COoperative Cyber prOtectiON for modern power grids

D4.2 COCOON System Architecture

Distribution Level	PU
Responsible Partner	University of Glasgow (UGLA)
Prepared by	Filip Holik (UGLA), Tahira Mahboob (UGLA), Awais Aziz Shah (UGLA), Dimitrios Pezaros (UGLA)
Checked by WP Leader	Angelos Marnerides (UCY)
Verified by Reviewer #1	Angelos Marnerides (UCY) 10/09/2024
Verified by Reviewer #2	Elvira Sanchez Ortiz (ENCS) 10/09/2024
Approved by Project Coordinator	Angelos Marnerides (UCY) 16/09/2024

Co-funded by the European Union

Disclaimer

Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the Directorate General for Communications Networks, Content and Technology. Neither the European Union nor the Directorate General for Communications Networks, Content and Technology can be held responsible for them.

Deliverable Record

Planned Submission Date	17/09/2024
Actual Submission Date	16/09/2024
Status and version	FINAL

Version (Notes)	Date	Author(s)	Notes
0.1 (Draft)	13/08/2024	Filip Holik (UGLA), Tahira Mahboob (UGLA), Awais Aziz Shah (UGLA), Dimitrios Pezaros (UGLA)	ToC, initial structure, work allocation
0.2 (Draft)	21/08/2024	Tahira Mahboob (UGLA), Filip Holik (UGLA)	Main sections
0.3 (Draft)	30/08/2024	Filip Holik (UGLA), Tahira Mahboob (UGLA), Georgios Kryonidis (AUTH), Himanshu Goyel (TU Delft), Vetrivel S. Rajkumar (TU Delft), Alfan Presekal (TU Delft), Alex Stefanov (TU Delft), Viktor Piotr Papadopoulos (Selene CC)	First draft - main sections completed, partners inputs
0.4 (Draft)	04/09/2024	Filip Holik (UGLA), Tahira Mahboob (UGLA), Luna Moreno Diaz (ING), David Senas Sanvicente (ING), Esther Ayas Iglesias (CUE)	Draft edits, partners contributions
0.5 (Draft)	08/09/2024	Filip Holik (UGLA), Tahira Mahboob (UGLA)	Draft sent to the reviewers
1.0 (Final)	13/09/2024	Filip Holik (UGLA), Tahira Mahboob (UGLA)	Final version

Contents

Executive Summary **10**

1 Introduction **11**
1.1 Scope of the deliverable . . . 11
1.2 Relation with other work packages and tasks . . . 11
1.3 Methodology . . . 12

2 CPN system architecture **13**
2.1 Architecture overview . . . 14
2.2 Architecture requirements . . . 15
2.3 Services decomposition . . . 15
2.4 Programmable networking devices . . . 16
2.4.1 Traditional networking devices . . . 16
2.4.2 Programmable networking devices . . . 17
2.4.3 COCOON Programmable Node platforms . . . 18

3 COMML architecture **20**
3.1 COMML agent . . . 21
3.1.1 Agent functions . . . 21
3.1.2 Southbound API (SBI) . . . 22
3.1.3 eBPF implementation types . . . 23
3.1.4 eBPF instruction set . . . 25
3.1.5 eBPF verifier . . . 26
3.1.6 eBPF maps . . . 27
3.1.7 eBPF helper functions . . . 28
3.2 COMML switch . . . 28
3.2.1 Switch variables . . . 30
3.2.2 Switch structures . . . 31
3.2.3 Switch functions . . . 31

4 IOL architecture **32**
4.1 Service broker . . . 32
4.2 Event handler and listeners . . . 33
4.3 Libraries and tools . . . 34
4.3.1 The Twisted framework . . . 34
4.3.2 The Protocol Buffer . . . 36
4.3.3 Twisted and Protocols Buffer for CPN implentations . . . 37
4.3.4 LLVM . . . 38
4.3.5 Clang . . . 38
4.4 eBPF μNF . . . 39
4.4.1 eBPF μNF code structure . . . 40
4.4.2 eBPF μNF code snippets . . . 40
4.5 Northbound API (NBI) . . . 41

5 CPN pilot use cases **43**
5.1 Mininet emulation (SGSim) 43
5.2 Digital substation 44
5.3 Energy communities 45
5.4 PV power plants 46
5.5 Secure regional electricity data operations 47

6 Conclusions **49**

Bibliography **50**

List of Figures

1.1 The relationship of D4.2 with other tasks, deliverables and WPs 12

2.1 Overall CPN system architecture in a realistic topology scenario 13
2.2 Detailed CPN system architecture 14
2.3 Services decomposition in the CPN system architecture 16
2.4 Exemplar of an industrial unmanaged switch 17
2.5 Exemplar of managed switches 17
2.6 Exemplar of a Raspberry Pi SBC 18
2.7 Exemplar of a NUC 19
2.8 Exemplar of a server 19

3.1 The COMML architecture 20
3.2 SBI communications between CPN controller and CPN 23
3.3 eBPF implementation types 24
3.4 Program verification by eBPF verifier 26
3.5 Example of Memory Leak attempt rejected by the verifier 27
3.6 eBPF programs and applications can access the eBPF maps via system calls . . 28

4.1 The IOL architecture 32
4.2 Example of TCP handshake between CPN and the CPN controller during initial the connection establishment 33
4.3 Programming paradigms for networking applications 34
4.4 Data Serialization with Protocol Buffer 36
4.5 HLL (high level programming language) code to bytecode conversion paradigm adhered by the CPN implementation style on the IOL 38

5.1 Smart Grid Simulator (SGSim) platform for CPN development testing 43
5.2 Digital substation topology 44
5.3 Energy Community pilot configuration. 45
5.4 Electrical and communication layout of the pilot PV plant. 46
5.5 Secure regional electricity data operation pilot environment 48

List of Tables

3.1 SBI Message Definitions 22
3.2 Helper eBPF functions 29
3.3 Helper packet manipulation functions 29
3.4 Helper network control functions 30
3.5 Other function 31

Definition of Acronyms

μNF Micro Network Function. 10, 11, 14–16, 20–23, 27–29, 31–34, 36, 38–41, 49

AC Alternating Current. 47

AD Anomaly Detection. 15, 42, 43

API Application Programming Interface. 14, 15, 25, 42, 49

ARM Advanced RISC Machine. 18, 21

AUTH Aristotle University of Thessaloniki. 12

COCOON COoperative Cyber prOtectiOn for modern power grids. 10, 11, 13, 15, 16, 40, 42–44, 47, 49

COMML Control Measurement and Monitoring Layer. 10–16, 20–22, 28, 30, 31, 36–38, 41, 43, 49

CPN COCOON Programmable Node. 10–25, 28, 30–39, 42–44, 46–49

CPU Central Processing Unit. 18, 19, 21, 25

CSL Cybersecurity Services Layer. 10, 11, 13–16, 20, 21, 27, 32, 33, 39, 41, 42, 44, 48, 49

DPDK Data Plane Development Kit. 18–20, 24, 25, 49

DPID Datapath identitifier. 22, 30, 36, 37

DRL Deep Reinforcement Learning. 15

eBPF extended Berkeley Packet Filter. 10, 11, 15, 17, 18, 20–29, 32–34, 36, 38–41, 49

ELF Executable and Linkable Format. 38

EPES Electrical Power and Energy Systems. 10, 13, 45, 49

FDII False Data Injection Identification. 38, 46, 47

FIFO First In First Out. 25, 31

GOOSE Generic Object-Oriented Substation Event. 13, 39, 43, 44

GPL General Public License. 41

HEDNO Hellenic Electricity Distribution Network Operator. 12, 45, 46

HIL Hardware in the Loop. 44

HLL High level programming language. 38

HTML HyperText Markup Language. 42

HTTP Hypertext Transfer Protocol. 35

I/O Input/Output. 34

ICMP Internet Control Message Protocol. 40

IEC International Electrotechnical Commission. 13

IEC104 IEC 60870-5-104. 13, 43, 46

IED Intelligent Electronic Device. 39, 44

IKE I&K Electrical Engineering Systems. 45

ING Ingelectus. 12, 47

IOL Instrumentation and Orchestration Layer. 10–16, 20–22, 27, 30–32, 34, 36–38, 41, 42, 48, 49

IP Internet Protocol. 30, 40

ISO/OSI International Organization for Standardization / Open Systems Interconnection. 13

JiT Just in Time. 20, 21, 24

JSON JavaScript Object Notation. 36, 42

LLVM Low Level Virtual Machine. 38

LV Low-Voltage. 45, 47

MAC Medium Access Control. 40, 41

ML Machine Learning. 14

MV Medium-Voltage. 45–47

NBI Northbound Application Programming Interface. 10, 11, 14, 15, 32, 33, 41, 43, 49

NETCONF Network Configuration Protocol. 22

NIC Network Interface Cards. 18, 19, 23, 24

NOS Network Operating System. 19

NUC Next Unit of Computing. 10, 18, 46

OF OpenFlow. 22, 23, 28

ONF Open Networking Foundation. 22

ONL Open Network Linux. 19

OS Operating System. 24

OT Operational Technology. 24

P4 Programming Protocol-independent Packet Processors. 17

PC Personal Computer. 43

PDP Programmable Data Plane. 10, 15, 20, 22, 23, 28, 33, 34, 36–38, 49

PLC Programmable Logic Controller. 47

POI Point of Interconnection. 46, 47

PPC Power Plant Controller. 47

PV Photovoltaic. 12, 13, 45–47, 49

RAM Random-Access Memory. 18, 25

rBPF Rust userspace Berkeley Packet Filter. 24

REST Representational State Transfer. 42, 49

RSC Regional Security Coordinators. 47

RTDS Real-Time Digital Simulator. 44

SBC Single Board Computer. 10, 18, 44

SBI Southbound Application Programming Interface. 10, 15, 21–23, 30, 32, 33, 38, 42, 43

SCADA Supervisory Control and Data Acquisition system. 46

SDG SCADA Data Gateway. 45, 46

SDN Software Defined Networking. 17, 22

SELENE CC Southeast Electricity Network Coordination Centre. 12, 49

SGSim Smart Grid Simulator. 43, 44, 49

SS Secondary Substations. 47

SSL Secure Socket Layer. 34

SV Sampled Values. 13, 43, 44

TCP Transmission Control Protocol. 30, 33–35, 40

TCP/IP Transmission Control Protocol / Internet Protocol. 31, 47

TLS Transport Layer Security. 34

TRL Technology Rediness Levels. 10, 18, 25, 43, 49

TUD TU Delft. 12

uBPF userspace Berkeley Packet Filter. 24, 25

UDP User Datagram Protocol. 34

UGLA University of Glasgow. 19

VM Virtual Machine. 43

WP Work Package. 11, 12, 47

XDP Express Data Path. 20, 23, 25

Executive Summary

The COoperative Cyber prOtectiOn for modern power grids (COCOON) project main goal is to use an inter–disciplinary approach to deliver a solution for cyber protection of modern and future Electrical Power and Energy Systems (EPES) networks. This solution will utilize a Programmable Data Plane (PDP) paradigm which will allow deployment of custom compute–intensive cyber protection applications equipped with accelerated data processing, forwarding and control functionalities. Unlike current traditional networks, which require dedicated middleboxes to deliver specific security functionality, a PDP device can support multiple security functions dynamically as per the operator's requirements supported by vast number of device types, such as the Single Board Computer (SBC), the Next Unit of Computing (NUC) and the programmable switches. Such a holistic network solution will provide up–to–date protection against modern threats emerging from the digitization of EPES networks.

The main objective of this deliverable, D4.2: *COCOON system architecture*, is to define an architecture for the COCOON Programmable Node (CPN) - a device utilizing the PDP paradigm and serving as a platform for the COCOON Toolset deployed on a centralized controller which will integrate cyber security services used by the EPES operators. The deliverable elaborates on foundations already described in previous deliverables, specifically deliverable D4.1 presenting CPN architectural abstractions, blueprints and requirements; and deliverable D1.1 presenting the Control Measurement and Monitoring Layer (COMML) and its architecture requirements and properties. The proposed system architecture will be used in development process of the entire CPN system. Moreover, the project participants can use this deliverable as a guide for developing Instrumentation and Orchestration Layer (IOL) Micro Network Function (μNF) supporting high–level Cybersecurity Services Layer (CSL) services.

The deliverable structure follows the layers in System architecture layers. Section 1 introduces scope of the deliverable, its relation with other work packages and tasks and the methodology. Section 2 presents the overall CPN system architecture and explains how CSL services are translated into COMML packet–level primitives. This process is called services decomposition and spans across all the architectural layers. Lastly, description of the devices that can support CPN deployment is provided. Section 3 describes the COMML of the CPN architecture and its two main components – the agent and the switch. An introduction to extended Berkeley Packet Filter (eBPF) implementation is provided, such as the instruction set, kernel helper functions, maps and the verifier. These details provide crucial set of information that is required for the development of μNFs. This section introduces the Southbound Application Programming Interface (SBI) and enlists supported message types.

Section 4 details the IOL and its elements, including the service broker, the event handler and the listener functions, leveraged libraries and tools. Implementation details of eBPF μNFs is provided and explained via an exemplar code of an eBPF μNF. Additionally, this section introduces the Northbound Application Programming Interface (NBI) definitions, required for communicating with the CSL. It is noted that, at this stage of the project, the NBI has not been finalized. This stems from close dependence of the NBI's modules' definitions and related implementation, to the requirements from CSL services. A comprehensive detail on the NBI will be defined as part of the proceeding deliverables. Section 5 presents use cases of the CPN deployment including an emulated environment for development purposes and the four pilot topologies for the Technology Rediness Levels (TRL) 7 deployment. Finally, Section 6 concludes the deliverable and summarizes its main contributions.

1 Introduction

The key aspect of the COCOON project is development of the CPN system composed of nodes and the controller. This system will act as a platform supporting network interaction for all the services. The goal of this deliverable D4.2: *COCOON system architecture* is to provide detailed system architecture which will support the development of the CPN adhering to the properties of the DevOps processes defined within task T4.1 and deliverable D4.1.

This deliverable follows up the work laid out in the previous deliverable D4.1 *COCOON Development Blueprint* where generalized key concepts of the system architecture, its requirements and blueprints were presented. The COMML requirements and properties were then expanded upon and presented in the deliverable D1.1 *Control, Measurement and Monitoring* together with computer network background used in the project. This deliverable finalizes the CPN system architectural description. Building on this, it also continues to provide useful background information for the project member in terms of CPN–compatible devices and a guide for writing custom μNF. Both of these properties will be utilized in pilot demonstrators.

This deliverable describes all aspects of the COMML architecture and the IOL architecture and presents in detail how all the components and modules will be implemented in COCOON pilots as well as in real networks.

1.1 Scope of the deliverable

This deliverable is focused at IOL and COMML of the CPN system architecture. The CSL is out of the scope of this deliverable, but its elements related with the remaining layers are still mentioned. This includes NBI and CSL services decomposition to μNFs located in IOL and COMML of a CPN.

Theoretical material presented in this deliverable is limited to programmable networking devices which can be used for CPN deployment and eBPF-related information such as its instruction set, verifier functionality, maps for persistent data storage and eBPF helper functions. A summary of these features is crucial for future development of μNFs which will be done by various project members.

The rest of the information is related to the CPN system architecture design which will form a baseline for the system development and deployment. Several parts such as target implementation devices and final structure of NBI have not been finalized and this deliverable points this fact out in relevant sections.

1.2 Relation with other work packages and tasks

This section describes deliverable's position in relation with other tasks and deliverables within the project. This deliverable is the second deliverable within the Work Package (WP) 4 and it expands the foundation explained in the deliverable D4.1 *COCOON Development Blueprint* produced by T4.1 and the COMML description provided in the deliverable D1.1 *Control, Measurement and Monitoring* produced by T1.1 as depicted in Figure 1.1.

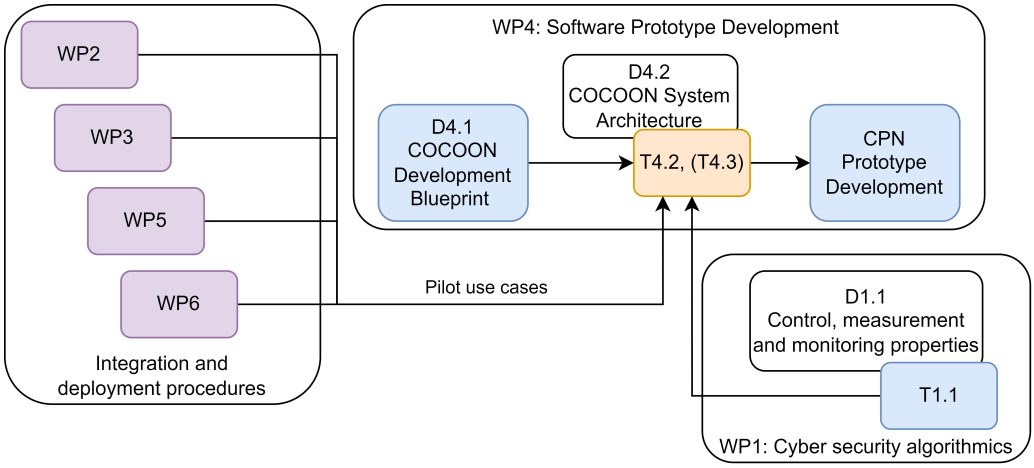

Figure 1.1: The relationship of D4.2 with other tasks, deliverables and WPs

The goal of this deliverable is to utilize the blueprint from deliverable D4.1 to further refine the low-level software architecture of the CPN COMML defined in deliverable D1.1 and to propose the IOL architecture. The COMML together with the IOL provides the complete system architecture, that will be further utilized for the development of the CPN prototype.

This deliverable includes inputs provided by partners. The details include WP2, WP3, WP5 and WP6 and presents pilot's topologies and the CPN placement within them.

1.3 Methodology

The methodology used for this deliverable combines offline and online discussions between partners with the goal of identifying pilot requirements for the CPN system architecture. Based on these discussion, the CPN system architecture is defined in this deliverable.

Additionally, partners were highly involved in developing Section 5. Specifically, TU Delft (TUD) provided expertise on topology of a digital substation and their lab setup. Aristotle University of Thessaloniki (AUTH) and Hellenic Electricity Distribution Network Operator (HEDNO) provided expertise on energy communities topology, Ingelectus (ING) provided description of Photovoltaic (PV) power plants and finally, Southeast Electricity Network Coordination Centre (SELENE CC) provided information about their secure regional electricity data operations pilot. In summary, this deliverable was written in a collaborative effort following an iterative development approach with several online meetings for feedback.

2 CPN system architecture

This section presents an overall high-level view on the CPN system architecture and its components, elaborated further in Sections 3 and 4. Figure 2.1 presents this overview and it will be referenced to in the following sections. It shows an exemplar topology of an EPES network composed from a primary substation, distributed PV generation and a secondary substation in the left part. Two main components are highlighted in colors in this part - the CPN controller (orange) and programmable nodes (green). These components are then mapped to the right part of the figure which shows the CPN architecture composed of the three layers - CSL, IOL and COMML.

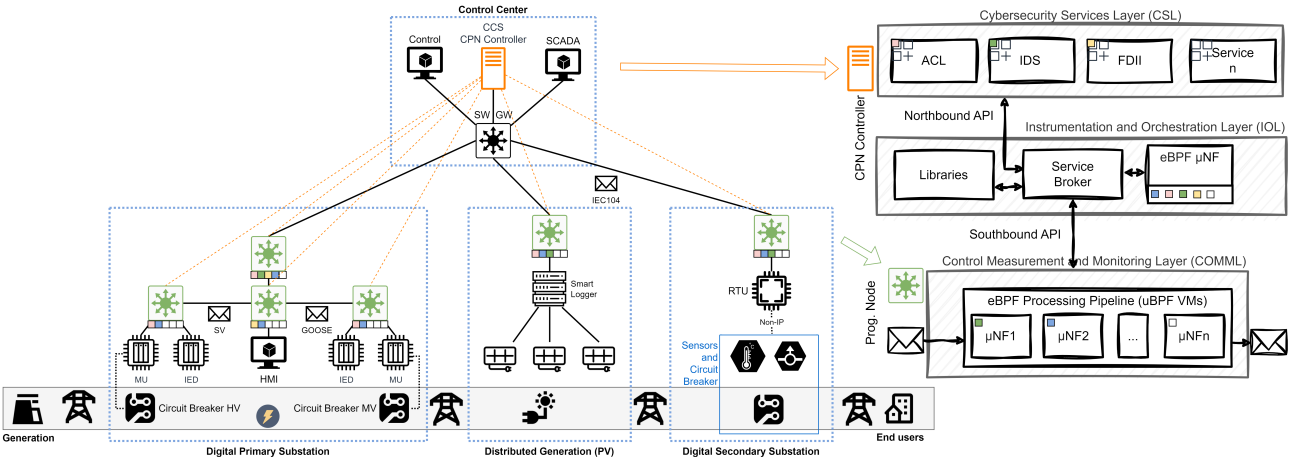

Figure 2.1: Overall CPN system architecture in a realistic topology scenario

The EPES topology represents the power transmission and distribution all the way from a power generation (including distributed power generation) to end users - as illustrated in the bottom part, grayed section. This section is composed from analog devices such as circuit breakers and sensors which report grid operation values to devices which will transform these values into a digital form and encapsulate within International Organization for Standardization / Open Systems Interconnection (ISO/OSI) communication protocols. These protocols, which are part of the International Electrotechnical Commission (IEC) 61850 standard, such as Sampled Values (SV) and Generic Object-Oriented Substation Event (GOOSE), and IEC 60870-5-104 (IEC104) use standard Ethernet connection represented by solid black links in the figure. Each part of the EPES can use different protocols. For example, a digital primary substation contains a more complex functionality than digital secondary substation, and therefore uses additional communication protocols used within the substation.

In the figure, the CPN controller is managing multiple CPN. This represents the "ideal" desirable architecture. Within the COCOON project scope and its pilots, the CPN controller and both its layers - CSL and IOL can be integrated within a single CPN. In this case, the CPN will be an autonomous device without connections to any controller. This will simplify system testing in pilot scenarios.

The desirable version shown in the figure will be tested in a virtualized environment as a concept for future EPES networks which will be built with centralized control and network programmability in mind. In this version, CSL and IOL placement is flexible and can either

be on a single server device, or two separate devices - either physical, or virtual. The CPN is always located on a programmable network device.

The right part of Figure 2.1 also shows the relationship between CSL services and μNF in both IOL and COMML. To provide a general overview, a CSL service must have one or more supporting μNFs which provides networking functions such as data collection. IOL functions as a library of these functions which can then be dynamically installed into a CPN pipeline implemented in the COMML. This relationship will be further elaborated in Section 2.3.

2.1 Architecture overview

The CPN system architecture is composed of three main layers as shown in Figure 2.2, which is a more detailed depiction of the right part of Figure 2.1. These layers are: the CSL, the IOL and the COMML. On this architecture representation, CSL is running externally from the CPN controller due to its NBI flexibility.

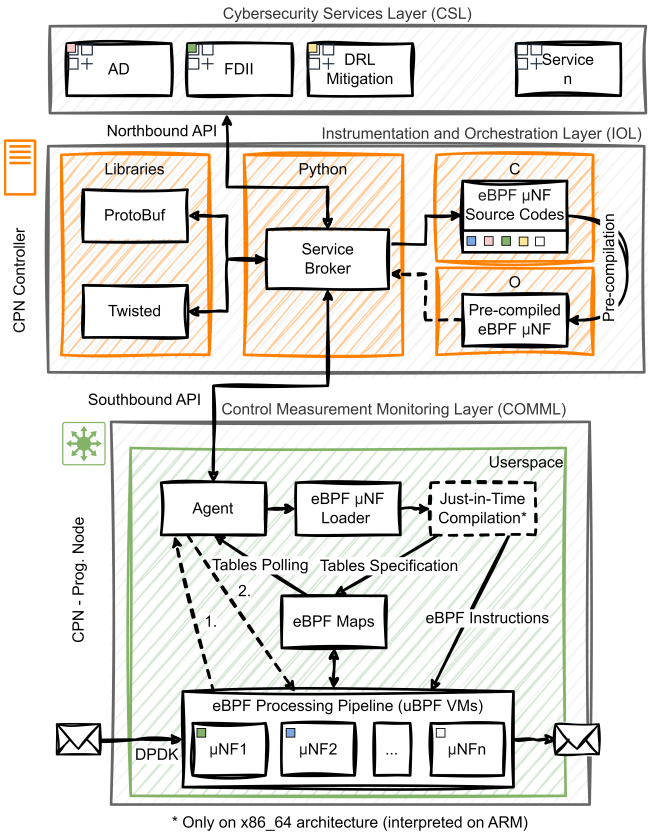

Figure 2.2: Detailed CPN system architecture

The IOL and the COMML will be explained in detail in Section 3 and Section 4 respectively, but in principle, essential functionality provisioned by these layers is:

- The CSL provides a platform for developing third party services, independent of the CPN platform. These services can be implemented in any programming language and the only requirement is a compatible Application Programming Interface (API). The CSL will be primarily implemented on a server as part of the CPN controller and therefore can support computationally intensive cyber protection applications including the Machine Learning (ML)–based algorithms.

- The IOL is an abstraction layer between CSL and COMML and it provides NBI and SBI, respectively, for communication between the layers. It includes the service broker responsible for handling API requests and μNFs management. This includes installation of μNF into the COMML PDP of a CPN and their removal. It also processes all data exchange between CSL services and storage located on the CPNs.

- The COMML utilizes PDP to implement low–layer packet–level primitives and a processing pipeline composed of active μNFs. These functions perform basic network operations such as extraction of specific packet fields, or their modification. The COMML is also responsible for efficient data storage of network parameters which can then be provided to the higher layers.

2.2 Architecture requirements

Architecture requirements were defined in D4.1 in details, but the most important ones, which were marked as "M" (must have) are summarized in this section.

The IOL must have the following four requirements: (i) an ability to utilize custom eBPF functions for support of services in the CSL, (ii) include required Linux kernel header files for providing the CPN functionality, (iii) an ability to manage order of execution of micro-network functions, (iv) the service broker for management of micro-network functions and APIs.

The COMML must have the following four requirements which are focused on packet processing, forwarding, control and monitoring: (i) ability to read and write on network interfaces and to parse and de–parse packet information for further analysis, (ii) forward the traffic with minimum overhead, (iii) ability to define basic packet actions - drop and re–route, (iv) ability to collect and provide various data using eBPF map structures.

2.3 Services decomposition

The COCOON solution in form of the CPN system architecture is based on decomposition of high-level generic services such as Anomaly Detection (AD) into low-level packet level primitives such as drop packet which can be supported by a PDP. This is achieved with eBPF μNF which can be composed into a PDP pipeline.

Figure 2.3 shows the relationship between services, μNFs in IOL, illustrated in orange, and μNFs in COMML, illustrated in green. A service is always part of the CSL and based on its complexity, might require one or more μNFs as shown in colored boxes. Note that this layer is part of the CPN controller and it might be located on the same physical or virtual device as the IOL as explained in Section 2 and Figure 2.1. An example of a service is "Deep Reinforcement Learning (DRL) Mitigation" which requires two μNFs - one for data collection, the "yellow box" and one for blocking functionality, the "brown box". These eBPF are pre-defined at the IOL, as "DRLM Collection" and "DRLM Blocking", respectively. Services can be dynamically added without requiring any change in the architecture provided that supporting μNF are added as well. This might include services such as encryption, authentication, asset discovery, firewall, network scanning, etc.

Every service must have corresponding eBPF μNF(s) which reside in the IOL and are saved as a source code in a "C file". Prior to their usage by the COMML, they need to be precompiled into an "object file". Note that the example in Figure 2.3 shows the forwarding μNF which does not have corresponding CSL service. Every networking functionality has to be programmed into an μNF and these μNFs serve as a building block, that are then used by CSL services, or the service broker to provide network operations.

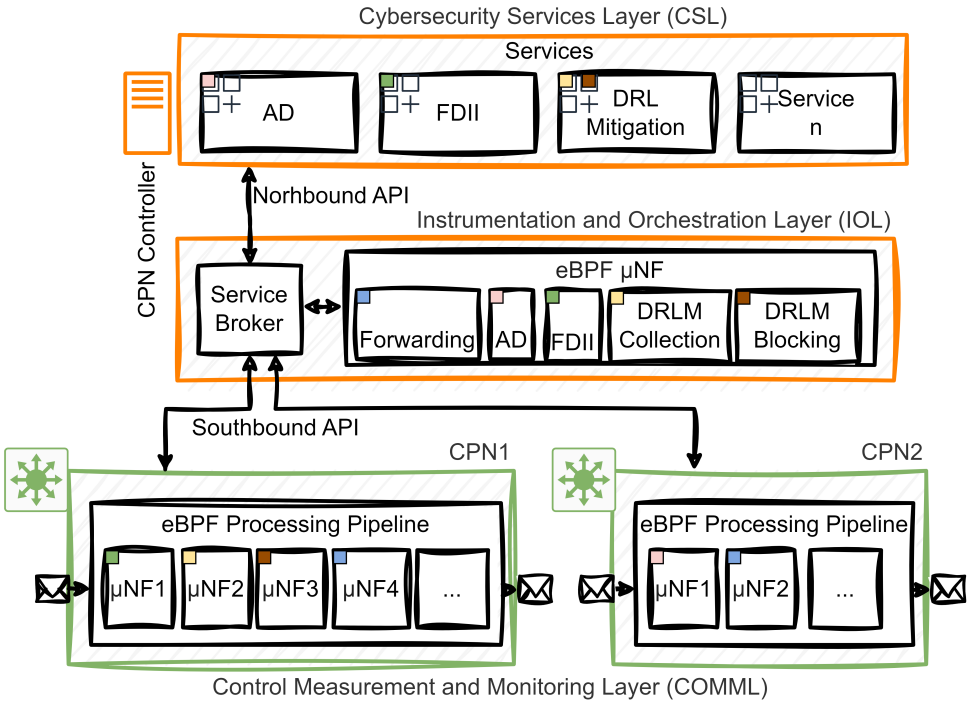

Figure 2.3: Services decomposition in the CPN system architecture

Finally, Figure 2.3 shows an example of two CPNs which are both managed by a single CPN controller. Every CPN can have a different processing pipeline which is composed of μNFs from the IOL. These can be installed and uninstalled dynamically upon the service broker request. The request is triggered from the CSL dynamically. In the example, the CPN1 has 4 installed μNFs while the CPN2 has 2. It is worth mentioning that the order of μNFs in the pipeline matters as they are processed sequentially. For example, if the forwarding μNF would be the first and its result would be to forward the packet to an interface, none of the following μNF would be executed. This logic of μNF sequencing will be handled by the COMML network function chaining process, controlled by the service broker.

2.4 Programmable networking devices

This section describes basic types of networking devices used in the COCOON project and presents devices suitable for physical deployment of the CPN. The section aims at supporting future terminology for pilots building as the information provided will help in pilots configuration and deployment.

2.4.1 Traditional networking devices

A traditional networking device is a box which contains both data and control plane as explained in deliverable D1.1. This means that the entire logic and configuration of how to process network traffic is present on the device. There are two types of these devices, unmanaged and managed which will both be used in the project.

Unmanaged devices

These are vendor–specific devices with an integrated set of unchangeable features. Unmanaged devices are typically only used for simple traffic forwarding without any additional functionality such as security features. These devices might be used in the project in limited

scenarios such as transitions between different network types. An exemplar of an unmanaged industrial switch is shown in Figure 2.4.

Figure 2.4: Exemplar of an industrial unmanaged switch

Managed devices

Managed devices provide an option to configure available features. This can include security features, depending on the device complexity. These devices will be used for most networking devices in the project, which does not need to run the CPN. An exemplar of managed switches is shown in Figure 2.5.

Figure 2.5: Exemplar of managed switches

2.4.2 Programmable networking devices

Programmable network devices are different types of devices than traditional networking devices as they offer customization of network processing. Unlike traditional networking devices that have a pre-defined set of configurable features, programmable networking devices can be used to implement user-defined features.

Programmable networking devices can be separated into three categories based on supported architecture: (i) Software Defined Networking (SDN), (ii) Programming Protocol-independent Packet Processors (P4) and (iii) eBPF. Only devices with the eBPF support will be utilized within the project.

eBPF devices

An eBPF device enables full programmability in the most flexible instruction set and is selected for the CPN implementation. An eBPF device can be any device which runs a Linux

kernel with the eBPF support, which is standard in the most common kernels. This makes eBPF suitable for various range of scenarios including the CPN implementation. Possible deployment options for the CPN are described in the following section.

2.4.3 COCOON Programmable Node platforms

eBPF is a technology of Linux kernel and it can therefore run on any such device. This means that traditional and programmable networking devices are usually not supported as most of them are based on proprietary firmware. The following device types can be used for the CPN.

Single Board Computer (SBC)

A SBC is type of a computer with all the components such as Central Processing Unit (CPU), Random-Access Memory (RAM) and Network Interface Cards (NIC) integrated on a single circuit board. These are typically relatively low-performance, small factor and reasonably-priced devices suitable for portable demonstrations and development testing. The best example is Raspberry Pi, shown in Figure 2.6, which uses Advanced RISC Machine (ARM) architecture and can have up to 8 GB of RAM while costing under 80 GBP [1]. This provides a suitable platform for initial real device deployment testing of the CPN within TRL 6 and selected pilots such as the digital substation described in Section 5.2.

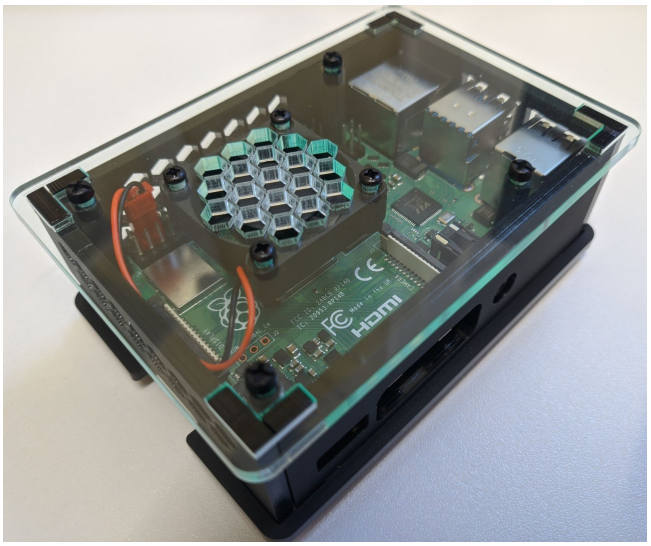

Figure 2.6: Exemplar of a Raspberry Pi SBC

Next Unit of Computing (NUC)

NUC is a small-form-factor bare bone computer which uses laptop components including CPU and RAM. These components have significantly higher performance and power consumption than SBC and are suitable for more demanding scenarios. These can be used in the same way as the SBC, but in scenarios where more processing power is required such as in pilots with higher data throughput. Both SBC and NUC might have limited support of Data Plane Development Kit (DPDK). They might be used for initial CPN testing in pilots due to their relatively low cost. An exemplar of a NUC from Nvidia is shown in Figure 2.7.

Figure 2.7: Exemplar of a NUC

Servers

Servers in various forms can also be used for the CPN implementation. Their advantage is potentially much higher performance and variability of NIC configurations which can support DPDK. Disadvantages are higher price, larger dimensions and higher power consumption. A server implementation will be used for initial DPDK testing at University of Glasgow (UGLA) and might also be chosen for some pilots in case of problems with other types. An exemplar of a server is shown in Figure 2.8.

Figure 2.8: Exemplar of a server

Whitebox / bare metal switches

The last category are whitebox or bare metal switches with open design and x86 CPU architecture. These are manageable switches and typically run Open Network Linux (ONL) - an open-source platform for modular Network Operating System (NOS) architecture on open networking hardware. An example switch can be the Edgecore CSR320 [2] with temperature hardened operations, redundant and hot swappable power modules and redundant 4+1 fans for high availability. A whitebox switch might be used in pilots deployment if high performance in hundreds of Gbps will be required from the CPN.

3 COMML architecture

The COMML represents a PDP and its overall architecture and properties were described in deliverable D1.1. The main purpose of the COMML is to provide PDP pipeline functionality for a network programming device. This will be used by the IOL which will install μNF into this pipeline based on requests from the CSL services. This section describes COMML architecture and its components in detail. Figure 3.1 shows the architecture separated into two main parts: agent and switch.

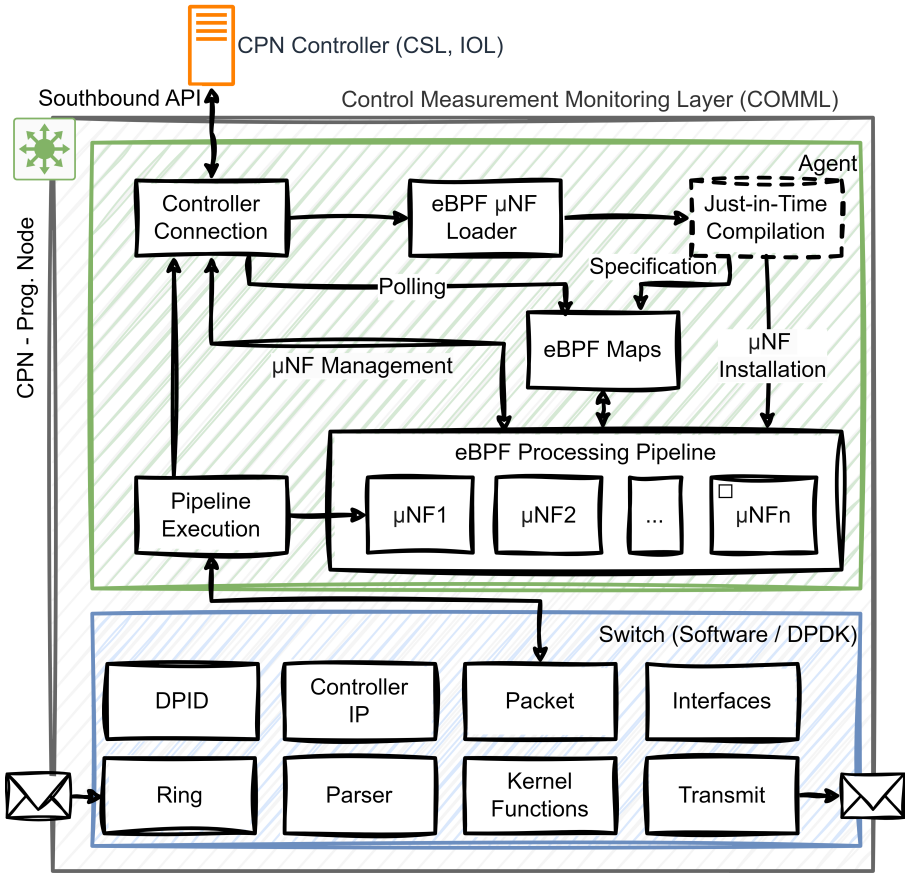

Figure 3.1: The COMML architecture

The agent part depicted by green color is responsible for communication with the CPN controller's IOL and for management of eBPF–related aspects. This includes eBPF loader, Just in Time (JiT) compiler, eBPF maps and the eBPF processing pipeline managed by the pipeline execution application. This is a software component which stays the same for all switch types. It will use C language for the implementation.

The switch part depicted in blue color can be represented by a software switch, or a DPDK switch depending on the target implementation. Displayed components will be the same for both switch types, but their implementation will be different depending on the switch architecture. If there would be a different switch architecture considered, for example Express Data Path (XDP), this would require a new switch part of the CPN architecture.

3.1 COMML agent

The COMML agent is a component responsible for providing an abstraction layer between the COMML switch, which is target specific, and the IOL. It is a component of the CPN with which the IOL interacts if there is any request from a CSL service or in case IOL itself needs to perform an operation on the CPN. The COMML agent uses the C language for optimal performance and it handles multiple functions described in the rest of this section.

3.1.1 Agent functions

The COMML agent is responsible for a diverse set of functions which are shown in Figure 3.1. The core functionality is the eBPF processing pipeline which presents a place for μNF installation. The controller connections manages the COMML part of the SBI. This can, upon a request, trigger a μNF installation via the eBPF μNF loader. Additionally, the controller connection can manage and poll eBPF maps which correspond to μNF in the eBPF processing pipeline. Finally, the pipeline execution is responsible for forwarding received packets to the corresponding stages in the eBPF processing pipeline and if required, to the controller connection for a processing by the IOL. These functions are described in detail below.

Controller connection

The controller connection component provides the SBI on the COMML. It defines functions for handling receive and send functions for SBI messages described in Section 3.1.2. Based on the received requests, it can trigger an interaction with eBPF μNF loader, eBPF maps, or the eBPF processing pipeline.

Pipeline execution

Pipeline execution is a function which is called from the COMML switch layer when a new packet is received. The function ensures sequential traversal through the eBPF processing pipeline. This includes all installed functions. If there is no function installed, packet is dropped. The packet can also be forwarded to an output port at any time during the pipeline traversal.

As mentioned in Section 2.3, eBPF μNF located in the pipeline are composed of network primitives (such as to extract a header field, or store a defined parameter) and they form basic building blocks for network functionality of the CSL.

eBPF micro network function (μNF) loader

When an μNF installation request is received by the COMML, the eBPF μNF loader ensures that the function is installed into an appropriate slot in the eBPF processing pipeline. This might include compilation if the target platform is the x86_64 architecture. On other platforms, including ARM, eBPF are installed directly without this compilation. This is because compiled code is translated into machine code before they are executed, while in case of interpretation only, the codes are translated into machine code at runtime by an interpreter.

Just in Time (JiT) compilation

The JiT compiler translates generic bytecode of the eBPF μNF from the IOL into the machine specific instruction set. The purpose is to optimize execution speed of the μNF and it makes eBPF μNF run as efficiently as natively compiled kernel code or as code loaded as a kernel module. The userspace JiT compiler is used only for x86_64 architecture of the COMML switch as ARM architecture of the switch is interpreted. This is due to the smaller instruction set of the ARM CPUs which affects the overall performance.

Table 3.1: SBI Message Definitions

Southbound API (SBI)		
Message Type	**Direction**	**Description**
Hello	C↔N	Handshake message between CPN and CPN controller
Install	C→N	Install eBPF ELF on the CPN
PacketIn	N→C	Packet sent from CPN to CPN controller
PacketOut	C→N	Packet sent from CPN controller to CPN output port
TablesList	C→N	List all the instantiated tables
TableList	C→N	List the content of the tables specified
TableEntryGet	C→N	Get a single entry from the table specified
TableEntryInsert	C→N	Update or Insert an entry with the key and value provided
TableEntryDelete	C→N	Remove an entry from the table specified
Notify	N→C	Asynchronous notification of an event with the user defined payload

N: CPN at the COMML, C: CPN controller at the IOL.

3.1.2 Southbound API (SBI)

The CPN controller needs to have a global view of the network topology, where the CPN controller maintains a persistent–connection with the controlled CPNs in the network. A CPN persistent–connection is maintained by the CPN controller to transmit synchronous messages to the CPN, and for the CPN to raise events and notifications to the CPN controller. Analogous to the SBI defined for the SDNs, namely, OpenFlow (OF) by the Open Networking Foundation (ONF) [3], or Network Configuration Protocol (NETCONF) by Cisco [4], this connection is referred to as the SBI, defined specifically for CPN implementations.

The types of the messages defined for the SBI, utilized for the communication between the CPN controller at the IOL and the CPN at the COMML are presented in the Table 3.1 and their functionality is depicted in Figure 3.2. This serves as a reference to SBI message definitions. Features of the SBI are as follows:

- The CPN SBI supports both in–band and out–of–band control plane implementations, and the choice depends on the infrastructure availability, and the security requirements.
- At the first step of the communication, the persistent connection is setup via a handshake between the CPN controller and the CPN at the PDP. This is required to establish the version compatibility between the endpoints, and acquiring the Datapath identitifier (DPID) of the connected CPNs at the PDP. Once the connection is established between the CPN controller and the CPN, programmability can be introduced via μNFs, installed at the PDP pipeline using an "install" SBI message. For instance, the "L2 forwarding μNF" will be installed automatically once the "Hello" message is received from the switch.
- The CPN controller maintains global view of the controller, wherein each eBPF program keeps its internal state in a set of tables, and contents of these tables can be retrieved utilizing SBI messages. For example, "TablesList" lists all tables maintained at the CPN. The controller can also access the contents of a specific Table by TableList, access an entry within that Table. The controller can also insert an entry into the Table, or delete an entry from the Table upon receiving a message from the controller.

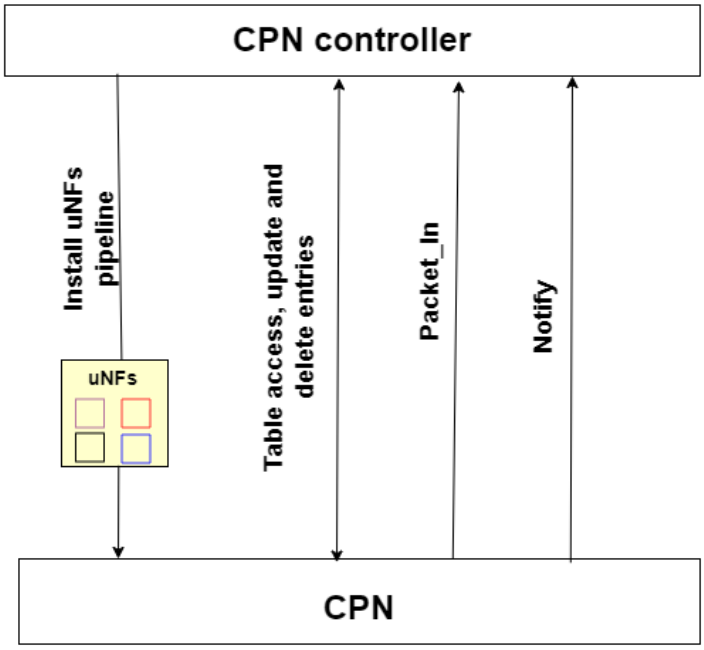

Figure 3.2: SBI communications between CPN controller and CPN

- In addition, the SBI for CPN implementations supports asynchronous message events, such as "notify" message transmitted from CPN to the CPN controller with user–defined payload embedded into it. Also, the "PacketIn" message, analogous to "packet_in" message in OF, is defined to transmit a message from the CPN to the CPN controller, in case of a table–miss or explicit–forwarding to the controller for further processing. An example would be a self-learning switch that will require installing the related μNF on the PDP. The decision to add an entry into the Table is provided to the CPN via SBI install message type.

The SBI will require an eBPF loader that works as an agent to be installed on the switch. This agent is similar in functionality to the OF based agent, working as in intermediary between the CPN controller and the CPN PDP.

3.1.3 eBPF implementation types

Deliverable D1.1 presented a high-level overview of possible eBPF implementations within a system composed from three elements; the userspace, the kernel, and the NIC. To briefly summarize, Figure 3.3 presents a general system overview with the *bee* symbol representing a possible eBPF implementation. Every implementation has unique features which were described in more details in D1.1 as:

- **Userspace**: This is the most flexible implementation supported on all devices and architectures, but with lower performance.
- **Generic**: This is a kernel implementation which does not require any support of specific features and has similar performance to the userspace implementation.
- **Native**: This is an efficient kernel implementation which utilizes XDP to avoid slow and expensive kernel network stack processing, but requires the XDP support.

- **Offloaded**: This is a hardware-accelerated implementation which is currently supported only on a very limited number of NICs, but offers the best performance.

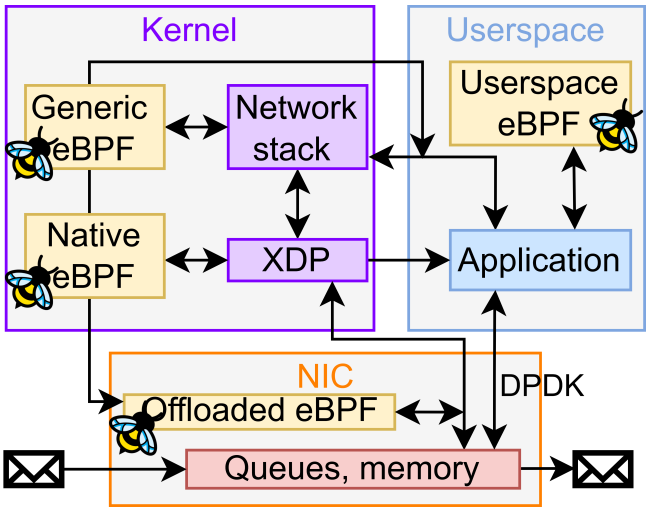

Figure 3.3: eBPF implementation types

The userspace implementation was selected as the most suitable for the project requirements and the rationale is presented in the following section which will explain the selected userspace type - userspace Berkeley Packet Filter (uBPF) and its combination with the DPDK framework which can eliminate the userspace disadvantage in lower performance.

Userspace implementation

Traditional implementation of eBPF is within the system kernel. While this provides good performance and kernel safety verification, the main disadvantage is limited flexibility and dependence on the kernel version. This might limit the Operating System (OS) update potential which could cause CPN incompatibility. This might be a problem in Operational Technology (OT) networks where compatibility and proper functionality must be ensured.

The userspace eBPF implementation, on the other hand, runs in the userspace as any other program [5]. Therefore, it is independent of the kernel version and it enables customization. Moreover, a code execution in userspace does not require root access. This aligns well with the security requirements of the project goals. Finally, CPN implementation requires designing custom eBPF functions, necessitating userspace implementation. For instance, uBPF with customized eBPF functionality allows eBPF maps to be allocated if they have not been created, and relocate them before the code is JiT compiled. This enables flexibility for future services being integrated into the existing deployments.

However, the main disadvantage of the userspace implementation is lower performance when compared with kernel implementations. Since DPDK passes packets directly between NIC and userspace, avoiding the kernel completely, combining the userspace implementation with the DPDK technology eliminates this drawback, detailed in Section 3.1.3. Figure 3.3 details the implementation method. DPDK technology

uBPF

There are several userspace eBPF implementations such as uBPF, Rust userspace Berkeley Packet Filter (rBPF) and bpftime. We selected uBPF for the CPN deployment mainly for its maturity (Microsoft's efforts to port eBPF to Windows operating environment and integration with DPDK). uBPF is available under the Apache License and provided via GitHub [6].

DPDK

DPDK is an open source framework consisting of a set of libraries for accelerating packet processing on a wide range of CPU architectures. The CPN architecture will utilize DPDK to avoid expensive kernel processing as DPDK bypasses kernel completely and passes traffic directly to the userspace layer. DPDK consists of the following components [7]:

- **The Ring Manager** provides a multi-producer and multi-provider First In First Out (FIFO) queue implemented as a table optimized for fast, bulk operations.
- **Memory pool manager** allocates objects in memory in a ring structure which can be spread across different RAM channels.
- **Network packet buffer manager** provides an API for allocation of message buffers stored in a memory pool. These are used for manipulation of network packet contents.
- **The Timer manager** provides an interface for precise time reference and asynchronous or periodical function calls.

DPDK will be utilized for the CPN TRL 6 and above after a successful testing of the plain uBPF implementation. From our previous experiments, presented in [8], DPDK achieved slightly higher performance than the XDP implementation and approximately twice to that of the plain userspace implementation. This proves the suitability of uBPF implementation and overcoming its performance disadvantage by integrating DPDK.

3.1.4 eBPF instruction set

An eBPF program is a sequence of instructions [9]. An eBPF program can be written in high-level language such as C, but before its use must be compiled into a byte code which uses the eBPF instruction set.

- The eBPF instruction set consists of eleven 64–bit registers, a program counter, and an implementation–specific amount (e.g., 512 bytes) of stack space.
- The eBPF programs needs to spill/fill the registers, if necessary, across calls. The reason for spilling/filling is due to the limited number of registers.
  - Spilling means that the value in the register is moved to the eBPF stack.
  - Moving a variable from eBPF stack to the register is called filling.

eBPF instruction set can be separated into several instruction classes. These contain load operations, 32-bit and 64-bit arithmetic operations and jump operations. These support underflow and overflow where the value will wrap. If a division by zero operation is detected, the destination register is set to 0.

Arithmetic instructions: Some of the basic arithmetic instructions with mathematical expressions with two variables, dst (destination) and src (source), are:

- BPF_ADD adds source to the destination (dst += src)
- BPF_SUB subtracts source from the destination (dst -= src)
- BPF_MUL multiplies source with the destination (dst *= src)

- BPF_DIV divides destination by the source if the source is not 0 (dst = (src != 0) ? (dst / src) : 0)
- BPF_AND performs logical AND operation (dst &= src)
- BPF_OR performs logical OR operation (dst | = src)
- BPF_XOR performs logical XOR operation (dst ∧ = src)
- BPF_MOV moves the source into the destination (dst = src)

Atomic operations: Some of the arithmetic instructions can be executed within a single instruction cycle. These operate on memory and can not be interrupted or corrupted by other access to the same memory region by other eBPF programs or means outside of this specification. Supported atomic operations are ADD, OR, AND and XOR.

3.1.5 eBPF verifier

The eBPF verifier is a core module in eBPF–based implementations. An eBPF program before being loaded in the kernel–space, needs to pass a set of requirements. The verification ensures the the program is safe to execute. The process is illustrated in the Figure 3.4. Since eBPF programs can be translated into native machine and execute in the kernel mode, it necessitates the eBPF verifier utility. If the programs are not thoroughly verified this could lead to several critical breaches/errors, such as, memory corruption, information leakage, leading to a kernel crash or a kernel deadlock/hang. A significant advantage of a verifier is that the programs can run at native speed once the program is verified and there would no computationally expensive runtime checks.

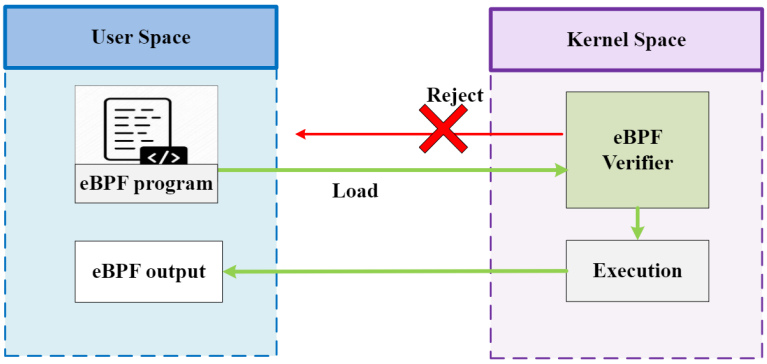

Figure 3.4: Program verification by eBPF verifier

Further elaboration is presented on the "safe" execution of an eBPF program. The main idea is that the program should not violate the security model of the system.

- **Path check**: Verifying potential paths the program would take when executed in–kernel.
- **Loops**: Ensures that the program runs to a completion, without loops which might result in kernel lookup.
- **Memory access**: The program can access the memory in a structured way. Since reading a memory could potentially lead to leak of sensitive information, the programs are not allowed to read an arbitrary memory. Also, uninitialized memory cannot be read since it could lead to leaking of sensitive information.

- **Deadlock**: Programs are not allowed to reach a state of deadlock, so it is verified that only one lock is held at a time to avoid deadlocks with multiple programs.

A functional implementation verifier's processing is given in Listing 3.1 as an example, where a program tries to leak a kernel address on the line 4.

```
1 SEC("socket1")
2 void *bpf_prog(struct __sk_buff *skb)
3 {
4     return (void*) skb;      /*Try to leak pointer*/
5 }
```

Listing 3.1: Exemplar code tries to leak kernel address

```
1 0000000000000000 <bpf_prog>:
2     0: r0 = r1; BPF_ALU64_REG(BPF_MOV, BPF_REG_0, BPF_REG_1)
3     1: exit; BPF_EXIT_INSN()
```

Listing 3.2: The compiled eBPF instructions

The program in Listing 3.1 tries to cheat the compiler by casting. The code given in Listing 3.1 is compiled to the two BPF instructions, given in Listing 3.2. Graphically presented in Figure 3.5, the first instruction moves whatever is in register r1 into r0, which is allowed by the verifier, however, the second instruction, the exit is not allowed, since registr r0 should contain a scalar to exit in principle. In this way, the eBPF verifier rejects the code by type checking.

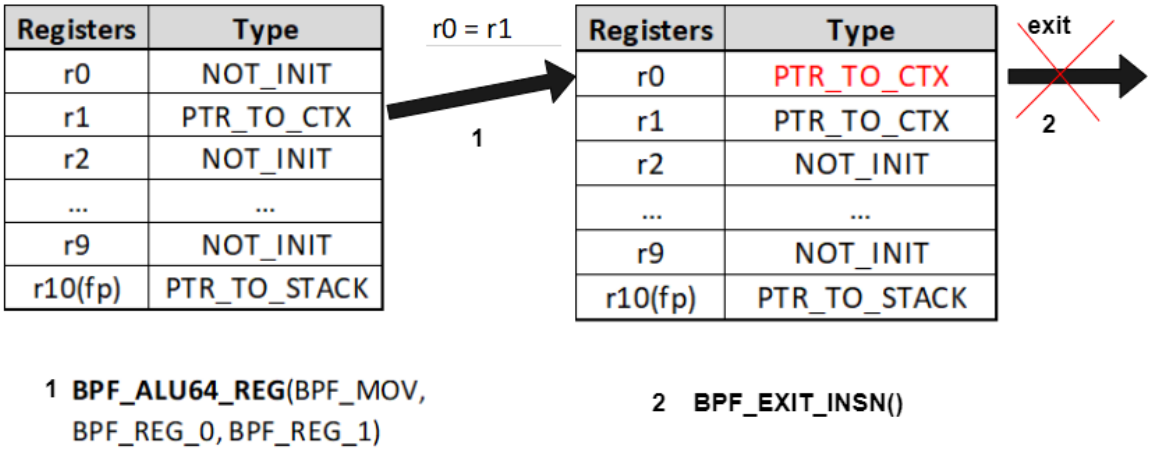

Figure 3.5: Example of Memory Leak attempt rejected by the verifier

3.1.6 eBPF maps

The eBPF maps are leveraged by eBPF programs to share collected information and storing the state. They are the only mechanism for persistent data storage between every μNF cycle and they will be crucial for providing data to the CSL and IOL. Their main features are:

- The data can be stored, manipulated, retrieved, deleted, and so on, using a variety of data structure, such as, arrays, stack, trees, hash tables, and so on.
- eBPF maps can be accessed from eBPF programs and applications in the user space, via system calls. The process of accessing the eBPF maps is illustrated in Figure 3.6.
- Some of the supported map types are:

- BPF_MAP_TYPE_HASH
- BPF_MAP_TYPE_ARRAY
- BPF_MAP_LRU_HASH
- BPF_STACK_TRACE

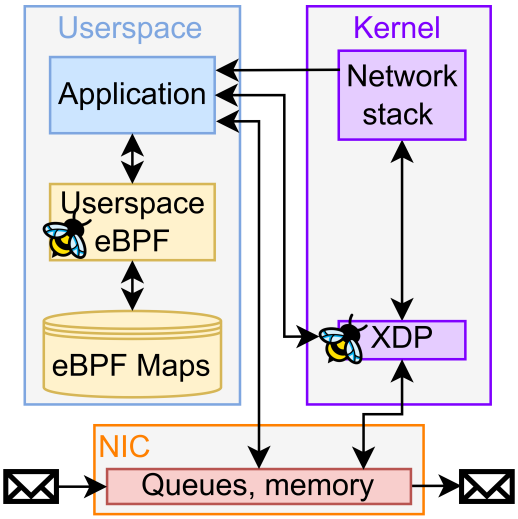

Figure 3.6: eBPF programs and applications can access the eBPF maps via system calls

As detailed in Section 3.1.3, an advantage of userspace implementation is that the eBPF maps are located in the userspace. This enables easy addition or modification of used eBPF maps, relevant for the required data plane switching scenarios. The CPN will enable implementation of wide range of applications, including, data plane switching scenarios. For example, an intelligent "self–learning" switch can feasibly be implemented using the PDP' ability to self–insert and update the contents in the switch's tables. For an incoming packet, the source and destination addresses insertion decisions can be delegated to the switch, unlike traditional OF–based implementations where the controller has all the responsibility.

3.1.7 eBPF helper functions

This section briefly lists some of the eBPF helper functions [10] required to interact with the data structures, perform packet manipulations, interact with the system, and so on. These functions are restricted to a white–list of helpers defined in the system kernel.

Tables 3.2, 3.3, 3.4, and 3.5 list some of the highlighted kernel helper functions that are used to interact with the eBPF maps, such as, retrieve, update, delete, or add packet data, computing or to manipulate the data packets, such as, updating hash values for layer 3 and layer 4 packets, and the supporting functions. These helpers are used by eBPF programs to interact with the system and implement custom μNFs.

3.2 COMML switch

The COMML switch is a component responsible for handling the ingress traffic. This is the only component which is dependent on the target platform. It has two independent implementations, one for a software switch and another one for the DPDK implementation. This component interacts with kernel and agent functions to achieve the PDP functionality. Its components can be separated into three sections: variables, structures, and functions.

Table 3.2: Helper eBPF functions

eBPF map interactions	
return	**Definition**
void	***bpf_map_lookup_elem(struct bpf_map *map, const void *key)**
This function performs a lookup in the map for an entry associated to key and returns the map value associated to key, or NULL if no entry was found. This function will be used by all μNF requiring data collection from associated maps.	
long	**bpf_map_update_elem(struct bpf_map *map, const void *key, const void *value, u64 flags)**
This function add or update the value of the entry, associated to the *key* in the *map* with the value provided. Returns 0 on success or negative error on failure.	
long	**bpf_map_delete_elem(struct bfp_map * map, const void *key)**
This function deletes the entry with *key* from the map. This function will be used by μNF to delete an entry from the map. Returns 0 on success or negative error on failure.	
long	**bpf_map_push_elem(struct bpf_map * map, const void *key)**
This function *push* and element value in the map. Here, if stack/queue is full, the oldest element is removed. Returns 0 on success or negative error for failure.	
long	**bpf_map_pop_elem(struct bpf_map * map, void *value)**
This function is used to *pop* an element from the map. Returns 0 on success or negative error on failure.	
long	**bpf_map_peek_elem(struct bpf_map * map, void *value)**
This function retrieves an element from *map* without removing it. Returns 0 on success, or negative error for failure.	
long	**bpf_for_each_map_elem(struct bpf_map * map, void callback_fn, void *callback_ctx, u64 flags)**
This function calls the callback function and other map-specific parameters for each map element.	

Table 3.3: Helper packet manipulation functions

Packet manipulations	
return	**Definition**
void	**bpf_l3_csum_replace(struct sk_buff *skb, u32 offset, u64 from, u64 to, u64 size)**
This function recomputes layer 3 (e.g. IP) checksum for the packet associated to skb.	
long	**bpf_l4_csum_replace(struct sk_buff *skb, u32 offset, u64 from, u64 to, u64 size)**
This function recomputes layer 4(e.g. TCP, UDP) checksum for packet associated to skb.	
u32	**bpf_get_hash_recalc(struct sk_buff *skb)**
This function retrieves the hash of the packet, skb $\rightarrow$ hash. Returns 32-bit hash.	
s64	**bpf_csum_update(struct sk_buff *skb, __wsum csum)**
This function adds checksum into the packet associated with *skb*$\rightarrow$csum.	

Table 3.4: Helper network control functions

Network control functions	
return	**Definition**
void	**bpf_skb_skb_vlan_push(struct sk_buff *skb, __be16 vlan_proto, u16 vlan_tci)**
This function pushes a vlan tag control information of the vlan_proto to the packet associated to skb. Return a 0 on success or negative error on failure.	
long	**bpf_skb_vlan_pop(struct sk_buff *skb)**
This function pops a VLAN header from a packet associated to skb. It returns a 0 on success, or a negative error on failure.	
long	**bpf_lwt_push_encap(struct sk_buff *skb, u32 type, void *hdr, u32 len)**
This function encapsulates the packet associated to skb within layer 3 protocol header. The header, provided in the buffer at address hdr, with len as its size in bytes, where type indicated the protocol of the header, IPV6 or GRE encapsulation. It returns a 0 on success, or a negative error on failure	
long	**bpf_skb_load_bytes(const void *skb, u32 offset, void *to, u32 len)**
This helper function is used to load data from a packet. It can load *len* bytes from offset from packet associated to *skb* into the buffer indicated by *to*.	
long	**bpf_redirect(u32 ifindex, u64 flags)**
This helper function redirects packet to another network device of index *ifindex*. XDP supports redirection to the egress interface and accepts no flag. The *bpf_f_ingress* value in *flag* is used for ingress path if the flag is present and egress otherwise. Returns XDP_REDIRECT on success or XDP_ABBORT on error.	
long	**bpf_tcp_send_ack(void *tp, u32 rcv_nxt)**
This helper function sends out a tcp ack, where *tp* is the in-kernel struct tcp_sock and rcv_nxt is the ack_seq. Returns 0 on success or negative error on failure.	

3.2.1 Switch variables

Switch variables store static configuration data which are required for correct functionality. A COMML switch keeps track of the DPID, the controller Internet Protocol (IP), and the interfaces variables, as illustrated in Figure 3.1. These have the following purpose:

- DPID is an unique identifier of every switch and the IOL of the CPN controller uses this information to keep track of every connected device. The DPID is represented as a number and can be stored in the *long* format.
- Controller IP defines the IP address of the IOL CPN controller and its Transmission Control Protocol (TCP) port. This is the address and port which will be used for all SBI messages which are being sent by the CPN.
- Interfaces is a number identifying how many interfaces the CPN has.

These variables are stored within the switch class as shown in the exemplar code listing 3.3. In this example, DPID is assigned by a function providing a random number. The IP address is set to "127.0.0.1", which represents the local system. In this case, the CPN controller runs on the same device as the node. The TCP port number is set to 9000. Finally, the interface count is set to default 0.

Table 3.5: Other function

Other functions	
return	**Definition**
u32	**bpf_get_prandom_u32(void)**
This function generates a pseudo-random number, i.e., a 32 bit unsigned value.	
long	**bpf_tail_call(void *ctx, struct bpf_map *prog_array_map, u32 index)**
This function triggers a tail call, i.e., jumps to another BPF program or μNF. This allows for program chaining, essential for μNF chaining. For security, there is an upper limit to successive tail calls. Once called, the program jumps to referenced program indexed by *index* in the program *prog_array_map*.	

```
variables.dpid = random_dpid();
variables.controller = "127.0.0.1:9000";
variables.interface_count = 0;
```

Listing 3.3: An exemplar of COMML switch variables

3.2.2 Switch structures

Switch structures are data structures optimized for efficient storage of received packets and their data. The COMML switch contains two main data structures, as illustrated in Figure 3.1. This includes the ring and the packet, where:

- Ring is a data structure for storing incoming packets for further processing. The ring structure is selected for more efficient insert, access and remove operations than traditional queues such as the FIFO queue.
- Packet is a data structure for storing a single packet information which can be accessed by the IOL of the CPN controller. This is a more selective data structure used only for packets which are identified to be of interest.

3.2.3 Switch functions

Switch functions are operations which the COMML switch can perform. This includes functions to process the ingress traffic, the kernel functions which provide supporting functionality and the transmit functionality responsible for sending the packet out of an interface. The functionality of these functions is:

- Parser is responsible for initial packet processing. It will store the packet into the ring structure and append additional metadata to the packet. This will include information such as port number on which the packet was received and a timestamp of when the packet was received.
- Kernel functions include all libraries and methods required for the switch functionality. This includes kernel header files for Transmission Control Protocol / Internet Protocol (TCP/IP) operations
- Transmit function is responsible for sending the packet out of an interface. This can either be a physical port, all the ports (flood), or the virtual port to the CPN controller.

4 IOL architecture

The IOL can be located either locally within the CPN, or on a dedicated server, referred to as the CPN controller. The later is preferred due to its alignment with the overall project goals. The advantage of having a controller is global management of all CPNs in the network. IOL is responsible for communication with CPNs, and handling requests from CSL.

The IOL architecture is shown in Figure 4.1 and is composed from 4 main components which are described in this section: (i) service broker responsible for main functionality and communication with all the other components, (ii) event handler and its components responsible for NBI and SBI communication, (iii) libraries supporting the functionality and (iv) eBPF μNFs source codes which presents a library of all available functions which can be installed into CPNs.

Figure 4.1 shows NBI towards CSL which can be placed on the same device as IOL, or separately; and SBI towards CPN as already explained in Section 2 and Figure 2.1. Most IOL internal components are written in Python, but eBPF μNF are written in C. These files must be pre-compiled into object files (.o) which can then be sent to CPNs for installation.

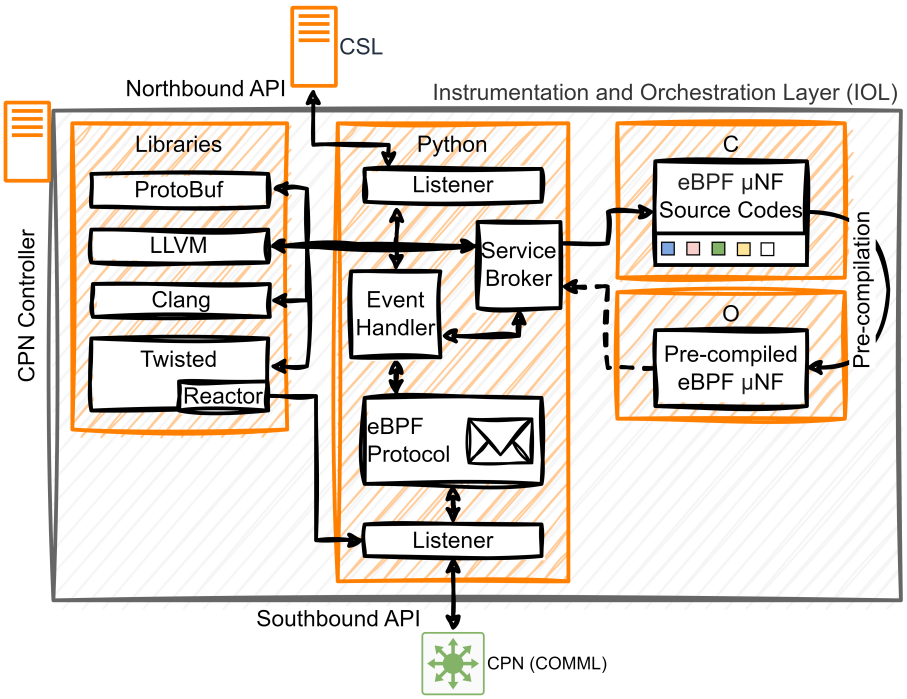

Figure 4.1: The IOL architecture

4.1 Service broker

The service broker is the main component of the IOL and is responsible for entire functionality of the layer. It uses all the other components to perform required operations. Firstly, it uses event handler for registering requests from CSL and messages sent by CPNs. These are processed with the ProtoBuf[1] and Twisted[2] libraries. Subsequently, the service broker decides

[1] `https://protobuf.dev/overview/`
[2] `https://twisted.org/`

what to do with the request. This might include installation of a new μNF, sending a query to a CPN map, or providing data to a CSL service.

4.2 Event handler and listeners

The controller logic will be implemented using Twisted Framework, explained in detail in Section 4.3.1, which is an event–based programming paradigm. The CPN controller internal control logic and interactions between the modules will follow the popular python–based controllers, such as the RYU[3] or the POX controller[4]. Twisted's event–based mechanism allows eBPF programs to listen and react to specific events generated by the CPN at the CPN PDP. The event handler uses listeners on both the SBI and the NBIs and dispatches a particular event to process the requests. Furthermore, the event handler uses custom eBPF function to handle the packets when a packet-in or notify messages is received embedded with contents in the payload.

The event handler, depicted in Figure 4.1, will be used to create independent events.

- The event handler dispatches a particular event.
- An event is dispatched for each message type. Example of an event will be CPNs connect or disconnect with the CPN controller. The "connection" event corresponds to a TCP connection establishment and a disconnection event to a connection closed event.
- CPN transmitted event: Other example of events could be, a "Hello" message transmitted by the CPN at the time of connection handshake between the CPN controller and the CPN, generated when a CPN connects to the operator network. Or it could be an "install" message event to install the eBPF μNFs onto the CPN eBPF processing pipeline.

An example of connection establishment between the CPN and CPN controller is presented in Figure 4.2. In this instance, a CPN initiates a connection request to the CPN controller.

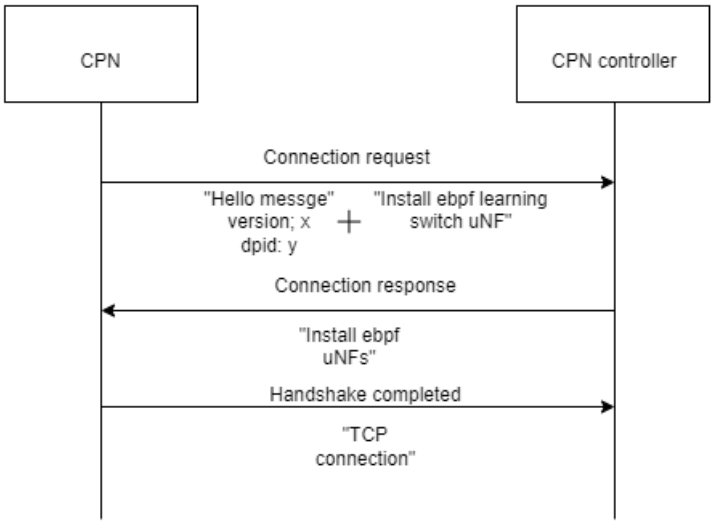

Figure 4.2: Example of TCP handshake between CPN and the CPN controller during initial the connection establishment

[3] https://ryu-sdn.org/
[4] https://github.com/noxrepo/pox

The self–learning switch μNF is installed automatically when a connection request is generated. The CPN controller further installs the required μNFs as a PDP pipeline as per the requirements of the operator. Connection establishment success is indicated by transmitting an eBPF TCP connection established message.

4.3 Libraries and tools

In this section, we first introduce the Python–based Twisted framework and then discuss Google's Protocol Buffer. The section concludes with the applicability of these libraries for functional implementations within the IOL layer.

4.3.1 The Twisted framework

Twisted is an event–driven network programming framework written in Python and licensed under the MIT License [11]. This framework is used for writing asynchronous, event–driven networked programs in Python for both clients and servers. Twisted projects support TCP, User Datagram Protocol (UDP), Secure Socket Layer (SSL), Transport Layer Security (TLS) implementations and so on.

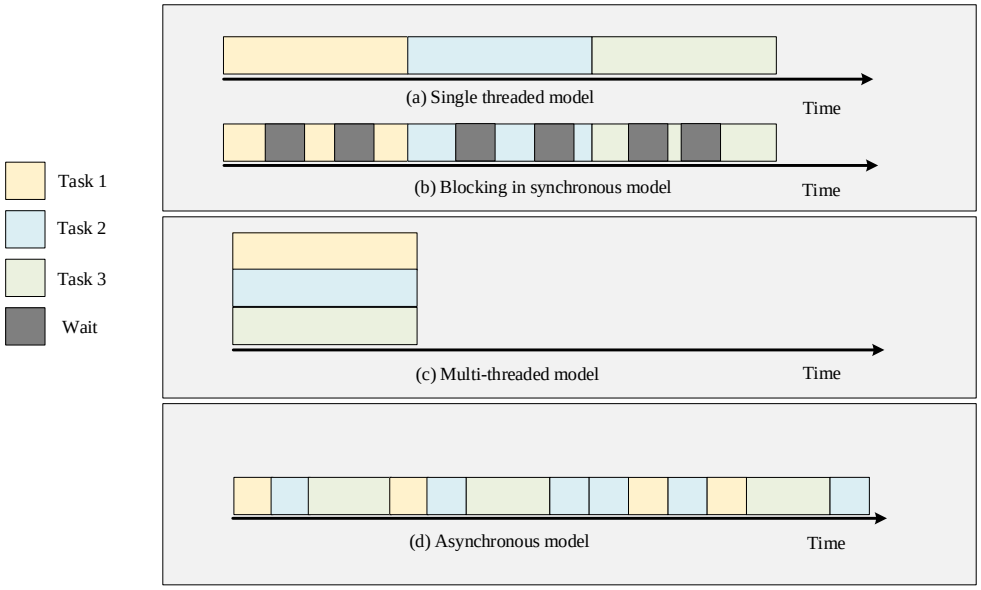

Figure 4.3: Programming paradigms for networking applications

Programming paradigms for networking applications: Tasks can be executed sequentially in a single–threaded program, concurrently on multiple processors in multi–threaded programs, or asynchronously following an event–driven mechanism, presented in Figure 4.3. An example of synchronous execution is an Input/Output (I/O) operation. In an I/O operation, synchronous request may result in blocking a task for some time, all other tasks may only be executed after the I/O is completed, as presented in Figure 4.3(b). Consequently, a single–threaded program may lead to delayed processing despite simplified reasoning. In contrast to this, multi–threaded programs execute tasks concurrently using separate threads executing on multiple processors, presented in Figure 4.3(c). In multi–threaded programs, tasks are executed

at the same time, each assigned to a thread, and, managed by the operating system, to avoid blocking and save time, presented in Figure 4.3(c). In an asynchronous implementation, the tasks are interleaved and supported by a single thread of control, presented in Figure 4.3(d).

Event-driven programming using Twisted: The Twisted framework offers an asynchronous *event–driven* networking platform. Hence, flow of the programs using the programming framework is determined by external events, characterized by event loops and callbacks to invoke actions when external events take place. It offers asynchronous behavior when an action is performed on completion of an event. This behavior enables programs to continue execution without the need for additional threads. An example of a TCP server is given in Listing 4.1, where Echo protocol simply sends back whatever it receives, the EchoFactory generates instances of Echo, and reactor listens on TCP port 8000.

```
%%%%% Example TCP Server %%%%%%%%%%%%%%

from twisted.internet import reactor, protocol

class Echo(protocol.Protocol):
    def datareceived(self, data):
        self.transport.write(data)

clas EchoFactory(protocol.Factory):
    def buildProtocol(self, addr):
        return Echo()

reactor.listenTCP(8000,EchoFactory())
reactor.run()
```

Listing 4.1: An examplar TCP server using Twisted

Twisted supports client–side operations. An example of web server Hypertext Transfer Protocol (HTTP) request is presented in Listing 4.2, where an HTTP agent is created to generate an HTTP request to "example.com" and prints the response details.

```
%%%%%%%%%% An example of Twisted client-side request %%%%%%

 from twisted.web.client impprt Agent
 from twisted.internet import reactor

 def handleResponse(response):
    print('Response version:', response version)
    print('Response code:', response code)
    print('Response phrase:', response.phrase)
    reactor.stop()

 agent = Agent(reactor)
 d = agent.request(b'GET, b'http://expample.com')
 d.addCallback(handleResponse)

 reactor.run()
```

Listing 4.2: An examplar of client-side application using Twisted

Twisted handles the asynchronous operations by using a reactor pattern which is an event loop that listens to events and dispatches them to event handlers.

The Twisted Reactor: The event loop is at the center of Twisted's operations. The concept of a reactor involves distributing events from multiple sources to their recipients within a single-threaded environment. In CPN, multiple switches will generate requests to the service

broker at the IOL. In this case, the Twisted reactor will respond to the requests from the underlying programmable switches at the PDP.

Twisted framework applicability: The Twisted framework is suitable for event–driven programming, such as networking applications, characterized by independent execution of several tasks, asynchronous sharing of mutable data between tasks, or encounter blocking while waiting for events. Twisted provides the essence of multi–threading supporting the concept of parallelism together with ease of reasoning offered by single–threaded programs. It is suitable for CPN to follow a synchronous event–based programming for communications between between the CPN PDP at the COMML and the CPN controller at the IOL.

4.3.2 The Protocol Buffer

The extensible mechanism for serializing structured data, is the Protocol Buffer, licensed by the BSD [12]. Protocol buffers generate native language bindings, similar to those generated by JavaScript Object Notation (JSON) or XML. However, compared to JSON/XML, the Protocol Buffers offer compactness via binary encoding leading to efficient encoding and parsing, faster processing speeds and schema evolution via integrating newer data structures without disrupting existing applications. It is noted that protocol buffers are composed as schema; where the data is separated from the context, thus offering light and compact message definitions.

The schema of the data is structured once, which is then used to generate codes using a compiler, such as, "protoc" or "clang" compiler, invoked at build time, creating source files. Source files of various programming languages can be generated, such as, Java, Python, C++, and so on. The data serialization process for the Protocol Buffer is presented in Figure 4.4.

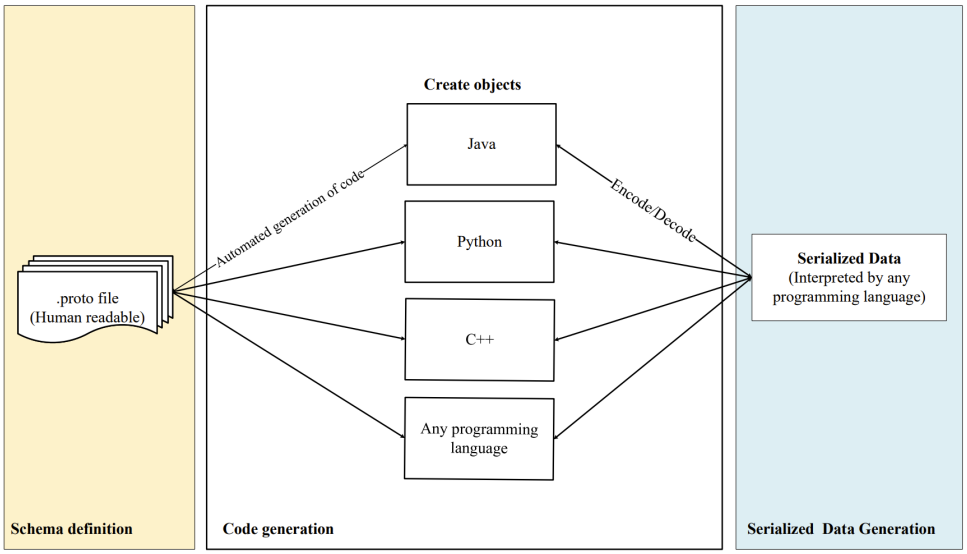

Figure 4.4: Data Serialization with Protocol Buffer

Definition: Owing to their language and platform independence, Protocol Buffers will be utilized for inter–server communications and storage data on disks. An example message entity, defined for the CPN at the PDP, is given in Listing 4.3. The *Hello* message, sent during the connection handshake that will be used to automatically install an eBPF μNF onto a CPN with the following elements "version" and DPID, using proto3 version of Google's Protocol Buffer.

```
1 %%%%%%%     Hello.proto    %%%%%%%%
2     syntax = "proto3";
3         message Hello {
```

```
4             uint32 version = 1;
5             uint64 dpid = 2;
6         }
```

Listing 4.3: Hello message entity schema definition

Attributes of CPN Protocol Buffer The main attributes of defining a protocol buffer are the message, field, and data types.

- **Message**: Protocol Buffers use messages as a fundamental unit for data encapsulation. A message is a unique information dataset comprising numerous tag-value pairs.
- **Field**: Messages are composed of fields. A field is unique because of its numeric tag and name–value pairing. In Listing 4.3, two fields, version and DPID are defined.
- **Data Types**: Protocol Buffers handle an array of basic data types, including booleans, integer values, string data type, and floating-point numbers, among others.
- **Encoding**: Protocol Buffers leverage a binary encoding method.
- **Schema**: Any data architecture within the Protocol Buffers is determined by a schema. It is this schema that is depicted in the ".proto" file.
- **Serialization and Deserialization**: Serialization refers to the conversion of structured data into binary data, while deserialization is the inverse process of turning binary data into structured data.

Applicability of Protocol Buffers: Protocol Buffers are well suited to applications related to communications and data storage, where it is needed to serialize record–like, structured typed data in a language and platform-independent extensible presentation. This is supported by fast parsing, and functional efficiency via auto–generated classes in various programming languages, in a compact storage structure.

Backward compatibility: Backward compatibility is standard for software products, in contrast, Protocol Buffers offers forward compatibility. For example, the old code is able to read new messages while ignoring newly added fields, provided some simple practices are followed when updating the old code for the ".proto" schema definitions. In this case, the default values will be considered for deleted fields, while the deleted repeated field will be empty. Consequently, the newly defined code with effectively interpret older messages. The performance efficiency, flexibility, language and platform–independence offered by Protocol Buffers, make it suitable candidate for structuring messages for the CPN implementations.

4.3.3 Twisted and Protocols Buffer for CPN implentations

Introducing programmability into CPN is supported by an interactive management and control entity, i.e., the service broker/orchestrator, at the IOL layer, programmable CPN, i.e., the PDP, at the COMML layer, and supporting mechanisms. In this context, the asynchronous event-based programmability offered by the Twisted framework and the schematic structuring of messages, which also offer serialization/deserialization using Protocol Buffers, make them suitable for implementation for persistent communications. Also, the asynchronous communication can be utilized to add flexibility of communication from the CPN to CPN controller.

- The Service broker, the core entity, at the IOL, will be developed utilizing the event–based Twisted framework, and will operate at the server–side.

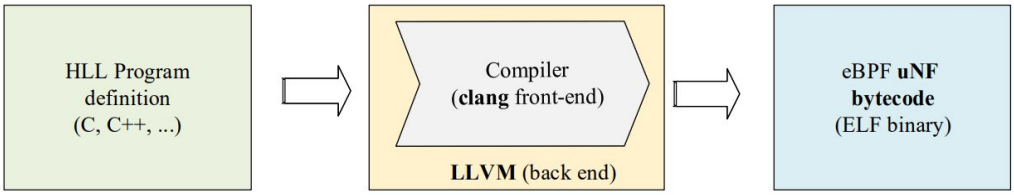

Figure 4.5: HLL (high level programming language) code to bytecode conversion paradigm adhered by the CPN implementation style on the IOL

- The eBPF Agent at the PDP works at the client side of the Twisted–framework. The eBPF agent at the COMML will be embedded into the programmable switch. It will receive the binaries, i.e. Executable and Linkable Format (ELF) files, and will processes them into a suitable format for the CPN platform and will allocate necessary tables. It is noted that the agent will communicate with the service broker at the IOL via the SBI.
- eBPF μNF loader at the PDP, i.e., the eBPF μNF ELF Loader, is responsible for allocating the eBPF tables required by the eBPF processing pipeline and finally transforming the eBPF bytecode it into a fast and usable format suitable for the device.
- eBPF μNF Service codes, i.e., the eBPF μNFs service codes are defined at the IOL layer. These service codes defined utilize the protocol buffers to create eBPF μNFs. These μNFs are pre–compiled ELF binaries.
- The CPN PDP: μNFs, such as, the Layer 2 learning switch, Layer 3 switch, packet size distributions, False Data Injection Identification (FDII), and so on, are defined at the IOL. These eBPF μNFs are pre–compiled ELF binaries, ready to be embedded/installed on the eBPF Processing pipeline at the PDP via the μNF loader at the COMML. Additionally, these μNFs can be installed as chains at the eBPF processing pipeline at the PDP.

Owing to the flexibility offered by the service implementation, the PDP is enabled to run any pipeline as defined by the network operator.

4.3.4 LLVM

The Low Level Virtual Machine (LLVM) is a toolkit and the backend for the Clang compiler. Therefore, implementation of the CPN will require the LLVM to convert the High level programming language (HLL) code for eBPF μNFs to bytecodes using the Clang library. The tools, libraries, and header files, including the assembler, disassembler, bitcode analyzer, and bitcode optimizer will be supported by the LLVM.

4.3.5 Clang

Clang is a compiler for c-based programs, written in HLL, that cooperates with LLVM. The eBPF μNF ELF binaries that will be installed, via the eBPF loader, on to the switches at the PDP processing pipeline will be pre-compiled utilizing the clang compiler.

Listing 4.4 shows examples of compilation of two μNF - *learning switch* and *adcollect* (anomaly detection collection). The first part of the command specifies the source C code (.c) and the last part the Clang output as an object file (.o). The object file can then be used by the IOL and installed into the COMML pipeline.

```
clang -O2 -target bpf -I ../includes -c learningswitch.c -o learningswitch.o
clang -O2 -target bpf -I ../includes -c adcollect.c -o adcollect.o
```

Listing 4.4: Exemplar compilation of two eBPF μNF for specific functionalities of learning switch and anomaly detection in the CPN with the Clang compiler

4.4 eBPF μNF

This section explains the structure of eBPF μNF. The purpose is to provide a material which can be used for development of custom eBPF μNF. This will be used by project partners in supporting the pilot functionality as well as in future CPN system updates which might require development of new CSL services and therefore corresponding μNFs. eBPF μNFs must have specific structure in order to comply with the eBPF requirements.

Code listing 4.5 show an exemplar of an eBPF μNF which performs asset discovery of Intelligent Electronic Device (IED) devices within a substation based on the GOOSE protocol number identification at line 23. Specific parts of the function are explained below.

```
#include <linux/if_ether.h>
#include <linux/ip.h>
#include <linux/icmp.h>
#include "ebpf_switch.h"

struct countentry
{
    int bytes;
    int packets;
};

struct bpf_map_def SEC("maps") assetdisc = {
    .type = BPF_MAP_TYPE_HASH,
    .key_size = 6,
    .value_size = sizeof(struct countentry),
    .max_entries = 256,
};

uint64_t prog(struct packet *pkt)
{
    struct countentry *item;

    if(pkt->eth.h_proto == 47240)
    {
        if (bpf_map_lookup_elem(&assetdisc, pkt->eth.h_source, &item) == -1)
      {
            struct countentry newitem = {
                .bytes = 0,
                .packets = 0,
            };
            bpf_map_update_elem(&assetdisc, pkt->eth.h_source, &newitem, 0);
            item = &newitem;
        }
    item->packets++;
    item->bytes += pkt->metadata.length;
    bpf_notify(0, pkt->eth.h_source, sizeof(pkt->eth.h_source));
    }
    return NEXT;
}
```

```
40 char _license[] SEC("license") = "GPL";
```

Listing 4.5: Exemplar of asset discovery function

4.4.1 eBPF μNF code structure

An eBPF program, or in case of the COCOON project, an eBPF μNF has four main code sections. These are:

- Imports - set libraries and files to be included in the program. This includes Linux Kernel header files which define implementation of network protocols such as Ethernet, IP, Internet Control Message Protocol (ICMP), and TCP.
- Data structures and maps definitions - allows definition of data structures with life scope of the program and persistent maps.
- The main code - defines all the eBPF program logic within a single function called "prog".
- License information - the last line of an eBPF code has to provide a license type which must be compatible with used kernel functions (otherwise the program is rejected).

4.4.2 eBPF μNF code snippets

This section describes parts of the code provided in Code listing 4.5 divided into section of the eBPF μNF code structure.

Imports

Code listing 4.6 shows how to import external files such as kernel header files which define common communication protocols such as Ethernet, IP and ICMP. Similarly, custom files and libraries can be imported as shown on the line 4, where an ebpf_switch header file is loaded.

```
1 #include <linux/if_ether.h>
2 #include <linux/ip.h>
3 #include <linux/icmp.h>
4 #include "ebpf_switch.h"
```

Listing 4.6: Exemplar of imported files

Data structures and maps definition

This code section allows definition of generic data structures and eBPF maps. Defined data structures can be inserted as values into the eBPF maps as shown in Code listing 4.7 which show an example of the *countentry* structure used as value in the *assetdisc* eBPF map. The *countentry* structure contains two variables for counting number of received bytes and packets. This structure can then be accessed in the map by the key in the Medium Access Control (MAC) address format. The map in this example has the hash format and maximum size of entries which can be stored of 256.

```
1 struct countentry
2 {
3     int bytes;
4     int packets;
5 };
6
7 struct bpf_map_def SEC("maps") assetdisc = {
```

```
    .type = BPF_MAP_TYPE_HASH,
    .key_size = 6, // MAC address
    .value_size = sizeof(struct countentry),
    .max_entries = 256
};
```

Listing 4.7: Exemplar of data structure and eBPF map definition

The main code

Code listing 4.8 shows an example of the main code function *prog* which defines the logic of the μNF. The code defines the *countentry* structure and then checks if an element with received destination MAC address exits in the map. If it does not, the *countentry* structure is initialized and the eBPF map is updated on the line 13 using the *bpf_map_update_elem* kernel helper function. Finally, the structure and the map are updated with increased number of packets and bytes received for this element. Line 18 show the *bpf_notify* function which sends an information message to the IOL for further processing. The command *NEXT* instructs the COMML pipeline execution component to continue in the eBPF processing pipeline.

```
1  uint64_t prog(struct packet *pkt)
2  {
3      struct countentry *item;
4
5      if(pkt->eth.h_proto == 47240)
6      {
7          if (bpf_map_lookup_elem(&assetdisc, pkt->eth.h_source, &item) == -1)
8        {
9              struct countentry newitem = {
10                 .bytes = 0,
11                 .packets = 0,
12             };
13             bpf_map_update_elem(&assetdisc, pkt->eth.h_source, &newitem, 0);
14             item = &newitem;
15         }
16     item->packets++;
17     item->bytes += pkt->metadata.length;
18     bpf_notify(0, pkt->eth.h_source, sizeof(pkt->eth.h_source));
19     }
20     return NEXT;
21 }
```

Listing 4.8: Exemplar of the main code function

License information

Finally, license information must be provided as shown in Code listing 4.9. The General Public License (GPL) license is recommended as some kernel function might require it and if the license is not provided, the μNF will be rejected.

```
1  char _license[] SEC("license") = "GPL";
```

Listing 4.9: Exemplar of program license

4.5 Northbound API (NBI)

The NBI will provide communication between the IOL and CSL which will be handled by service brokers in both layers. This will require inter-process communication as these two

components run continuously and interactively react on real-time events - either from CPNs, or from CSL services. For this reason, Representational State Transfer (REST) API is proposed at this stage. This can be conveniently integrated within the IOL listener and the service broker. While the concrete functionality of this interface will still depend on final CSL requirements, in general, there will be two groups of functions: general and service-specific.

The general functions will provide similar operations as in case of the SBI, but forward the information all the way from the CPN to the CSL. These will include:

- List, add and remove function(s)
- List, update and delete table(s)

Functions tailored for COCOON upper level applications underpinned by CSL services will be defined in future deliverables. An example of two functions for the AD service start and data retrieval is provided in Code sample 4.10. The first function instructs the IOL to start the AD service and the line 4 returns a HyperText Markup Language (HTML) reply. The second function is a request for IOL to get the monitor data, which are returned in the JSON format and includes list of connected devices, log and the asset discovery variable.

```
1  @app.get("/cocoon_start_ad")
2  def cocoon_start_ad():
3      //Do the Anomaly Detection start logic
4      return '<h2> AD service started </h2>'
5
6  @app.get("/cocoon_monitor_get_data")
7  def cocoon_monitor_get_data():
8      return json.dumps({"connected_devices" : list(connected_devices),
9          "log" : log,
10         "asset_discovery" : asset_discovery
11 })
```

Listing 4.10: Exemplar instantiation of two upper layer COCOON applications through the CPN REST-based northbound API (NBI)

5 CPN pilot use cases

This section describes CPN pilot use cases where CPN will be deployed and tested and an emulated scenario for the CPN development which is presented first. This will be used for functionality validation prior any pilot deployments and thus ensuring solid foundations for addressing requirements related to achieving the desired TRL of the COCOON solution.

5.1 Mininet emulation (SGSim)

In the first stages of the CPN development, the Smart Grid Simulator (SGSim) platform [13] will be used. This platform creates a smart grid topology with a digital primary substation, two digital secondary substations and realistic emulated communication including IEC104, GOOSE and SV protocols. SGSim uses the Mininet network emulator [14] and is distributed in form of a Virtual Machine (VM), which can be easily deployed on an average Personal Computer (PC). This enables efficient development and testing of the CPN functionality in near real-world conditions, corresponding to the TRL 5, without the need of specialized hardware and risk of a real network downtime.

The SGSim default emulated topology is shown in Figure 5.1 and bounded by the blue frame. The topology creates 9 software switches, which will be used for CPN implementation before the deployment on real devices. The large number of nodes will enable verification of various COMML pipelines, composed of different μNF as depicted by colored boxes under every node. For example, DPS HV shows installation of two μNF - AD and forwarding. This will also test parallel communication between the server and all the nodes, the SBI and the NBI, and functionality of the dashboard.

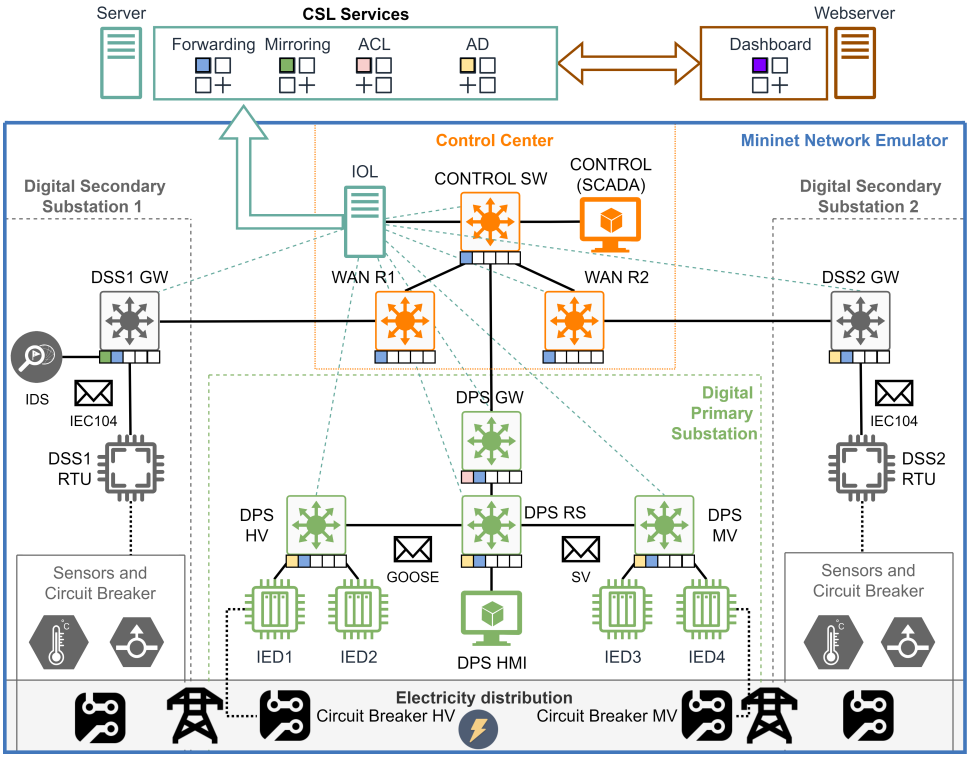

Figure 5.1: Smart Grid Simulator (SGSim) platform for CPN development testing

The SGSim topology and protocol communication patterns can be easily modified to represent realistic scenarios including the COCOON pilots. This will be utilized in final phases before the real device deployment to represent topology of the target COCOON pilot. Topologies of these pilots are described below.

5.2 Digital substation

An overview of digital substation topology is presented in Figure 5.2. The digital substation comprises a power system simulator utilizing Real-Time Digital Simulator (RTDS), physical IED, and workstations. All components are integrated into a Hardware in the Loop (HIL) co-simulation. The IEDs are fully IEC 61850 compliant, meaning, the relay has the capability for GOOSE messaging and uses SV for measurements. Using the received SV input, it calculates the fault condition and trip status, which are then communicated via GOOSE. The GOOSE communication represents critical substation communication, i.e., trip and block commands through switched Ethernet. As shown in Figure 5.2, the relay data links are connected to a network switch which also has a connection to the RTDS GTNET 2x card. The card is interfaced to the RTDS through an internal optical fibre connection.

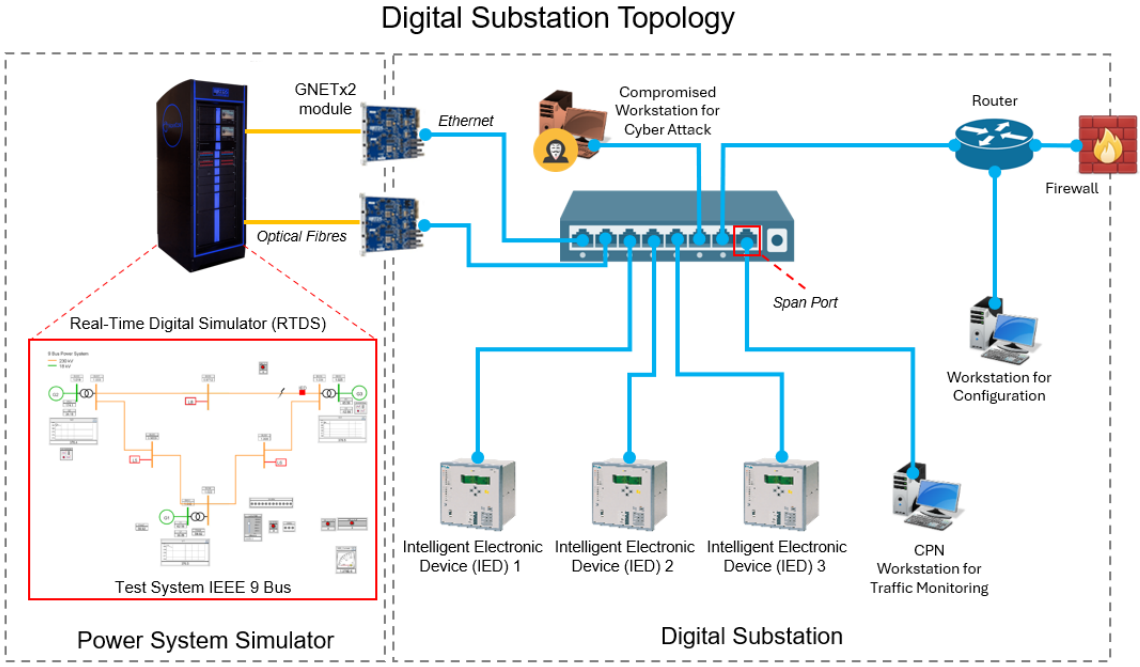

Figure 5.2: Digital substation topology

The digital substation contains three workstations. The first workstation is designated for the configuration of digital substations, i.e., for adjusting the settings of IEDs. The second workstation is specifically assigned as CPN to observe and analyze the flow of data within the digital substation's network. This is achievable because this CPN workstation is connected to a span port on the switch which mirrors all the traffic to the CPN. The CPN in this pilot will integrate CPN controller so all the layers of the CPN system architecture will run on a single device. CSL in this case will run only the anomaly detection service which is not resource intensive and the CPN can be therefore implemented on a relatively low-performance device such as a SBC. At last, the third workstation serves as a compromised device within the digital substation. This machine is used to orchestrate a cyber attack that specifically targets components in the digital substation.

5.3 Energy communities

A brief overview of the pilot configuration is presented in Figure 5.3. Specifically, the energy community consists of six PV plants that are connected to a Medium-Voltage (MV) EPES through Low-Voltage (LV)/MV step-up transformers. Details regarding the electrical configuration of the energy community are presented in deliverable D5.1 - *Secure Energy Communities Pilot Design, AS orchestration and infrastructure Configuration.*

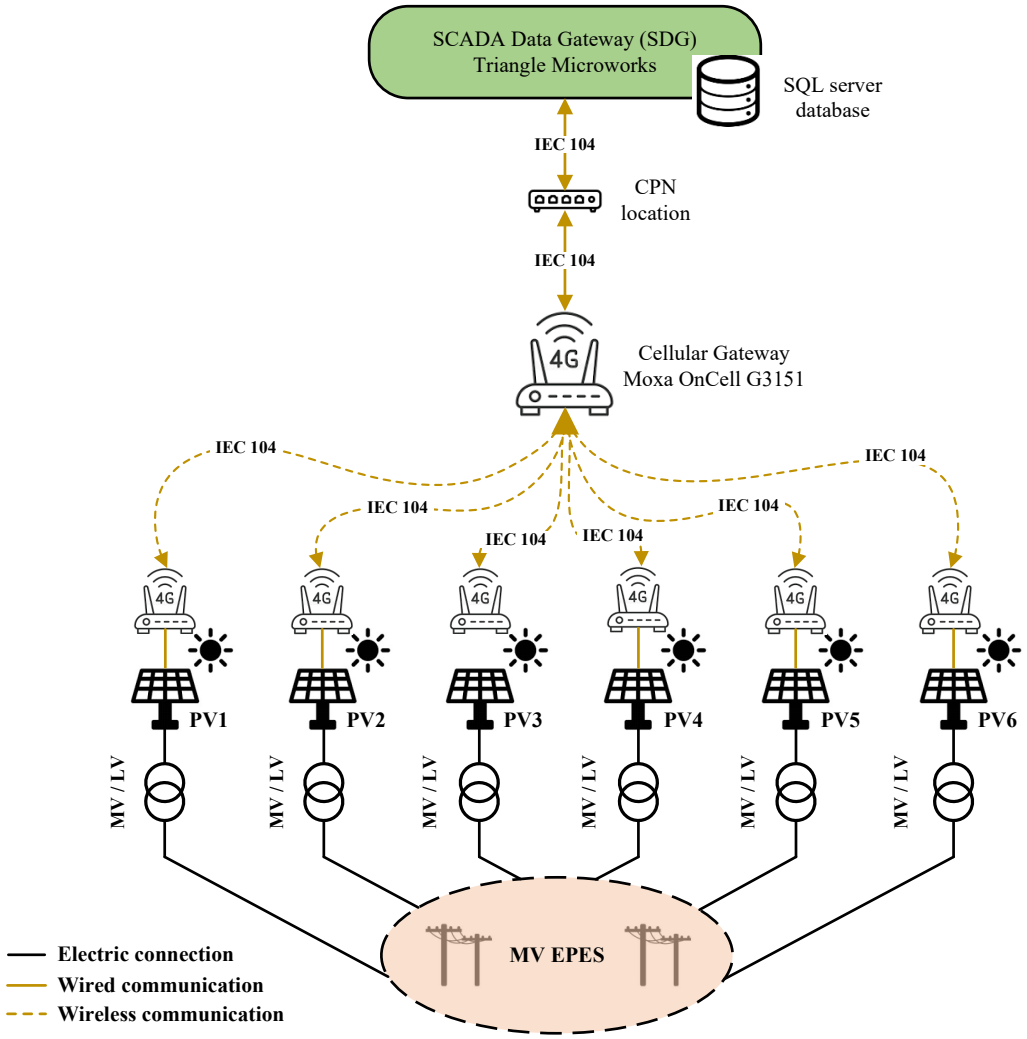

Figure 5.3: Energy Community pilot configuration.

Each PV plant is monitored and controlled through a smart logger. This device belongs to the energy community owner, i.e., I&K Electrical Engineering Systems (IKE) and can be used either for acquiring measurements or for sending operating set-points, e.g., active and reactive to each individual PV inverter of the plant. Note that the smart logger provides the possibility of monitoring and controlling the PV plant at an aggregated level, e.g., receiving and sending operating set-points at the point of interconnection with EPES.

A 4G router is physically attached to each smart logger using a wired connection. This router belongs to the HEDNO, and is used to provide access to the smart logger, thus enabling the remote monitoring and control of each PV plant of the energy community. Using a 4G cellular network, the 4G router of all PV plants communicate wirelessly with a cellular gateway (Moxa OnCell G3151) that is located at the premises of HEDNO. A SCADA Data Gateway

(SDG) is physically attached to the cellular gateway using a wired connection. The main scope of the SDG is to monitor and control in real-time the operation of each PV plant, and thus of the energy community. All the data exchanged between SDG and each PV plant are stored in a local SQL server database, as shown in Figure 5.3. It is worth mentioning that the IEC104 Standard is used as the main communication protocol. The CPN will be installed between the cellular gateway and SDG to monitor all the traffic exchanged between the HEDNO Supervisory Control and Data Acquisition system (SCADA) and the PV plants of the energy community. The CPN controller will be placed either on the CPN itself, or remotely on the SDG. The CPN in this use case will be assessed based on the performance of the FDII service aiming to recognise anomalous signals on setpoints. Due to the fact that the CPN is placed in-line and all the traffic must be handled by it, it will require usage of a device with a high throughput capability. Hence, a whitebox switch or a NUC is recommended for this pilot.

5.4 PV power plants

This pilot consists of a PV power plant called "Hoyas Grandes I" with a nominal active power of 5 MW at the Point of Interconnection (POI) which feeds power into the MV distribution system at 20 kV. This power plant is of the string PV inverter type and is located in Granada, Spain. The electrical layout of the plant is shown in Figure 5.4.

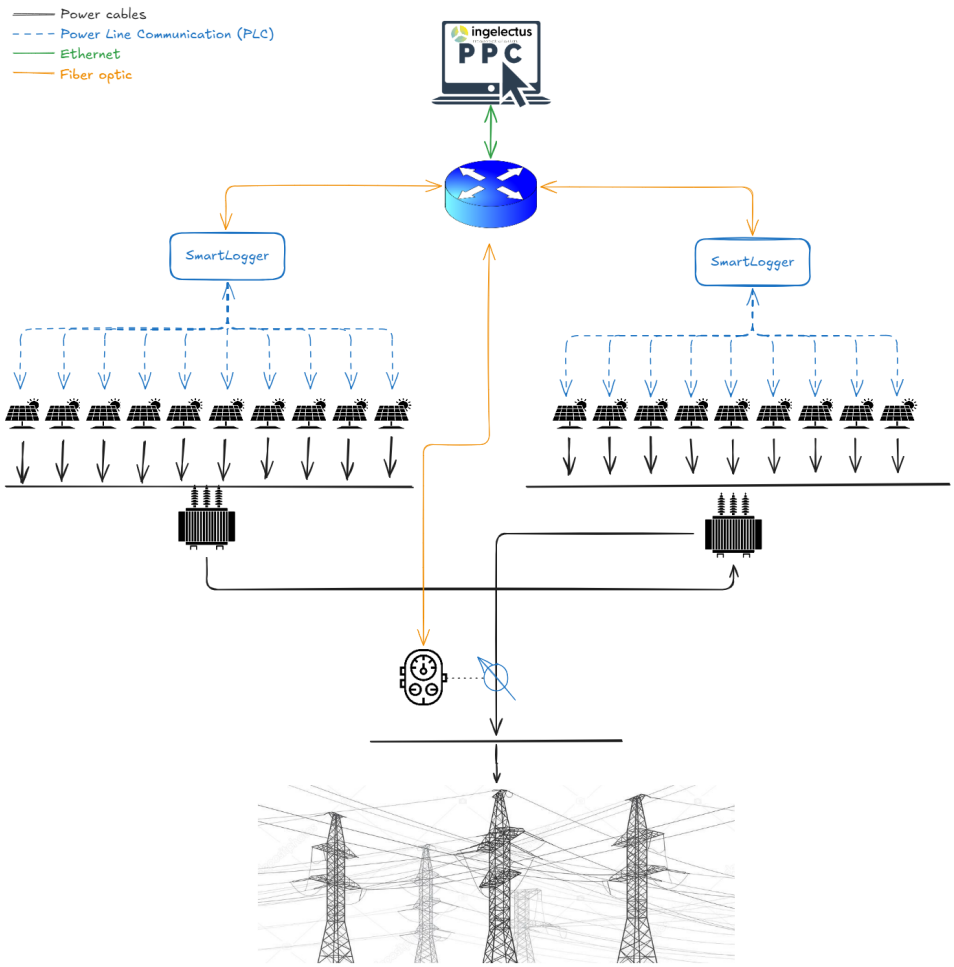

Figure 5.4: Electrical and communication layout of the pilot PV plant.

The plant is composed of 24 electrical nodes, 21 branches, and 19 PV inverters, with two

voltage levels interconnected by the corresponding transformers: medium voltage at 20 kV and low voltage at 0.8 kV. The MV collector system is composed of 2 feeders that evacuate power from the Secondary Substations (SS) to a disconnection center. Each feeder consists of several radial MV conductors of type RHZ1-20L 12/20KV 1X150 mm2. At the LV level, each PV inverter is connected to a SS by a LV conductor of type RV-Al 300 mm2.

There are two SS to increase the voltage from 0.80 kV (LV inverter level) to 20 kV (MV distribution level). Each transformer has a rated power of 3 MVA and a short circuit impedance of 6.25 %. There are 19 PV string inverters manufactured by Huawei, model SUN2000-330KTL-H1. The inverter has a nominal Alternating Current (AC) voltage of 800 V and a nominal power of 300 kVA at the plant design temperature of 40°C. There are 10 inverters connected to one SS and 9 inverters connected to the other SS.

Based on the Spanish grid connection code and the European Network Code on Requirements for Generators, this plant is considered a Type B power plant, which means that it must be able to perform the following functionalities at the POI: over-frequency response, reactive power control, power factor control, voltage (Q-V) control and reactive power capacity within the maximum and minimum active power dispatched. The plant level control is performed by a Power Plant Controller (PPC) from Ingelectus. The PPC is responsible for compliance with the grid code technical requirements at the POI by sending the necessary active and reactive power setpoints to the inverters.

In this pilot, a communication system has been installed for the monitoring and control of the photovoltaic plant, as described in the communication diagram of Figure 5.4. Through the defined communication architecture, the PPC is capable of sending setpoints to each photovoltaic inverter, collecting measurements from the POI, and complying with the requirements of the Spanish grid code related to the operation of the plant at that point.

To communicate with the inverters, a Huawei SmartLogger 3000 was installed at each SS, using Programmable Logic Controller (PLC) technology. In the first SS, the SmartLogger communicates with the 10 inverters connected to it, while in the second SS, the SmartLogger communicates with the 9 inverters connected to that center. The SmartLogger is also responsible for the conversion from fiber optics to PLC.

At the POI, there is a network analyzer for collecting measurements and sending them via Modbus TCP/IP over fiber optics to the plant switch. The ING PPC connects to the fiber optic network through the plant switch, using an Ethernet cable.

The CPN placement within this pilot is an item to be addressed in the subtask 8.1.4 *COCOON platform deployment* of WP8. For this reason, the placement is not finalized at this stage and not shown in the Figure 5.4. The expected location at this stage is between the router depicted in blue color and the ING PPC - on the green link. This represents a similar setup like in case of the energy communities pilot and the same device is recommended for the implementation. The CPN in this pilot will also support the FDII service.

5.5 Secure regional electricity data operations

The pilot demonstrator sets the stage for validating that the COCOON solution can be installed and interact effectively in a real-world operational environment, particularly in safeguarding critical infrastructure against cyber threats. For this purpose, we have created a replica of the SEleNe CC environment, one of the six European Regional Security Coordinators (RSC), which follows rigorous security standards concerning coordination processes, to facilitate realistic testing and validation of the COCOON solution.

The pilot environment consists of four basic components:

- Firewall
- Switch
- Storage
- Server

The firewall, which is the first layer of security, protects the pilot environment from external threats and is positioned between the pilot environment and the internet. Next, we have the Layer 2/3 switch, meaning it can handle both switching and routing functions. It is connected to the server and the storage, allowing high-speed communication between these two components, and is also connected to the firewall to allow data to flow between the pilot environment and external networks. Then we have the storage, which combines flash and traditional hard disk storage to provide a balance of performance and capacity. Finally, we have the server, which has Hypervisor-1 installed, to create and manage virtual machines that will host the CPN controller. The CPN controller located on the server will run all the services on the CSL and will provide the IOL functionality managing the connection to all the CPNs in other pilots. At this stage of the testing, CPNs from the previous pilots will have CSLs and IOLs moved to this server. Figure 5.5 illustrates the pilot architecture.

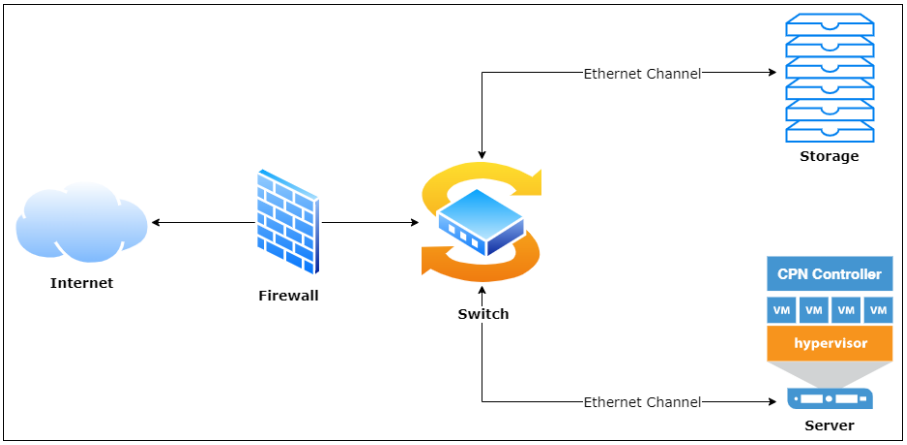

Figure 5.5: Secure regional electricity data operation pilot environment

6 Conclusions

This deliverable presented the COCOON system architecture which will be used for the CPN and the CPN controller development. Section 1 introduced the deliverable, its relations with the other tasks and deliverables, scope and the used methodology. Section 2 presented an overall view of the CPN system architecture and explained services decomposition with examples of how a service from the CSL relates to a μNF in the IOL and how they are then installed into a PDP in the COMML. Finally, this section also presented types of networking devices and stated how each type will be used in the project.

Section 3 described the COMML architecture in detail. The section introduced two components of the COMML - the agent and the switch which can be represented by a software switch, or DPDK for better performance. The section described all components within the COMML as well as functionality of the southbound API with its supported message type. This also included various low-level details of eBPF such as its instruction set, verifier, maps and helper functions. Information in this section is crucial for understanding possibilities and limitations of eBPF μNF.

Section 4 described the IOL architecture and its components. This included the NBI, used libraries, service broker with its elements and explanation on the eBPF μNF source codes and the pre-compiled object files. The main goal of this section was to give an overview of IOL functionality managed by the service broker which interacts with other components. This section also provided examples of μNFs and their source code structure which can then be used by the project partners when developing μNF for the CSL services. The NBI was also presented with suggested implementation technology to be REST based on the CSL needs. This area is still subject to changes and will be further elaborated in the following deliverables focused on the CSL and its components.

Finally, section 5 presented CPN pilot use cases for CPN deployment. This included deployment within an emulated environment of SGSim - a Mininet-based tool providing realistic EPES communication network composed of three digital substations and a control center. This platform provides a suitable environment for efficient CPN development and testing on the TRL 5 and allows easy topology modification according to the pilot needs. Section 5 also included four pilots, described with figurative representations, presenting the topologies and the CPN placement. This included TU Delft's digital substation, HEDNO's energy communities, Ingelectus's PV power plants and SELENE CC's secure regional electricity data operations.

In general, information entailed within this deliverable will act as the basis for the development of the entire CPN solution including its controller parts composed of the CSL and IOL as well as the CPN itself and its COMML. Finally, this deliverable will act as a guide for all the partners which will need to develop supporting μNF for their CSL services.

Bibliography

[1] The Pi Hut. *Raspberry Pi 5*. Online (accessed 2024-09-11): `https://thepihut.com/products/raspberry-pi-5`. 2024.

[2] Edgecore Networks Corporation. *300G CELL SITE ROUTER CSR320*. Online (accessed 2024-09-11): `https://www.edge-core.com/product/csr320/`. 2024.

[3] Open Networking Foundation. *OpenFlow Switch Specification Version 1.5.1 (Protocol version 0x06)*. Online (accessed 2024-08-20): `https://opennetworking.org/`. 2015.

[4] Internet Engineering Task Force (IETF), Request for Comments: 6241. *Network Configuration Protocol (NETCONF)*. Online (accessed 2024-07-18): `https://cisco.com/`. 2011.

[5] Y. Zheng. *Userspace eBPF Runtimes: Overview and Applications*. Online (accessed 2024-08-22): `https://eunomia.dev/blogs/userspace-ebpf/`. 2024.

[6] Big Switch Networks, Inc. *iovisor/ubpf: Userspace eBPF VM*. Online (accessed 2024-08-22): `https://github.com/iovisor/ubpf`. 2024.

[7] DPDK Project. *DPDK*. Online (accessed 2024-08-22): `https://www.dpdk.org/`. 2024.

[8] K. A. Simpson et al. "Galette: a Lightweight XDP Dataplane on your Raspberry Pi". In: *2023 IFIP Networking Conference (IFIP Networking)*. 2023, pp. 1–9.

[9] D. Thaler. *BPF Instruction Set Architecture (ISA)*. Online (accessed August 14). 2024.

[10] Linux kernel source code (GPLv2). *BPF Documentation*. Online (accessed August 18). 2024.

[11] A. Fettig. *Twisted network programming essentials*. O'Reilly Media, Inc., 2005.

[12] BSD. *Protocol Buffers*. Online (accessed July 26, 2024. 2001.

[13] F. Holik, S. Y. Yayilgan, and G. B. Olsborg. "Emulation of Digital Substations Communication for Cyber Security Awareness". In: *Electronics* 13.12 (2024). DOI: `10.3390/electronics13122318`.

[14] Mininet Project Contributors. *Mininet An Instant Virtual Network on your Laptop (or other PC)*. Online (accessed 2024-06-26): `https://mininet.org/`. 2022.