are sent encrypted and therefore known only to the communicating devices. Moreover, in order to prevent information leakage, all variables $seed$, key, and $idvv$ in the algorithms below should have the same length, which we chose to be 256 bits in our design. This length is commonly considered robust. From our experiments discussed in Section 4.2.3, the hashing primitive to be used is SHA512, which yields 512 bits, of which we will use the most-significant q bits if we need to reduce the output length to q (as recommended by IETF RFC 4880). For example, we use the most-significant 256 bits of the SHA512 output as the key for symmetric ciphers.

The initial iDVV value is deterministically created at both ends of the association between two devices[5], by calling function `idvv_init`, which performs hashing on the concatenation of the initial *seed* and *key*, as illustrated by algorithm 9. After set-up, the generator is ready for first use, as described in the following section.

Algorithm 9: iDVV set-up

```
1: idvv_init()
2:     idvv ← H(seed || key)
```

4.2.2.2 iDVV generation

After the bootstrap with the initial *idvv* value, the `idvv_next` function is invoked on-demand (again, synchronously at both ends of the channel) to autonomously generate authentication or encryption keys that will be used for securing the communications, as illustrated by algorithm 10.
The *key* remains the only constant shared secret between the devices. The *seed* evolves to a new indistinguishable-from-random value each time `idvv_next` is invoked to generate a new iDVV. The new seed is the outcome of a hashing primitive H over the current *seed* and current *idvv* (line 2). The *new idvv*, output of function `idvv_next`, is the outcome of a hashing primitive H over the concatenation of the *new seed* and association key *key*.

Algorithm 10: iDVV generation

```
1: idvv_next()
2:     seed ← H(seed || idvv)
3:     idvv ← H(seed || key)
```

4.2.2.3 iDVV synchronization

The iDVV mechanism is agnostic w.r.t. secure communication protocols, and can be used in a number of ways, in a number of protocols, as a key-per-message or key-per-session, etc. The only key issue about iDVV generation, is to keep it synchronized in both ends of the channel. So, we present some recommendations in this regard.
The most general style of iDVV use is *Indexed iDVV*: iDVVs are indexed by the generation number, and they are operated in "one key per direction" mode, i.e., at each end, one iDVV is generated for each communication direction. This way, they support competitive, non-synchronized correspondents. This mode also supports unreliable, connectionless protocols like UDP. Each iDVV generated is indexed by a sequence number (the initial iDVV being $idvv^0$) and the sequence number is included in the message where the respective *idvv* is used. This way, each receiving end (this works in either direction, as we have two pairs of iDVVs) can know the exact *idvv* number that should be used and, for example, detect and recover from omissions, by generating *idvv*'s the necessary number of times to resynchronize.
iDVVs can get out of sync for a number of reasons, such as speed differences, omission errors, or even DoS attacks. When de-synchronization happens, a baseline technique consists of advancing the iDVV of the "slower" end, to catch up. The process is made robust by two techniques. First, communication should be authenticated (encrypt-then-MAC recommended), such that any messages failing crypto (decryption or MAC verification), can be simply discarded. Second, when say, $idvv^k$ is advanced to $idvv^l$ ($k < l$) to re-synchronize, and the operation is not successful (crypto fails), the old $idvv^k$ is restored (and the message motivating the recovery, is discarded, as per above). This restoration does not affect the PFS of communications because the $idvv^k$ (or newer) has not yet been used to secure the traffic between the two communicating devices. Finally, in the case of attacks, these robustness techniques also help to foil them, since the attacker cannot mimic valid crypto, so the message is discarded, and the node returns to the original iDVV state.

[5]For readability, we omit the device-identifying subscripts in the variables.

4.2.2.4 iDVV implementation and application

iDVVs require minimal resources, which means that they can be implemented on any device, from a simple and very limited smart card to most existing devices. In other words, they are a simple and viable solution that can be embedded in any networking device. Just three values per association have to be securely stored — the seed, the association key and the iDVV itself — in order to use iDVV continuously. Furthermore, only hash functions, simple to implement and with a very small code base, are required to generate iDVVs. Such kind of resource is already available on all networking devices that support traditional network protocols and basic security mechanisms.
We advocate (and demonstrate in Section 4.2.3.2) that iDVVs are inexpensive and, as a result, can be used on a per-message basis to secure communication. It is worth emphasizing that, from a security perspective, one fresh iDVV per message makes it much harder for attacks such as key recovery, advanced side channel attacks, among other general HMAC attacks, to succeed. In fact, the one-time key approach was initially used for generating MACs. Yet, it was set aside (i.e. replaced by keys with a longer lifetime) due to performance reasons. However, as the iDVV generation has a low cost, we incur a lower penalty.
Finally, iDVVs can have further practical applications. For instance, the TLS handshake can be used to bootstrap the iDVV. After that, iDVVs can be used as session keys, i.e., in security mechanisms such as encrypt-then-MAC.

4.2.3 On the cost of security

In this section we provide a quantitative analysis of the impact of cryptographic primitives on control plane communication. Although the number of use cases is expanding, SDN has been mainly targeting data centers. As such, SDN controllers have to be capable of dealing with the challenging workloads of these large-scale infrastructures. In these environments new flows[6] can arrive at a given forwarding device every 10 µs, with a great majority of mice traffic lasting less than 100ms [19]. This means that current data centers need to handle peak loads of tens of millions of new flows/s. The control plane has to meet both the network latencies and throughputs required to sustain these high rates. Current controllers are capable of achieving a throughput of up to 20M flows/s using TCP [85].
So any effort to systematically secure control plane communications has to meet these challenges. In the following we try to put the problem in perspective, by analysing the effect of including even the most basic security primitives to ensure authenticity, confidentiality and integrity when considering peak loads of this magnitude. We start by analyzing the latency impact of TLS, relative to TCP, and then we focus on hashes and MACs as they are the essential primitives for authenticity and integrity of communication.

4.2.3.1 The cost of secure channels

Our first experiments assess the compared average latency of TCP and TLS on control plane communication. We analyse the latency of connection setup and of OpenFlow `PACKET_IN`/`FLOW_MOD` messages. The OpenFlow `PACKET_IN` message is used by switches to send packets to the controller (e.g. when there is no rule matching the packet received in the switch). `FLOW_MOD` messages allow the controller to modify the state of an OpenFlow switch.
The connection setup time for TLS is two orders of magnitude higher than for TCP, since TLS has a more elaborate handshake protocol between the devices [84]. Also, PolarSSL (a library used in systems from companies such as Gemalto, ARM, and Linksys) induces nearly twice the overhead of OpenSSL. However important, a high connection cost can be amortized by maintaining persistent connections. As such, we focus on the communications cost. Figure 4.5 shows the latency of `FLOW_MOD` messages, averaged over 10k messages. The results with `PACKET_IN` messages were similar so we omit

[6]In spite of the fact that there are several definitions of flow in SDN [85], we equate SDN flow with TCP flow for the sake of simplicity.

them for clarity. The costs of TCP, OpenSSL and PolarSSL grow nearly linearly with the number of forwarding devices. OpenSSL latency is approximately 3x higher than TCP. This is explained by the high overhead of cryptographic primitives, as we further analyse in the next section. PolarSSL is significantly worse, increasing the latency by up to 7x when compared with TCP.

Conclusions: The main findings of this analysis can be summarised in two points. First, different implementations of TLS present very different performance penalties. Second, the additional computation required by the cryptographic primitives used in TLS leads to a non-negligible performance penalty in the control plane. In consequence, we turn to lightweight cryptographic libraries, such as NaCl [24] and TweetNaCl [25], which are starting to be used in different applications.
NaCl has been designed to be secure [11, 24] and to be embedded in any system [25], taking a clean slate approach and avoiding most of the pitfalls of other libraries (e.g. OpenSSL – misuse issues):

- First, it exposes a simple and high-level API, with a reduced set of functions for each operation.
- Second, it uses high-speed and highly-secure primitives, carefully implemented to avoid side-channel attacks.
- Third, NaCl is less error-prone because low-security options are eliminated and it also provides a limited number of cryptographic primitives. In other words, users do not need deep knowledge regarding security to use it correctly. This is one of the major differences between it and other libraries such as OpenSSL. For instance, it has been recurrently shown that developers have been using OpenSSL in incorrect ways, leading to several security issues.
- Fourth, it has already been shown that secure and high-performance network protocols, outperforming OpenSSL, can be designed and implemented using NaCl [119].

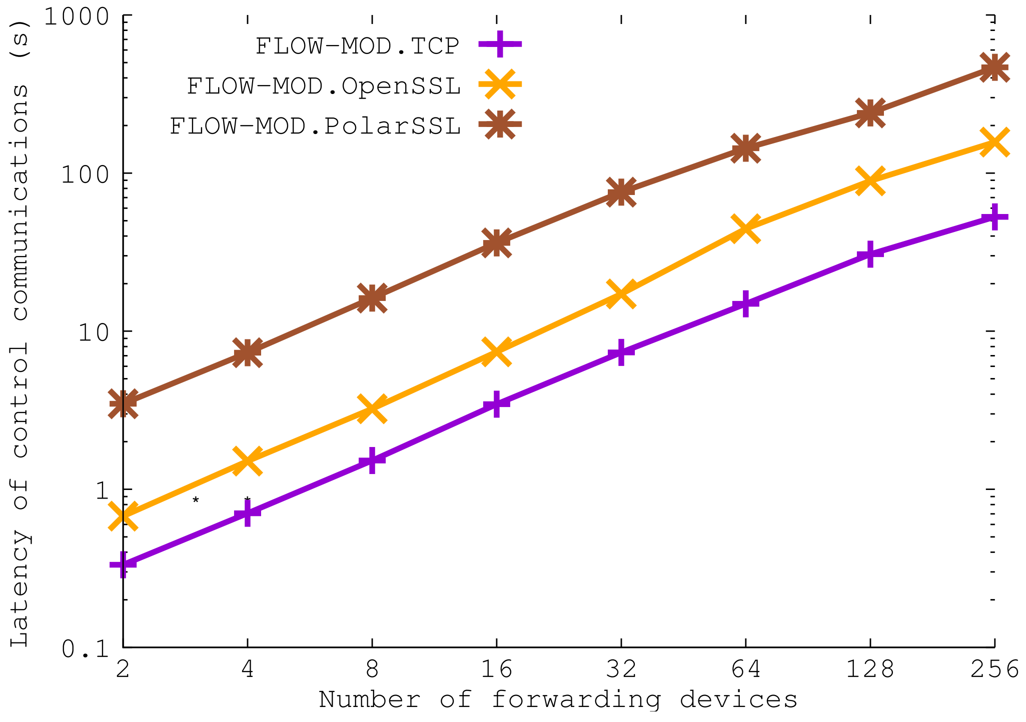


Figure 4.5: `FLOW_MOD` latency (in log scale)

4.2.3.2 A closer look at the cost of cryptography

To understand in more detail the cause of the previous findings we now perform a fine-grained analysis of two main classes of security primitives used in secure channel protocols: hashing and MAC.
We analyse the performance of nine hashing primitives. The results are presented in Figure 4.6. The red bars represent primitives that are provided by OpenSSL, while white bars (BLAKE and KECCAK) indicate the original implementation of primitives that are not part of OpenSSL. From Figure 4.6, we observe that the primitives with smaller digest sizes (SHA-1 and MD5) achieve better performance, as expected. The stronger versions of the SHA and BLAKE families achieve comparable performance (slightly slower), with higher security guarantees. Interestingly, SHA-512 outperforms SHA-256. This behavior is explained by the fact that on a 64-bit processor each round can process twice as much data (64-bit words instead of 32-bit words). However, SHA-256 is faster on a 32-bit processor.
To understand the variance between different implementations, we present in Figure 4.7 the costs of the five hashing primitives for which different implementations were available. The OpenSSL implementation shows the best performance performance for hashing primitives. With the exception of RIPEMD160, the PolarSSL implementation always presented higher message latencies.

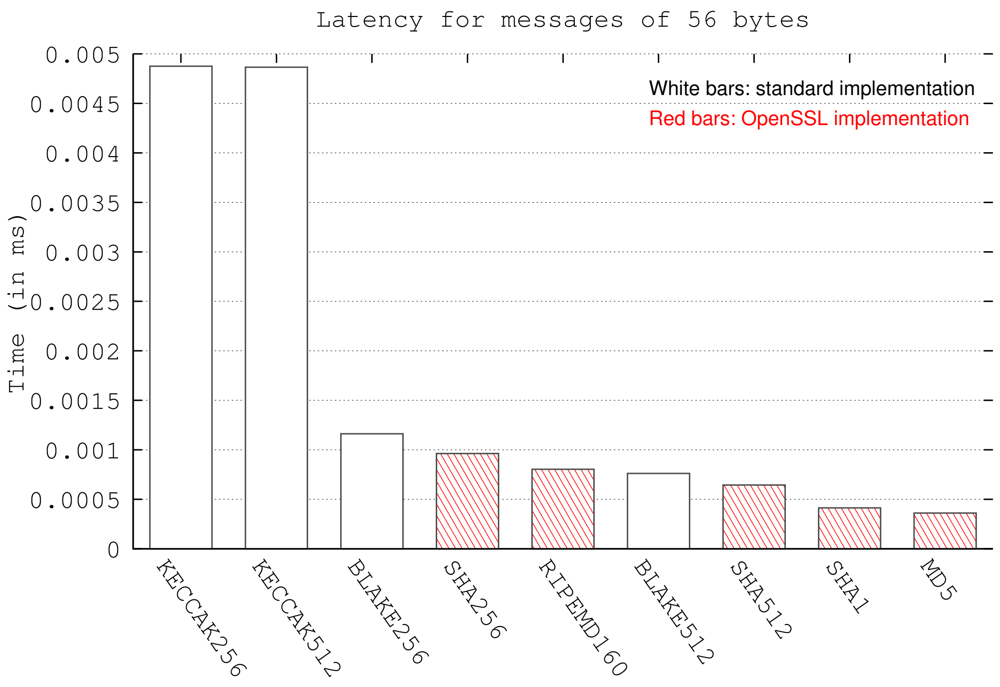


Figure 4.6: Hashing primitives

Finally, Figure 4.8 shows the results of the latency analysis of six MAC primitives. It is clear that Poly1305 outperformed all other primitives, being approximately two times faster than OpenSSL's HMAC-SHA1, and close to four times faster than HMAC-SHA512, for instance.

Conclusions: From the results of Figure 4.8, considering the MAC primitive with best performance in the analysis (Poly1305 with 0.001ms per message), around 20 dedicated cores are needed to compute a MAC in order to maintain a rate of 20M flows/s. To understand the importance of judiciously selecting the security primitives implementation, the HMAC-SHA512 OpenSSL (worst case performance in the analysis) would require over three times more cores (up to 65) to compute MACs at these rates. From the hashing primitive analysis in Figures 4.6 and 4.7, of the strong primitives (i.e. all except SHA1 and MD5), SHA-512 performs the best. However, concerning MAC primitives, the performance of HMAC-SHA512 disappoints, and it is clear that Poly1305 outperformed all other primitives, providing security with high speed and low per-message overhead.

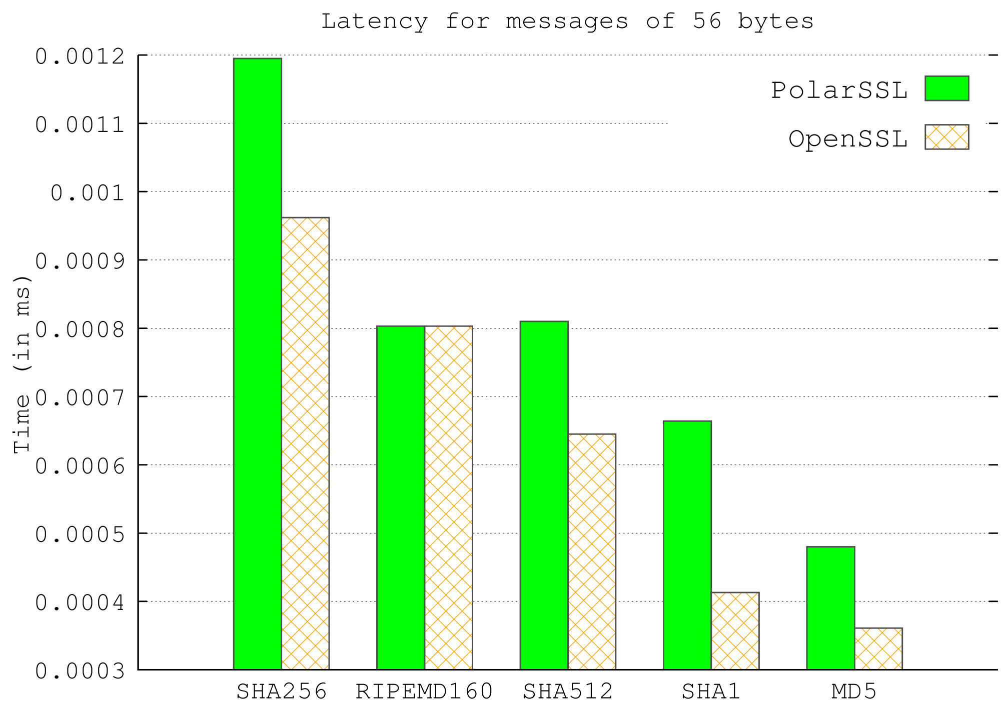


Figure 4.7: Implementations of hashing primitives

Figure 4.9 shows the performance of different primitives for generating cryptographic material. We compare the iDVV generator using SHA512 (iDVV-S5), with an implementation of a common key derivation function (KDFx) with different values for the exponent c (128, 64, 32, and 16, respectively), the Diffie-Hellman implementation used by OpenSSL (DH-OSSL), and the `randombytes()` function (NaCl-R) provided by NaCl. The latencies of the several primitives are significantly higher than iDVV. Even the *randombytes()* primitive of NaCl, the second fastest after iDVV, still presents a latency at least 2.6x higher.
In summary, our findings in this section indicate that:

- the inclusion of cryptographic primitives results in a non-negligible performance impact on the latency and throughput of the control plane; and that
- a careful choice of the primitives used and their respective implementations can significantly contribute to reduce this performance penalty and enable feasible solutions in certain scenarios.

Taking the outcome of our analysis into consideration, and given the benefits of NaCl described in Section 4.2.3.1, we have selected the NaCl lightweight cryptographic library, and the MAC and strong hash primitives with best performance – Poly1305 and SHA512 OpenSSL – as the baseline SC secure channel component technologies. NaCl is complemented in our architecture with the iDVV mechanism to generate cryptographic material (e.g. keys) used by NaCl ciphers. Taken together they provide, as per our evaluation, the best trade-off between security and performance for control plane communications in SDN.

4.2.4 Discussion

4.2.4.1 On the security of iDVV

With respect to the secrecy of iDVVs, it is ensured from initialization and so long as neither the KDC, controller, nor forwarding device is compromised. Our scheme also achieves perfect forward secrecy in the face of compromise of either KDC, controller, or forwarding device. In short, when the KDC is

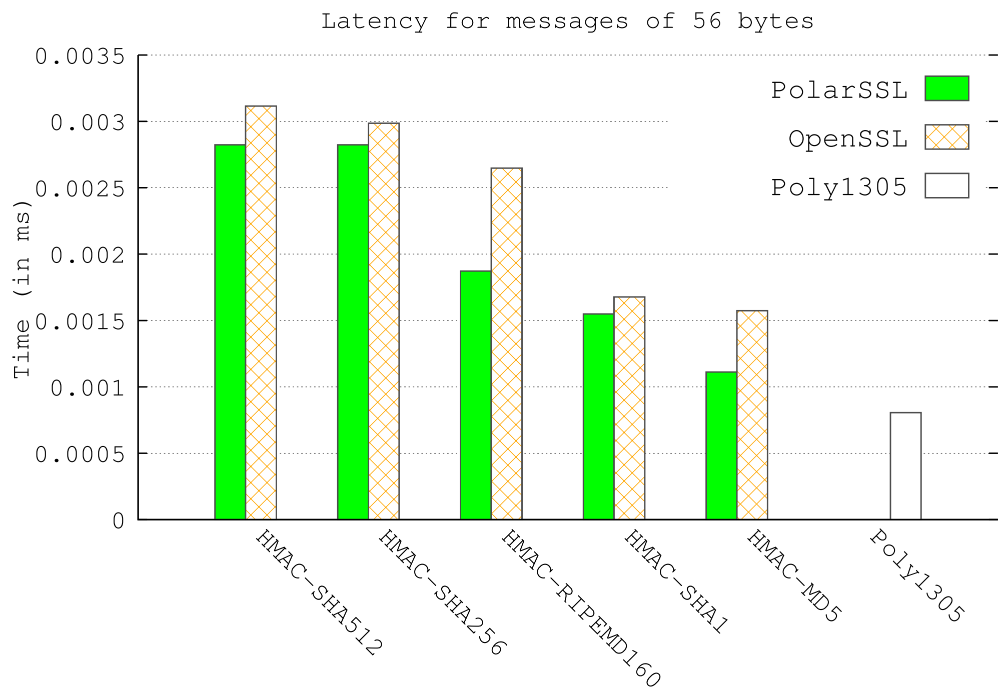


Figure 4.8: MAC primitives

compromised, then the attacker would be able to obtain all the shared secrets (between the authority and registered devices), decrypt the past communication that delivered the initial *seed* and *key* to the associated devices, re-generate iDVVs and, in consequence, decrypt past conversations.
We provide a simple mechanism for providing PFS even when the authority is compromised: we update the shared key each time a forwarding device is associated with a controller. The key is updated as follows: $K_{kc} \leftarrow H(K_{kc})$ and $K_{kf} \leftarrow H(K_{kf})$. This way, a shared key captured cannot decrypt any past messages, since they have been encrypted with previous generations of that key, which have been "forgotten" in the system, given the irreversible nature of hashes.
As far as devices are concerned, when they are compromised, the current values of *seed*, *key* and *idvv* are captured. Note that *key* stays as the original secret, but *seed* is rolled forward everytime a new iDVV is generated. So, the attacker will be unable to synthesize any past iDVVs since day one and so, cannot decrypt past conversations, achieving PFS, as we desired.

4.2.4.2 On the solution robustness

Our proposal compares well with traditional solutions such as EJBCA (`http://www.ejbca.org/`) and OpenSSL, two popular implementations of PKI and TLS, respectively.
The first interesting take away is that our solution has nearly one order of magnitude less LOC (85k) and uses four times less external libraries and only four programming languages. This makes a huge difference from a security and dependability perspective. For instance, to formally prove more than 717k LOC (OpenSSL + EJBCA) is by itself a tremendous challenge. And it gets considerably worse if we take into account eighty external libraries and eleven development languages. Moreover, it is worth emphasizing that libraries such as OpenSSL suffer from different fundamental issues such as too many legacy features accumulated over time, too many alternative modes as result of tradeoffs made in the standardization, and too much focus on web and DNS names.
Second, OpenSSL is complex and highly configurable. This has been also the source of many security incidents, i.e., developers and users frequently use the library in an inappropriate way. It has also been shown that the majority of the security incidents are still caused by errors and misconfiguration of systems. Lastly, recent research has uncovered new vulnerabilities on TLS implementations [29].

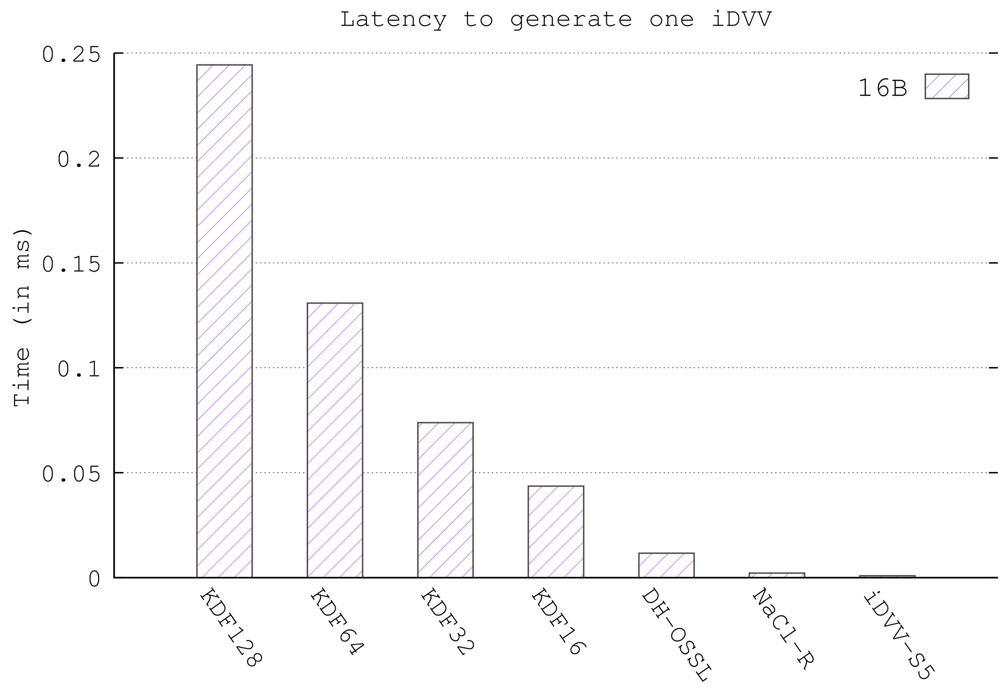


Figure 4.9: Latency to generate keys

In contrast, our proposed architecture exhibits gains in both performance and robustness, contributing to solving the dilemma we outlined in the introduction. By having less LOC, we significantly reduce the threat surface – by one order of magnitude – and by combining NaCl and the iDVV mechanism, we provide a potentially equivalent level of security, but quite increased performance/robustness product, as keys can be rolled even on a per message basis.

4.2.4.3 On the cost of iDVV

Similarly to iCVVs, iDVVs are a low overhead solution that requires minimal resources. This solution is thus feasible to be integrated into compute-constrained devices as commodity switches. Our preliminary evaluation has revealed that the iDVV mechanism is faster than traditional solutions, namely, the key-exchange algorithms embedded in the OpenSSL implementation. Considering a setup with 128 switching devices, sending `PACKET_IN` messages to and receiving `FLOW_MOD` messages from the controller, our results shows our proposed solution (iDVV + NaCl's ciphers) to be more than 30% faster than an OpenSSL-based implementation using AES256-SHA (the most common high performance cipher suite, used by IT companies such as Google, Facebook, Microsoft, and Amazon). Importantly, we were able to outperform OpenSSL-based deployments while still providing the same security properties: authenticity, integrity, and confidentiality. In addition, we achieved this result not only while offering the same properties, but also with stronger security guarantees: the tests were made by generating one iDVV *per packet*, while the OpenSSL-based implementation uses a single key (for symmetric ciphering) for the entire communication session.

4.2.5 Related work

There are several feasible attacks against the SDN control plane [134]. Most of them explore vulnerabilities such as the lack of authentication, authorization and other essential security properties. However, almost no attention has been paid to the security requirements of control plane associations and communication between devices. For instance, only recently, the use of secrecy through obscurity has been proposed to protect SDN controllers from DoS attacks [4]. In this case, the switch authentication ID is hidden in a specific field in the IP protocol. It is assumed that the devices share a look-up table and unique IDs. However, in spite of being capable of mitigating DoS attacks, this technique does not address the security issues of control plane communications.

4.2.6 Conclusions

In this section, we set out to explore and confirm our intuition for the possible reasons behind a slower than expected adoption of security mechanisms in SDN, and based on those findings, we proposed KISS, a secure and efficient solution for SDN control plane communications.
We started by investigating the impact of essential cryptographic primitives and TLS implementations on the control plane performance. We showed that whilst even the most basic security primitives add a non-negligible degradation of performance, a judicious choice of these primitives and their specific implementations can mitigate the penalty significantly. This is particularly important for the typical SDN scenario that resorts to commodity hardware, sometimes with modest computing capabilities.
The second problem we explored in this work was the complexity of the centralized support infrastructure for authentication and key distribution. We proposed iDVV, a simple and robust decentralized mechanism for generating and verifying the secrets necessary for secure communications between network devices.
Our results are encouraging in terms of an increase of performance — 30% improvement over OpenSSL — and robustness — an order of magnitude reduction in the number of LOC, and implied cyclomatic complexity. This also means that formal verification is more tractable, which is one of our future goals for iDVV, for instance.
An extended report of this work can be found in [84], extending discussion of the iDVV performance, forward secrecy, randomness, and proofs of its security properties.

4.3 Fault-tolerant control plane

Software-Defined Networking (SDN) decouples the network control plane from the data plane via a well-defined programming interface (such as OpenFlow). This decoupling allows the control logic to be logically centralized, easing the implementation of network policies, enabling advanced forms of traffic engineering (e.g., Google's B4 [74]), and promoting innovation.
The controllers are the crucial enabler of the SDN paradigm: they maintain the logically centralized network state to be used by applications and act as a common intermediary with the data plane. Figure 4.10 shows the normal execution in an SDN environment. Upon receiving a packet it does not know how to handle, the switch sends an *event* to the controller. The controller delivers the event to the applications, which afterwards apply their logic based on this event, and eventually instruct the controller to send *commands* to the switches (e.g., to install flow rules).

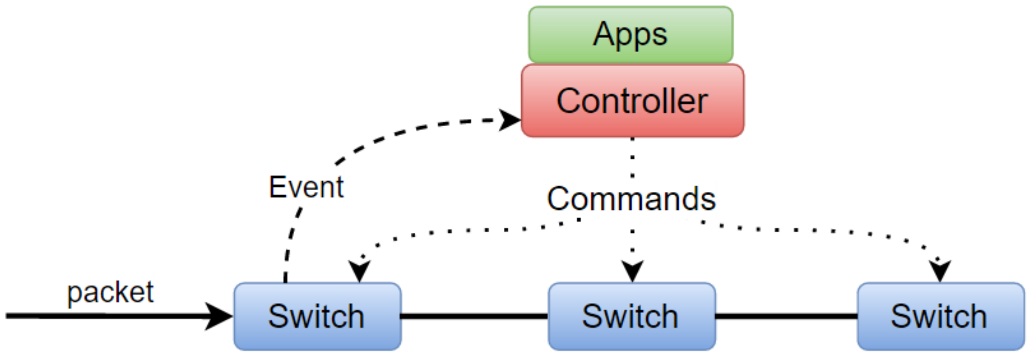


Figure 4.10: SDN flow execution

A trivial implementation of SDN using a centralized controller would lead to an undesirable outcome:

a *single point of failure*. To guarantee the required availability of network control, it is necessary the controller platform to be made fault-tolerant. Fault tolerance demands transparency: for a controller that claims having such ability – in other words, for it to be logically centralised – applications that run on it should operate correctly in the presence of faults. This is a fundamental requirement, as in case of controller failures the network view needs to be maintained consistent, otherwise applications will operate in a stale network view, leading to network anomalies that can have undesirable consequences (e.g., security breaches) [93].
To address this problem, traditional replication techniques are usually employed [110, 92, 114]. However, achieving a consistent network view in the controllers is not enough to offer logically centralised control – as such existing approaches are prone to the above-mentioned anomalies. In SDN, it is necessary to include switch state into the system model to fulfil this goal. Since switches are programmed by controllers (and controllers can fail), there must be mechanisms to ensure that the entire event-processing cycle of SDN is handled consistently.
A correct, fault-tolerant SDN environment needs to ensure *observational indistinguishability* [76] between an ideal central controller and a replicated controller platform. Informally, to ensure observational indistinguishability the fault-tolerant system should behave the same way as a fault-free SDN for its users (end-hosts and network applications). For this purpose, it is necessary the following three properties to be met: *total event ordering*; *exactly-once event processing*; and *exactly-once execution of commands*. We elaborate on these requirements in Section 4.3.1.
To the best of our knowledge, the problem of correct, fault-tolerant SDN control has only been addressed in the work by Katta *et al.* [76]. The proposed solution, Ravana, handles the entire event-processing cycle as a transaction – either all or none of the components of this transaction are executed. By correctly handling switch state this system guarantees SDN correctness even under fault. To achieve these properties, however, Ravana requires modifications to the OpenFlow protocol and to existing switches. These requirements preclude its adoption on existing systems.
Faced with this challenge, we propose Rama, a fault-tolerant SDN controller platform that, similar to Ravana, offers a transparent control plane that allows unmodified network applications to run in a consistent and fault-tolerant environment. The novelty of the solution lies in Rama not requiring changes to OpenFlow nor to the underlying hardware, allowing immediate deployment. For this purpose, Rama exploits existing mechanisms in OpenFlow and orchestrates them to achieve its goals. The main contributions of this work can be summarized as follows:

- A protocol for fault-tolerant SDN that provides the correctness guarantees of a logically centralised controller *without* requiring changes to OpenFlow or modifications to switches.
- The implementation and evaluation of a prototype controller – Rama – that demonstrates the overhead of the solution to be modest.

4.3.1 Fault-tolerant SDN

Traditional techniques for replicating controllers do not ensure correct network behaviour in case of failures. The reason is that these techniques address only part of the problem: maintaining consistent state in controller replicas. By not considering switch state (and the interaction controller-switches) inconsistencies may arise, resulting in potentially severe network anomalies. In this section we present a summary of the problems of using techniques that do not incorporate switches in the system model, which lead to the design requirements of a *correct* fault-tolerant SDN solution. We also present Ravana [76], the first fault-tolerant controller that achieves the required correctness guarantees for SDN.

4.3.1.1 Inconsistent event ordering

Since OpenFlow 1.3, switches can maintain TCP connections with multiple controllers. In a fault-tolerant configuration switches can be set to send all their events to all known controller replicas.

As replicas process events as they are received, each one may end up building a different internal state. Although TCP guarantees the order of events delivered by each switch, there are no ordering guarantees between events sent to controllers by the different switches, leading to the problem.
Consider a simple scenario with two controller replicas (c1 and c2) and two switches (s1 and s2) that send all events to both controllers. Switch s1 sends two events – e1 and e2, in this order – and switch s2 sends two other events – e3 and e4, in this order. One possible outcome where both controllers receive events in a different order while respecting the TCP FIFO property is c1 receiving events in the order e1, e3, e2, e4 and c2 receiving in the order e3, e4, e1, e2. Unfortunately, an inconsistent ordering of events can lead to incorrect packet-processing decisions. As a result of this consistency problem we derive the first design goal for a fault-tolerant and correct SDN controller:

Total event ordering: controllers replicas should process the same (total) order of events and subsequently all controller application instances should reach the same internal state.

4.3.1.2 Unreliable event delivery

In order to achieve a total ordering of events between controller replicas two approaches can be used:

1. The master (primary) replica can store controller state (including state from network applications) in an external consistent data-store (as in Onix [82]);
2. The controller state can be kept consistent using replicated state machine protocols.

Although both approaches ensure a consistent ordering of events between controller replicas, they are not fault-tolerant in the standard case where only the master controller receives all events.
If we consider - for the first approach – that the master replica can fail between receiving an event and finishing persisting the controller state in the external data-store (which happens after processing the event through controller applications), that event will be lost and the new master (i.e., one of the other controller replicas) will never receive it. The same can happen in the second approach: the master replica can fail right after receiving the event and before replicating it in the shared log (which in this case happens before processing the event through the controller applications). In these cases, since only the crashed master received the event, the other controller replicas will not have an updated view of the network. Again, this may cause severe network problems. Similar problems can occur in case of repetition of events. These problems lead to the second design goal:

Exactly-once event processing: All the events sent by switches are processed, and are neither lost nor processed repeatedly.

4.3.1.3 Repetition of commands

In either traditional state machine replication or consistent storage approaches, if the master controller fails while sending a series of commands, the new elected master may send repeated commands. This may happen when the old master fails before informing the slave replica of its progress. Since some commands are not idempotent, its duplication can lead to undesirable network behaviour. This problem leads to the third and final design goal:

Exactly-once command execution: any series of commands are executed only once on the switches.

4.3.1.4 Existing approaches

Ravana [76] is the first controller to provide correct fault-tolerant SDN control. To achieve this, the system processes control messages transactionally and exactly once (at both the controllers and the switches) using a replicated state machine approach, but without involving the switches in an expensive consensus protocol.

The protocol used by Ravana is briefly explained here. Switches buffer events (as they may need to be retransmitted) and send them to the master controller that will replicate them in a shared log with the slaves. The controller will then reply back to the switch acknowledging the reception of the events. Then, events are delivered to applications that may after processing require one or more commands to be sent to switches. Switches reply back to acknowledge the reception of these commands and buffer them to filter possible duplicates.
While Ravana allows unmodified applications to run in a fault-tolerant environment, it requires modifications to the OpenFlow protocol and to switch hardware. Namely, Ravana leverages on buffers implemented on switches to retransmit events and filter possible repeated commands received from the controllers. Also, explicit acknowledgement messages must be added to the OpenFlow protocol so that the switch and the controller acknowledge received messages. Unfortunately, these requirements preclude immediate adoption of Ravana. For instance, it is not antecipated OpenFlow to be extended to include the required messages anytime soon. These limitations are the main motivation for our proposal.

4.3.2 Design

Our proposal, Rama[7], is driven by the following four requirements. First, the system should maintain a correct and consistent state even in the presence of failures (in both the controllers and switches). Second, the consistency and fault-tolerance properties should be completely transparent to applications. Third, the performance of the system should not degrade as the number of network elements (events and switches) grows. Fourth, the solution should work with existing switches and not require new additions to the OpenFlow protocol, for immediate deployability.

4.3.2.1 Architecture

The main components of the architecture of Rama are: (i) *OpenFlow enabled switches* (switches that are implemented according to the OpenFlow specification), (ii) *controllers* that manage the switches and (iii) a *coordination service*. In our model, we consider only one network domain with one primary controller and one or more backup controllers, depending on the number of faults to tolerate. Each switch connects to one primary controller and multiple (f to be precise) backup controllers (to tolerate up to f crash controller faults). This primary/backup model is supported by OpenFlow in the form of master/slave and allows the system to tolerate controller faults. When the master controller fails, the remaining controllers will elect a new leader to act as the new master for the switches managed by the crashed master. This election is supported by the coordination service.
The coordination service offers strong consistency and abstracts controllers from complex primitives like fault detection and total order, making them simpler and more robust. Note that the coordination system requires a number of replicas equal to $2f+1$, with f being the number of faults to tolerate. The strong consistency model assures that updates to the coordination service made by the master will only return when they are persistently stored. This means that slaves will always have the fresh modifications available as soon as the master receives confirmation of the update. This results in a consistent network view among all controllers even if some fail. In addition to the controllers' state, the switches also maintain state that must be handled consistently in the presence of faults. Fulfilling this request is the main goal of the protocol we present next.

4.3.2.2 Rama protocol

In an SDN setting, switches generate events (e.g., when they receive packets or when the status of a port changes) that are forwarded to controllers. The controllers run multiple applications that process the received events and may send commands to one or more switches in reply to each event. This cycle repeats itself in multiple switches across the network as needed.

[7]In the Hindu epic Ramayana, Rama is the hero whose wife (Sita) is abducted by Ravana.

In order to maintain a correct system in the presence of faults, one must handle the state in the controllers and the state in the switches consistently. To ensure this, the entire cycle presented in Figure 4.11 is processed as a *transaction*: either all or none of the components of this transaction are executed. This means that (i) the events are processed exactly once at the controllers, (ii) all controllers process events in the same (total) order to reach the same state, and (iii) the commands are processed exactly once in the switches. Because the standard operation in OpenFlow switches is to simply process commands as they are received, the controllers must coordinate to guarantee the required exactly-once semantics.

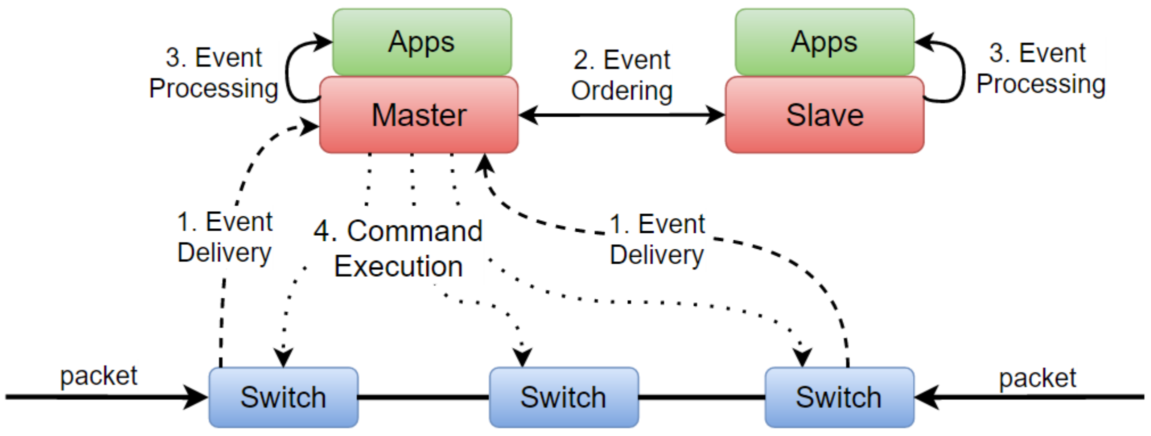


Figure 4.11: Rama control loop

By default, in OpenFlow a master controller receives all asynchronous messages (e.g., `OFPT_PACKET_IN`), whereas the slaves controllers only receive a subset (e.g., port modifications). With this configuration only the master controller would receive the events generated by switches. There are two options to solve this problem. One is for slaves to change this behaviour by sending an `OFPT_SET_ASYNC` message to each switch that modifies the asynchronous configuration. As a result, switches send all required events to the slaves. Alternatively, all controllers can set their role to `EQUAL`. The OpenFlow protocol specifies that switches should send all events to every controller with this role. Then, controllers need to coordinate between themselves who the master is (i.e., the one that processes and sends commands). We have opted for the second solution and use the coordination service for leader election amongst controllers.
The fault-free execution of the protocol is represented in Figure 4.12. In the figure we consider a switch to be connected with one master controller and a single slave controller. The main idea is that switches must send messages to *all controllers*, so that they can coordinate themselves even if some fail at any given point.
The master controller then replicates the event in a shared log with the other controllers, imposing a total order on the events received (to simplify, the coordination service is omitted from the figure). When the event is replicated to the shared log between controller replicas, it is processed by the master controller applications, which will generate zero or more commands. To guarantee exactly-once semantics, the commands are sent to the switches in bundles (a feature introduced in OpenFlow 1.4). With this feature a controller can open a bundle, add multiple commands to it and then instruct the switch to commit all commands present in the bundle in an atomic and ordered fashion.
Rama uses bundles in the following way. When an event is processed by all modules, the required commands are added by the master controller to a bundle. The master then sends an `OFPBCT_COMMIT_REQUEST` message to each switch affected by the event. The switch processes the request and tries to apply all the commands in the bundle in order. Afterwards, it sends a reply message indicating if the Commit Request was successful or not. This message is used by Rama as an acknowledgement.
Again, we need to make sure that this reply message is sent to all controllers. This is a challenge,

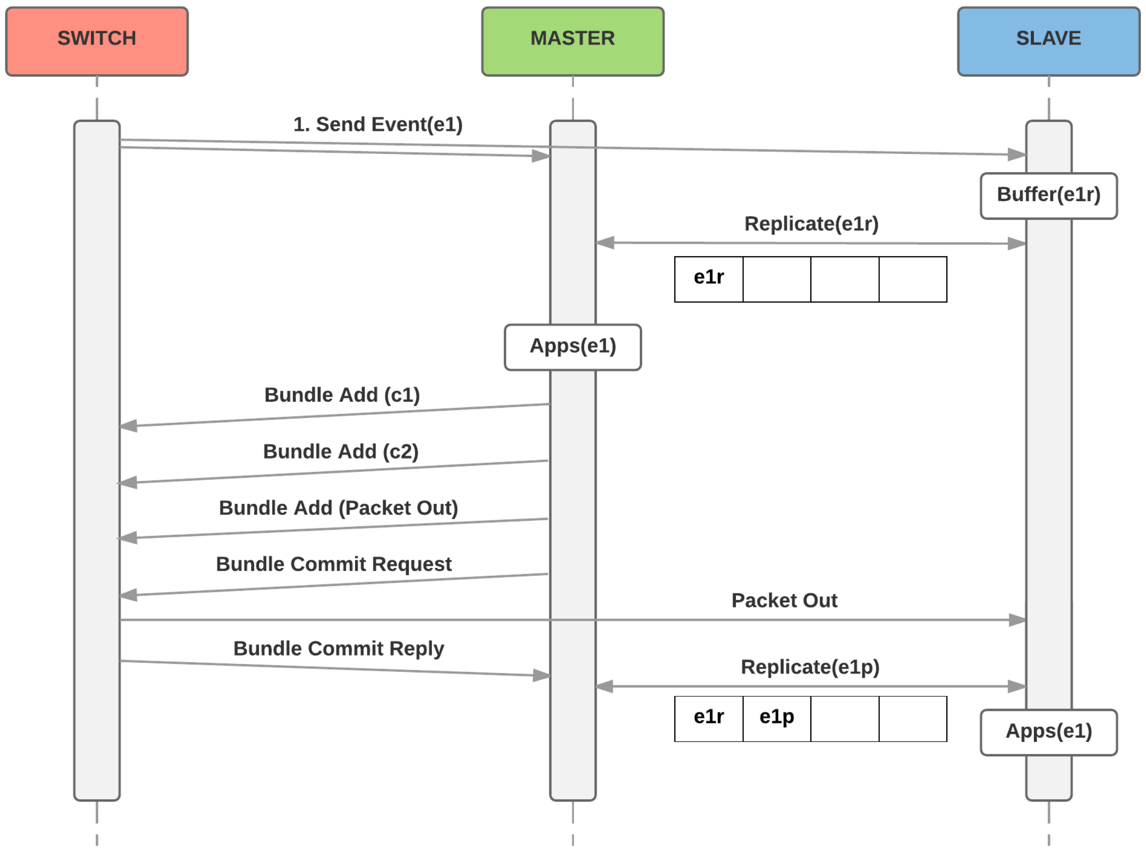


Figure 4.12: Fault-free case of the protocol

because Bundle Replies are Controller-to-Switch messages and hence are only sent to the controller that made the request (using the same TCP connection). To overcome this challenge we introduce a new mechanism in Rama. The way we inform other controllers if the bundle was committed or not (so that they can decide later if they need to resend specific commands) is by including one `OFPT_PACKET_OUT` message in the end of the bundle with the action `output=controller`. The outcome is that the switch will send the information included in the `OFPT_PACKET_OUT` message to all connected controllers in a `OFPT_PACKET_IN` message. This message is set by the master controller to inform slave controllers about the events that were fully processed by the switch (in this bundle). This prevents a new master from sending repeated commands, thus guaranteeing exactly-once semantics.
The master finishes the transaction by replicating an `event processed message` in the log, informing backup controllers that they can safely feed the corresponding event in the log to their applications. This is done to simply bring the slaves to the same updated state as the master controller (the resulting commands sent by the applications are naturally discarded).

Fault cases. When the master controller fails, the backup controllers will detect the failure (by timeout) and run a leader election algorithm to elect a new master for the switches. Upon election, the new master must send a Role Request message to each switch, to register as the new master. There are three main cases where the master controller can fail:

1. Before replicating the received event in the distributed log (Figure 4.13);
2. After replicating the event but before sending the Commit Request (Figure 4.14);
3. After sending the Commit Request message.

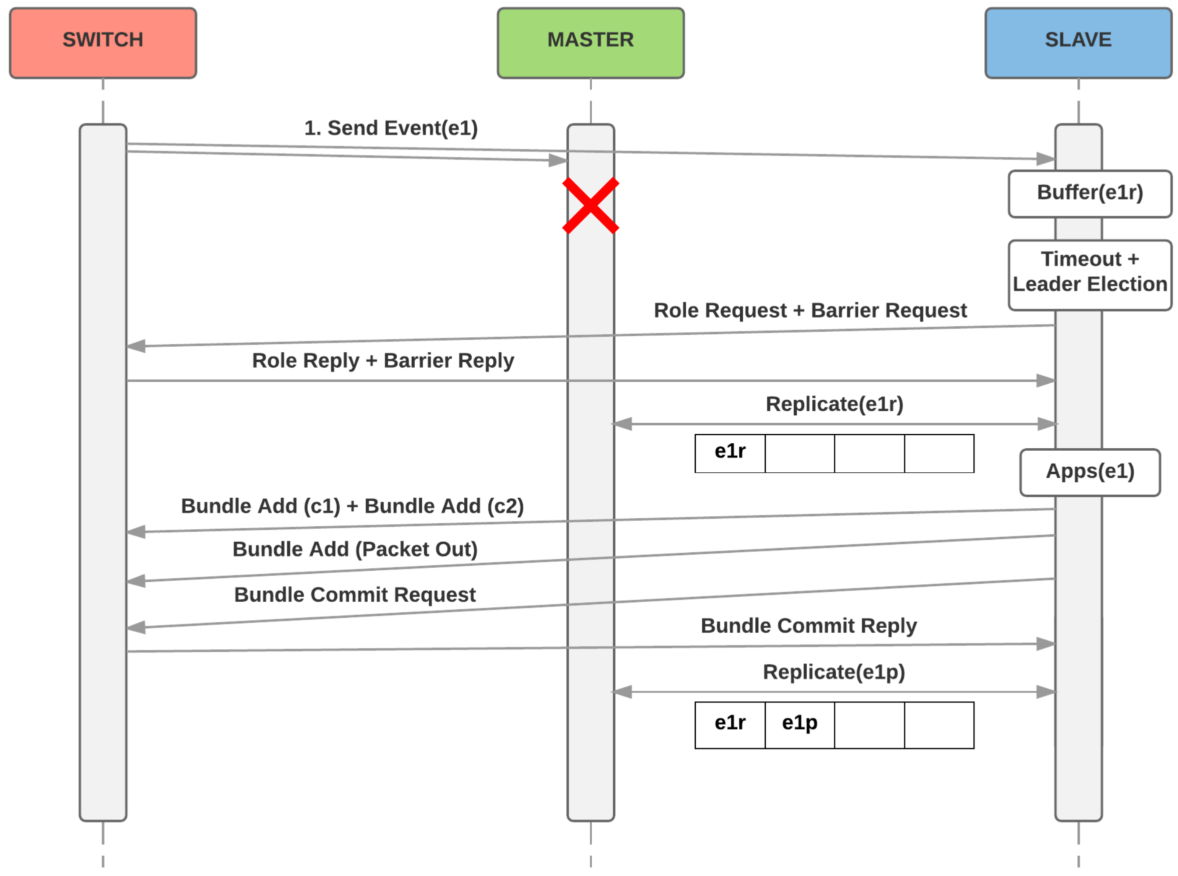


Figure 4.13: Case of the protocol where the master fails *before* replicating the event received

In the first case, the master failed to replicate the received events to the shared log. As slave controllers receive and buffer all events, no events are lost. First, the new master must finish processing any events logged by the older master. Note that events marked as processed have their resulting commands filtered. This makes the new master reach the same internal state as the previous one before choosing the new order of events to append to the log (this is valid for all other fault cases). The new elected master then appends the buffered events in order to the shared log and continues operation (feeding the new events to applications and sending commands to switches).
In the cases where the event was replicated in the log (cases 2 and 3), the master that crashed may or may not have issued the Commit Request message. Therefore, the new master must carefully verify if the switch has processed everything it has received before re-sending the commands and the Commit Request message. To guarantee ordering, OpenFlow provides a Barrier message, to which a switch can only reply after processing everything it has received before. If a new master receives a Barrier Reply message without receiving a Commit Reply message (in form of `OFPT_PACKET_OUT`), it can safely assume that the switch did not receive nor execute a Commit Request for that event from the old master (case 2)[8]. Even if the old master sent all commands but did not send the Commit Request message, the bundle will never be committed and will eventually be discarded. Therefore, the new master can safely resend the commands. In case 3, since the old master sent the Commit Request before crashing, the new master will receive the confirmation that the switch processed the respective commands for that event and will not resend them (guaranteeing exactly-once semantics for commands).

[8]This relies on the FIFO properties of the controller-switch TCP connection.

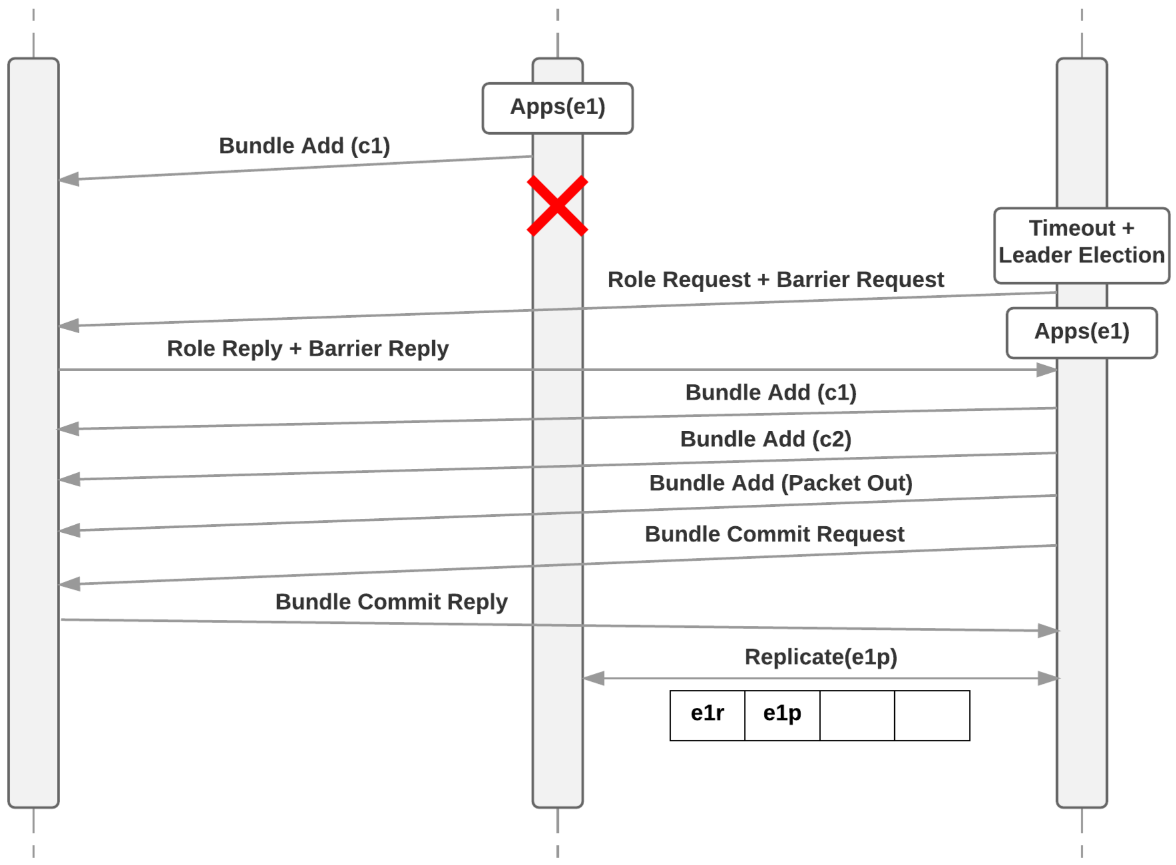


Figure 4.14: Case of the protocol where the master fails *after* replicating the event

Property	Ravana	Rama
At least once events	Buffering and retransmission of switch events	Switches send events to every controller with role EQUAL
At most once events	Event IDs and filtering in the log	
Total event order	Master appends events to a shared log	
At least once commands	RPC acknowledgments from switches	Bundle commit is known by every controller by piggybacking PacketOut in OpenFlow Bundle
At most once commands	Command IDs and filtering at switches	

Table 4.3: How Rama and Ravana achieve the same consistency properties using different mechanisms

4.3.3 Correctness

The Rama protocol we propose in this work was designed to guarantee correctness of fault-tolerant SDN control. We define correctness as in [76], where the authors introduce the concept of observational indistinguishability in the SDN context, defined as follows:
Observational indistinguishability: If the trace of observations made by users in the fault-tolerant system is a possible trace in the fault-free system, then the fault-tolerant system is observationally indistinguishable from a fault-free system.

For observational indistinguishability, it is necessary to guarantee transactional semantics to the entire control loop, including (i) *exactly-once event delivery*, (ii) *event ordering and processing*, and (iii) *exactly-once command execution*. In this section we summarize how the mechanisms employed by our protocol fulfil each of these necessary requirements. For a brief comparison with Ravana, see Table 4.3.

Exactly once event processing: events cannot be lost (processed *at least once*) due to controller faults nor can they be processed repeatedly (they must be processed *at most once*). Rama does not need switches to buffer events neither that controllers acknowledge each received event to achieve *at-least once event processing* semantics. Instead, Rama relies on switches sending the generated events to *all (f+1)* controllers (considering that the system tolerates up to f crash faults) so that at least one will know about the event. Upon receiving these events, the master replicates them in the shared log while the slaves add the events to a buffer. As such, in case the master fails before replicating the events, the new elected master can append the buffered events to the log. If the master fails after replicating the events, the slaves will filter the events in the buffer to avoid duplicate events in the log. This ensures *at-most once event processing* since the new master only processes each event in the log once. Together, sending events to all controllers and filtering buffered events ensures *exactly-once event processing*.

Total event ordering: to guarantee that all controller replicas reach the same internal state, they must process any sequence of events in the same order. For this, Rama relies on a shared log across the controller replicas (implemented using the external coordination service) which allows the master to dictate the order of events to be followed by all replicas. Even if the master fails, the new elected master always preserves the order of events in the log and can only append new events to it.

Exactly once command execution: for any given event received from one switch, the resulting series of commands sent by the controller are processed by the affected switches exactly *once*. As Rama cannot rely on switches acknowledging and buffering the commands received from controllers (to filter duplicates), it uses OpenFlow Bundles to guarantee transactional processing of commands. Additionally, the Commit Reply message, which is triggered after the bundle finishes, is sent to *all* controllers (by including the Packet Out message at the end of the bundle) and thus acts as an acknowledgement that is independent of controller faults. This way, upon becoming the new master, the controller replica has the required information to know if the switch processed the commands inside the bundle or not, without relying on the crashed master. Furthermore, the new master sends a Barrier Request message to the switch. Receiving the corresponding Barrier Reply message guarantees that commands will be processed by the switches *exactly-once*.
It is important to note that we also consider the case where switches fail. However, this is not a special case of the protocol because it is already treated by the OpenFlow protocol under normal operation. A switch failure will generate an event in the controller which will be delivered to applications, for them to act accordingly (e.g., re-route traffic around the failed switch). A particularly relevant case is when a switch fails before sending the Commit Reply to the master and the slave controllers. Importantly, this event does not result in transaction failure. Since this is a normal event in SDN, the controller replicas simply mark pending events for the failed switch as processed and continue operation.
While we detail our reasoning as to why our protocol meets the correctness requirements of observational indistinguishability in SDN, modelling the Rama protocol and giving a formal proof is left as future work.

4.3.4 Implementation

We have built Rama on top of the Floodlight controller. For coordination, we opted for ZooKeeper [72]. This service abstracts controllers from fault detection, leader election, and event transmission and storage (for controller recovery). Rama introduces two main modules into Floodlight: the *Event Replication* module and the *Bundle Manager* module. Additionally, the Floodlight architecture was

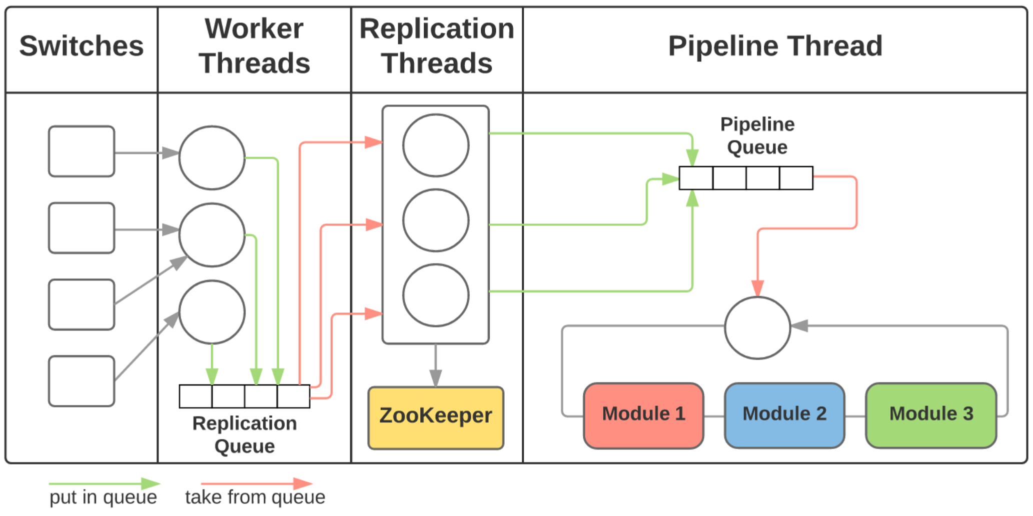


Figure 4.15: Rama thread architecture

optimised for performance by introducing parallel network event collection and logging (Rama's multi-thread architecture is shown in Figure 4.15), and by batching events. The multi-thread paralelism is introduced carefully, not to break TCP FIFO order of event processing, as will be explained next. An additional requirement is to make the control plane transparent for applications to execute unmodified.

Parallelism. In the original Floodlight, worker threads are used to collect network events and to process the modules pipeline (in Floodlight network applications are called "modules"). This design precludes event batching and other optimisations. Ideally, we want to free the threads that collect network events as soon as possible so that they can keep collecting more events. For this purpose, the worker threads' only job in Rama is to push events to the Replication Queue. Events for a particular switch are collected always by the same thread (although each thread can be shared by several switches) and thus TCP FIFO order is guaranteed in the Replication Queue. Next, the Rama runtime imposes a total order on the events by giving them a monotonically increasing ID. As such, several Replication threads can then take events from this queue and execute the logic in the Event Replication module, which will send the events to ZooKeeper in batches, without breaking the required total order for correctness. When ZooKeeper replies to the request, the events are added to the Pipeline Queue to be processed by the Floodlight modules. A single thread is used in this step, to guarantee the total order. The slave replicas also follow the total order from the IDs assigned by the master.
The Event Replication module is transparent to other modules as it acts before the pipeline. The modules will continue to receive events as usual in Floodlight and process them by changing their internal structures and sending commands to switches.

Event Replication and ZK Manager. The Event Replication module is the bridge between receiving events from the worker threads and pushing them into the pipeline queue to be processed by Floodlight modules. Events are only added to the pipeline queue after being stored in ZooKeeper. To separate tasks, Event Replication leverages on the ZK Manager, an auxiliary class that acts as ZooKeeper client (establishing connection, making requests and processing replies) and keeps state regarding the events (an event log and an event buffer in case of slaves) and switch leadership. Event Replication and the ZK Manager work together to attain exactly-once event delivery and total order as follows.
When an event arrives at the Event Replication module, we check whether the controller is in master or slave mode. In master mode the event is replicated in ZooKeeper and added to its in-memory log. The events are replicated in ZooKeeper in batches, so each replication thread simply adds an event to the current batch and becomes free to process a new event. Eventually the batch will be sent to

ZooKeeper containing one or more events to be stored. Upon receiving the reply, the events are pushed to the pipeline queue, ordered according to the identifier given by the master to guarantee total order. In slave mode, the event is simply buffered in memory (to be used in the case where the master controller fails). A special case is when the event received is the Packet Out that the master controller included in the bundle. In this case, the slave marks that this switch already processed all commands for this event. Slaves also keep an event log as the master, but only events that come from the master are added to it. An important detail is that event identifiers are set by the master controller. Therefore, the events will be queued in the same order as they were in the master controller replica.

Bundle Manager. This module keeps state related to the open bundles for each switch (as result of an event) and is responsible for adding messages to the bundle, closing and committing it. We modified the write method in `OFSwitch.java` (the class that is used by all modules to send commands to switches) to call the Bundle Manager. This makes the process transparent to applications. This module will wrap the message sent by application modules in a `OFPT_BUNDLE_ADD_MESSAGE` and send it to the switch. Before committing the bundle, the Bundle Manager also adds the required `OFPT_PACKET_OUT` message.

Event batching. Floodlight thread architecture was modified to allow event batching, for performance reasons. The ZKManager groups events before sending them to ZooKeeper in batches. Batches are sent to ZooKeeper using a special request called `multi`, which contains a list of operations to execute (e.g., create, delete, set data). For event replication, the multi request will have a list with multiple create operations as parameter. This request is sent after reaching the maximum configured amount of events (e.g., 1000) or some time after receiving the first event in the batch (e.g., 50ms). This means that each event has a maximum delay bound (regarding event batching).

4.3.5 Evaluation

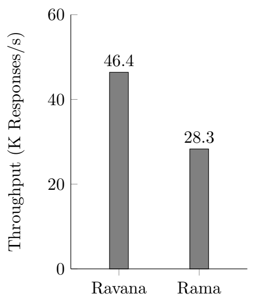
In this section we evaluate Rama to understand the costs associated with the mechanisms used to achieve the desired consistency properties.

4.3.5.1 Setup

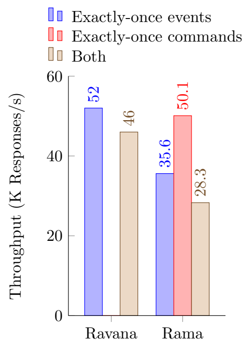
For the evaluation we used 3 machines connected to the same switch via 1Gbps links. Each machine has an Intel Xeon E5-2407 2.2GHz CPU and 32 GB (4x8GB) of memory. Machine 1 runs one or more Rama instances, machine 2 runs ZooKeeper 3.4.8, and machine 3 runs Cbench to evaluate controller performance. This setup tries to emulate a realistic scenario with ZooKeeper on one machine for fault-tolerance purposes, and Cbench on a different machine to include network latency.

4.3.5.2 Rama performance

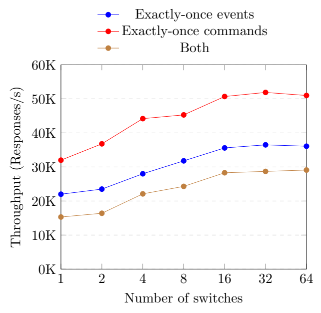
We have compared the performance of Rama against Ravana [76]. Figure 4.16a shows the throughput for each controller (for Ravana we use the results reported in the paper, as its authors considered a similar setup). For Rama measurements we run Cbench emulating 16 switches.
Rama achieves a throughput close to 30K responses per second. This figure is lower than Ravana's, as our solution incurs in higher costs compared to Ravana for the consistency guarantees provided. The additional overhead is caused by two requirements of our protocol. First, current switches' lack of mechanisms to allow temporary storage of OpenFlow events and commands require Rama to instruct switches to send all events to all replicas, increasing network overhead. Second, the lack of acknowledgement messages in OpenFlow leads Rama to a more expensive solution – bundles – to achieve its goals. The overhead introduced by these mechanisms is translated into reduced throughput when compared with Ravana.
In figure 4.16b we show, separately, throughput results considering the different levels of consistency provided by both Rama and Ravana. The exactly-once events consistency level (▮▮) ensures that no



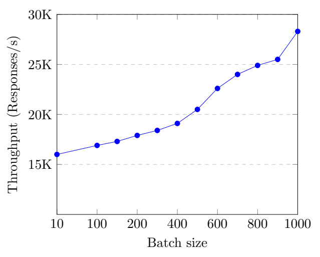
(a) Fault-tolerant controllers throughput



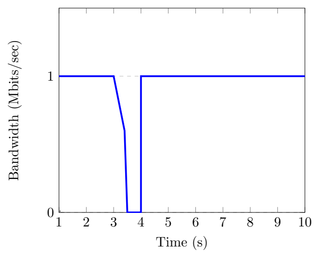
(b) Throughput with different consistency guarantees



(c) Rama throughput with different number of switches



(d) Variation of Rama throughput with batch size



(e) Rama failover time

Figure 4.16: Evaluation results

events are lost and that controllers do not process repeated events. Additionally, controllers must agree on a total order of events to be delivered to applications. For the latter, both Rama and Ravana rely on ZooKeeper to build a shared log across controllers. Note that neither Rama nor Ravana wait for ZooKeeper to persistently store requests on disk (they both use ZooKeeper in-memory).

The Exactly-once commands semantics (🟥) ensures that commands sent by controllers are not lost and that switches do not receive duplicate commands. Ravana relies on switches to explicitly acknowledge each command and filter repeated ones. For Rama, this includes maintaining state of all opened bundles for switches, and sending additional messages to the switches. Instead of replying only with a Packet Out as in Floodlight, Rama must send messages to open the bundle, add the Packet Out to it, close the bundle and commit it. To evaluate this case, we modified Cbench to make switches increase their counters only when they receive a Commit Request message from the controller. This allows a fair evaluation of the performance of Rama in a real system – indeed, in Rama a packet will only be forwarded after committing the bundle on the switch to guarantee consistent processing.

As shown in Figure 4.16b, some guarantees are costlier to ensure than others[9]. For instance, the cost of providing Exactly-once events semantics is higher than Exactly-once commands semantics. This result brings with it an important insight: the system bottleneck is the coordination service. In other words, the additional mechanisms Rama uses to guarantee the desired consistency properties add overhead but, crucially, system performance is not limited by these mechanisms.

Figure 4.16c shows how maintaining multiple switch connections affects Rama throughput. As switches send events at the highest possible rate, the throughput of the system saturates with around 16 switches. Importantly, the throughput does not decrease with a higher number of switches.

[9]Note that we do not include the results from Exactly-once commands in Ravana as these are not available in [76]. It is possible, however, to extrapolate that the results will be inline with the rest of the analysis.

4.3.5.3 Event batching

Rama batches events to reduce the communication overhead of contacting ZooKeeper. In practice, events are sent to ZooKeeper after reaching a configurable number of events in the batch (batching size) or after a configurable timeout (batching time).
To evaluate batching we conducted a series of tests with different configurations to understand how the batching size and time affects Rama performance (Figure 4.16d). Intuitively, a larger batching size will increase throughput, but as downside will also increase latency. As batching size increases, throughput increases due to the reduction of RPC calls required to replicate events.

4.3.5.4 Failover Time

To measure the time for Rama to react to failures we use the network emulator Mininet, OpenvSwitch [120], and iperf. We setup a simple topology in Mininet with one switch and two hosts, one to act as iperf server and another as client. We start the client and server in UDP mode, with the client generating 1 Mbit/sec for 10 seconds. The switch connects to two Rama instances and sends all events to both controllers. Each Rama instance is connected to the ZooKeeper server running on another machine (as before) with a negotiated session timeout of 500ms. To make sure that no rules are installed on the switch – so that events are sent to the controllers each time a packet arrives – we run Rama with a module that only forwards packets (using Packet Out messages) without modifying the switch's tables.
Figure 4.16e shows the reported bandwidth from the iperf server and indicates the time taken by Rama to react to failures. Namely, the slave replica takes around 550ms to react to faults. This includes the time for: (a) ZooKeeper to detect the failure and notify the slave replicas (500ms); (b) electing a new leader for the switches; (c) the new leader to transition to master (finish processing logged events from the old master to reach the same internal state); (d) append buffered events to the log and start delivering unprocessed events in the log to applications so they start sending commands to the switches. As is clear, the major factor affecting failover time is the time ZooKeeper needs to detect the failure of the master controller.

4.3.5.5 Summary

Rama comes close, but does not achieve the performance of Ravana. This is due to the fact that our system incurs in higher costs. Rama requires more messages to be sent over the network and introduces new mechanisms, such as bundles, which increase the overhead of the solution. Importantly, the overall cost to achieve the same consistency properties without changes to switches or OpenFlow is relatively modest.

4.3.6 Related work

Consistent SDN. Levin et al. [93] have explored the trade-offs of state distribution in a distributed control plane, motivating the importance of strong consistency in applications' performance. However, as noted in the CAP theorem, a system can not provide availability while also achieving strong consistency in the presence of network partitions. As such, fault-tolerant SDN architectures must use techniques to explicitly handle partitions in order to optimize consistency and availability (and thus achieving a tradeoff between them) [38]. OF.CPP [117] explores the consistency and performance problems associated with packet processing at the controller and proposes the use of transactional semantics to solve them. However, the proposed semantics in packet processing are not enough: controllers should also coordinate to guarantee the same semantics in the switches' state, a fundamental requirements our work fulfils.

Consistent network updates. The concepts of per-packet and per-flow consistency in SDN were introduced in [126] to provide a useful abstraction for applications: consistent network updates. With consistent updates, packets or flows in flight are processed exclusively by the old or by the new network policy (never a mix of both). The main mechanism used to guarantee consistent network updates is the use of a two-phase protocol to update the rules on the switches. In [42], Canini et al. extend this idea to a distributed control plane and formalize the notion of fault-tolerant policy composition. This class of proposals addresses consistent network updates, which is an orthogonal problem to the one addressed here.

Fault-tolerance in SDN. Botelho et al. [34] address fault tolerance in the control plane while achieving strong consistency with SMaRtLight, a fault-tolerant controller architecture for SDN. The controllers are coordinated through a data store, used to achieve durability and strong consistency. In [35] the authors extend their solution to a distributed deployment. In contrast to our solution, SMaRtLight requires applications to be modified to use the data store directly. More importantly, the solution does not consider the consistency of switch state in the system model. Ravana [76] was the first fault-tolerant controller that integrates switches into the problem. The techniques proposed by its authors guarantee correctness of event processing and command execution in SDN. The main differentiating factor of our work against Ravana is that our solution does not require changes to the OpenFlow protocol nor to switches.

Traditional fault-tolerance techniques. Viewstamped Replication [110], Paxos [92], and Raft [114] are well-known distributed consensus protocols used for replication of state machines in client-server models. None of these widely-used protocols is directly applicable in the context of SDN, where to guarantee correctness it is necessary not only to have consistent controller state, but also switch state.

4.3.7 Conclusions

In a fault-tolerant SDN, maintaining consistent controller state is not enough to achieve correctness. Unlike traditional distributed systems, in SDN it is necessary to consistently handle switch state to avoid loss or repetition of commands and events under controller failures. To address these challenges we proposed Rama, a consistent and fault-tolerant SDN controller that handles the entire event processing cycle transactionally.
Rama differs from existing approaches by not requiring modifications to the OpenFlow protocol nor to switches. This comes at a cost, as the techniques introduced in Rama incur in a higher overhead. As the overhead leads to a relatively modest decrease in performance, we expect this to be compensated by the fact that our solution is immediately deployable. We make our software available open source to further foster adoption of fault-tolerant SDN.
As for future work, besides devising a formal proof on the consistency guarantees Rama provides, we plan to address correctness in distributed SDN deployments and to consider richer fault models.

After presenting the techniques proposed to enhance the security and dependability of the multi-cloud network virtualization infrastructure, we now turn to the modules of our solution that deal with self-management of network security, before closing this deliverable.

Chapter 5 Self-Management of Network Security

This chapter presents the autonomic security management and enforcement framework for a SUPERCLOUD SDN-based network environment. First, we introduce a security agent for a Opendaylight cloud platform for flexible service chaining (Section 5.1). We then describe the framework for security policy management and enforcement (Section 5.2). Finally, we describe a use case on DDoS attack mitigation (Section 5.3) and present experimental results (Section 5.4).

5.1 OpenDaylight Security Agent

This security agent enables the definition of flexible chains of network security services, building on features already provided by an SDN such as OpenDaylight, instead of re-implementing them from scratch. We first introduce the context (Section 5.1.1). We then present OpenDaylight protocols used by the agent (Sections 5.1.2, 5.1.3, and 5.1.4) and how they may be combined (Section 5.1.5).

5.1.1 Motivation

Thousands of *network service functions*, especially security service functions like firewall, load balancing, NAT, HTTP enrichment, etc., are currently deployed. The way they are deployed today is static: each network service function is dedicated to one delivered service of a unique client, and setting this involves direct network configuration. Thereby, deploying a service can take long, is hardly scalable, is not resource-optimized, and may have a negative impact on time-to-market.

The purpose of *service chaining* is to simplify the way service functions are deployed and to make them application-driven. It consists of a *service function domain* containing a large group of service functions that can be dynamically linked together to deliver a service. This domain is bound to classifiers that decide which type of traffic enters into a chain of the domain. Service functions are presented as resources available for consumption that can be linked together based on policy. It provides a more generic approach to think about a service.

However, deployed service functions today are quite limited. Network traffic for each service delivered is not isolated and flows through all services regardless of need. Deploying granular and isolated services defined by a service policy increases configuration complexity. Service chains are still limited to acyclic graphs and there is no service policy that can be expressed in terms of intent rather than network configuration. More importantly, there is no service plane (like subscriber/employee information) carried and exchanged between service functions, drastically limiting possibilities in terms of QoS, policy and security enforcement.

OpenDaylight (ODL) modules for Service Function Chaining (SFC) and Group Based Policy (GBP) with Network Service Header (NSH) aim at enabling this approach. The SFC module used with NSH allows the definition of flexible service chains with a true service plane in an application-driven way. GBP allows policy definition, service chain selection, and service plane without requiring network knowledge or configuration.

The OpenDaylight security agent illustrates this approach, showing how these elements work together on a real implementation.

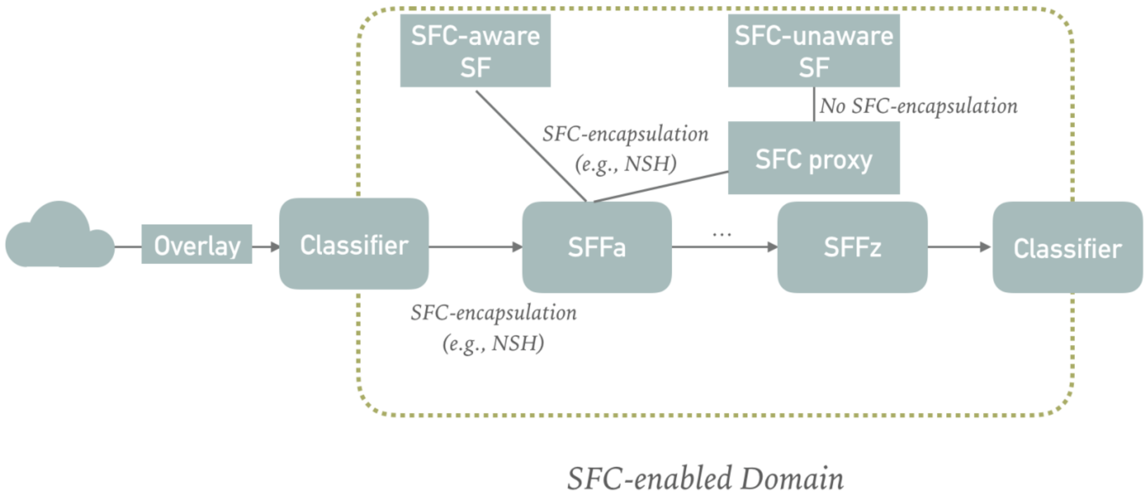


Figure 5.1: SFC domain with its components

5.1.2 SFC and NSH with OpenDaylight

A network integrating SFC allows to define an ordered list of network services through which certain network flows must pass. Services are then linked together in the network to create the service chain. ODL SFC can meet a wide range of architecture requirements. It ignores all the network complexity (VLAN, overlay, etc...) and allows users to focus on configuration and on links between the services in the chain. With SFC, the service chain becomes "application-driven" (i.e., application A dialogues with application B through a dedicated chain). Services are built using flexible, not linear, service chains. The insertion of services is done without having to take into account the topology of the network. To aim towards "application-driven" service chaining, the SFC relies on an SFC *encapsulation header*. This header/protocol gives information on the graph that the packet passes through and on its precise location in this graph. The Network Service Header is an example of SFC encapsulation that allows the user to define a complete service plane carrying granular service function information.
To achieve dynamic service chaining, the ODL SFC module includes the following components, shown in Figure 5.1:

- *Classifier:* determines which types of traffic need to be chained based on a policy table.
- *Service Function Forwarder:* redirect packets to the right service function based on the SFC encapsulation information.
- *Service Function Proxy:* handles SFC encapsulation information for SFC-unaware service function.
- *Service Function:* a function responsible for specific treatment of packets.

These components handle the graph defined by the user. Graphs define a *Service Function Chain (SFC)*, where each graph node represents a *Service Function (SF)*. Such SF graph nodes can be part of zero, one, or many SFCs. A given graph node can appear one or multiple times in a given SFC.

5.1.3 Network Service Header

The Network Service Header is a data plane protocol and SFC encapsulation used by ODL for representing a service chain in the network. This header is added to the packets transmitted in the chain. It indicates which service functions (Firewall, DPI, Load Balancer, etc...) they must pass through. Figure 5.2 presents the content of an NSH header.

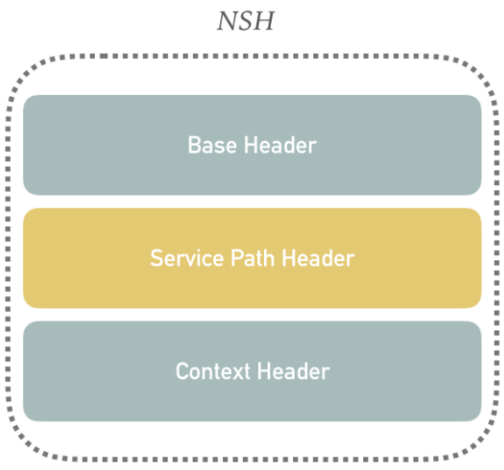


Figure 5.2: Network Service Header protocol

NSH is added to the package via a *Service Classifier.* The Service Classifier is an element that will enrich packets with information from the service chain it borrows and the metadata defined by the Group Based Policy module. The *Base Header* provides information about the service header and the payload protocol. The *Service Path Header* reflects the selection of a service path through a unique *Service Path Identifier (SPI).* The location of the packet in the SPI is given by the *Service Index.* SPI and SI tell the packet where to go without the need for configuration flow metadata or packet information itself. Finally, the *Context Header* carries metadata information along with a service path that can be shared between NSH-aware service functions. Such metadata can be derived from different sources (external system, network nodes, devices, orchestrator, service functions, etc.) and be used for QoS, policy and security enforcement, like in the use case shown in Figure 5.3.

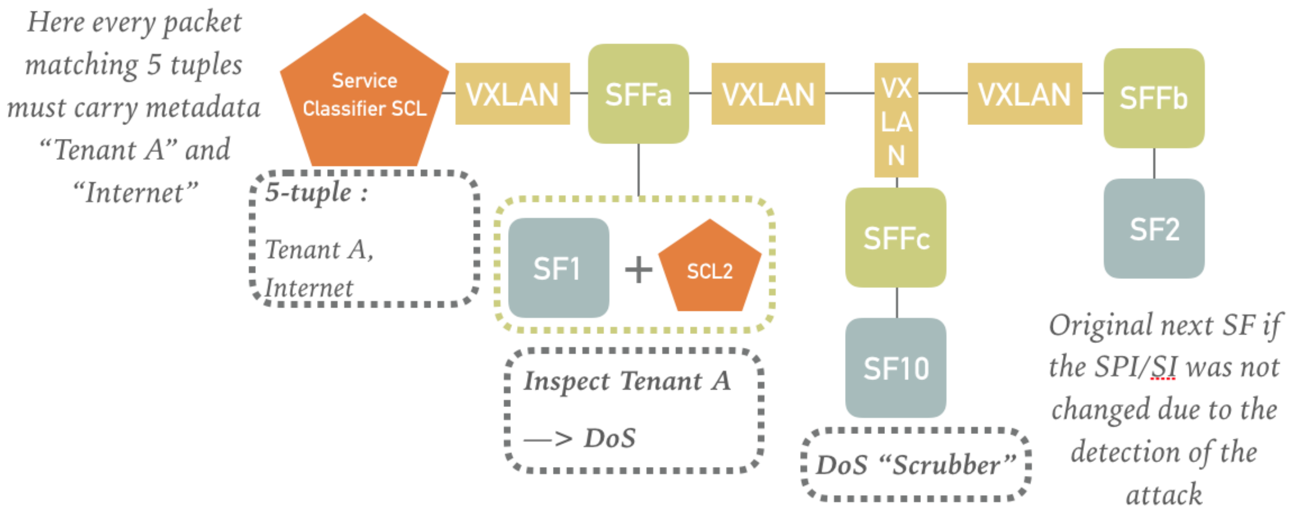

Here every packet matching 5 tuples must carry metadata "Tenant A" and "Internet"
Service Classifier SCL
VXLAN
SFFa
VXLAN
VX LA N
VXLAN
SFFb
5-tuple :
Tenant A, Internet
SF1
+
SCL2
SFFc
SF2
SF10
Inspect Tenant A —> DoS
DoS "Scrubber"
Original next SF if the SPI/SI was not changed due to the detection of the attack

Figure 5.3: Security use case by Network Service Header

In this use case, SF_1 reads the metadata and processes to inspection if the metadata matches "*Tenant A*", detects an attack and changes metadata from "*Internet*" to "*attack*". Based on the new information, SCL_2 changes SPI and SI in order to send the packet to a scrubber (SF_{10}). As shown in this example, NSH is not a transport or network header. NSH is never used by network nodes to transfer packets, but only to build the service plane.

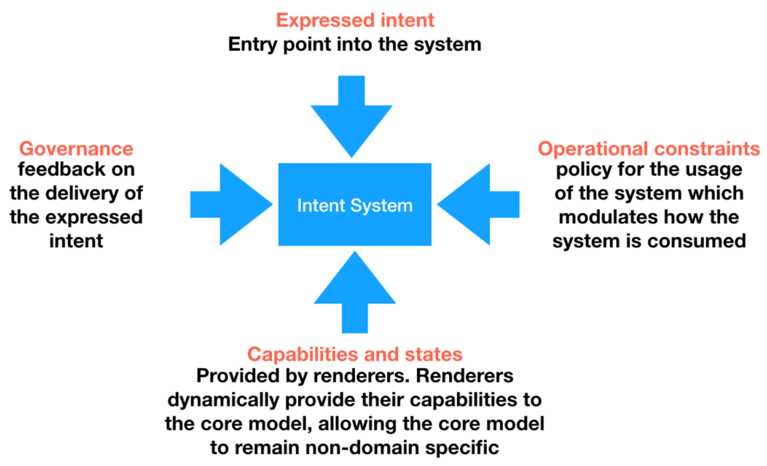

Expressed intent
Entry point into the system
Governance
feedback on the delivery of the expressed intent
Intent System
Operational constraints
policy for the usage of the system which modulates how the system is consumed
Capabilities and states
Provided by renderers. Renderers dynamically provide their capabilities to the core model, allowing the core model to remain non-domain specific

Figure 5.4: Intent System description

5.1.4 GBP: a new declarative way of expressing network configuration

GBP is at the core of OpenDaylight. It allows users to express network configuration in a declarative vs. imperative manner (i.e., "what you want" rather than "how to do it"). GBP lets the user focus on modeling, enabling processing and automation of policies without the hassle of network operations and configurations. GBP is achieved with the implementation of an Intent System, which can be seen as a process around an intent-driven data model containing no domain specifics and able of addressing multiple semantic definitions of intent (see Figure 5.4).
The GBP architecture is based on two models: the *Access Model* which is the core of the GBP Intent System policy resolution process, and the *Forwarding Model* which addresses networking topology description.

5.1.4.1 GBP Access Model

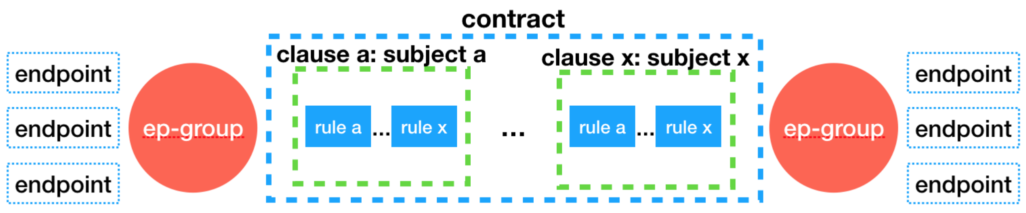


Figure 5.5: GBP Access Model

The GBP Access model, shown in Figure 5.5, allows a user to define how entities in a network can communicate. A user can set *End Points* which describe an entity and *contracts* describing the communication. In this model, the underlying network used is not specified.
Each entity is seen as a unique End Point. End Points with common policy rules are grouped together in an *End Point Group (EPG)*.
Contracts describe how End Points and/or End Point Groups can communicate. As shown in Figure 5.6, a contract consists of a set of *subjects* which describes in which way EPG can communicate. A subject is triggered by a *clause* matching against *requirements* and *capabilities* exposed by EPGs. Based on the traffic EPGs want to use, the subject applies the corresponding *rule* and performs any necessary actions on that traffic. A rule consists of a set of *classifiers* and *actions* applied to the traffic based on its classification.

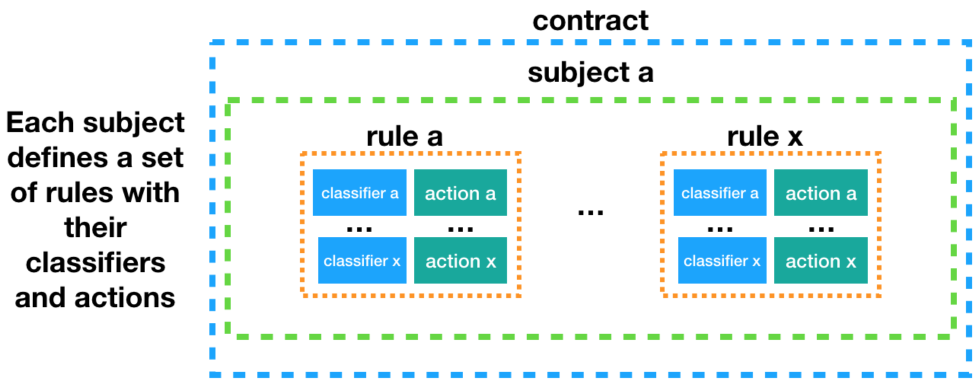


Figure 5.6: GBP Access Model contract

An sample classifier would match against all TCP traffic on port 80. An action describes what the operations to perform on the traffic before it reaches its destination. Actions could include tagging or encapsulating the traffic, redirecting the traffic, or applying a Service Function Chain.

5.1.4.2 GBP Forwarding Model

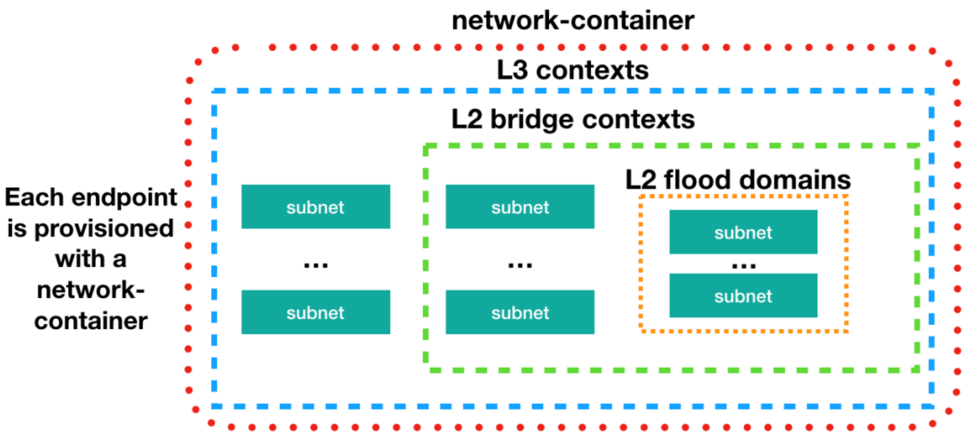


Figure 5.7: GBP Forwarding Model

This model, shown in Figure 5.7 describes different network containers that contracts and End Points can leverage.
It consists of an *L3 context* which is a namespace where traffic is passed at Layer 3. Several subnets can be included in an L3 context. Under this context, one finds the *L2 bridge context* where traffic can be sent at Layer 2. Several different subnets may also be defined. Finally, the *L2 flood domain* describes the network flooding behavior.

5.1.5 Combining GBP and SFC

Service policies can easily be applied to service chains, e.g., to add a policy to a firewall via information from the Group Based Policy module. For example, using Group Based Policy, we can express relationships such as "the EPG_A group can communicate with the EPG_B group" without having to worry about the underlying exchanges between the service layer and the groups. To achieve this, SFC integrates metadata representing EPG_A and EPG_B, the *representational metadata*. These metadata, generated by GBP, will transport information such as subscribers or employees information to the various services and enable use cases such as the one described Section 5.1.3. This ability to transmit high-level information is essential for SFC.

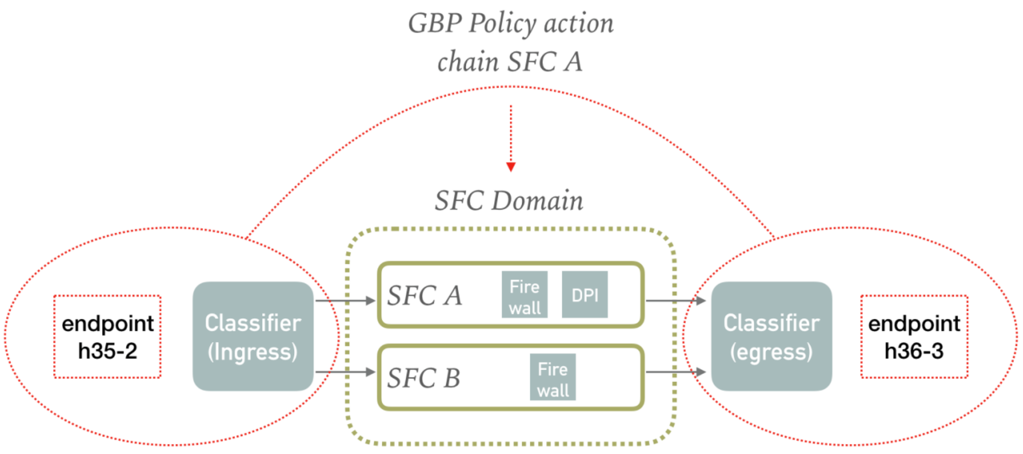


Figure 5.8: Interaction between GBP and SFC

Figure 5.8 describes a typical interaction and the scope of each module. Concerning the contract between End Point $h35-2$ and $h36-3$, GBP defined the action "*SFC A*". GBP asks SFC if it has the chain "*SFC A*". SFC then creates the needed overlay bridges and forwarding rules into flow tables. After that, SFC return path-ID, starting index, first hop IP:Port and encapsulation to GBP. Finally, GBP creates the necessary classifier rules to direct packets according to the contract.

5.2 Policy Driven Management and Enforcement Framework

In this Section, we will firstly present a short description of the policy management and enforcement framework defined for the SUPERCLOUD project. Secondly, we will introduce the two use cases and finally, we will detail the experimental results which demonstrate that our framework can successfully reduce the collateral damage on a customer network caused by the attack traffic targeting another customer network.

5.2.1 Design Components

The functional components of the policy management framework are described as follows:

- The **SDN Controller** provides an interface with the switches. The controller is logically centralized, and multiple controllers can also be used to provide scalability. When a message is received by the framework to push or modify the rule, the controller is responsible for sending the messages to the switches.
- The **Monitoring Component (MC)** is responsible for receiving alerts and notifications from the different customers. Particularly, it extracts the events and conditions from the security alerts and QoS request messages, which are used by the Policy Decision Point (PDP) in order to select the policy from the policy database.
- The **Policy DataBase (PDB)** is essentially a repository containing the high-level security and network policies defined by the network operator, without detailing the specific deployment strategy. Policies can be specified in any format defined by the network administrator. A Policy

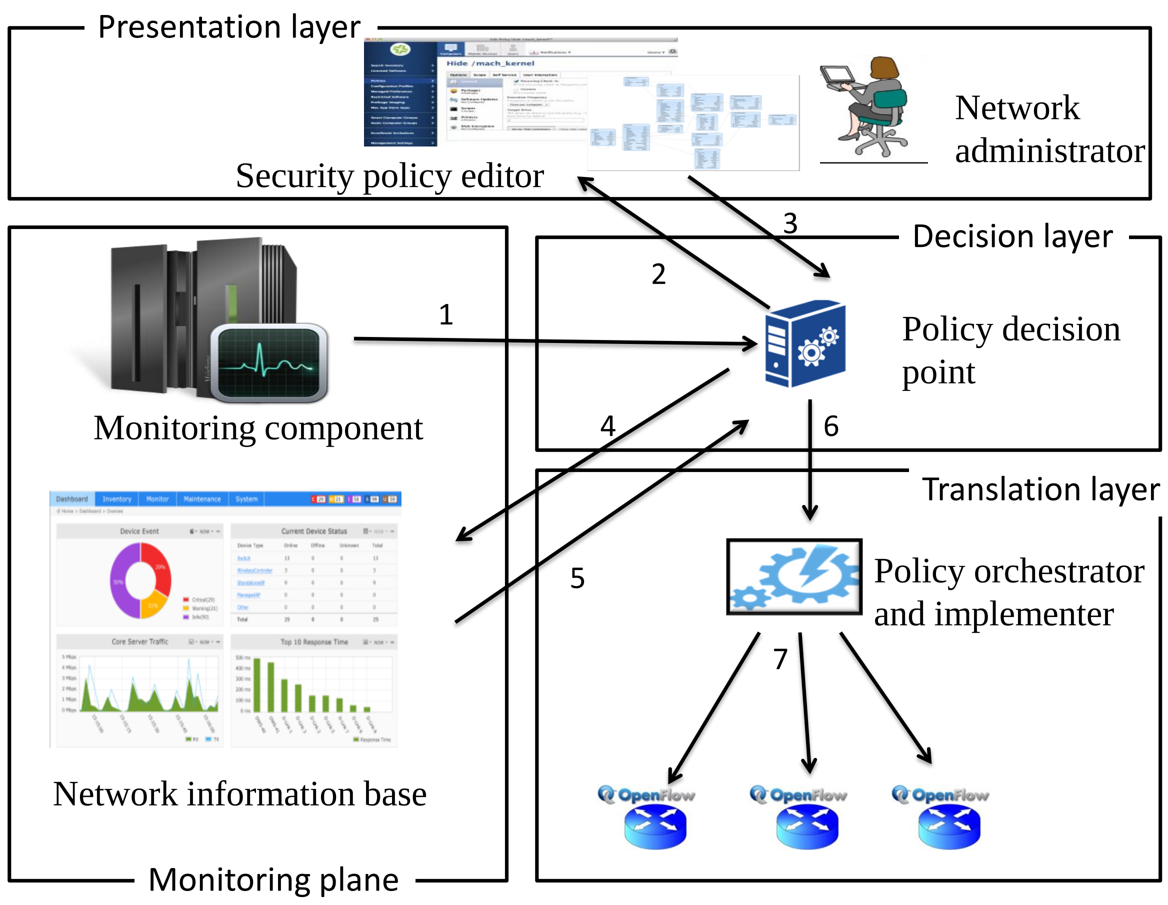


Figure 5.9: Policy Management and Enforcement Framework

ID is used to index the policies in the database, which helps in retrieving the policy when an event occurs.

- The **Policy Decision Point (PDP)** is in charge of the global policy decision. It acts as an orchestrator between different components in the framework. Moreover, the PDP maintains a table containing the list of middleboxes that can be traversed in the network for the different types of suspicious traffic. For example, the PDP can specify that a suspicious traffic should traverse a firewall or a Network Address Translator (NAT).
- The **Network Information Base (NIB)** maintains a table containing the list of paths ordered by bandwidth guarantees. It computes paths in the list depending on the network status of the ISP. Network status information may be obtained from external tools, such as OpenNetMon [152], which allow to maintain a traffic matrix for different paths and switches in the network. NIB also maintains a mapping table containing the list of middleboxes and their deployment location in the network to where suspicious flows can be steered.
- The **Policy Orchestrator and Implementer (POI)** contains the OpenFlow rule templates for high-level actions to be enforced in the data plane devices. `OpenFlow` rule templates contain the guidelines to specify how the different activities can be executed in the network. It provides an additional level of modularity to define the format for the different types of tasks that can be performed. It is an important part of the translation process as it provides an abstract view for the concrete rules which are to be deployed.

5.2.2 Operational Workflow

The design overview of our framework is shown in Fig. 5.9, consisting of several functional components as described in Section 5.2.1. To better illustrate the specific functions of the different components, a detailed operational workflow is given as follows:

1. A policy database is populated by the administrator with the high-level network and security policies.
2. An event is triggered at the ISP controller when a security alert is received by the `Monitoring component` from the customer. Then, some information is extracted from the alert. It contains a number of conditions such as the attack type (referred to as event), the flow class (e.g., suspicious) and the impact severity (e.g., low), as well as the flow information (i.e., source and destination IP addresses). This information is then forwarded to the PDP [68].
3. Based on the event and conditions received from the MC, the PDP activates a policy in the PDB which outputs a high-level action (e.g., forward, redirect or drop) to be enforced. Then the PDP forwards the high-level action, along with the flow information and the bandwidth to be provided, to the NIB to obtain the details of a concrete path to route the flow.
4. The NIB computes one or multiple best-fitted paths based on the transmitted parameters. Flow information allows to identify the ingress and egress nodes within the ISP networks; bandwidth requests indicate a required level of QoS or a required degradation against a suspicious flow.
5. After computation, the NIB forwards the path details containing the switch IDs and associated output ports and a Network Service Header (NSH) to the PDP. If all the paths in the ISP network are congested then the NIB returns a message stating that "no paths are available" to the PDP.
6. The PDP forwards the path details, NSH and flow information as an OpenFlow match field to the POI for enforcing low-level rules. In the case when no paths are available to route the traffic, the PDP outputs a rate-limit action against the flows which are causing the congestion in the path, in order to maintain a fair share of bandwidth to all flows.

5.3 Use Case

This Section gives a use case about DDoS attack mitigation. The network topology is illustrated in Fig. 5.10, which consists of one ISP, three customers (C_1, C_2, and C_3) and four external hosts (H_1, H_2, H_3, and H_4). In this use case, we consider that hosts H_1 and H_2 send malicious traffic towards customer C_1. H_2 aims to congest C_1's network as well as the ISP network, while H_3 and H_4 generate legitimate traffic towards C_2 and C_3 respectively. In the following, we firstly describe the settings with respect to the deployment of our translation mechanism. We will then give a step-by-step example to showcase how a high-level mitigation policy can be translated into low-level rules for enforcement.

5.3.1 Settings for On-demand Attack Mitigation

Security alert sent by the customer: Security alerts are used to report attacks. They contain the network and assessment attributes. Security alerts are described using the IDMEF [57] format. Listing 5.1 shows a security alert forwarded by the customer C_1 to the ISP.

- The network attributes are comprised of the IP addresses and the attack type. The security alert contains network information such as the source IP address, here 10.0.0.2 (H_2), and the destination IP address, here 10.0.0.3 (C_1), while the attack type is `ICMP Flood`.

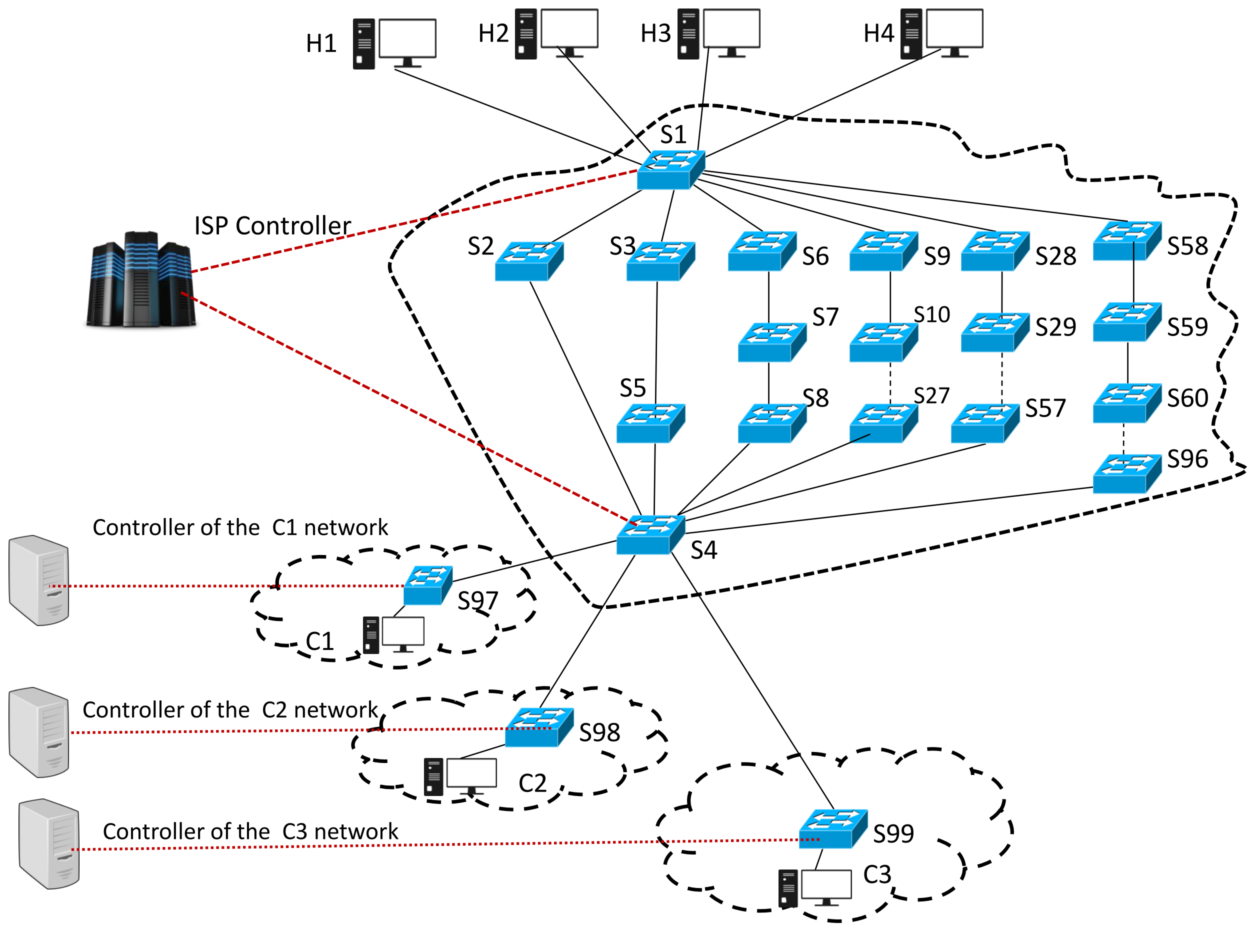

H1
H2
H3
H4
S1
ISP Controller
S2
S3
S6
S9
S28
S58
S7
S10
S29
S59
S5
S8
S27
S57
S60
S96
S4
Controller of the C1 network
S97
C1
Controller of the C2 network
S98
C2
Controller of the C3 network
S99
C3

Figure 5.10: Experimental scenario: One ISP with three customers

- The assessment attributes describe the effect of the attack on the network. For instance, a low impact severity, as shown in the alert, represents the fact that the congestion level in the customer network is 70%. Moreover, the flow class indicates that the flow of concern is considered suspicious.

DDoS attack mitigation policy at the ISP controller: This is the policy to process the received security alerts. The example policy shown in Listing 5.2 provides a high level action, here `redirect`, that processes ICMP flood traffic with a low impact severity.

Table 5.1: Traffic generation: source and destination

Hosts	Destination network
H_1	C_1
H_2	C_1
H_3	C_2
H_4	C_3

Table 5.2: Path descriptions

Path	Bandwidth	Destination Network
P_4	Low	C_1, C_2, C_3
P_5	Medium	C_1, C_2, C_3
P_6	High	C_1, C_2, C_3

Listing 5.1: Security alert sent by customer C1

```
<IDMEF-Message version="1.0">
<Alert>
 <Analyzer analyzerid="CUSTOMER_C1"/>
 <Source>
   <Address category="ipv4-addr">
          <address>10.0.0.2</address>
   </Address>
 </Source>
 <Target>
   <Address category="ipv4-addr">
         <address>10.0.0.3</address>
   </Address>
 </Target>
 <Classification event="ICMP_Flood">
 </Classification>
 <Assessment>
        <Impact severity="Low"/>
  </Assessment>
 <AdditionalData>type="string"meaning="flow_class">
 <string>Suspicious</string>
 </AdditionalData>
</Alert>
</IDMEF-Message>
```

Listing 5.2: Policy to redirect suspicious traffic in the ISP network

```
<Policy PolicyID="Mitigation">
          <Event Type = "ICMP_Flood">
     </Event>
          <Condition>
        <flow class="suspicious"/>
        <Impact severity="Low"/>
     </Condition>
     <Actions action="Redirect"/>
     </Actions>
</Policy>
```

Instantiation of the template to redirect the suspicious traffic: This information describes how to redirect the suspicious traffic through the low bandwidth path or through the middleboxes. Table 5.2 provides the path to route the flow depending on the impact severity mentioned in the alert sent by the customer. Furthermore, Listing 5.3 provides the concrete path details including the switch IDs and output ports to steer the flow.

Listing 5.3: Paths of low bandwidth

```
P4:{Switch:S1,output(8);Switch:S9,output(2),Switch:S10,output(2);...;
     Switch:S27;Switch:S4,output(4)}
P5:{Switch:S1,output(9);Switch:S28,output(2);Switch:S29,output(2);...;
     Switch:S57,output(2);Switch:S4,output(4)}
P6:{Switch:S1,output(10);  Switch:S58,output(2);  Switch:S59,output(2);
     Switch:S60,output(2);...;  Switch:S96,output(2);Switch:S4,output(2)}
Path={5:P5,  6:P6,  7:P7}}
```

5.4 Experimental Results

This section describes the results of our experimentations to assess both the performance of the high-level policy translation mechanism, and the effectiveness of the mitigation policies in reducing collateral damage. The scenarios are created using the Mininet emulator [2]. Three scenarios with different numbers of switches are specified for the mitigation policies as shown in Fig. 5.10. As explained earlier, suspicious flows have been classified in three classes according to their impact severity (low, medium and high). So, we provision three different bandwidths for the mitigation policies. The methodology to classify these flows in three different classes is outside the scope of this proposed framework. We assume that the customer uses some detection mechanism to classify the flows in different classes [37]. Moreover, three different scenarios with a varying number of switches are also specified for the QoS policies as shown in Table 5.3.

5.4.1 Evaluation metrics

The objective of our experiments is to evaluate the time to implement the high-level policies into low-level OpenFlow rules in the switches. Moreover, we also aim to evaluate how our framework can successfully reduce the collateral damage on a customer network caused by the attack traffic targeting another customer network. The metrics used to evaluate our prototype are specified in Table 5.4. They include the implementation time of the policy, the packet loss, the throughput and the network jitter.

Table 5.3: QoS Policies in different scenarios with varying number of switches in the path

QoS policies	A	B	C
Gold:No.of switches	3	5	7
Silver:No.of switches	4	8	15
Bronze:No.of switches	5	10	20

Table 5.4: Defined metrics to evaluate the prototype

Metric	Definition	Unit
Implementation time	It measures the time taken to translate the high-level policy and deploy them as OpenFlow rules in the switch. It increases with the increasing number of data plane devices for the deployment of the rules.	seconds
Packet loss	It evaluates the number of packets lost due to congestion or attack. Packet loss increases as long as the network is congested. Packet loss also occurs during the deployment of the rules in the switches because of some delay in the deployment.	number of packets
Throughput	Volume of traffic received in average unit of time. It decreases when the congestion increases in the network.	Mbps
Network jitter	Measures the variation in arrival time between packets and further provides the understanding of congestion or attack traffic on the system. It increases dramatically in the presence of congestion.	milliseconds

5.4.2 Implementation time of mitigation policy

From Fig. 5.11, we can see that the implementation time to deploy the policy to handle suspicious flows remains under 75 milliseconds, even for flows with a high impact severity. It is significantly higher than the implementation time of mitigation policies for flows with a low and medium impact severity, as well as the one to implement the policy to block the malicious flows. Since the path with the highest number of hops is provisioned for processing the high impact severity flows. If we consider the number of switches in the path for the deployment of the rules, it is still reasonable in between 75 to 80 millisecond. The implementation time to deploy the policy to handle low and medium impact severity flows are around 37 and 55 millisecond respectively.
Interestingly, the deployment time to block malicious flows is significantly lower as compared to the deployment time of other policies. It is because of the fact that the block action is only enforced at the ingress switch of the ISP and no further deployment is required.
We conclude that the deployment time of the policies is reasonable, but it can be further reduced if the number of devices for policy deployment can be reduced. This can be done with pre-installed rules in some devices of the network. If the policies are pre-deployed in the core switches, then only at the border switches policies need to be deployed dynamically reducing the overall implementation time.
We also conducted the experimentation to evaluate the deployment time of the policy to handle suspicious flows with varying traffic rates. We wanted to evaluate whether the traffic rate affects the

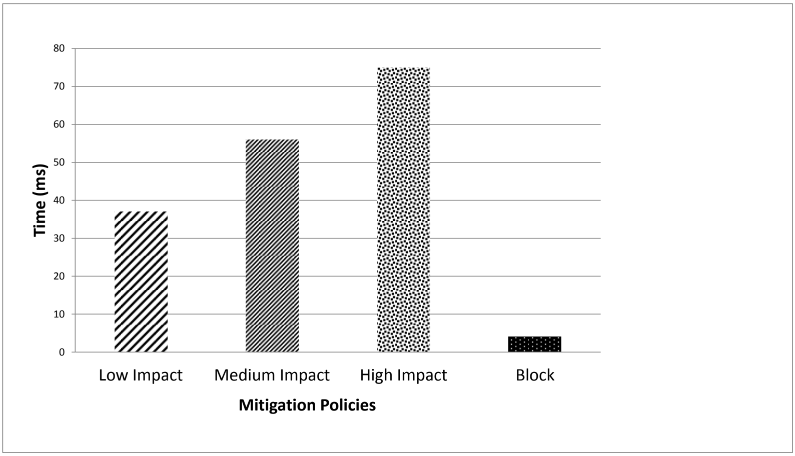


Figure 5.11: Implementation time of mitigation policies

implementation time of the policy. Fig. 5.12 shows that the deployment time of the low-level rules corresponding to the high-level policy to process the suspicious flows of high impact severity is close to constant. Experimentation was run with varying traffic rates from 1 Mbps to 15 Mbps. In all cases, it is around 75 milliseconds. It shows that the traffic rate does not affect our policy translation process and the deployment of corresponding low-level rules in the network.

5.4.3 Malicious traffic filtering

Filtering malicious flows at the border router of the ISP network is better for their customers, as it reduces the impact on the incoming legitimate traffic to the customer flows traversing through the ISP network. Therefore, we measured the number of packets that bypassed the ingress switch while the alert to drop the malicious flows was being processed. This was evalutated against varying traffic rates. As can be seen in Fig. 5.13, when the traffic rate is 1 Mbps then around 18 packets crossed the ingress switch and reached the customer network. It is worth noting that the number of packets that crossed the ingress switch and reached the customer network increases as the traffic rate increases. As shown in Fig. 5.13, when the traffic rate is at 5 Mbps then close to 125 packets are able to reach the customer network which was under attack. There is a sharp increase in the number of packets that reached the customer network when the traffic rate increases from 10 to 15 Mbps. It occurs because of the minimum delay in processing the security alert and the deployment low-level rules to drop the malicious flows.

5.4.4 Implementation time of QoS policies

Some experimentations were run to evaluate the implementation time of the different QoS policies. In a similar way, the high-level QoS policies were translated into low-level OpenFlow rules depending on the request from the customers of the ISP. It is worth noting that the implementation time of the gold QoS policy is the lowest in comparison to the silver and bronze QoS policies. The implementation time for the gold QoS policy is high in scenario C as it has more switches than scenarios A and B. The implementation time reported in Fig. 5.14 for the silver QoS policy also follows the same increasing trend with according to the number of switches in the topology. As can be seen in Fig. 5.14, the

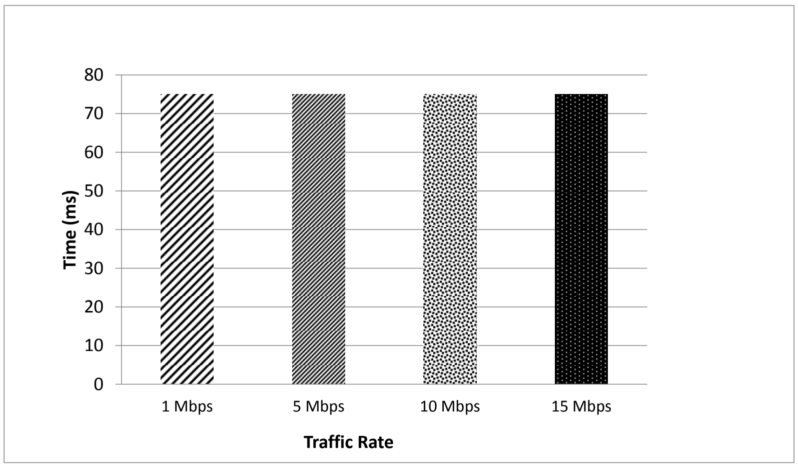


Figure 5.12: With varying traffic rate, time required to deploy the rules to process the flows

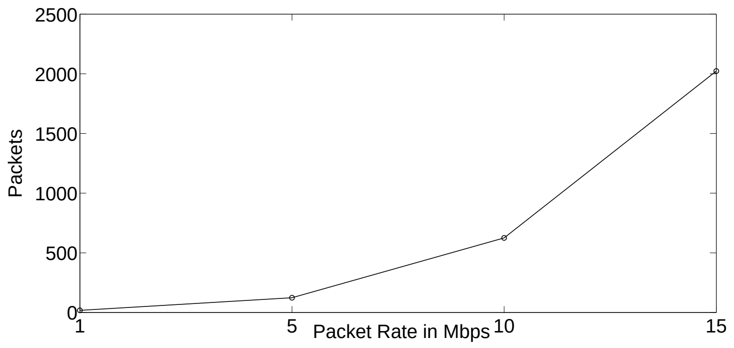


Figure 5.13: Number of Packets that bypass during the implementation of block action at the ingress switch

deployment time in scenario C is around 28 milliseconds, which is reasonable considering the number of switches in the path (15) for scenario C as specified in Table 5.3.
Similarly, the deployment time of the bronze QoS policy increases with the number of switches in the path. It takes around 36 milliseconds to deploy the low-level rules on the bronze path in scenario C (20 switches as indicated in Table 5.3). Considering the number of switches in the path, the deployment time of policy in the bronze path is still in the order of a few milliseconds.

5.4.5 Packet loss

We also measured the packet loss during the deployment of low-level rules according to our mechanism. As shown in Fig. 5.15, the packet loss also follows an increasing trend. But, with our mechanism, we are able to minimize this loss to a large extent. It can be seen, in Fig. 5.15, that there is no packet loss when the traffic rate is around 1 Mbps. Even at 15 Mbps of traffic rate, the packet loss is around 7.2 percents as compared to the 37 percents, without our mechanism. It shows that our mechanism greatly reduces the packet loss percentage while deploying the low-level rules in the OpenFlow switches.
Our strategy to reduce packet loss relies on deploying the low-level rules in the core switches first,

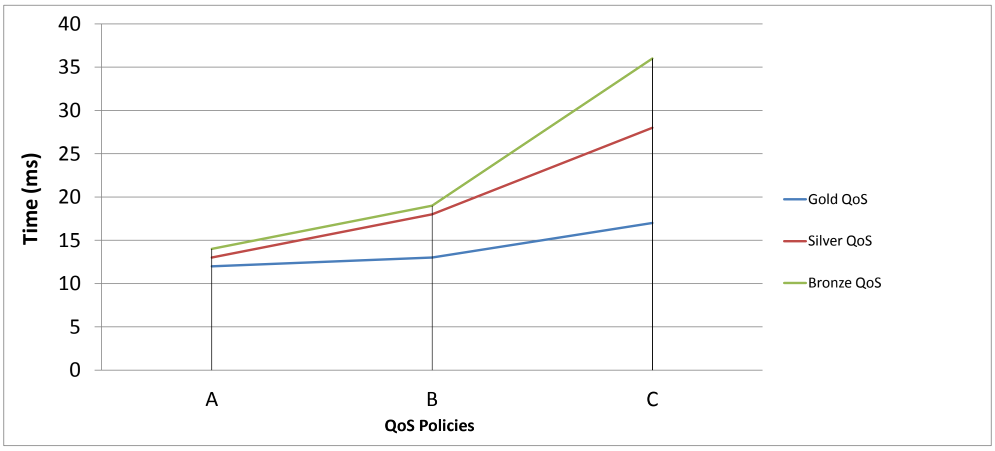


Figure 5.14: Implementation time of QoS policies with different scenarios

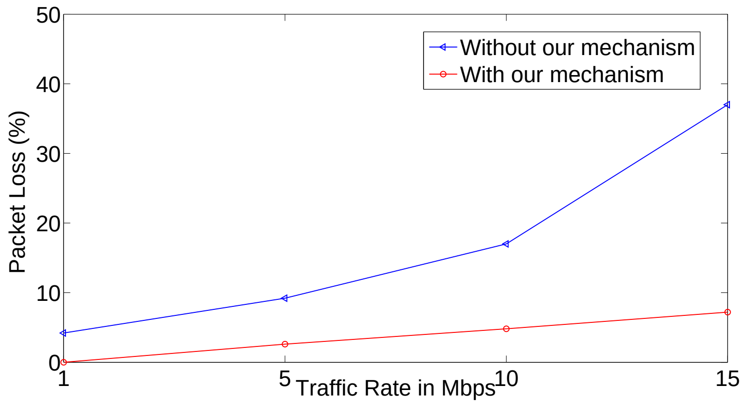


Figure 5.15: Packet loss during the deployment of the policy

before modifying the rules in the ingress and egress switches. While the mitigation rules are being deployed, the flows of concern still traverse the previous path, so as to reduce the packet loss. The packet loss still occurs because of the delay in deploying the low-level rules from the controller to the border switches.

5.4.6 Throughput of legitimate traffic

The throughput was measured in the presence of DDoS attacks. As shown in Figure 5.10, we used H_2 to generate DDoS attack traffic, and observed the impact on the legitimate traffic going to the customers C_1, C_2 and C_3. As we can see in Figure 5.16, the throughput of all the legitimate traffic

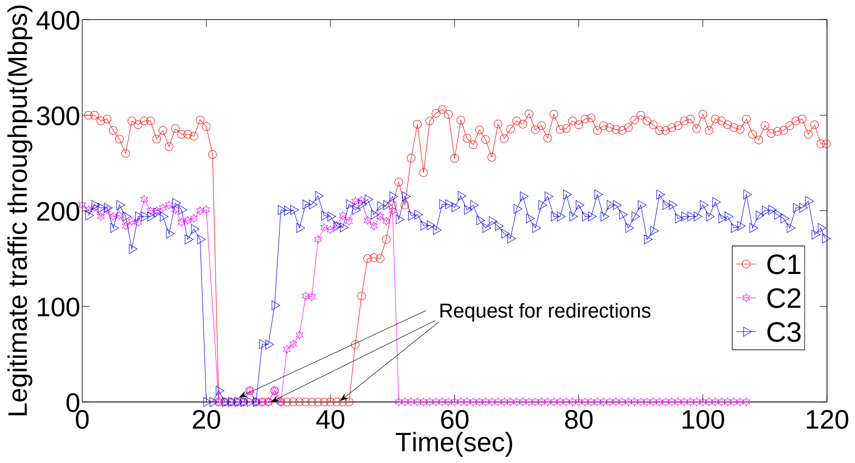


Figure 5.16: Throughput of legitimate traffic going towards customer network after redirection.

dropped sharply as soon as H_2 started to attack. As a result, the SDN controller of customer C_3 sends an alert, which contains the FlowID (source IP, destination IP) and the security class (legitimate), to the *MC* at the ISP controller, prompting the *PDP* to instruct the controller on redirecting the flow. Subsequently, the *NIB* computes the best path, i.e., P_3, and forwards the NSH with the path details to the PDP. Finally, the corresponding OpenFlow rules are loaded to the *PEPs*, namely the OpenFlow switches. As shown in Figure 5.16, the legitimate traffic heading to C_3 was thus able to quickly return to its normal level.
Similarly, the traffic flow going to C_2 was redirected through path P_2 upon the request of customer C_2, in order to restore its throughput to the normal level. Afterwards, the alert of customer C_1 reached the ISP controller, which interestingly redirected the traffic originating from host H_1 (which has the higher throughput) to path P_2 as well, degrading the throughput of the traffic (originating from host H_3) to customer C_2 down to zero, as shown in Figure 5.16. This indicates that, due to the limited availability of high QoS paths in the ISP network, ensuring the QoS for one customer may incur negative impact on other customers.

5.4.7 QoS provisioning of legitimate traffic

Following the previous experiment, we examine how the QoS of legitimate traffic can be provisioned if all the paths with high bandwidth are congested. In this experiment, we assume that customer C_1 requests a better QoS for the traffic sent from H_1. As shown in Figure 5.17, since the legitimate traffic going towards customers C_2 and C_3 was protected from collateral damage, the traffic from H_1 was redirected to the lower bandwidth path P_4, ensuring that the QoS was not heavily impacted despite the congestion of the legitimate path P_1.

5.4.8 Network jitter of legitimate traffic

Finally, we tested how the network jitter of legitimate traffic varies in the presence of congestion. As Fig. 5.18 shows, the network jitter of legitimate traffic going towards customers C_1, C_2, and C_3 started to increase when the attack traffic from H_2 congested the network. However, all of them immediately decreased when the ISP controller redirected the traffic flows upon receiving the mitigation requests from the customers. Despite the similar changing pattern, the network jitter of the traffic going to C_3 decreased earlier in comparison to those of C_2 and C_1. This is simply because customer C_3 sent an alert earlier than customers C_2 and C_1 did.

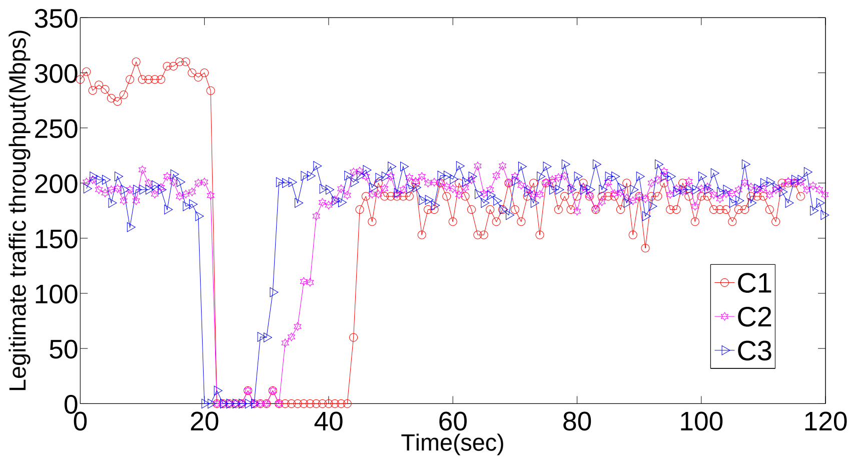


Figure 5.17: Throughput of legitimate traffic in the case the traffic going towards C_1 is redirected through the low suspicious path.

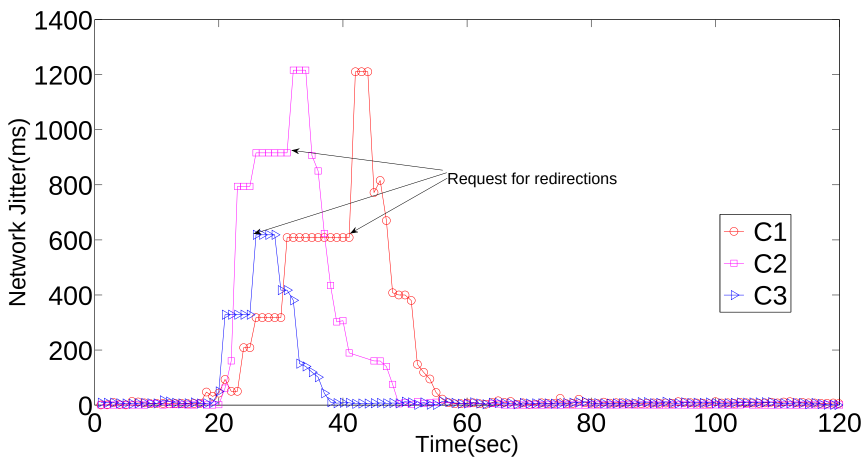


Figure 5.18: Network jitter of legitimate traffic.

Chapter 6 Conclusions

In this deliverable we have presented the architecture of the SUPERCLOUD network virtualization platform. We started with an overview of the architecture, followed by a description of our solution to the core component of the network hypervisor: the secure virtual network embedding. Third, we presented the design, implementation, and evaluation of the techniques introduced to improve the security and dependability of the infrastructure. Finally, we detailed the autonomic security management services for the virtualization platform.
The integration of the network hypervisor with components from other work packages has been positively tested, namely with:

- the authentication service (Work Package 2);
- the storage service Janus (Work Package 3);
- the Maxdata use case (Work Package 5); and
- the Phillips use case (Work Package 5).

Our multi-cloud network virtualization platform thus fulfilled (and extended) all objectives defined for Work Package 4.
As anticipation of future work, we expect the network hypervisor to be enhanced along three axis:

- First, to provide elasticity to virtual networks, by allowing tenants to scale up and down their virtual infrastructures.
- Second, to improve the efficiency of the substrate infrastructure by integrating mechanisms to migrate network and compute resources transparently.
- Third, to enhance virtual networks with advanced security services and other network functions, by leveraging the recent advances on programmable data planes [33].

Bibliography

[1] ViNE-Yard. `http://www.mosharaf.com/ViNE-Yard.tar.gz`.

[2] Mininet: An instant virtual network on your laptop, 2016.

[3] VIS.JS. `http://visjs.org/`, 2017. Accessed: 2017-10-25.

[4] O. I. Abdullaziz, Y. J. Chen, and L. C. Wang. Lightweight authentication mechanism for software defined network using information hiding. In *2016 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6, Dec 2016.

[5] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, and Paul Zimmermann. Imperfect forward secrecy: How diffie-hellman fails in practice. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 5–17, New York, NY, USA, 2015. ACM.

[6] Adnan Akhunzada, Ejaz Ahmed, Abdullah Gani, Muhammad Khurram Khan, Muhammad Imran, and Sghaier Guizani. Securing software defined networks: taxonomy, requirements, and open issues. *IEEE Communications Magazine*, 53(4):36–44, 2015.

[7] Ali Al-Shabibi, Marc De Leenheer, Matteo Gerola, Ayaka Koshibe, Guru Parulkar, Elio Salvadori, and Bill Snow. Openvirtex: Make your virtual sdns programmable. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 25–30, New York, NY, USA, 2014. ACM.

[8] Ali Al-Shabibi et al. OpenVirteX: Make your virtual SDNs programmable. In *HotSDN*, 2014.

[9] Max Alaluna et al. Secure and Dependable Multi-Cloud Network Virtualization. In *XDOM0*, 2017.

[10] Martin R Albrecht, Davide Papini, Kenneth G Paterson, and Ricardo Villanueva-Polanco. Factoring 512-bit RSA moduli for fun (and a profit of $9,000), 2015.

[11] J. Bacelar Almeida, Manuel Barbosa, Jorge S. Pinto, and Barbara Vieira. Formal verification of side-channel countermeasures using self-composition. *Science of Computer Programming*, 78(7):796 – 812, 2013. Special section on Formal Methods for Industrial Critical Systems (FMICS 2009 + FMICS 2010) & Special section on Object-Oriented Programming and Systems (OOPS 2009), a special track at the 24th ACM Symposium on Applied Computing.

[12] R. Alvizu, G. Maier, N. Kukreja, A. Pattavina, R. Morro, A. Capello, and C. Cavazzoni. Comprehensive survey on T-SDN: Software-defined networking for transport networks. *IEEE Communications Surveys Tutorials*, PP(99):1–1, 2017.

[13] Markku Antikainen, Tuomas Aura, and Mikko Srel. Spook in your network: Attacking an SDN with a compromised openflow switch. In Karin Bernsmed and Simone Fischer-Hbner, editors, *Secure IT Systems*, Lecture Notes in Computer Science, pages 229–244. Springer International Publishing, 2014.

[14] Cyril Arnaud and Pierre-Alain Fouque. Timing attack against protected rsa-crt implementation used in polarssl. In Ed Dawson, editor, *Topics in Cryptology - CT-RSA 2013*, volume 7779 of *Lecture Notes in Computer Science*, pages 18–33. Springer Berlin Heidelberg, 2013.

[15] Ahmad Aseeri, Nuttapong Netjinda, and Rattikorn Hewett. Alleviating eavesdropping attacks in software-defined networking data plane. In *Proceedings of the 12th Annual Conference on Cyber and Information Security Research*, CISRC '17, pages 1:1–1:8, New York, NY, USA, 2017. ACM.

[16] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards predictable datacenter networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 242–253, New York, NY, USA, 2011. ACM.

[17] Lawrence E. Bassham, III, Andrew L. Rukhin, Juan Soto, James R. Nechvatal, Miles E. Smid, Elaine B. Barker, Stefan D. Leigh, Mark Levenson, Mark Vangel, David L. Banks, Nathanael Alan Heckert, James F. Dray, and San Vo. Sp 800-22 rev. 1a. a statistical test suite for random and pseudorandom number generators for cryptographic applications. Technical report, Gaithersburg, MD, United States, 2010.

[18] M. Ben-Yehuda et al. The turtles project: Design and implementation of nested virtualization. In *9th USENIX - OSDI'10*, 2010.

[19] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pages 267–280, New York, NY, USA, 2010. ACM.

[20] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Understanding data center traffic characteristics. *SIGCOMM Comput. Commun. Rev.*, 40(1):92–99, January 2010.

[21] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, et al. ONOS: towards an open, distributed sdn os. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 1–6. ACM, 2014.

[22] Daniel J. Bernstein. *Introduction to post-quantum cryptography*, pages 1–14. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[23] Daniel J Bernstein, Tanja Lange, and Ruben Niederhagen. Dual EC: a standardized back door. In *The New Codebreakers*, pages 256–281. Springer, 2016.

[24] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In *Progress in Cryptology - LATINCRYPT*, volume 7533 of *L. N. in CS.* Springer, 2012.

[25] Daniel J. Bernstein, Bernard van Gastel, Wesley Janssen, Tanja Lange, Peter Schwabe, and Sjaak Smetsers. TweetNaCl: A crypto library in 100 tweets. In Diego F. Aranha and Alfred Menezes, editors, *Progress in Cryptology - LATINCRYPT 2014*, volume 8895 of *Lecture Notes in Computer Science*, pages 64–83. Springer International Publishing, 2015.

[26] A. Bessani, J. Sousa, and E. E. P. Alchieri. State machine replication for the masses with BFT-SMART. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362, June 2014.

[27] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. Depsky: Dependable and secure storage in a cloud-of-clouds. *Trans. Storage*, 9(4):12:1–12:33, November 2013.

[28] Alysson Bessani, Ricardo Mendes, Tiago Oliveira, Nuno Neves, Miguel Correia, Marcelo Pasin, and Paulo Verissimo. Scfs: A shared cloud-backed file system. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 169–180, Berkeley, CA, USA, 2014. USENIX Association.

[29] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A messy state of the union: Taming the composite state machines of TLS. In *2015 IEEE Symposium on Security and Privacy*, pages 535–552. IEEE, 2015.

[30] Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, Rustan Leino, Jay Lorch, et al. Everest: Towards a verified, drop-in replacement of https. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 71. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[31] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. Implementing TLS with verified cryptographic security. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 445–459. IEEE, 2013.

[32] KEVIN BOCEK. Infographic: How an attack by a cyber-espionage operator bypassed security controls, Jan. 2015.

[33] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3), July 2014.

[34] F. Botelho, A. Bessani, F.M.V. Ramos, and P. Ferreira. On the design of practical fault-tolerant sdn controllers. In *Third European Workshop on Software Defined Networks*, EWSDN '14, 2014.

[35] F. Botelho, T. A. Ribeiro, A. Bessani, F. M. V. Ramos, and P. Ferreira. Design and implementation of a consistent datastore for a distributed sdn control plane. In *EDCC16: The 12th European Conference on Dependable Computing*, 2016.

[36] Fábio Botelho, Tulio A Ribeiro, Paulo Ferreira, Fernando MV Ramos, and Alysson Bessani. Design and implementation of a consistent data store for a distributed SDN control plane. In *Dependable Computing Conference (EDCC), 2016 12th European*, pages 169–180. IEEE, 2016.

[37] R. Braga, E. Mota, and A. Passito. Lightweight DDoS flooding attack detection using NOX/OpenFlow. In *35th IEEE Conference on Local Computer Networks (LCN)*, pages 408–415, Oct 2010.

[38] E. Brewer. Cap twelve years later: How the "rules" have changed. *Computer*, 45(2):23–29, 2012.

[39] BillyBob Brumley and Nicola Tuveri. Remote timing attacks are still practical. In Vijay Atluri and Claudia Diaz, editors, *Computer Security - ESORICS 2011*, volume 6879 of *Lecture Notes in Computer Science*, pages 355–371. Springer Berlin Heidelberg, 2011.

[40] D. Buhov, M. Huber, G. Merzdovnik, E. Weippl, and V. Dimitrova. Network security challenges in android applications. In *2015 10th International Conference on Availability, Reliability and Security*, pages 327–332, Aug 2015.

[41] C. Cachin and A. Samar. Secure distributed dns. In *International Conference on Dependable Systems and Networks, 2004*, pages 423–432, June 2004.

[42] Marco Canini, Petr Kuznetsov, Dan Levin, and Stefan Schmid. Software transactional networking: Concurrent and consistent policy composition. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, 2013.

[43] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '07, 2007.

[44] M. Chowdhury et al. PolyViNE: Policy-based Virtual Network Embedding Across Multiple Domains. In *ACM SIGCOMM VISA*, 2010.

[45] N. M. M. K. Chowdhury et al. Virtual network embedding with coordinated node and link mapping. In *INFOCOM 2009, IEEE*, pages 783–791, April 2009.

[46] Cisco. Annual security report, 2014.

[47] P. A. R. S. Costa, F. M. V. Ramos, and M. Correia. Chrysaor: Fine-grained, fault-tolerant cloud-of-clouds mapreduce. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 421–430, May 2017.

[48] Bob Cromwell. Massive failures of internet PKI, 2017. `http://cromwell-intl.com/cybersecurity/pki-failures.html`.

[49] M. C. Dacier, H. Konig, R. Cwalinski, F. Kargl, and S. Dietrich. Security challenges and opportunities of software-defined networking. *IEEE Security Privacy*, 15(2):96–100, March 2017.

[50] Quynh Dang. Recommendation for existing application-specific key derivation functions. *NIST Special Publication*, 800:135, Dec 2010.

[51] DigiCert Inc. Enabling perfect forward secrecy, 2017. `https://www.digicert.com/ssl-support/ssl-enabling-perfect-forward-secrecy.htm`.

[52] Yevgeniy Dodis, David Pointcheval, Sylvain Ruhault, Damien Vergniaud, and Daniel Wichs. Security analysis of pseudo-random number generators with input: /dev/random is not robust. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS '13, pages 647–658, New York, NY, USA, 2013. ACM.

[53] Benjamin Dowling, Douglas Stebila, and Greg Zaverucha. Authenticated network time synchronization. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 823–840, Austin, TX, 2016. USENIX Association.

[54] Chris Edwards. Researchers probe security through obscurity. *Communications of the ACM*, 57(8):11–13, 2014.

[55] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in Android applications. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 73–84, New York, NY, USA, 2013. ACM.

[56] Shuqin Fan, Wenbo Wang, and Qingfeng Cheng. Attacking openssl implementation of ecdsa with a few signatures. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1505–1515. ACM, 2016.

[57] Benjamin Feinstein, David Curry, and Herve Debar. The Intrusion Detection Message Exchange Format (IDMEF). RFC 4765, October 2015.

[58] Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. *Cryptography engineering: design principles and practical applications*. John Wiley & Sons, 2011.

[59] Andreas Fischer et al. Position paper: Secure virtual network embedding. *Praxis der Informationsverarbeitung und Kommunikation*, 2011.

[60] Andreas Fischer et al. Virtual Network Embedding: A Survey. *IEEE Communications Surveys Tutorials*, 15(4):1888–1906, 2013.

[61] A. Froehlich. 9 spectacular cloud computing fails. Information Week, July 2015.

[62] C. Fuerst, S. Schmid, L. Suresh, and P. Costa. Kraken: Online and elastic resource reservations for multi-tenant datacenters. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9, April 2016.

[63] GLPK. GNU Linear Programming Kit. `http://www.gnu.org/software/glpk/`, 2008.

[64] Google. Protocol buffers, 2017.

[65] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: A scalable and flexible data center network. *SIGCOMM Comput. Commun. Rev.*, 39(4):51–62, August 2009.

[66] Marcella Hastings, Joshua Fried, and Nadia Heninger. Weak keys remain widespread in network devices. In *Proceedings of the 2016 ACM on Internet Measurement Conference*, pages 49–63. ACM, 2016.

[67] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 35–35, Berkeley, CA, USA, 2012. USENIX Association.

[68] Shai Herzog. The COPS (Common Open Policy Service) Protocol. RFC 2748, January 2000.

[69] Brad Hill. Failures of trust in the online PKI marketplace cannot be fixed by "raising the bar" on certificate authority security, 2013.

[70] Jaap-Henk Hoepman and Bart Jacobs. Increased security through open source. *Commun. ACM*, 50(1):79–83, January 2007.

[71] L. S. Huang, S. Adhikarla, D. Boneh, and C. Jackson. An experimental study of TLS forward secrecy deployments. *IEEE Internet Computing*, 18(6):43–51, Nov 2014.

[72] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*, volume 8, page 9, 2010.

[73] IEEE Spectrum. Special report: 50 years of Moore's law, 2015.

[74] Sushant Jain et al. B4: Experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, 2013.

[75] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, SIGCOMM '13, pages 3–14, New York, NY, USA, 2013. ACM.

[76] N. Katta, H. Zhang, M. Freedman, and J. Rexford. Ravana: Controller fault-tolerance in software-defined networking. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, 2015.

[77] Z. K. Khattak, M. Awais, and A. Iqbal. Performance evaluation of OpenDaylight SDN controller. In *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, pages 671–676, Dec 2014.

[78] Soo Hyeon Kim, Daewan Han, and Dong Hoon Lee. Predictability of Android OpenSSL's pseudo random number generator. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 659–668, New York, NY, USA, 2013. ACM.

[79] Timo Kiravuo, Mikko Sarela, and Jukka Manner. A survey of ethernet lan security. *IEEE Communications Surveys & Tutorials*, 15(3):1477–1491, 2013.

[80] Rowan Klöti, Vasileios Kotronis, and Paul Smith. OpenFlow: A security analysis. In *IEEE Proc. Wkshp on Secure Network Protocols (NPSec)*, 2013.

[81] Teemu Koponen et al. Network virtualization in multi-tenant datacenters. In *USENIX NSDI*, 2014.

[82] Teemu Koponen and other. Onix: A distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, 2010.

[83] D. Kreutz, A. Bessani, E. Feitosa, and H. Cunha. Towards secure and dependable authentication and authorization infrastructures. In *2014 IEEE 20th Pacific Rim International Symposium on Dependable Computing*, pages 43–52, Nov 2014.

[84] D. Kreutz, P. Esteves-Verissimo, C. Magalhaes, and F. M. V. Ramos. The KISS principle in Software-Defined Networking: An architecture for Keeping It Simple and Secure. *ArXiv e-prints*, June 2017.

[85] D. Kreutz, F.M.V. Ramos, P. Esteves Verissimo, C. Esteve Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1), Jan 2015.

[86] D. Kreutz, J. Yu, P. Esteves-Verissimo, C. Magalhaes, and F. M. V. Ramos. The KISS principle in software-defined networking: a framework for secure communications. *IEEE Security & Privacy*, 2017. Accepted for publication.

[87] D. Kreutz, J. Yu, F. M. V. Ramos, and P. Esteves-Verissimo. ANCHOR: logically-centralized security for Software-Defined Networks. *ArXiv e-prints*, November 2017.

[88] Diego Kreutz et al. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, Jan 2015.

[89] Diego Kreutz, Oleksandr Malichevskyy, Eduardo Feitosa, Hugo Cunha, Rodrigo da Rosa Righi, and Douglas D.J. de Macedo. A cyber-resilient architecture for critical security services. *Journal of Network and Computer Applications*, 63:173 – 189, 2016.

[90] Diego Kreutz, Fernando M.V. Ramos, and Paulo Verissimo. Towards secure and dependable software-defined networks. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 55–60, New York, NY, USA, 2013. ACM.

[91] M. Lacoste et al. User-Centric Security and Dependability in the Clouds-of-Clouds. *IEEE Cloud Computing*, 3(5):64–75, 9 2016.

[92] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2), May 1998.

[93] Dan Levin, Andreas Wundsam, Brandon Heller, Nikhil Handigol, and Anja Feldmann. Logically centralized?: state distribution trade-offs in software defined networks. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 1–6. ACM, 2012.

[94] Shuhao Liu et al. Security-Aware Virtual Network Embedding. In *IEEE ICC*, June 2014.

[95] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *Commun. ACM*, 58(2):44–46, January 2015.

[96] Chowdhury M. et al. ViNEYard: Virtual Network Embedding Algorithms with Coordinated Node and Link Mapping. *IEEE/ACM Transactions on Networking*, 20(1):206–219, February 2012.

[97] D. Mahu, V. Dumitrel, and F. Pop. Secure entropy gatherer. In *2015 20th International Conference on Control Systems and Computer Science*, pages 185–190, May 2015.

[98] Aanchal Malhotra, Isaac E Cohen, Erik Brakke, and Sharon Goldberg. Attacking the network time protocol. *IACR Cryptology ePrint Archive*, 2015:1020, 2015.

[99] Konstantinos Manousakis and Georgios Ellinas. Attack-aware planning of transparent optical networks. *Optical Switching and Networking*, (0):–, 2015.

[100] G. Markowsky. Was the 2006 Debian SSL debacle a system accident? In *2013 IEEE 7th International Conference on Intelligent Data Acquisition and Advanced Computing Systems (IDAACS)*, volume 02, pages 624–629, Sept 2013.

[101] G. McGraw. Software security. *IEEE Security Privacy*, 2(2):80–83, Mar 2004.

[102] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[103] MEF. MEF, 2017.

[104] Michael Mimoso. GPG patches 18-year-old libgcrypt RNG bug, 2016.

[105] M. Naldi. Connectivity of Waxman Topology Models. *Computer Communications*, 29(1):24–31, December 2005.

[106] Namecheap.com. Cipher suites configuration (and forcing perfect forward secrecy), 2015. `https://www.namecheap.com/support/knowledgebase/article.aspx/9601//cipher-suites-configuration-and-forcing-perfect-forward-secrecy`.

[107] David Naylor, Alessandro Finamore, Ilias Leontiadis, Yan Grunenberger, Marco Mellia, Maurizio Munafo, Konstantina Papagiannaki, , and Peter Steenkiste. The cost of the "S" in HTTPS. In *Proceedings of the Tenth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, page 7, New York, NY, USA, 2014. ACM.

[108] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12), December 1978.

[109] NIST. NIST statistical test suite, 2017.

[110] Brian M Oki and Barbara H Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 8–17. ACM, 1988.

[111] ONF. Openflow switch specification (version 1.5.0), Dec. 2014.

[112] ONF. Principles and practices for securing software-defined networks. Technical report, Open Networking Foundation, Feb. 2015. ONF TR-511.

[113] ONF. Open Networking Foundation, 2017.

[114] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, 2014.

[115] OpenSSL.org. OpenSSL security advisory [10 nov 2016], November 2016.

[116] Dave Otway and Owen Rees. Efficient and timely mutual authentication. *SIGOPS Oper. Syst. Rev.*, 21(1), January 1987.

[117] Peter Perešíni, Maciej Kuzniar, Nedeljko Vasić, Marco Canini, and Dejan Kostiū. Of. cpp: Consistent packet processing for openflow. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 97–102. ACM, 2013.

[118] Adrian Perrig, Robert Szewczyk, J. D. Tygar, Victor Wen, and David E. Culler. Spins: Security protocols for sensor networks. *Wirel. Netw.*, 8(5):521–534, September 2002.

[119] W. Michael Petullo, Xu Zhang, Jon A. Solworth, Daniel J. Bernstein, and Tanja Lange. MinimaLT: Minimal-latency networking through better security. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 425–438, New York, NY, USA, 2013. ACM.

[120] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J. Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Jonathan Stringer, Pravin Shelar, Keith Amidon, and Martín Casado. The design and implementation of open vswitch. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 117–130, Berkeley, CA, USA, 2015. USENIX Association.

[121] Google Cloud Platform. Google compute engine incident 16007. `https://status.cloud.google.com/incident/compute/16007`, April 2016.

[122] Philip Porras, Seungwon Shin, Vinod Yegneswaran, Martin Fong, Mabry Tyson, and Guofei Gu. A security enforcement kernel for OpenFlow networks. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 121–126, New York, NY, USA, 2012. ACM.

[123] PwC, CSO magazine and CERT/CMU. US cybercrime: Rising risks, reduced readiness. Technical report, PwC, 2014.

[124] M. Rahman et al. Survivable Virtual Network Embedding. In *NETWORKING*, pages 40–52, May 2010.

[125] Abbas Razaghpanah, Arian Akhavan Niaki, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Johanna Amann, and Phillipa Gill. Studying TLS usage in android apps. In *Proceedings of the 13th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '17, page 7, New York, NY, USA, 2017. ACM.

[126] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 323–334. ACM, 2012.

[127] RightScale. 2017 state of the cloud report, February 2017.

[128] Francisco Javier Ros and Pedro Miguel Ruiz. Five nines of southbound reliability in software-defined networks. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 31–36. ACM, 2014.

[129] Matthias Rost, Carlo Fuerst, and Stefan Schmid. Beyond the stars: Revisiting virtual cluster embeddings. *SIGCOMM Comput. Commun. Rev.*, 45(3):12–18, July 2015.

[130] Dominik Samociuk. Secure communication between OpenFlow switches and controllers. *AFIN 2015*, page 39, 2015.

[131] Bruce Schneier. Lousy random numbers cause insecure public keys, Feb 2012.

[132] Bruce Schneier. *Data and Goliath: The hidden battles to collect your data and control your world.* WW Norton & Company, 2015.

[133] J. Schonwalder and V. Marinov. On the impact of security protocols on the performance of SNMP. *IEEE Trans. on Net. and Service Management*, 8(1), 2011.

[134] S. Scott-Hayward, S. Natarajan, and S. Sezer. A survey of security in software defined networks. *IEEE Communications Surveys Tutorials*, 18(1):623–654, Firstquarter 2016.

[135] S. Scott-Hayward, G. O'Callaghan, and S. Sezer. SDN security: A survey. In *Future Networks and Services (SDN4FNS), 2013 IEEE SDN for*, pages 1–7, Nov 2013.

[136] N. Shahriar et al. Connectivity-Aware Virtual Network Embedding. In *IFIP Networking*, June 2016.

[137] Alexander Shalimov, Dmitry Zuikov, Daria Zimarina, Vasily Pashkov, and Ruslan Smeliansky. Advanced study of SDN/OpenFlow controllers. In *Proceedings of the 9th Central and Eastern European Software Engineering Conference in Russia*, CEE-SECR '13, pages 1:1–1:6, New York, NY, USA, 2013. ACM.

[138] Prateek Sharma, David Irwin, and Prashant Shenoy. How not to bid the cloud. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[139] Y. Sheffer, R. Holz, and P. Saint-Andre. Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). RFC 7525, May 2015.

[140] C. Shen, E. Nahum, H. Schulzrinne, and C. P. Wright. The impact of TLS on SIP server performance: Measurement and modeling. *IEEE/ACM Trans. on Networking*, 20(4), Aug 2012.

[141] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. Can the production network be the testbed? In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 365–378, Berkeley, CA, USA, 2010. USENIX Association.

[142] Seugwon Shin, Phillip Porras, Vinod Yegneswaran, Martin Fong, Guofei Gu, and Mabry Tyson. FRESCO: Modular composable security services for software-defined networks. In *Internet Society NDSS*, Feb. 2013.

[143] Seungwon Shin and Guofei Gu. Attacking software-defined networks: A first feasibility study. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 165–166, New York, NY, USA, 2013. ACM.

[144] Seungwon Shin, Yongjoo Song, Taekyung Lee, Sangho Lee, Jaewoong Chung, Phillip Porras, Vinod Yegneswaran, Jisung Noh, and Brent Byunghoon Kang. Rosemary: A robust, secure, and high-performance network operating system. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, Nov. 2014. *To appear.*

[145] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 183–197, New York, NY, USA, 2015. ACM.

[146] Drew Springall, Zakir Durumeric, and J. Alex Halderman. Measuring the security harm of tls crypto shortcuts. In *Proceedings of the 2016 Internet Measurement Conference*, IMC '16, pages 33–47, New York, NY, USA, 2016. ACM.

[147] Philip B. Stark. Don't bet on your random number generator, Mar 2017.

[148] Harlan Stenn. Securing network time protocol. *Commun. ACM*, 58(2):48–51, January 2015.

[149] Raphael E. Steuer. *Multiple Criteria Optimization: Theory, Computation and Application.* John Wiley, New York, 546 pp, 1986.

[150] USA Today. Massive Amazon cloud service outage disrupts sites. `https://www.usatoday.com/story/tech/news/2017/02/28/amazons-cloud-service-goes-down-sites-scramble/98530914/`, February 2017.

[151] J. Tsidulko. The 10 biggest cloud outages of 2016 (so far). The Channel Company, July 2016.

[152] N. L. M. van Adrichem, C. Doerr, and F. A. Kuipers. Opennetmon: Network monitoring in openflow software-defined networks. In *2014 IEEE Network Operations and Management Symposium (NOMS)*, pages 1–8, May 2014.

[153] Nicole van der Meulen. DigiNotar: Dissecting the first dutch digital disaster. *Journal of Strategic Security*, 6(2), 2013.

[154] Apostol Vassilev and Timothy A. Hall. The importance of entropy to information security. *Computer*, 47(2):78–81, 2014.

[155] Paulo Verissimo, Miguel Correia, Nuno Ferreira Neves, and Paulo Sousa. Intrusion-resilient middleware design and validation. In *Information Assurance, Security and Privacy Services*, volume 4 of *Handbooks in Information Systems*, pages 615–678. Emerald Group Publishing Limited, May 2009.

[156] Verizon. 2015 data breach investigations report. Technical report, Verizon, 2015.

[157] T. Wan, A. Abdou, and P. C. van Oorschot. A Framework and Comparative Analysis of Control Plane Security of SDN and Conventional Networks. *ArXiv e-prints*, March 2017.

[158] Huangxin Wang, Quan Jia, Dan Fleck, Walter Powell, Fei Li, and Angelos Stavrou. A moving target DDoS defense mechanism. *Computer Communications*, 46(0):10 – 21, 2014.

[159] Ahmad Samer Wazan, Romain Laborde, Franois Barrere, Abdelmalek Benzekri, and DavidW. Chadwick. PKI interoperability: Still an issue? a solution in the x.509 realm. In *Info. Assurance and Sec. Education and Training*, volume 406. Springer, 2013.

[160] Dan Williams, Hani Jamjoom, and Hakim Weatherspoon. The xen-blanket: Virtualize once, run everywhere. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 113–126, New York, NY, USA, 2012. ACM.

[161] Dan Williams and Ricardo Koller. Unikernel monitors: extending minimalism outside of the box. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. USENIX Association, 2016.

[162] Di Xie, Ning Ding, Y. Charlie Hu, and Ramana Kompella. The only constant is change: Incorporating time-varying network reservations in data centers. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 199–210, New York, NY, USA, 2012. ACM.

[163] Yuval Yarom and Naomi Benger. Recovering openssl ecdsa nonces using the flush+reload cache side-channel attack. *IACR Cryptology ePrint Archive*, 2014:140, 2014.

[164] H. Yu et al. Cost Efficient Design of Survivable Virtual Infrastructure to Recover from Facility Node Failures. In *IEEE ICC*, June 2011.

[165] Jiangshan Yu, Mark Ryan, and Cas Cremers. DECIM: Detecting endpoint compromise in messaging. Cryptology ePrint Archive, Report 2015/486, 2017. `http://eprint.iacr.org/2015/486`.

[166] Jiangshan Yu, Mark Ryan, and Cas Cremers. DECIM: Detecting endpoint compromise in messaging. *IEEE Trans. Information Forensics and Security*, 2017.

[167] Jiangshan Yu and Mark Dermot Ryan. Device attacker models: Fact and fiction. In *Security Protocols XXIII - 23rd International Workshop, Cambridge, UK, March 31 - April 2, 2015, Revised Selected Papers*, pages 158–167, 2015.

[168] L. Yu and Z. Cai. Dynamic scaling of virtual clusters with bandwidth guarantee in cloud datacenters. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9, April 2016.

[169] M. Yu et al. Rethinking Virtual Network Embedding: Substrate Support for Path Splitting and Migration. *ACM SIGCOMM Computer Communication Review*, 38(2):17–29, April 2008.

[170] Ellen W. Zegura et al. How to Model an Internetwork. In *IEEE INFOCOM*, pages 594–602, March 1996.

[171] Kim Zetter. Researchers solve juniper backdoor mystery; signs point to NSA, Dec 2015.

[172] Y. Zhao, L. Iannone, and M. Riguidel. On the performance of SDN controllers: A reality check. In *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*, pages 79–85, Nov 2015.

[173] L. Zheng et al. How to bid the cloud. In *ACM SIGCOMM*, 2015.

[174] Lidong Zhou, Fred B. Schneider, and Robbert Van Renesse. Coca: A secure distributed online certification authority. *ACM Trans. Comput. Syst.*, 20(4):329–368, November 2002.

[175] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *2012 IEEE Symposium on Security and Privacy*, pages 95–109, May 2012.