Abstract:

Network operators are often very cautious before deploying any novel networking service. This is done only if the new networking solution is fully monitored, secured and can provide rapid return on investments. By adopting the emerging Network Functions Virtualization (NFV) concept, network operators will be able to overcome this constraint by allowing them to deploy solutions at lower costs and risks. Indeed, NFV involves implementing network functions in software that can rely on virtualization techniques to run on standard server hardware, and that can then be deployed in, or moved to, various network location as required.

This document presents the deployment of the two DOCTOR NDN/NFV testbeds accessing Web content in both UTT and University of Lorraine/TELECOM Nancy. Details about the incremental deployment of the project outcomes are provided. It includes the basic deployment of docker as a container-based virtualization framework and the instantiation of several NDN nodes in a functional topology. Integrating security components like monitoring probes (MMT) for network monitoring and CyberCAPTOR for vulnerability assessment is also described. Details about NDNPerf tool for NDN performance evaluation are also presented.
TABLE OF CONTENTS

1 Introduction .............................................................................................................................. 5
2 The DOCTOR Testbed ............................................................................................................. 6
  2.1 UTT Testbed ......................................................................................................................... 6
  2.1.1 Hardware equipment ......................................................................................................... 6
  2.1.2 Servers installation and configuration ............................................................................... 7
  2.1.3 A first network topology ................................................................................................. 8
  2.2 LORIA Testbed ..................................................................................................................... 9
  2.2.1 Hardware equipment ....................................................................................................... 9
  2.2.2 Servers installation and configuration ............................................................................. 10
  2.3 Interconnection between testbeds ..................................................................................... 12
3 Integration of NDN and/or IP routers .................................................................................... 14
  3.1 Virtual switch: OVS ............................................................................................................ 14
  3.2 Routing VNFs ..................................................................................................................... 15
    3.2.1 IP Routing VNF ............................................................................................................. 15
    3.2.2 NDN Routing VNF ....................................................................................................... 15
4 Integration of the HTTP/NDN gateways ............................................................................... 17
  4.1 Functionality overview ....................................................................................................... 17
  4.2 Implementation details ....................................................................................................... 17
    4.2.1 Technical overview .......................................................................................................... 17
    4.2.2 Inside the ingress Gateway iGW ..................................................................................... 18
    4.2.3 Inside the egress Gateway eGW ................................................................................... 19
  4.3 Data collection and analysis ............................................................................................... 20
    4.3.1 Scraper .......................................................................................................................... 21
    4.3.2 Performance criteria ...................................................................................................... 21
    4.3.3 Results .......................................................................................................................... 21
5 Integration of the MMT based monitoring solution ............................................................... 26
  5.1 Functionality overview ....................................................................................................... 26
  5.2 Implementation details ....................................................................................................... 27
    5.2.1 NDN Plugin implementation .......................................................................................... 27
    5.2.2 IFA attack detection capability ....................................................................................... 27
    5.2.3 Centralized monitoring server ....................................................................................... 28
6 Integration of CyberCATOR ................................................................................................... 29
  6.1 Functionality overview ....................................................................................................... 29
    6.1.1 Attack graph generation ............................................................................................... 29
Deliverable D4.1: Description of the DOCTOR Testbed

6.1.2 Attack path extraction ................................................................. 30
6.1.3 Scoring .................................................................................. 30
6.1.4 Remediation ........................................................................... 30
6.1.5 Visualization .......................................................................... 30
6.2 Implementation details .................................................................. 31

7 NDNperf ....................................................................................... 33
7.1 Introduction ............................................................................... 33
7.2 NDNperf features ...................................................................... 33
7.3 Server-side performance evaluation of NDN .................................. 34
7.3.1 Experimental environment .................................................... 34
7.3.2 NDN performances according to the Data source ................... 34
7.3.3 NDN performances according to the signature configuration .... 35
7.3.4 Exploiting multi-threaded signature of NDN Data .................. 36

8 Conclusion .................................................................................... 37
9 References ................................................................................... 38
## TABLE OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Overview of the DOCTOR virtualized network infrastructure</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Physical OpenStack Testbed Architecture (OpenStack Juno)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>(a) Connection between containers using a VXLAN tunnel; (b) Connection between containers using switch concatenation</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Testbed NDN network</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>DOCTOR testbed at LORIA</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Example of script to link two Docker containers with an ad hoc link</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>UTT and LORIA testbeds interconnection</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Interface connection between OVS and Docker VNFs</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Concept of network slicing in DOCTOR</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>The NDN VNF building</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Ingress and egress HTTP/NDN gateways</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Overview of the Data flow in the gateway</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>NDN Name to notify a new HTTP request</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>NDN Name to retrieve a response</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>NDN Name to retrieve a request</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>(a) Frequency distribution of the percentage of successful web site retrieval (semi-log scale); (b) Frequency distribution of image errors (log-log scale)</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>Success percentage against number of parallel requests</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>95% confidence interval for TimeNGW, TimeGW1 and TimeGW2.</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>Histograms of the number of successfully packets as a function of time (a) without GW; (b) first round by GW</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>Histograms of the number of successful packets as a function of time. (a) second round by GW, Timer=0; (b) second round by GW, Timer=5sec</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>(a) TimeGW1 as function of TimeNGW (log-scale) ; (b) TimeGW2 as function of TimeGW1 (log-scale)</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>Functional Monitoring Architecture</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>MMT architecture deployment for SDN</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>MMT-Operation adapted to NDN stack</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>Simple attack graph</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>&quot;Attack path&quot; tab</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>CyberCAPTOR pipeline</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>Experimental environment</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>Local cache throughout</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>Distant cache throughput</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>Throughput with DigestSha256</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>Throughput with Sha256WithRsa</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>Throughput with Sha256WithRsa (multi-thread)</td>
<td></td>
</tr>
</tbody>
</table>

Deliverable D4.1: Description of the DOCTOR Testbed
1 Introduction

This deliverable D4.1 is the first result of the subtask T4.1 that aims at deploying and operating the NDN/NFV testbed providing access to HTTP based services. The objectives of this subtask are two-fold. The first objective is the quantitative evaluation of a NDN/NFV deployment scenario in order to provide concrete feedback to potential stakeholders. This will be achieved through QoS measurements and QoE evaluations (collected by the regular probing of end-users). The second objective is the use of the testbed to reproduce selected observable attacks in situ and detecting them. This data will provide input for refining the models and detection algorithms designed in Tasks 2 and 3.

The DOCTOR testbed described in this document hosts the different modules conceived in the technical tasks. It relies on the DOCTOR architecture (cf. Figure 1) we have designed in task 1 and detailed in deliverable D1.2. Notice that the building of this testbed is done incrementally since we integrate in it, the prototype versions of the implemented modules and perform their integration following an agile development.

The document is organized as following: The section 2 presents the deployment of the two DOCTOR NDN/NFV testbeds accessing Web content in both UTT and University of Lorraine/TELECOM Nancy. Details about the incremental deployment of the project outcomes are provided. It includes the basic deployment of docker as a container-based virtualization framework and the instantiation of several NDN nodes in a functional topology. It also discusses the IP and NDN routing VNFs (section 3). Integration of the HTTP/NDN gateways in both testbed is presented in section 4. Sections 5 and 6 details the integration of security components like monitoring probes (MMT) for network monitoring and CyberCAPTOR for vulnerability assessment is also described. Finally, the implementation of one of the contributions of DOCTOR project i.e. NDNPerf tool for NDN network performance assessment is presented in the section 7.

Figure 1. Overview of the DOCTOR virtualized network infrastructure
2 The DOCTOR Testbed

One of the main outcomes of the DOCTOR project is to build a real testbed offering an experimental environment to develop and test the different VNF (Virtual Network Function) envisioned in the project (routing, monitoring, security, etc.), with the goal to offer an end-to-end NDN connectivity over a virtualized infrastructure following the latest NFV standards. To this end, we ordered and installed several servers to support our future experiments and we carefully configured their virtualization and network layers. The testbed is bi-located in Nancy and Troyes and will ultimately involve external users to generate real traffic through the infrastructure. The testbed is composed of a set of servers on each site, interconnected by a VPN tunnel. Following the NFV philosophy, the infrastructure is composed of standard x86 servers.

The following section details the hardware and software infrastructure we set up and that constitutes the backbone of the DOCTOR testbed.

2.1 UTT Testbed

The UTT testbed is composed of several servers that allow, thanks to virtualization technologies, to easily reproduce realistic networks, thus facilitating the development, tests and validation of subsequent algorithms.

Two separated NDN networks are deployed in the testbed. The NDN nodes of the first network are hosted in virtual machines (VMs) that are created and managed using OpenStack platform, while the nodes of the second network are directly hosted on a single physical server (Baremetal). In both cases, each NDN node is hosted in a docker container as respect to the requirements identified in the first task of the Doctor project. Consequently, the UTT testbed is designed to serve two scenarios. The first one is based on an OpenStack and Docker implementation, will enable the implementation and test of several virtual topologies. Here each Virtual Machine will stand for a physical server taking part of a Point of Presence of an ISP. As for the second scenario, since it solely uses one physical server directly hosting all Docker containers, it will be devoted to benchmark issues. One can note that in both scenarios, the Docker containers are connected by Open vSwitch (OVS) bridges.

2.1.1 Hardware equipment

OpenStack-virtualization

For a reliable virtual implementation and to support scalability needs (for instance with tests open to real users), physical servers must have substantial computational capabilities. The UTT testbed servers were chosen to meet these needs. The OpenStack architecture is composed of 10 physical servers (1 Controller, 1 Network, 4 computers and 4 Storages). Table 1 shows the hardware configurations.

<table>
<thead>
<tr>
<th>OpenStack hardware configurations</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>CPU</th>
<th>Memory</th>
<th>Disk</th>
<th>Network cards</th>
<th>OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dell PowerEdge R720 (Controller, Network, Computes)</td>
<td>2x Intel Xeon E5-2650v2 (2,6GHz, 8C, Cache 20Mo)</td>
<td>128 Go</td>
<td>Intel X520, double port 10Gb (SFP+), double port 1GbE</td>
<td>Ubuntu 14.04 LTS</td>
</tr>
<tr>
<td>Dell PowerEdge R720xd (Swift, Cinder)</td>
<td>2x Intel Xeon E5-2650v2 (2,6GHz, 8C, Cache 20Mo)</td>
<td>128 Go</td>
<td>Intel X520, double port 10Gb (SFP+), double port 1GbE</td>
<td>Ubuntu 14.04 LTS</td>
</tr>
</tbody>
</table>
Baremetal virtualization

In this scenario, all containers are hosted on single physical server. The characteristics of the physical server are given in Table 2.

<table>
<thead>
<tr>
<th>CPU</th>
<th>Memory</th>
<th>Disk</th>
<th>OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>R420</td>
<td>2x Intel Xeon E5-2430 (2.50GHz, 6C, Cache 15Mo)</td>
<td>64 Go</td>
<td>4To Ubuntu 14.04 LTS</td>
</tr>
</tbody>
</table>

### 2.1.2 Servers installation and configuration

As described in D1.1 of the doctor project, OpenStack is defined as cloud management system including multiple components. It offers multiple tools for managing virtual environments. It proposes a modular architecture allowing to manage heterogeneous cloud infrastructures. The OpenStack modules communicate using an AMQP protocol. This standardized protocol creates communication channels between the OpenStack modules (or services) and abstracts the underlying layers such as the OS or the network on which pass messages.

The physical architecture of our OpenStack platform deployed in the OpenStack virtualization scenario is illustrated in Figure 2. This figure shows the different physical servers and the network resources allocated to OpenStack to work properly. This architecture is based on the Juno release of OpenStack. We used four computes nodes running multiple VMs connected by OpenStack Network services and four storage nodes to record experimental results.

In the OpenStack-Virtualization scenario, containers hosted in VMs need to be connected. We have implemented two solutions to do this, each related to an ISP deployment scenario that can be further considered in the following of the project. The first solution consists in emulating what an ISP could implement to emulate a local network composed of different servers hosting docker containers but which are actually separated by the Internet. To that aim, we have setup a VxLan tunnel between the containers (Figure 3 a). The script below is used to create an OVS bridge (OVS1) in VM1
Deliverable D4.1: Description of the DOCTOR Testbed

and connect it to the hosted Docker containers (Container-1, 2). Next, we connect OVS1 to the created VXLAN Tunnel (vxlan10). The same process is done on VM2 (OVS2 bridge, Container-3, 4).

```
#!/bin/bash
IP_HOST="172.25.1.1"
IP_CONTAINER="172.25.1.11"
CONTAINER="Container1"
sudo ovs-vsctl del-br OVS1
sudo ip link del vxlan10
sudo ovs-vsctl add-br OVS1
sudo ip link add vxlan10 type vxlan id 10 group 239.0.0.10 ttl 4 dev eth0
sudo ovs-vsctl add-port OVS1 vxlan10
sudo ifconfig vxlan10 up
```

By contrast, to rather emulate a local interconnection of containers hosted in the same network, it is also possible to concatenate virtual switches and Linux bridges as shown in Figure 3 b. This last solution needs however to open some security groups in OpenStack. Each VM may have many network interfaces. We attach one of them (eth1) to the virtual switch OVS1. This interface is now considered as a part of OVS1 and cannot be used anymore by VM1 (the other interfaces of VM1 are left for other connections, e.g. Internet or SSH). As a result, OVS1 is now connected to interface eth0 of Compute1, as well as the physical switch. This process is done by executing the following commands on VM1 and VM2:

```
ovs-vsctl add-br OVS1
ovs-vsctl add-port OVS1 eth1
ifconfig eth1 0.0.0.0
ifconfig OVS1 172.25.1.1
```

![Figure 3. (a) Connection between containers using a VXLAN tunnel; (b) Connection between containers using switch concatenation.](image)

2.1.3 A first network topology

We present here the network topology used for this study, whose objective is to enable us to collect early data while conducting the first performance evaluation tests of the project. This topology is hosted in the server R420 shown in Table 2. Physical host configuration, where the virtualization layer is only docker (scenario Baremetal virtualization defined previously).
The NDN network topology is composed of two HTTP/NDN gateways (GW) and two NDN routers. GWs and NDN routers are hosted in dedicated Docker containers. The Gateways forward NDN traffic over the NDN network and translate it to classical HTTP requests/responses before forwarding it to the client or to external Internet servers as shown in Figure 4. The client is configured to accept and send HTTP traffic using the Ingress-Gateway (iGW) node as proxy. The communication between the containers are ensured through the use of an Open vSwitch (OVS) who acts as a networking infrastructure-layer.

2.2 LORIA Testbed

2.2.1 Hardware equipment

The servers, on the LORIA site, are two R730 servers with the following configuration:

- CPU: Xeon E5-2630 v3 (8c@2.4GHz)
- RAM: 64GB DDR4@2133MHz
- Storage: 2x400Go SAS SSD + 4To SATA HDD
- Network: Intel X540 2x10Gbps + Intel i350 2x1Gbps

These two servers are fully dedicated to the DOCTOR project and have been designed with the purpose to offer realistic performances regarding the specific application envisioned in DOCTOR. More precisely, our goal is to deliver over NDN around 1GB/s of cached content or 50MB/s of new content to the final users of our testbed. This subsection presents the particular sizing we made for the servers hosted by the LORIA.

2.2.1.1 Processor sizing

Our performance evaluation of NFD, run on a modern processor, shows that it can deliver \(~1.35\text{Mo/s/Ghz}\) of new NDN data (a 3.7Ghz processor delivers \(~5\text{Mo/s}\) of data). This high cost in mainly due to the fact that new NDN data need to be encrypted before being sent, causing a high overhead. Consequently, a throughput of 50Mo/s would need approximately 37Ghz of computation power which can be achieved with some modern Intel CPUs with 14 cores or 16 cores. However, they are expensive and a dual-CPU (2*8 cores) architecture exhibits the same range of performance for cheaper. We finally chose two Xeon E5-2630v3 (8 cores at 2.4GHz) to empower our two servers.

2.2.1.2 RAM sizing

Our architecture will consume RAM for two main tasks. First, for the deployment of our network functions in the virtualized environment but they should not consume much RAM if well designed. Second, NFD relies on the available RAM to cache the data what enhance NDN performances. We chose a regular sizing of 2GB per core for the RAM, leading to 64GB per server.
Deliverable D4.1: Description of the DOCTOR Testbed

2.2.1.3 Network interfaces sizing

Based on our preliminary measurements, when data are cached (best case scenario), NFD can deliver over 280MB/s of data which quickly exceeds the capacity of a 1GB interface. Therefore, we chose to add two Intel X540 10 GB/s network cards: one for the inter-servers communications, and another for the outside.

2.2.1.4 Storage sizing

To extend our caching capacity, we chose two 400GB SSD configured in RAID0. This setup allows the server to deliver cached content at a high speed close to the limit of the network interface (10GB/s). Moreover, SSD can handle more operations per second than regular drives (between \(10^4\) and \(10^7\) vs \(10^2\) for classic HDD) and deal better with random accesses, which make it a better solution for intensive caching.

2.2.1.5 Evaluation

Our evaluation shows that the infrastructure can deliver on average 49MB/s of new NDN data (including a RSA-2048 encryption process) which is perfectly in line with our sizing choices. To evaluate the maximum throughput achieved by the infrastructure when cached content must be delivered, we reproduced the behaviour of a search of random NDN Data packets with 16 threads randomly reading 8kb on the SSD drives. The achieved throughput is of 875MB/s, which is ~70% of the 10Gb/s link capacity. This validates the sizing of the servers for our project objectives.

2.2.2 Servers installation and configuration

![Diagram of DOCTOR testbed at LORIA]

Figure 5. DOCTOR testbed at LORIA

The network topology of the LORIA part of the DOCTOR is defined as follow. We set up a dedicated DMZ for the needs of the project with a dedicated /27 sub-network. IP addresses in the DMZ are
reachable from outside the laboratory for a given set of ports (80, 443, 1194, 6363, 8080, 9388, 10000-10020) for both TCP and UDP.

The two servers run Ubuntu server 16.04 LTS with Docker engine 1.12. The early versions of Docker (<1.09), did not make it possible to create and configure networks thanks to the Docker API, so the only way to make containers communicate through different networks than the default one (docker0) was to create ad hoc links (as illustrated by the script in Figure 6). The main limitation of this method is that the links are not persistent, so that when a container stops, it loses all the established network links. But in counterpart, network becomes fully configurable by chaining Docker containers and specializing some containers into logical network components.

```
#!/bin/bash
if [[ $(/usr/bin/id -u) -ne 0 ]]; then
  echo "please run this as root"
  exit
fi
if [ "$#" -ge 5 ]
then
  if [ "$3" -gt 16 ]
  then
    # recuperation des pid des containers
    pid1=$(docker inspect -f '{{.State.Pid}}' "$1")
    pid2=$(docker inspect -f '{{.State.Pid}}' "$2")

    if [ ! -d /var/run/netns ]
    then
      mkdir /var/run/netns
    fi

    # creation des namespaces
    ln -s="/proc/$pid1/ns/net" /var/run/netns/$pid1 &> /dev/null
    ln -s="/proc/$pid2/ns/net" /var/run/netns/$pid2 &> /dev/null

    # calcul du sous reseau
    if [ "$3" -gt 30 ]
    then
      c=30
    else
      c=$3
    fi
    z=4
    for (( i=($30 - $c); i > 0; i-- ))
    do
      z=$(( $z * 2 ))
    done
    m=$z
    z=$(( ($z * $4 ) )
    m=$(( $m + $z - 1 ))
    y=$(( ($z % 256 ) ))
    x=$(( ($z % 256 ) ))
    my=$(( $m / 256 ))
    mx=$(( $m % 256 ))
    echo "in subnet 172.17.$y.$x/$c:"

    # definition du lien
    if [ "$#" -ge 6 ]
    then
      B=$6
    else
      B=$5
    fi

    ip link add "A$c-$4-$5-$B" type veth peer name "B$c-$4-$5-$B"

    # definition des points
    ax=$(( ($x + $5 ) ))
    ay=$y
    if [ "$ax" -ge 256 ]
    then
      ay=$(( ($ay + $ax / 256 ) )
    ax=$(( ($ax % 256 ) )
```
Deliverable D4.1: Description of the DOCTOR Testbed

Recently, Docker added a network stack to its core-engine. Now, specific sub-networks can be created and containers can be easily added to these sub-networks thanks to the Docker API. The network interfaces created with this method are persistent because Docker keeps track of the containers that are attached to the sub-networks. The next version of the DOCTOR testbed will use the new Docker API to build the virtual network.

2.3 Interconnection between testbeds

The DOCTOR testbed is bi-localized between UTT and LORIA laboratories and host software components from all the different partners, making it a truly common testbed. The UTT and LORIA testbeds are connected through a VPN tunnel as shown in Figure 7. This tunnel is created using the OpenVPN tool. The UTT iGW can request NDN content through this tunnel. It is necessary to create the NFD route between the VPN client on the UTT side and the VPN server on the LORIA side. Using eGW1 as the prefix, the route is created as follows:

nfdc register /eGW1 tcp://192.168.128.1, where 192.168.128.1 is the private VPN address.

Figure 6. Example of script to link two Docker containers with an ad hoc link
Deliverable D4.1: Description of the DOCTOR Testbed

Figure 7. UTT and LORIA testbeds interconnection

The configuration of the OpenVPN client is given in Table 2.3.1.

<table>
<thead>
<tr>
<th>Remote</th>
<th>Protocol</th>
<th>Port to listen</th>
<th>Routed IP tunnel</th>
<th>SSL/TLS parms</th>
<th>Private IP address</th>
</tr>
</thead>
<tbody>
<tr>
<td>152.81.47.226</td>
<td>tcp</td>
<td>1194</td>
<td>Dev tun</td>
<td>ca ca.crt cert utt.crt key utt.key</td>
<td>192.168.128.10/24</td>
</tr>
</tbody>
</table>

Table 2.3.1: OpenVPN client configuration
3 Integration of NDN and IP routers

Based on the architecture presented in Figure 1, we provide details in this section about the virtual switch OVS used in the DOCTOR testbed. We also present the IP and NDN routing VNFs.

3.1 Virtual switch: OVS

The virtualization technology we selected is Docker. We do not explain it in this deliverable, but the reader can know more about Docker and our choice in D1.2.

Regarding the virtual switch, we decided to use OpenVSwitch (OVS), an open source virtual switch, largely used in the virtualization area. We decided to use this switch, rather than the native Docker switch, since it allows more programmability, more functions. Amongst many, one very useful for us is the ability to select the IP addresses for the components and the IP addresses range for the network. Since we want to deploy network slices, we aimed to have different IP ranges as different ISP can have. Furthermore, it is more realistic to an operational networking configuration.

For using OVS, we deactivate the network interface of the physical machine and connect it to the virtual network interface of the virtual switch OVS. We then activate the OVS network interface, which will act as the machine interface. This virtual switch can then be seen by other entities in the network.

We then have to connect our VNFs to this virtual switch. Since the VNFS are docker components, which do not use the docker bridge, we need to connect those components to the OVS switch. For this, we use a script which enables to connect the VNFS components network interfaces to the switch via virtual interfaces, to affect the VNFS IP and MAC addresses in the range we want and make all necessary additional configuration steps. The Figure 8 show this.

![Figure 8. Interface connection between OVS and Docker VNFs](image)

To route data packets from OVS to the appropriate VNF, we configure OVS route packets according to the `dl_type` field of the Ethernet header.
If \( dl\_type \) is 0x800, then it is an IP packet, then forward it to the IP router VNF.
If \( dl\_type \) is 0x800 and \( ip\_address \) is the IP address of the HTTP/NDN gateway, then forward it to the HTTP/NDN Gateway VNF.
If \( dl\_type \) is 0x8624 , then it is an NDN packet, then forward it to the NDN router VNF.

3.2 Routing VNFs

In the DOCTOR project, we aim to set up different virtualized networks running on the same physical network. This means that we have physical nodes, which will host different virtualized component, the so-called VNF (Virtual Network Function), interconnected to each other, via physical links, on which will be set up different network slices.

The following figure presents this concept of network slicing, with 3 network slices (e.g.; the IP network slice in blue, the NDN network slice in Orange and any other network slice in grey).

![Concept of network slicing in DOCTOR](image)

In the next two sub-sections, we present how we build our two VNFs: IP and NDN components.

3.2.1 IP Routing VNF

For this first demonstrator, we simply dockerized a light linux image, enabling the routing facilities. In this Docker image, we just installed the routing module and configured it as necessary to be able to forward IP packets from one interface to another one. We also installed the module that enables to configure the routing tables so as to be able to configure it dynamically.

This IP Routing VNF docker image is then very simple to build. Using the Quagga open source router is an option we envisioned, but in this first step, we wanted something easy to implement. Furthermore, having the SDN controller to configure the IP routers, we do not really need the routing protocols to discover the topology and adapt it to changes.

For the configuration, we dynamically add the routing by specifying the next hop (IP address of the next IP router in the network slice). This is as it is in current IP configuration.

3.2.2 NDN Routing VNF

For the NDN routing VNF, we decided to base our VNF on the NFD open source software, which is a well-known NDN implementation and largely used by the research community.
For NFD, we need a linux system. Our docker image for the NDN VNF then includes a light linux system and the NFD software. We also create a specific file to be able to start the NFD daemon, when the docker component is instantiated. A similar file for the configuration is also done. These files are included in the dockerized VNF and made executable remotely, so that to be able to remotely deploy, launch, start and configure the NDN VNF. The needed modules can then be schematized as in Figure 10.

![Figure 10. The NDN VNF building](image)

In this VNF, we should note that depending on the configuration, the NFD daemon can be configured to route NDN packets directly over Ethernet, or over IP. In the first case, in the configuration phase, we just configure the MAC address of the network interface of the NDN VNF. And for the routing, we configure the FIB of NFD to route packet toward a given interface (named of the interface). In the second case, we also configure an IP address for the network interfaces and configure the NFD daemon to route packet toward the node having the specified IP address.
4 Integration of the HTTP/NDN gateways

4.1 Functionality overview

In our work published in [3], we defined two kinds of gateways: 1) one ingress gateway, aiming at converting HTTP users’ requests into NDN interest messages and converting NDN Data messages into HTTP replies, sent to the end-users. 2) One egress gateway, the counterpart of the first one, aiming at converting NDN interest messages into HTTP requests towards public web site and converting HTTP replies into NDN Data messages.

The ingress gateway (iGW) is then the one closer to the end-user, while the egress gateway (eGW) is the one closest to the web site. The following picture presents those 2 gateways integrated in the network, on the path between the end-users and the web server.

![Figure 11. Ingress and egress HTTP/NDN gateways](image)

4.2 Implementation details

4.2.1 Technical overview

Each part of the gateway (i.e. iGW and eGW) is composed of 3 sub-components:
- A HTTP client-side or a HTTP server-side for respectively iGW or eGW
- a NDN client-side
- a NDN server-side

The gateways are written in C++11. The NDN part (client-side and server-side) is based on the ndn-cxx library\(^1\) (v0.4.1), while the HTTP side uses raw TCP sockets implemented thanks to the boost.asio library\(^2\) (v1.55). As illustrated in Figure 12, performance-critical components are highly multi-threaded. We use several thread pools implemented by using boost library.

---

\(^1\) [http://named-data.net/doc/ndn-cxx/current/](http://named-data.net/doc/ndn-cxx/current/)

In order to communicate with each other, these components use queues that carry references to objects named “Query” that contains all the useful information needed to fulfil a request. A “Query” object is composed of the following elements:

- a Status in order to keep track of the processing steps of the request (ex: timed_out, is_completed, etc.)
- Method, URL and header hash of the request
- The raw content of the request and response
- Current and maximum segment number for both request and response
- A map of Data packets (request for iGW or response for eGW)
- A last access timestamp for removal purpose

In the current preliminary version of the gateway, the two parts of the gateway can only speak to each other but in a near future, we will add the possibility for any NDN client to communicate with eGW and iGW to retrieve web content.

4.2.2 Inside the ingress Gateway iGW

The HTTP server component is built over a bare TCP socket server because we want to access to the raw requests of the clients. The gateway manipulates raw requests because it is not its job to handle the full analysis of the HTTP protocol and it is easier to handle than complex objects returned by high level http libraries.

When the HTTP server component receives a request, a new thread is created to handle the request. After having received the header part of the request, the HTTP server component can modify the request header if needed (for example, changing the connection from keep-alive to close) in order to simplify the process. Then it computes a sha1 of the header, creates a Query object with this information and sends a reference of this object to the two other components. If the request contains additional data in its body, the component appends them as soon as the data are available. In this way, the gateway does not block the traffic until the full reception of the request. After receiving the request, the HTTP server component waits for part of the response and sends them as soon as possible (it does not wait the full response).
Deliverable D4.1: Description of the DOCTOR Testbed

The NDN client component is built with the NDN library and uses only one thread to manage this part because we show in section 7.3.2 that a single thread is enough to handle the client-side of NDN communications. When the NDN client component receives a new Query reference, it starts by sending a NDN Interest packet to eGW in order to notify a new user request. Figure 13 shows how the name is built in this case. The NDN client component appends to the eGW name a special name component “req” in order to allow eGW to know it is a notification of a new request. Then the NDN client component adds the prefix of the iGW (the one used to register the route) followed by the request method, the request URL and the SHA1 of the header. We use a SHA1 hash to differentiate some requests for a given content. Indeed, when two clients ask for the same content but have different cookies (for example, for authentication purpose), they may receive different contents. Without a hash computed on the full header, the second may receive the cached response of the first and this can create some privacy problems. Unfortunately, this also makes requests almost unique, each user having a different header when requesting a same web content. This results in a bad caching efficiency because the NDN name will be different for each user. This problem defined by “How to efficiently transport HTTP content over NDN to benefit from caching?” will be investigated in future works.

Figure 13. NDN Name to notify a new HTTP request

After this first NDN Interest packet, the NDN client component sends another one in order to retrieve the server response. For these NDN Interest packets, their name structure illustrated in Figure 14 is similar to the first one but with an additional segment number at the end in order to ask a specific chunk of the response. The received data will be encapsulated in the Query object in order to be used by the HTTP server component.

Figure 14. NDN Name to retrieve a response

The NDN server component is composed of a thread pool (of limited size) dedicated to NDN Data packet generation and signature, and another thread for the network part. When the NDN server component receives a Query reference, a thread from the pool handles it. To store de ong-ing queries, a new entry is created in a dictionary with the NDN name as key. It is used by the network thread that processes the Data packets when it can. The network thread always tries to make the largest possible NDN Data packet in order to reduce signature overhead. Finally, the processed Query reference is put back in the queue, waiting for more data, while the others Queries are evaluated and processed in the meantime.

When the NDN server component receives a NDN Interest packet, the network thread looks into the dictionary for an entry corresponding to the NDN name. If an entry exists, the thread sends the corresponding NDN Data packet if it has already been generated. Otherwise, the NDN Interest packet will be put in a queue until it can be answered or a timeout.

4.2.3 Inside the egress Gateway eGW
The HTTP client component is also directly built over a TCP socket because its sole purpose is to forward users’ requests to the right web servers. Advanced features, like auto redirection, provided by HTTP libraries are not needed for our current usage.

When receiving a Query reference from the NDN server component, the HTTP client component creates a thread that initiates a communication with the corresponding web server. Since the requested data are delivered progressively, the thread performs asynchronous tasks that are triggered only if new data are available. A similar procedure is done for the reception of the response, but these tasks can timeout when the web server sends no response. A thread may also stop if the NDN client component encounters problem with the request reception.

The NDN client component of eGW is very similar to the one of iGW. The main difference is that it generates the NDN name based on the one sent by iGW, by extracting the sub-string after the special name component “req” and appending a segment number in order to ask a specific chunk of the request (Figure 15).

![Figure 15. NDN Name to retrieve a request](image)

The NDN server component of the eGW is also very similar to the one of iGW but it has to perform additional tasks since it has two kinds of name to respond. When the iGW asks for a response chunk, the behavior is the same than the the NDN server component of thr iGW. When the iGW notifies a new request, the NDN server component of eGW creates a new Query object and sends a reference to both the NDN client and the HTTP client components. It also puts the Query object in an internal queue used for the generation and signature of NDN Data packet based on the server response. The NDN server component replies with a NDN Data packet that contains a status code for the iGW. The status code can be one the three values listed below:

- OK, represented by 0x01, means that eGW will proceed the request and will retrieve the user request from the world-wide web;
- ALREADY ASKED, represented by 0x02, means that this request is currently in process or is finished but still in memory. eGW will not ask for the request and iGW can directly ask for response chunks;
- BAD REQUEST, represented by 0x03, means that eGW does not understand the name in the NDN Interest packet and will not process further.

### 4.3 Data collection and analysis

To validate the capability of gateways, we have conducted a set of functional tests which consists in determining the set of HTTP objects able to successfully cross our NDN network. We introduce here our data collection tools used to perform the tests and we provide the sets of data we obtained and a detailed analysis of results.

The first test that we have defined is to pass successfully the top-1000 most popular web sites (source: https://gtmetrix.com/top1000.html). Each website in the Top-1000 consists in a main web page composed of a set of HTTP objects. To retrieve the whole page content, we have implemented a dedicated web Scraper we present subsequently.
Deliverable D4.1: Description of the DOCTOR Testbed

4.3.1 Scraper

We developed a web scraper whose objective is to emulate the behavior of the Internet users which pass their HTTP traffic through the NDN. As such it is a component located out of the testbed infrastructures previously described (which may however be Openstack virtual machines). The latter can run a raw scraping of any set of web urls but it can also reproduce a realistic user behavior according to two parameters: (1) when searching a website among the Top-1000, it leverages a Zipf distribution to pick-up a web randomly and (2) the inter-request time distribution follows a Poisson law distribution. That way, beyond the raw scraping of web content it acts as a first step toward the run of the testbed in real conditions (while avoiding to imply real users who may face all the disruptions due to the testbed component adjustments).

The source code of the scraper was developed in Java. It uses the free Jaunt API (jaunt-api.com, release 1.2.3). This API allows recovering the objects constituting a web page such as images, css, javascript, or others. The Scraping algorithm works according to the following steps:

- Launch HTTP requests with particular rate (main website page)
- Get responses from the ingress gateway
- Get all individual objects in the page
- Check the existence of objects
- Check gateway responses against objects
- Determine unretrievable websites

4.3.2 Performance criteria

The first validation test aims to check that the gateway is actually able to map requests and responses to respectively NDN Interest and Data packets and to transit HTTP traffic across the testbed in a reliable way. For that purpose, we tested the gateway against the Top-1000 websites.

The second part of this validation is to measure the gateway performance based on the following criteria:

- Capability to process simultaneous requests
- Impact of the number of simultaneous requests
- Maximum supported rate (number of parallel requests)
- Impact of the NFD cache
- Average time consumed by single request with and without cache

4.3.3 Results

4.3.3.1 Top-1000

The obtained results presented in this section and depicted in Figure 16 are based on the experiments using the web scraper and passing web traffic through NDN network shown in Figure 4. We found that among the 1000 tested web sites, 101 sites were unavailable during the experimentation due to network or server side issues. 132 were only accessible through HTTPS. Thus, we have considered 767 HTTP websites for our tests. Among them, we have collected 47 364 HTTP objects, 92.55% of them were correctly retrieved and are distributed as follows:

- 75.22% are images,
- 11.68% are scripts
- 10.05% are links toward external resources (e.g. CSS sheets).
- 3% are heterogeneous external resources embedded in tags such as iframe, area, span, etc.

Figure 16.a shows the frequency distribution of the percentage of web page content, given by the number of HTTP objects retrieved over the set of all objects. The result (plotted in semi-log scale)
Deliverable D4.1: Description of the DOCTOR Testbed

shows that most of top-1000 websites (more precisely among the 767 websites considered in this study) can be retrieved entirely or almost while only a few websites give poor results. Since the main part of objects that cannot be retrieved are images, among other content, Figure 16.b shows the distribution of images errors plotted in log-log scale for all websites. We can see that the gateway enables a perfect or almost perfect retrieval of the images from the majority of web sites, thus demonstrating its global good performance and that only a few websites among our dataset contain a large set of images (more than 50) than cannot be retrieved. Consequently, it appears that the testbed successfully retrieves most of the reachable websites while errors are located in very small subpart of the dataset.

![Graph](image1.png)

Figure 16. (a) Frequency distribution of the percentage of successful web site retrieval (semi-log scale); (b) Frequency distribution of image errors (log-log scale)

### 4.3.3.2 Scalability support of the GW

The previous results validated the good ability of the gateway to translate HTTP to NDN and vice versa. In this second part, we evaluate its scalability in order to dimension the load it can support when connect to real users. The tests consist in sending multiple requests with different rates. The goal is to test the gateway’s ability to respond to multiple requests simultaneously and to find any collapse point.

The test we have implemented and repeated for different concurrency levels considers a subset of the Top-1000 data set composed of 4096 HTTP objects the testbed can successfully retrieve to let the concurrency test be independent from any other disrupting effect such as timeouts due to the unavailability of a web content. To assess the support of a given concurrency level, we basically consider the percentage of objects the testbed has successfully retrieved.

Figure 17 shows the percentage of success as a function of the number of concurrent requests. We can see that we have a success percentage greater than 98% when the number of concurrent requests is less than 32. We also notice that the greater the number of concurrent requests is, the higher gateways’ success rate falls, until reaching the collapse point when the number of requests in parallel is greater than 256. This fall in performance is mainly due to the way the gateway has been implemented, and specifically the part that performs the translation of HTTP content to NDN. Also, these results can be significantly improved by a new, more efficient, implementation of the gateway that would support multithreading, unlike the current prototype version which has been implemented in a single thread. This new implementation is planned for the next version of the Gateway.
4.3.3.3 Cache efficiency

One of the ground NDN principles consists in the ability of the network nodes to cache content (standing for Data packets). As such, retrieved Data packets are cached in order to satisfy future Interests that request the same content. This feature is thus intended to reduce congestions and improves throughput and latency for popular content. In this section, we present the results we have obtained when testing the NDN cache efficiency in case of HTTP traffic according to the basic HTTP/NDN translation scheme presented in section 4.2. We define the following notations:

- **TimeNGW**: the time spent to correctly get an HTTP object without using the gateway; in other words, by transiting through a standard IP network.
- **TimeGW1**: the time spent to retrieve an HTTP object at the first round by the gateway (NDN network).
- **TimeGW2**: the time spent to retrieve an HTTP object at the second round by the gateway (NDN network). We retransmit the same previously emitted Interest.
- **Timer**: waiting time before resending an interest.

To show the cache efficiency, we compare the consumed time during the first and the second round through the GW. To assess the accuracy around the average values, we calculate the 95% confidence interval which provides a range that is highly likely to contain the true population quantity. Table 3 shows the average time consumed by a request during the first and second round by the Gateway and the average spent time without GW (standard IP network). We can see that the average time required to process a request that passes for the first time through the NDN gateway (i.e., TimeGW1) is slightly more important than in the case of a request that does not pass through the gateway (i.e., TimeNGW). This additional cost is due, in particular, to the additional translation operations that are performed by the gateway. Nevertheless, this additional cost is largely amortized during the second round (TimeGW2) where the caching mechanism allows the NDN network to respond to HTTP requests much more quickly than in the case of a traditional IP network. Figure 18 shows graphically the confidence interval around the average spent time.

<table>
<thead>
<tr>
<th>Table 3. Confidence interval limits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
</tr>
<tr>
<td>TimeNGW</td>
</tr>
<tr>
<td>TimeGW1</td>
</tr>
<tr>
<td>TimeGW2</td>
</tr>
</tbody>
</table>
In order to investigate more detailedly the impact of NDN caches, we conducted additional tests where we define a waiting timer between two consecutive Interests. The objective is to verify the presence of the Data packet in the NFD cache and to show the utility of the parameter FreshnessPeriod measuring the lifetime of the Data packet. We denote by Timer the waiting time between two interest packets requesting the same data. As defined in NDN, the content might be fetched from an NDN router’s cache rather than from its original producer. So, we test here the impact of the defined Timer on the average request time previously studied (see Table 3). In our NDN testbed, we consider a cache policy with FreshnessPeriod=10 seconds.

Figure 19.a and Figure 19.b show, respectively, the histograms of the number of successfully packets as a function of time for IP Internet and NDN network at first round respectively. Those histograms show a bimodal distribution. This means that the distribution of data is around two populations. This phenomenon can be explained by the presence of contents in intermediate caches, provided by the Content Store of NDN nodes, that are closer than the data origins. We note that, the processing time of the GW stills higher than classical IP Internet. This can be explained by the HTTP-NDN conversion and processing time.

Figure 18. 95% confidence interval for TimeNGW, TimeGW1 and TimeGW2.

Figure 19. Histograms of the number of successfully packets as a function of time (a) without GW; (b) first round by GW
Figure 20.a and Figure 20.b show the histograms of the number of successfully packets as a function of time for the second round through the GW for different values of the Timer (0 seconds and 5 seconds). The bimodal distribution disappears. The content is retrieved from the nearest NDN node which is one of our testbed nodes. It is clearly shown that when the Timer is less than the FreshnessPeriod, the average time consumed by a packet during the second round is almost unchanged. When the Timer is greater than FreshnessPeriod, the average spent time is equal to average time spent at first round (packet that has been in the network too long should be discarded).

![Histograms of the number of successful packets as a function of time. (a) second round by GW, Timer=0; (b) second round by GW, Timer=5sec](image1.png)

Figure 21.a shows the distribution of the consumed time during the first GW passage as a function of time spent without GW. To better show the benefit of the NDN cache, we plot on Figure 21.b the time spent during the second round as function of the time spent at the first round. In the majority of cases, the time consumed during the first passage is much higher than the time consumed during the second passage when the waiting Timer is less than FreshnessPeriod.

![TimeGW1 as function of TimeNGW (log-scale) ; (b) TimeGW2 as function of TimeGW1 (log-scale)](image2.png)
5 Integration of the MMT based monitoring solution

5.1 Functionality overview

The virtualization of the monitoring function leads to several challenges. Existing solutions need to be adapted and correctly controlled since they were meant mostly for physical and not virtual systems and boundaries, and do not allow fine-grained analysis adapted to the needs of SDN and virtualized networks. Malfunctioning or even minor problems in a VNF could introduce vulnerabilities and instability of other VNFs, as well as the integrity of the host machine.

In DOCTOR, the monitoring function is needed to be able to precisely understand what is going on in the network, with a twofold objective. First, it is necessary for improving the security in the communications and services offered by the virtual environments. Second, from the administration and management’s point of view, it will help ensure the environment’s health and guarantee that the system functions as expected.

After having identified different possibilities [4], we decided to systematically deploy the monitoring components (MMT3 probes) in every VNF. This approach offers the best security visibility, since the system has a complete view of the internal state of every VNF, as well as the interactions with the host or any other VNF. Furthermore, its deployment is simpler than other approaches since it can be included in the software image of the VNF, so it is automatically initiated when instantiating each virtual machine with no further configuration needed. To reduce the impact on performance the deployed probes are made so that they can be easily configured to strictly capture and analyze only the essential information.

The MMT probe is composed of a set of complementary, yet independent, modules as shown in the following Figure 22. MMT-Extract allows the capture of network packets based on the libpcap library. It is based on Deep Packet and Flow Inspection (DPI/DFI) technology in order to classify and extract network protocol metadata and KPIs. In the context of Doctor, a new plugin to interpret NDN protocols has been developed for both native (direct layer 2 stack) and overlay (NDN/IP stack) cases.

---

3 MMT stands for Montimage Monitoring Tool – More details are available following this link: http://www.montimage.com/flyers/MMT_Brochure.pdf
es, using TLV-based signatures, extracting the different NDN protocol field values and performing basic statistics.

MMT-Security is a rule engine that analyses and correlates network events to detect operational and security incidents. Self-learning capabilities to derive the baseline network usage has been implemented detect Deny of Service (DoS) attacks (like interest flooding attacks) specific to NDN. MMT-QoS provides visibility on the quality of the network in terms of different KPI (Key Performance Indicators, like delays, jitter, response time).

The deployment of MMT monitoring solution in every VNF allows the focus of the analysis on metrics and security indicators related to the VNF itself. In this scenario, only parts of the parsing plugins in MMT-Extract are needed to fulfil the list of protocols used by the VNF. Besides, the security analysis and intrusion detection needs only to target the risks and vulnerabilities identified for the given VNF and differentiate abnormal activity from allowed activity. The security analysis methodology and properties of an NDN node are different from the ones for a firewall or a HTTP/NDN gateway.

The performance impact of the monitoring probe can be reduced when it focuses only on part of the network traffic. Besides, the monitoring tool can analyse specific VNF security issues and apply advanced algorithms to detect pre-identified risks and attacks targeting the single VNF. It can be adapted to the specific requirements of NDN nodes to analyse normal activity and detect any abnormal behaviour. However, the monitoring tool installed in each VNF consumes part of the memory and CPU allocated for the VNF. This can have an impact on the network operation and can add delays in communications. Furthermore, the monitoring tool will have only local visibility of the VNF traffic which compromises the detection of collaborative attacks or attacks involving different network paths. This last limitation is addressed by the sharing of data between MMT probes (P2P cooperation) and by performing centralized analysis (done by the MMT Operator) in order to improve intrusion detection capability.

5.2 Implementation details

5.2.1 NDN Plugin implementation

In the context of Doctor, a new plugin to interpret NDN protocols has been developed for both native (direct layer 2 stack) and overlay (NDN/IP stack) cases, using TLV-based signatures, extracting the different NDN protocol field values and performing basic statistics. This extracted metadata allows monitoring the NDN traffic and performing performance and security analysis of the communication between different NDN nodes to detect potential security flaws.

NDN plugin is available as open source solution in the Github platform following this link: https://github.com/Montimage/NDN_Plugin. Notice that MMT probe is implemented in C language and so is its plugins.

5.2.2 IFA attack detection capability

MMT tool has been extended to parse NDN based traffic and differentiate interest and data packets (and also extract NDN packet and session attributes). Thus, on each NDN node network interface, we are able to compute periodically the ratio defined by “Number of data packets / Number of interest packets”. By implementing the algorithm presented in [5], MMT was able to detect potential IFA attacks by comparing the variation of this ratio with a reference defined by a shifting time window.

---

4 http://named-data.net/doc/ndn-tlv/
Deliverable D4.1: Description of the DOCTOR Testbed

The refinement/configuration of the algorithm assets depends on the analysed network and the generated alerts are reported to be mitigated in a later stage.

5.2.3 Centralized monitoring server

MMT probes have the capacity of P2P communication, so they can share relevant information with the aim of increasing the efficiency of the security mechanisms and, thus, ensure the correct operation of the whole system. But to perform coordination and orchestration of the whole monitoring system, a central MMT Operator, which receives information from the distributed MMT probes, is provided. The MMT Operator is in charge of correlating events to create reports to inform network operators on the network state. It provides a holistic view with the ultimate objective of detecting complex situations that may compromise the overall system.

The monitoring system (MMT probes and MMT Operator) is schematized in Figure 23.

Figure 23. MMT architecture deployment for SDN

MMT-Operator have been adapted to display the NDN topology and the details of NDN communication. Besides, detection alerts related to IFA attack are also taken into account. An example of MMT-Operator GUI is presented in the Figure 24 where we can see the topology of an NDN network and the top active nodes.

Figure 24. MMT-Operation adapted to NDN stack
6 Integration of CyberCAPTOR

CyberCAPTOR is a security monitoring tool based on an attack graph model. Initially developed for physical networks, it was later adapted to virtualized networks and eventually NDN in the context of the DOCTOR project. It is composed of four main modules forming a data pipeline, and a graphical visualization interface. These modules are attack graph generation, attack path extraction, attack path scoring and remediation. The first three modules are automatically chained (with parameters given by the operator), while the remediation module requires manual validation to commit a remediation proposal.

CyberCAPTOR’s inputs are the network topology, vulnerability scans of the machines, fixed and variable costs for applying elementary remediation, operational costs for the infection of a given machine or denial of service and an up-to-date vulnerability database (the NVD database\(^5\)). Its outputs are the complete attack graph, all the extracted attack paths, their scores and a list of remediations (i.e., list of actions to perform) for each attack path.

6.1 Functionality overview

6.1.1 Attack graph generation

The attack graph approach allows a defender to enumerate all possible paths for an attacker, given a network topology (i.e., network and software configuration, VMs placement and domain dependencies). It relies on an up-to-date vulnerability database and a global knowledge of the network. CyberCAPTOR depends on the MulVAL attack graph engine\(^6\). It uses generic rules and vulnerability information from the system to produce attack graphs. A few dozen rules are enough to model most attack steps. System topology and vulnerability information are used as parameters for the generic rules, thus forming attack steps. These attack steps have several inputs, called preconditions, and an output, called postcondition. MulVAL then produces an AND-OR graph, composed of 3 types of nodes: AND nodes, OR nodes and LEAF nodes.

An attack step needs all its preconditions to be true to satisfy its postcondition. For this, “AND” logical nodes are used. On the other hand, “OR” nodes represent different ways for an attacker to gain some level of privileges on the network (e.g., different attack steps that lead to the same postcondition). LEAF nodes are nodes without preconditions. They correspond to elementary preconditions, or “facts”, i.e. information given as input. These facts are the conditions that can further be remediated. In the example shown by Figure 25, there are 4 leaves, 2 AND nodes and 2 OR nodes.

![Simple attack graph](https://example.com/attack_graph.png)

Figure 25. Simple attack graph

---

\(^5\) Website: [https://nvd.nist.gov/download.cfm](https://nvd.nist.gov/download.cfm)

**6.1.2 Attack path extraction**

The complete attack graph for a company network is very large (potentially millions of edges for a few hundred machines), so that it is not relevant to present it to an operator. Due to the complexity of many information systems, focusing interest on particular subgraphs of the attack graph is necessary. A noticeable subgraph category is attack paths.

An attack path is a subgraph of an attack graph corresponding to all graph nodes an attacker can cross to reach a certain objective (generally execute code on a given machine). It is a DAG (Directed Acyclic Graph) rooted on the target machine. Its LEAFs are all facts of the topology that can be used to attack a particular target. Attack paths consequently show the subset of facts that can be changed in order to thwart the attack.

**6.1.3 Scoring**

Attack paths are scored in order to automatically present the most relevant paths to an operator. This is done by assessing the criticality of each attack path or the likelihood of their occurrence. Attack path scores have two components: impact score and risk score.

The impact score is defined as the sum of local impacts for all vertices of the attack graph. The local impact for each vertex is defined by the user, often motivated by operational aspects. By default, each rule (e.g., vulnerability exploitation, network access) has a constant local impact.

Risk scores model the likelihood of the realization of an attack path. It is computed from the leaves of the attack path to its root: each LEAF represents a fact, with a default risk (depending on the fact), and each AND and OR nodes has a risk depending on the corresponding fact or rule and the number of incoming and outgoing vertices of the node.

Each attack path is given a score, which are then normalized between 0 and 1, and sorted.

**6.1.4 Remediation**

CyberCAPTOR provides information on possible remediation actions to prevent exploitation of identified attack paths. This corresponds to a list of actions that need to be performed on the network topology that will disable the attack path. A remediation action is an elementary change in the topology. Each remediation action roughly corresponds to a different precondition. For instance, a patch remediates a vulnerability, a firewall rule remediates a network access, and moving a VM protects it from security incidents on a particular host.

Since one can apply multiple action combinations, all combinations are proposed so that the operator can choose the best one according to functional / business needs. Once a remediation has been chosen, the attack paths are recomputed to take into account the topology changes.

**6.1.5 Visualization**

The Cybercaptor-client component provides a web interface to feed input, configure the attack graph and remediation engines and visualize attack graphs. The client's main web page proposes 5 tabs, each corresponding to a feature described below:

- **Initialization:** A form to upload an XML topology file, in the format generated by cyber-data-extract. This first step is necessary (and the only one available when no attack graph is loaded) and the attack graph and paths are generated upon file submission. When the attack graph is ready, other tabs appear.

- **Configuration:** This tab shows 2 panels. The left panel displays all hosts from the topology (physical and virtual, independently of the protocol (NDN / IP), and allows the user to configure their criticity in the network. These weights are used by the scoring engine to assess the criticity of attack paths depending on the crossed hosts. The right panel allows the user to tune the remediation costs for each remediation type.

- **Attack graph:** This tab displays the complete attack graph, either on topological form (i.e. showing only hosts and attack step between hosts), or on logical form (i.e. showing the full AND-OR graph, with all preconditions and rules).
- **Attack path:** This tab shows attack paths individually, either on topological or on logical form, in a display similar to the attack graph tab. A selector allows the user to choose an attack path among the identified ones, and a gauge displays the criticality of the path (Figure 26. "Attack path" tab). At the bottom of the page are listed the possible remediations for this particular attack path, and a button allows the user to commit them in the topology.

![Figure 26. "Attack path" tab](image)

### 6.2 Implementation details

CyberCAPTOR's main component is a Java program, exposing a REST API through a Tomcat server. The Java program makes calls to Python scripts (cyber-data-extract) to prepare inputs and feeds them to the MulVAL attack graph engine, called in backend. It then loads MulVAL's output and performs attack path extraction and calculation. The Web client queries the API and provides configuration and graph visualization features. All the components are packaged in a Docker container that exposes two ports: the API port and the web interface.

![Figure 27. CyberCAPTOR pipeline](image)

CyberCAPTOR takes in input an XML file containing formatted topology information on the network (subnets, network interfaces configuration, virtual machines placement, NDN topology) and vulnerability information on the running services. Once this file is uploaded through the API, attack graph and paths are automatically generated. Remediation must however be called manually. Remedia-
Deliverable D4.1: Description of the DOCTOR Testbed

The testbed requires cost parameters, indicating the fixed, variable, operational, business, etc., costs of each type of remediation. These costs must be configured beforehand via the API.

More details are available on:
- https://github.com/DOCTOR-ANR/cybercaptor-server
- https://github.com/DOCTOR-ANR/cybercaptor-client
- https://github.com/DOCTOR-ANR/cyber-data-extract
7 NDNPerf

7.1 Introduction

While sizing and building the DOCTOR testbed, we faced a real lack of software dedicated to NDN performance evaluation. Indeed, most research papers are focused on NDN caching performance evaluation but none considers the performance of NDN Data packets generation while this is also of prime importance for applications with real-time constraints. In NDN, Data packets must have at least a SHA-256 hash that should be digitally signed to link the data and its name to the provider. Thus, this hash/signature must be verified in order to check the packet integrity, and, if signed, to authenticate the content provider. However, by using signatures, we found out that NDN exchanges can become CPU intensive when Data packets are generated. This is a critical constraint for real-time applications (live-video, online-games, VoIP, etc.) that generate fresh Data and cannot be signed in advance.

That is why we decided to build a specific performance evaluation tool and to conduct the first comprehensive evaluation of NDN throughput at the server side, while measuring the CPU consumption under different scenarios. This was performed thanks to NDNperf, an open source tool for NDN performance evaluation we made. Details about this tool have been published and presented at the ACM ICN 16 conference [2].

Although real performance evaluation was not at the center of current research efforts of the NDN protocol due to its youth, some studies show performance tests to highlight the benefits of their own solutions. Guimaraes et al. [8] extend the experimentation done by Van Jacobson in a virtual network over the Internet with their testbed named FITS. They highlight the poor performance of CCN in point-to-point transfers compared to TCP despite the latter being limited by a small link capacity (10 Mbps). Some studies considered both software and hardware performance evaluation of NDN packet forwarding. Yuan et al. [9] performed a study of the CCN forwarding daemon in a multi-client/server environment with throughput monitoring and profiling. They expose the operational flow of the forwarder and highlight issues regarding the software scalability, like too complex operational flows, and propose some ideas that will help to improve it. Won So et al. [10] work on an implementation of an NDN forwarder on Cisco routers with integrated service modules that can take advantage of multi-core processors. They report that their implementation can theoretically achieve high throughput (20 Gb/s) based on the number of packets forwarded. To our knowledge, no study investigated the impact of NDN Data packet generation on server performances.

7.2 NDNperf features

We designed NDNperf, an open source tool for NDN server-side performance evaluation and sizing purposes, in order to have an idea of the throughput a server can achieve when it has to generate and transmit NDN Data packets. It is very similar to iPerf and also needs a client and a server to perform the measurements while minimizing the number of instructions between Interest reception and Data emission. It exists in two flavors (Java and C++) and has the following features:

- Periodic report of performances: end-to-end throughput, latency, processing time;
- Fresh NDN Data generation or NDN Data delivery from caches;
- Multi-threaded (one main thread for event lookup and N threads for NDN Data generation);
• Able to use all available signatures implemented in the NDN library, choose the size of the key, and the transmission size of Data packets;
• Message broker implementation (Java version only).

NDNperf features many options regarding the signing process because we identified it as the main bottleneck of application performances. Indeed, code profiling using the Valgrind code profiling tool\textsuperscript{7} on a running NDN server showed that most of the processing time is dedicated to signing (between 87\% and 64\% respectively for a Data packet with a payload size of 1024 and 8192 octets) which constitutes the main driver of performance improvement. The second costliest operation is the wire encoding, accounting for 6\% for a payload size of 1024 octets and 30\% for 8192 octets.

7.3 Server-side performance evaluation of NDN

7.3.1 Experimental environment

Our evaluation uses the LORIA testbed described in section 2.2. NDNperf uses the version 0.4.0 of the NDN libraries and is used in its C++ version. Each server runs its own NDN Forwarding Daemon (NFD) instance (Figure 28).

![Figure 28. Experimental environment](image)

Before each test, we send a warm-up traffic to fill the data structures of the two NFD instances. In the upcoming experiments, we use a MTU of 1500 and an Interest window size of 8 packets at the client side. This window seems to be the best for our testbed due to no real gain past this value. When using the multi-threaded content-provider, we will multiply the window size by the number of threads to take advantage from the available CPU cores.

7.3.2 NDN performances according to the Data source

The next two Figure 29 and Figure 30 represent the NFD cache performance throughput. For each payload size, we generate Data packets with SignatureSha256WithRsa before doing our tests, and these packets will constitute the cache.

\textsuperscript{7} http://valgrind.org/
Deliverable D4.1: Description of the DOCTOR Testbed

Figure 29. Local cache throughput

Figure 30. Distant cache throughput

Figure 29 shows the throughput and CPU usage for the client application and the client-side NFD instance from which the NDN Data are retrieved. With a payload of 8192 octets, the throughput is about 1792Mbps. This difference can be explained because the client-side NFD only needs to look in its Content Store to retrieve the Data packets and doesn’t have to access the PIT nor the FIB. This also demonstrates that NDN performs well when cached data are transmitted.

Then, in Figure 30, we retrieve the NDN Data packets from the server-side NFD cache: the client-side NFD doesn’t have any of the requested Data packet and forwards the Interests to the serve-side NFD instance which directly answers from its Content Store filled with the needed packets. With a maximum of 671Mbps, the throughput is disappointing but is still above the throughput achieved with newly generated NDN Data. Like with DigestSha256, we can observe a slight decrease in packet throughput with the payload size, except for a payload of 1024 octets for which the result is surprisingly low. For larger payload sizes, the limiting factor is clearly the client-side NFD whereas the server-side NFD only uses around 60% of its CPU core. From this, we conclude that the PIT and FIB lookup process on the client-side NFD have a very high price.

7.3.3 NDN performances according to the signature configuration

Our first evaluation is based on a single-threaded version of NDNperf. The achievable throughput is tested in these four conditions: New Data generation with (1) DigestSha256 or (2) SignatureSha256WithRsa, and Content present in (3) client-side NFD cache or (4) server-side NFD cache.

Figure 31. Throughput with DigestSha256

Figure 32. Throughput with Sha256WithRsa

In the following experiment, we use a freshness value of 0 ms for NDN Data so that the client can never get Data packets from any cache. Figure 31 displays the throughput in packets per second and the percentage of CPU usage for each of the 4 running applications
(NDNperf client, client-side NFD, server-side NFD and NDNperf server), and this, for different payload sizes. Packet throughput decreases with the payload size but this is vastly counterbalanced by the fact that the payload size doubles each time, so the global data throughput still increases with a maximum average throughput of 487Mbps. In the case of a simple SHA-256 hash, a mono-threaded application is enough to saturate the NDN Forwarding Daemon (NFD) which constitutes the bottleneck of this experiment. In comparison with the previous tests, the achieved throughput is more than three times lower than the throughput achieved when data are retrieved from the local cache in Figure 29.

In the next experiment, a RSA digital signature is used. Signing packets can be very CPU intensive according to the RSA key length. For this test is applied the default key size used by NFD which is a RSA 2048 bits key. As shown in Figure 32, the limiting factor is now by far the server application that fully uses its allocated processor core in all configurations. The RSA signature costs too much to be handled by only one thread and the result is that we do not use the full capacity of the network and the throughput is limited around 550 packets per second which represents up to 34Mbps of new NDN Data with the largest payload size (8192 octets).

### 7.3.4 Exploiting multi-threaded signature of NDN Data

![Figure 33. Throughput with Sha256WithRsa (multi-thread)](image)

For the final experiment, we use NDNperf as a multi-threaded application and compare the throughput increase for authenticated packets compared to the single-thread setup of Figure 32. Figure 33 shows the throughput regarding the number of threads used on the server. The experiment was done with the default RSA 2048 bits key and a payload of 8192 octets while we run up to 32 signing threads concurrently to match the number of logical cores available on our server. In these conditions, we can saturate an instance of NFD with approximately 25 logical cores generating new NDN Data. But using nearly all the processing resources to prepare NDN Data packets seems inefficient.

In conclusion, NFD is the bottleneck of our testbed configuration when NDN Data are present in caches or generated without RSA signature. When digital signature is used, the content-provider clearly becomes the bottleneck unless it is multi-threaded and can allocate much computation power for signing purpose. This can be a major burden for applications using real-time authenticated NDN Data.
8 Conclusion

This deliverable presents the first version of the integrated DOCTOR testbed. The integration work follows an incremental methodology of the project outcomes starting with a basic deployment of NDN stack in docker-based virtualized environment.

OVS virtual switch has been selected for the DOCTOR testbed and IP/NDN routing VNFs has been also conceived and integrated. An HTTP/NDN gateway has been designed and implemented to allow the DOCTOR testbed interreacting with a regular IP-based environment. Furthermore, integrating security components i.e. MMT monitoring probe for network monitoring and CyberCAPTOR security monitoring tool for vulnerability assessment, has been also achieved for this first version of the testbed.

The DOCTOR testbed is intended to be evaluated with real users (student from UTT and University of Lorraine/TELECOM Nancy) and this will allow us to have relevant feedback to improve the testbed and its components. More work related to the integration of SDN and dynamic management of VNFs concepts is also planned for the future versions of the testbed.


9 References


