UNIVERSITÀ DI PISA

SEA++

USER MANUAL

SEA++ WITH SDN SUPPORT INET-BASED

EDITED BY

FRANCESCO RACCIATTI[1]
ALEXANDRA STAGKOPOULOU[2]
MARCO TILOCA[1,2]
GIANLUCA DINI[1]

UNIVERSITÀ DI PISA

SEA++ INET-based
User manual

Francesco Racciatti[1] · Alexandra Stagkopoulou[2] · Marco Tiloca[1,2] · Gianluca Dini[1]

AFFILIATIONS

1. Dept. of Ingegneria dell'Informazione, University of Pisa, Largo Lazzarino 1, 56100 Pisa, Italy
2. SICS Swedish ICT AB, Security Lab, Isafjordsgatan 22, SE-164 40 Kista, Sweden

CONTACTS

- racciatti.francesco@gmail.com · write to Francesco Racciatti
- alexandra.stagkopoulou@sics.se · write to Alexandra Stagkopoulou
- marco@sics.se · write to Marco Tiloca
- gianluca.dini@ing.unipi.it · write to Gianluca Dini

Contents

List of Figures

Chapter 1

Intoduction

SEA++ is an attack simulator based on INET/OMNeT++ network simulator. It allows the reproduction of effects of successfull attacks and the quantitative evaluation of their impact. SEA++ is a complete simulation tool which helps the user to evaluate the impact of cyber/physical attacks. It has to be clarified that SEA++ does not find new attacks neither evaluate the feasibility of them. The user does not need to implement the adversary model as the actual way of attack's execution is out of scope of SEA++. The user only describes the attack scenario and evaluates the impact of the successfull attack.

SEA++ consists of 3 basic components; (i) the high level *Attack Description Language (ADL)*, (ii) the *Attack Interpreter* and (iii) the *Engine*. The user describes the attack scenarios using the ADL. The interpreter converts the attack file .adl to the Attack Configuration File .xml, which is given as input to the Engine. The Engine injects the atomic events at runtime during the network simulator based on the Attack Configuration File. The two basic components of the Engine are the *Local Filter* and the *Global Filter*. The Local Filter is the component which communicates with all the layers of the OSI stack of a node being able to intercept, inject, modify or drop the packets. The Local Filters of all the nodes within the network communicate with the Global Filter which represents an external entity in the network. The two components handle the 3 different types of attacks; *physical*, *conditional* and *unconiditonal* attacks. The Local Filter is responsible to perform the physical and the conditional attacks based on the ADL primitives, while the Global Filter is responsible for the unconditional attacks.

SEA++ has been extended providing support for SDN architectures. The basic mechanism of SEA++ has been integrated to OpenFlow switches and SDN controllers and the user is able to describe attack scenarios against these units. This manual describes the ADL primitives of the attack simulator SEA++ and shows the steps to run an example either in traditional or SDN networks.

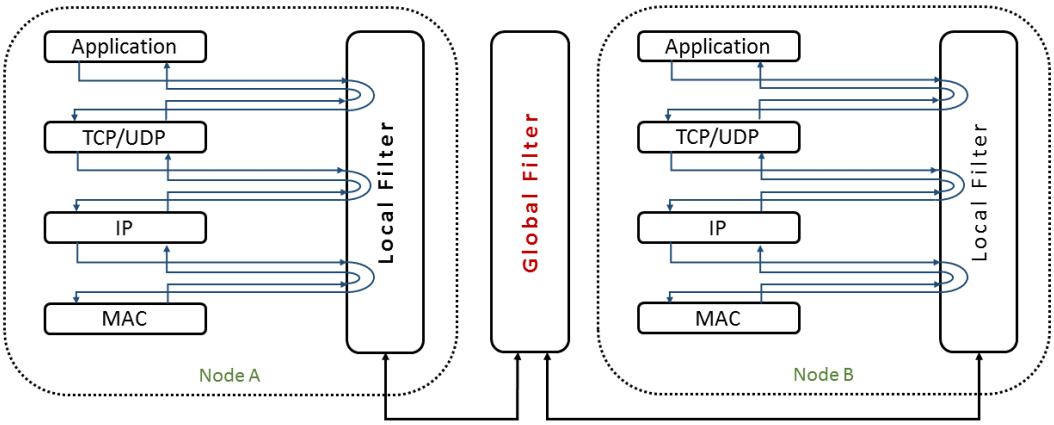

Figure 1.1: SEA++ Mechanism

Chapter 2

Build SEA++

SEA ++ is based on many components and makes use of multiple programming languages. In particular SEA++:

- is written in C++11 and Python 2.7.6;
- extends INET 2.6, which is based on OMNeT++ 4.x;
- requires the `libxml` library;

SEA++ is designed for Linux distributions based on Debian and it was not tested on other OSs. It is fully compatible with Ubuntu 14.04 LTS, but it is possible using it on older or newer versions by adopting several shrewdnesses.

2.1 Build SEA++ on Ubuntu 14.04 LTS

The steps to build SEA++ are the following:

1. get and install `libxml` library;
2. get and build OMNeT++ 4.x (preferibly 4.6 or greater);
3. get and build SEA++.

If errors occur during the installation of some component, it should refer to official guide:

- `libxml` - `http://www.xmlsoft.org`
- OMNeT++ - `http://omnetpp.org`
- SEA++ - `https://github.com/seapp/src/archive/v0.99.tar.gz`

2.1.1 Get and install `libxml` library

The `libxml` library can be installed by using the `apt-get`:

```
~$ sudo apt-get install libxml++2.6-dev libxml++2.6-doc
```

The `libxml` library sources are available at `http://libxmlplusplus.sourceforge.net/`.

2.1.2 Get and build OMNeT++

To get OMNeT++ 4.6 open your terminal, go in your home and type:

```
~$ wget http://omnetpp.org/omnetpp/send/30-omnet-releases/2290-omnet-4-6-source-
    ide-tgz
```

When the download finishes, untar and unzip the source files in your home:

```
~$ tar xvfz 2290-omnet-4-6-source-ide-tgz
```

It creates the directory `omnetpp-4.6`.

Set environment variables (assuming you are using bash as your shell):

```
~$ export PATH=$PATH:~/omnetpp-4.6/bin
~$ export LD_LIBRARY_PATH=~/omnetpp-4.6/lib
```

Append the above commands to the `.bash_profile` file.

It is now possible to build OMNeT++. If no graphical environment is available then follow the below commands:

```
~$ cd omnetpp-4.6/
~/omnetpp-4.6$ NO_TCL=1 ./configure
~/omnetpp-4.6$ make
```

Otherwise, configure OMNeT++ with the graphical interface:

```
~$ cd omnetpp-4.6/
~/omnetpp-4.6$ ./configure
~/omnetpp-4.6$ make
```

More details about OMNeT++ installation can be found in `http://omnetpp.org`.

2.1.3 Get and build SEA++

To get `SEA++` open your terminal, go in your home and type:

```
~$ wget https://github.com/seapp/seapp_stable/archive/master.zip
```

or visit `https://github.com/seapp/seapp_stable` to download the zip code.

When the download finishes, unzip and rename the master directory in your home:

```
~$ unzip master.zip
~$ mv seapp_stable-master seapp_stable
```

The new directory `INET_SDN_dev` has been created and it is now possible to build SEA++ supporting SDN architectures:

```
~$ cd seapp_stable
~/seapp_stable$ make makefiles
~/seapp_stable$ make
```

The `seapp_stable` directory on your home should contain the following subdirectories:

`seapp_stable/`	SEA++ root directory
`interpreter/`	ADL interpreter

src/	sources
actions/	action classes
actionbase/	action base class
change/	change class
clone/	clone class
create/	create class
destroy/	destroy class
disable/	disable class
drop/	drop class
move/	move class
put/	put class
retrieve/	retrieve class
seappexpression/	seappexpression class
send/	send class
attacks/	attack classes
attackbase/	attack base class
attackentry/	attack entry class
physicalattack/	physical attack class
conditionalattack/	conditional attack class
unconditionalattack/	unconditional attack class
exmachina/	exmachina class
globalfilter/	globalfilter class
localfilter/	localfilter class
parser/	parser class
openflow/	SDN directory
apps/	SDN applications
attackDetection/	sdn-based attacks' detection methods
ctrlApps/	SDN controller's application behaviors
images/	SDN images
messages/	OpenFlow messages
mitigation/	sdn-based attacks' mitigation methods
nodes/	SDN nodes
scenarios/	SDN topology scenarios
utility/	utility class
variable/	variable class
.../	standard INET classes
examples/	examples ready to use
...	inet examples
inet_sdn/	SDN-based examples
.../	standard INET classes

This version of SEA++ includes the `openflow` direcory which provides the support for the SDN architectures. All the components, nodes and applications for the new type of networks can be found under this directory.

2.2 Build SEA++ on older Ubuntu versions

To build SEA++ on older Ubuntu versions, in addition to the steps described above you may need to:

- upgrade the Python interpreter;
- upgrade the C++ compiler;
- change SEA++ makefile;
- change OMNeT++ configure file.

2.2.1 Upgrade Python interpreter

SEA++ uses features provided by Python 2.7.6, so it is necessary to install at least this version of its interpreter.

2.2.2 Upgrade C++ compiler

SEA++ uses features provided by C++11. It is necessary to install at least the version 4.7 of gcc/g++ compilers and set them as default.

2.2.3 Link libraries in SEA++ makefile

SEA++ is built and checked for Debian Linux Distributions and not for other OSs. It is necessary to check if libraries used by SEA++ are linked to the correct paths. After downloading SEA++, in the initial makefile, all the links to the libraries are under `-l/usr/lib/x86_64-linux-gnu/*` path.

You may need to change the path to `-l/usr/lib/i386-linux-gnu/*` depending on system's architecture.

2.2.4 Change CFLAGS in OMNeT++ configure file

SEA++ is based on INET and the `opp_makemake` tool of OMNeT++ is used in order to compile the framework. The file `configure.user` is the one where all the necessary parameters (compilers, flags, etc) are declared. To enable the features provided by C++11 you have to declare the option `'-std=c++11'` in the flag CFLAGS of OMNeT++'s `configure.user` file.

Chapter 3

Summary of ADL primitives

3.1 Actions

The set of the available actions is shown below.

destroy The *destroy* action is a node action used to destroy a node. The node discards all the packets and it is present in the simulation field.

Listing 3.1: ASL destroy example

```
#primitive: destroy(nodeId, t)
destroy(5, 200)
```

Listing 3.2: Interpreter output

```
<configuration>
  <Physical>
    <Attack>
      <start_time>200</start_time>
      <node>5<node>
      <action>
        <name>Destroy</name>
      </action>
    </Attack>
  </Physical>
</configuration>
```

disable The *disable* action is a node action used to remove the node from the simulation field.

Listing 3.3: ASL disable example

```
#primitive: disable(nodeId, t)
disable(5, 200)
```

Listing 3.4: Interpreter output

```
<configuration>
  <Physical>
    <Attack>
      <start_time>200</start_time>
```

```
      <node>5<node>
      <action>
        <name>Disable</name>
      </action>
    </Attack>
  </Physical>
</configuration>
```

In case of **disable** primitive it is necessary to import and declare the submodule **'ExMachina'** in the .ned file. The module does not need to be connected, but its presence is necessary as it is responsible to handle this action.

move The *move* action is a node action used to move a node.

Listing 3.5: ASL move example

```
#primitive: move(nodeId, t, x, y, z)
move(5, 100, 10, 10, 0)
```

Listing 3.6: Interpreter output

```
<configuration>
  <Physical>
    <Attack>
      <start_time>100</start_time>
      <node>5<node>
      <action>
        <name>Move</name>
        <parameters>10:10:0</parameters>
      </action>
    </Attack>
  </Physical>
</configuration>
```

retrieve The *retrieve* action is a packet action used to retrieve the content of a field of a target packet and to store it into a variable.

Listing 3.7: ASL retrieve example

```
#primitive: retrieve(packetName, fieldName, variableName)
list targetList = {1,2,5}
from 200 nodes in targetList do {
 filter("UDP.sourcePort" == "1000" or "UDP.sourcePort" == "1025" and "UDP.
      destinationPort" == "2000")
    var destPort
    retrieve(original, "UDP.sourcePort", destPort)
}
```

Listing 3.8: Interpreter output

```
<configuration>
  <Conditional>
    <Attack>
      <start_time>200</start_time>
      <node>1:2:5</node>
      <var>
        <name>destPort</name>
        <value></value>
        <type>NONE</type>
```

```
      </var>
      <filter>UDP.sourcePort:==:1000:UDP.sourcePort:==:1025:UDP.destinationPort
           :==:2000:AND:OR</filter>
      <action>
        <name>Retrieve</name>
        <parameters>packetName:original:field_name:UDP.sourcePort:varName:
             destPort</parameters>
      </action>
    </Attack>
  </Conditional>
</configuration>
```

In the ACF the conditional operators are stored in *reverse order* to that of the operands.

drop The *drop* action is a packet action used to discard a target packet.

Listing 3.9: ASL drop example

```
#primitive: drop(packetName)
list targetList = {1,2,5}
from 200 nodes in targetList do {
  filter("UDP.sourcePort" == "1000" or "UDP.sourcePort" == "1025" and "UDP.
       destinationPort" == "2000")
    drop(original)
}
```

Listing 3.10: Interpreter output

```
<configuration>
  <Conditional>
    <Attack>
      <start_time>200</start_time>
      <node>1:2:5</node>
      <filter>UDP.sourcePort:==:1000:UDP.sourcePort:==:1025:UDP.destinationPort
           :==:2000:AND:OR</filter>
      <action>
        <name>Drop</name>
        <parameters>packetName:original</parameters>
      </action>
    </Attack>
  </Conditional>
</configuration>
```

clone The *clone* action is a packet action used to create a packet that is a clone of a target packet.

Listing 3.11: ASL clone example

```
#primitive: clone(packetName, clonedPacketName)
list targetList = {1,2,5}
from 200 nodes in targetList do {
  filter("UDP.sourcePort" == "1000" or "UDP.sourcePort" == "1025" and "UDP.
       destinationPort" == "2000")
    packet dolly
    clone(original, dolly)
}
```

Listing 3.12: Interpreter output

```
<configuration>
  <Conditional>
    <Attack>
      <start_time>200</start_time>
      <node>1:2:5</node>
      <filter>UDP.sourcePort:==:1000:UDP.sourcePort:==:1025:UDP.destinationPort
          :==:2000:AND:OR</filter>
      <action>
        <name>Clone</name>
        <parameters>packetName:original:newPacketName:dolly</parameters>
      </action>
    </Attack>
  </Conditional>
</configuration>
```

The interpreter handles the variables and the packets in two different ways. In the ADL file, the user has to declare both the variables and the packets but in the ACF the declared packets do not appear, unlike the declared variables.

create The *create* action is a packet action used to create a packet 'ex-novo'.

Listing 3.13: ASL create example

```
#primitive: create(packetName, layer5.type, value5, layer4.type, value4, layer3.
    type, value3, layer2.type, value2)
list targetList = {1,2,5}
from 200 nodes in targetList do {
  filter("UDP.sourcePort" == "1000" or "UDP.sourcePort" == "1025" and "UDP.
      destinationPort" == "2000")
    packet exNovo
    create(exNovo, "APP.type", "0000")
}
```

Listing 3.14: Interpreter output

```
<configuration>
  <Conditional>
    <Attack>
      <start_time>200</start_time>
      <node>1:2:5</node>
      <filter>UDP.sourcePort:==:1000:UDP.sourcePort:==:1025:UDP.destinationPort
          :==:2000:AND:OR</filter>
      <action>
        <name>Create</name>
        <parameters>packetName:exNovo:APP.type:0000</parameters>
      </action>
    </Attack>
  </Conditional>
</configuration>
```

The default sizes of packets created by the *create* action are listed in table 3.1. The sizes can be changed by using the *change* action.

change The *change* action is a packet action used to change the content of a field of a target packet.

Listing 3.15: ASL change example

```
#primitive: change(packetName, fieldName, variableName)
list targetList = {1,2,5}
from 200 nodes in targetList do {
  filter("UDP.sourcePort" == "1000" or "UDP.sourcePort" == "1025" and "UDP.
       destinationPort" == "2000")
    change(original, "UDP.destinationPort", 5000)
}
```

Listing 3.16: Interpreter output

```
<configuration>
  <Conditional>
    <Attack>
      <start_time>200</start_time>
      <node>1:2:5</node>
      <var>
        <name>5000</name>
        <value>5000</value>
        <type>NUMBER</type>
      </var>
      <filter>UDP.sourcePort:==:1000:UDP.sourcePort:==:1025:UDP.destinationPort
          :==:2000:AND:OR</filter>
      <action>
        <name>Change</name>
        <parameters>packetName:original:field_name:UDP.destinationPort:value
            :5000</parameters>
      </action>
    </Attack>
  </Conditional>
</configuration>
```

In the ASL file, even if the user uses variables which has not been explicitly declared, the interpreter automatically declares and initializes the variables (if it is necessary).

SEA++ supports the assignement of random values to the fields. The keywords **"RANDOM_IP"**, **"RANDOM_MAC"**, **"RANDOM_INT"**, **"RANDOM_SHORT"** are used to generate random values of the specific *RANDOM_x* type. More details can be found in 3.3 .

As mentioned above, when a packet is created a default packet size is assigned to it. The size can be changed by using the keyword ***controlInfo.packetSize***.

Finally, SEA++ supports the encapsulation of packets using the *change* action. The feature allows the building of complete fake packets including all the layers of the OSI stack. An example of the feauture is:

Listing 3.17: ASL change example: Packet Encapsulation

```
#primitive: change(packetName, fieldName, variableName)
list targetList = {1,2,5}
from 200 nodes in targetList do {
    packet traPacket
    packet appPacket

    #create a new APP-layer packet
    create(appPacket, "APP.type", "1001")

    change(appPacket, "APP.info", 1)
    change(appPacket, "APP.name", "myPacket")

    change(appPacket, "controlInfo.packetSize", 1250)
```

```
    #create a new TRA-layer packet
    create(traPacket, "TRA.type", "0000")
    change(traPacket, "TRA.destinationPort", 123)

    ############ SET THE PAYLOAD ############
    change(traPacket, "TRA.payload", appPacket)
}
```

Using the keywork **payload** ("*layer_name.**payload***"), we can define the packet which will be encapsulated as payload in the current packet.

send Given a packet, cloned or created (and correctly fillled), which belongs to the layer L, the *send* action is used to send the packet to the bottom layer.

Listing 3.18: ASL send example

```
#primitive: send(packetName, forwardingDelay)
list targetList = {1,2,5}
from 200 nodes in targetList do {
  filter("UDP.sourcePort" == "1000" or "UDP.sourcePort" == "1025" and "UDP.
       destinationPort" == "2000")
    packet dolly
    clone(original, dolly)
    send(dolly, 0)
}
```

Listing 3.19: Interpreter output

```
<configuration>
  <Conditional>
    <Attack>
      <start_time>200</start_time>
      <node>1:2:5</node>
      <filter>UDP.sourcePort:==:1000:UDP.sourcePort:==:1025:UDP.destinationPort
           :==:2000:AND:OR</filter>
      <action>
        <name>Clone</name>
        <parameters>packetName:original:newPacketName:dolly</parameters>
      </action>
      <action>
        <name>Send</name>
        <parameters>packetName:dolly:delay:0</parameters>
      </action>
    </Attack>
  </Conditional>
</configuration>
```

put The *put* action is usefull to transmit packets from a node to a set of recipient nodes bypassing the communication channel.

Listing 3.20: ASL put example

```
#primitive: put(packetName, recipientNodes, direction, updateStats,
     forwardingDelay)
list targetList = {1,2,5}
list dstList = {6}
from 200 nodes in targetList do {
  filter("UDP.sourcePort" == "1000" or "UDP.sourcePort" == "1025" and "UDP.
       destinationPort" == "2000")
```

```
    packet dolly
    clone(original, dolly)
    put(dolly, dstList, TX, FALSE, 0)
}
```

Listing 3.21: Interpreter output

```
<configuration>
  <Conditional>
    <Attack>
      <start_time>200</start_time>
      <node>1:2:5</node>
      <filter>UDP.sourcePort:==:1000:UDP.sourcePort:==:1025:UDP.destinationPort
          :==:2000:AND:OR</filter>
      <action>
        <name>Clone</name>
        <parameters>packetName:original:newPacketName:dolly</parameters>
      </action>
      <action>
        <name>Put</name>
        <parameters>packetName:dolly:nodes:6:direction:TX:throughWC:false:delay
            :0</parameters>
      </action>
    </Attack>
  </Conditional>
</configuration>
```

3.2 Variables and expressions

3.2.1 Variables

The ADL handles variables which can store both numbers and strings. The user must declare the variable before using it. The syntax to declare a variable is:

Listing 3.22: Syntax to declare a variable

```
var foo
```

The AS introduces the type transparency than the user has not to declare the type of the content of the variable, e.g. integer, or double, or string.

Expressions The ADL handles expressions that make possible both operations and assignments. In the table 3.2 is shown the ADL expression table.

An example of ADL expression is:

Listing 3.23: Syntax expression example

```
var result
var operand1 = 2
var operand2 = 7
result = operand1 + operand2
```

As specified above, the ADL introduces the type transparency and a variable can store both numbers and strings. This feature makes possible to initialize a variable with a number and after assign a string to it:

Packet Type	**Packet Size**
Application	1 byte
TCP	20 bytes
UDP	8 bytes
IP	20 bytes
PPPFrame	7 bytes
EthernetFrame	18 bytes

Table 3.1: Default Packet Size

operator	**numbers**	**strings**
$=$	supported	supported
$+=$	supported	not supported
$-=$	supported	not supported
$\times=$	supported	not supported
$/=$	supported	not supported
$\div=$	supported	not supported

(a) Assignment operators

operator	**numbers**	**strings**
$+$	supported	supported
$-$	supported	not supported
$\times$	supported	not supported
$/$	supported	not supported
$\div$	supported	not supported

(b) Arithmetic operators

operator	**numbers**	**strings**
$<$	supported	supported
$>$	supported	supported
$<=$	supported	supported
$>=$	supported	supported
$==$	supported	supported
$!=$	supported	supported

(c) Comparison operators

operator	**numbers**	**strings**
AND	supported	supported
OR	supported	supported

(d) Logical operators

Table 3.2: ADL expression table

Listing 3.24: Legal expressions

```
var result
var operand1 = 2
var operand2 = 7
result = operand1 + operand2
result = "Hello, world!"  # legal expression
```

However the user has got the responsibility to ensure the consinstency of the expressions:

Listing 3.25: Illegal expressions

```
var result
var operand1 = 2
var operand2 = 7
result = operand1 + operand2
result = "Hello"    # legal expression
result += ", world!"    # legal expression
var operand3 = 5
resutl += operand3    # illegal expression
```

3.3 Random Values

The *change* action is a packet action used to change the content of a field of a target packet. SEA++ supports the assignement of random values to the fields. The random generator is based on the C++11 libraries which guarantee uniformly distributed non-deterministic random numbers within a predefined range. The below keywords are used to generate random values of the specific RANDOM_x type:

`RANDOM_IP`	Random IPv4 Addresses
`RANDOM_MAC`	Random Ethernet MAC Addresses
`RANDOM_INT`	Random positive integer numbers
`RANDOM_SHORT`	Random positive short numbers

A *RANDOM_INT* number will be a positive number within the range of 0 and the maximum integer supported and a *RANDOM_SHORT* value will be ranging among 0 and the maximun value for a variable of type short.

While the usage of "*RANDOM_INT*" and "*RANDOM_SHORT*" keywords is straightforward, there are some guidelines to help for the rest two keywords.

3.3.1 Assign random MAC addresses

The current implementation of SEA++ supports the assignment of random Ethernet MAC addresses. The change is based on the control information objects which are attached on packets while they are traversing the communication stack of a node. To this end, the interception of the packet is done between the network and mac layer of the stack. Below is an example of a conditional attack using the ADL and changing the source MAC address of the packet:

Listing 3.26: Assign random MAC source address

```
list dstList = {2}
from 21 nodes in dstList do {
  filter ("NET.destAddress" == "192.168.2.6" and "attackInfo.fromGlobalFilter" ==
        1)
    change(original, "controlInfo.src", RANDOM_MAC)
}
```

The compromised node 2 intercepts all the packets which are destined to host with IP address "192.168.2.6", but it changes the MAC source address only to packets which have been generated by the global filter. The keyword **attackInfo.fromGlobalFilter** is used to distiguish the genuine packets from packets which have been generated by the Global Filter through unconditional attacks.

3.3.2 Assign random IP addresses

The change of the IP address field can be performed in two different ways. One is based on the control information object that OMNeT++ provides and the second one is based on the packet header. The choice depends on the user's need and the attack's description.

Using the Control Object

In this case, the packet is intercepted between the transport and the network layer. The user has to be aware of the current topology and its specifications, mainly the interface table, because the correct interface id has to be defined during the description of the action. In this way, one compromised host can be used to generate and inject packets in the network with different IP source addresses using the same interface. An example of this case is shown below:

Listing 3.27: Assign random IP source address on control object

```
list dstList = {2}

from 20 every 0.4 do {

  # declare a packet
  packet fakePacket

  # create a new application packet
  create(fakePacket, "APP.type", "1001")

  # fill the new packet properly

  change(fakePacket, "APP.info", RANDOM_INT)
  change(fakePacket, "APP.name", "myPacket")

  change(fakePacket, "controlInfo.destAddr", "192.168.2.6")
  change(fakePacket, "controlInfo.destPort", 123)
  change(fakePacket, "controlInfo.sockId", 0)
  change(fakePacket, "controlInfo.interfaceId", 101)


  change(fakePacket, "sending.outputGate", "app_udp_inf$o[0]")

  put(fakePacket, dstList, TX, FALSE, 0)
```

```
}

from 21 nodes in dstList do {
  filter ("TRA.destinationPort" == "123" and "attackInfo.fromGlobalFilter" == 1)
    change(original, "controlInfo.srcAddr", RANDOM_IP)
}
```

In the example, the user defines the generation of application packets which will be forwarded to the destination "192.168.2.6" on port "123". The interface used to forward the packets has the id number "101". This is not an arbitrary number but the actuall interface id used by the simulation. The generated by the simulator packets will be injected in the reception buffer of the compromised host 2. The compromised host starts intercepting the packets between the transport and the network layer (conditional attack) and changes the IP source address of the packet to a random value.

If a different implementation than the one provided by INET 2.6 of Routing or Interface table is used, this change will not be supported.

HINT: OMNeT++ counts the interface ids starting from number 100. Higher priority is given to loopbacks, then the rest of the interfaces follow. In this example, we consider a host with one loopback and one Ethernet interface.

Using the packet header

An alternative way to change the IP source field of a packet is based on the direct manipulation of the packet header. In this case, the interception of the packet is performed between the network and mac layer of the communication stack. This change action can also be combined with the assignment of a random MAC source address of the packet.

Listing 3.28: Assign random IP source address on packet header

```
list dstList = {2}

from 20 every 0.4 do {

  # declare a packet
  packet fakePacket

  # create a new application packet
  create(fakePacket, "APP.type", "1001")

  # fill the new packet properly

  change(fakePacket, "APP.info", RANDOM_INT)
  change(fakePacket, "APP.name", "myPacket")

  change(fakePacket, "controlInfo.destAddr", "192.168.2.6")
  change(fakePacket, "controlInfo.sockId", 0)
  change(fakePacket, "controlInfo.interfaceId", 1)
  change(fakePacket, "controlInfo.destPort", 123)

  change(fakePacket, "sending.outputGate", "app_udp_inf$o[0]")

  put(fakePacket, dstList, TX, FALSE, 0)
}

from 21 nodes in dstList do {
```

```
  filter ("NET.destAddress" == "192.168.2.6" and "attackInfo.fromGlobalFilter" ==
        1)
    change(original, "NET.srcAddress", RANDOM_IP)
    change(original, "controlInfo.src", RANDOM_MAC)
}
```

The generated IP values are based on the network mask and the network address the user defines in the `.ini` file. The feature has been tested by using only the *'FlatNetworkConfigurator'* module. In case of *'IPv4NetworkConfigurator'*, arbitrary values will be generated.

Chapter 4

Set up and run a new SEA++ scenario

SEA++ is distributed with a complete set of examples that are ready to use. However, the process to build a new SEA++ scenario is substantially identical to that of INET.

4.1 Traditional Networks

In the following example is shown how to set up a new simple scenario in which there are a client and a server.

1st step - make the folder As in INET, the 1st step is to make the folder that will contain all the new files, e.g. scenario.

```
~/seapp_stable/inet/examples/ mkdir scenario
~/seapp_stable/inet/examples/ cd scenario
```

2nd step - network description As in INET, the 2nd step is to edit the `.ned` file that contains the network description, e.g. `scenario.ned`.

Listing 4.1: scenario.ned

```
package inet.examples.inet.scenario;

import inet.networklayer.autorouting.ipv4.IPv4NetworkConfigurator;
import inet.nodes.inet.StandardHost;
import inet.globalfilter.GlobalFilter;

network scenario
{
  parameters:
    string attackConfigurationFile = default("none");
    double n;

  submodules:
    globalFilter: GlobalFilter;
    client: StandardHost;
    server: StandardHost;
```

```
    configurator: IPv4NetworkConfigurator;

  connections allowunconnected:
    client.pppg++ <--> { datarate = 10Mbps; } <--> server.pppg++;
    globalFilter.nodes++ <--> client.global_filter;
    globalFilter.nodes++ <--> server.global_filter;

}
```

In this step, is fundamental to:

- add the the string parameter `attackConfigurationFile` to the network;
- import the GlobalFilter class, declare a GlobalFilter submodule and connect it to all the other nodes.

In case of **disable** primitive it is necessary to import and declare the submodule 'ExMachina' in the .ned file.

3rd step - edit omnetpp.ini As in INET, the 3rd step is to edit the `omnetpp.ini` file. In this step is fundamental to bind the configuration(s) with the ACF(s), by overwriting the name of the network parameter `attackConfigurationFile` with the name of a particular ACF.

```
[General]
network = scenario
sim-time-limit = 600s

// General settings ...

// Config(s) specific settings
[Config attack-example]
**.attackConfigurationFile = attack-example.xml

[Config attack-example2]
**.attackConfigurationFile = attack-example2.xml

// ...
```

4th step - add the ACF The 4th and last step is to add the ACF(s) in the folder.

Run the simulation The simulation is ready to run. In the terminal, call `run_inet` that is in the `src` folder:

```
~/seapp_stable/inet/examples/scenario ../../src/run_inet $*
```

It will start the simulation supported by the GUI, as shown in figure 4.1. Be carefull to specify well the path from the current folder (i.e. the simulation folder) to the `src` folder.

To run the simulation in express mode without use the GUI, type:

```
~/seapp_stable/inet/examples/scenario ../../src/run_inet -u Cmdenv -f omnetpp.
    ini
```

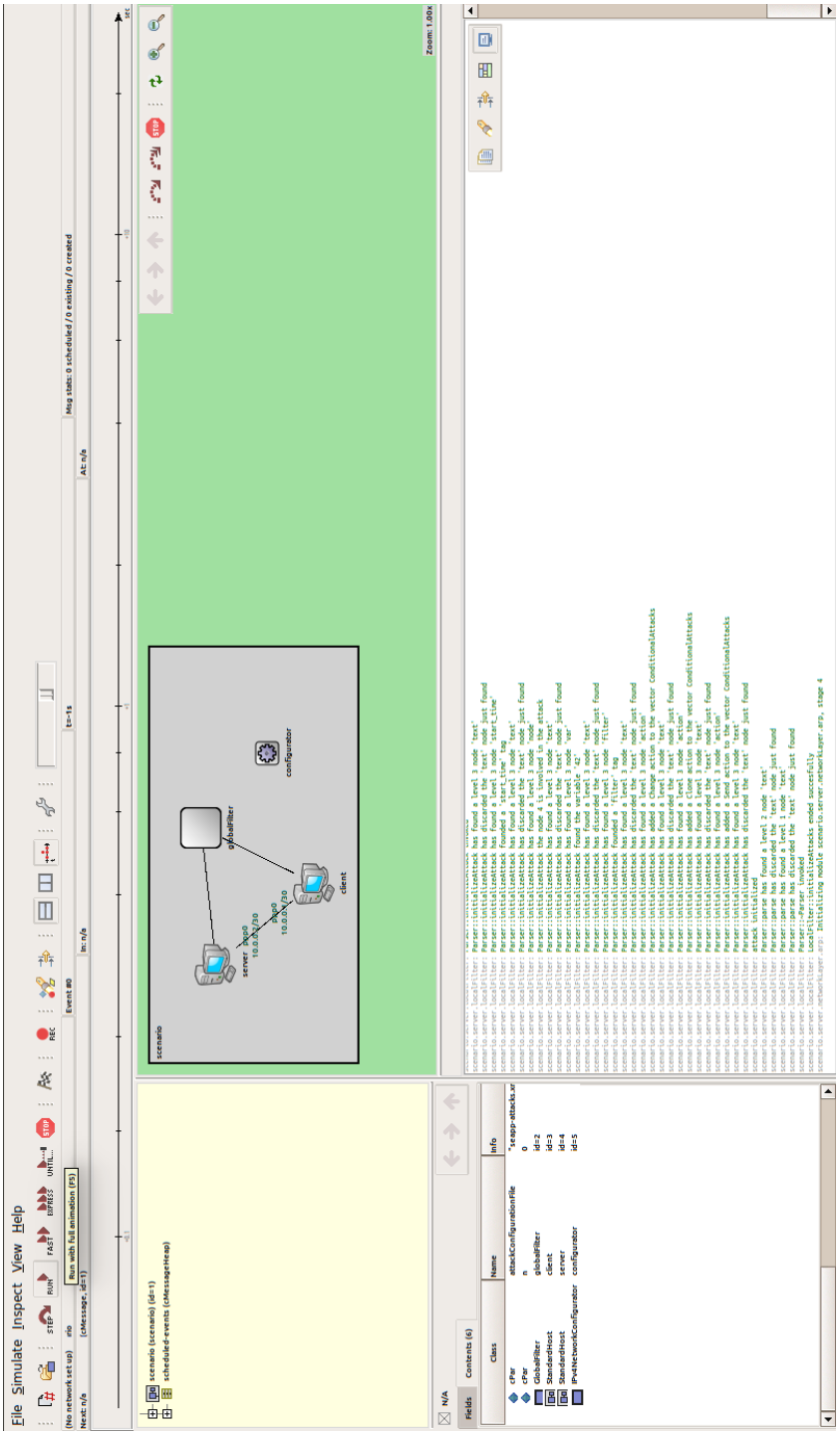

Figure 4.1: Simulation scenario

4.2 SDN Architectures

The same foundamental steps are followed in order to run an example in SDN architecture, with some small differences though. The main difference is that SDN nodes should be used. All the SDN-related nodes can be found under the */openflow* directory.

The below example shows a network with one client, one server, one OpenFlow switch and a SDN controller. The source code can be found under the */examples/inet_sdn* direcory.

1st step - make the folder As in INET, the 1st step is to make the folder that will contain all the new files, e.g. scenario.

```
~/seapp_stable/examples/inet_sdn mkdir scenario
~/seapp_stable/examples/inet_sdn cd scenario
```

2nd step - network description As in INET, the 2nd step is to edit the `.ned` file that contains the network description, e.g. `scenario.ned`.

Listing 4.2: scenario.ned

```
package inet.examples.inet_sdn.scenario;

import inet.nodes.inet.StandardHost;
import inet.networklayer.autorouting.ipv4.FlatNetworkConfigurator;
import inet.util.ThruputMeteringChannel;

import inet.globalfilter.GlobalFilter;
import inet.exmachina.ExMachina;

import inet.openflow.nodes.*;

network Topo
{
    parameters:
        string attackConfigurationFile = default("none");

    @display("bgb=2000,1000");
    types:
        channel ethernetline extends ThruputMeteringChannel {
            delay = 1us;
            datarate = 100Mbps;
        }
  submodules:
        configurator: FlatNetworkConfigurator {
            @display("p=1800, 500");
        }
        globalFilter: GlobalFilter {
            @display("p=1800,200");
        }
        exmachina: ExMachina {
            @display("p=1800, 700");
        }
        client: StandardHost {
           @display("p=150,250;i=device/laptop");
        }
        server: StandardHost {
            @display("p=1500,350;i=device/server");
```

```
        }
        open_flow_switch: Open_Flow_Switch_SEA {
            @display("p=1000,650");
        }
        controller: Open_Flow_Controller_SEA {
            @display("p=1000,100");
        }


    connections allowunconnected:

        client.ethg++ <--> ethernetline <--> open_flow_switch.ethg++;
        server.ethg++ <--> ethernetline <--> open_flow_switch.ethg++;

        controller.ethg++ <--> ethernetline <--> open_flow_switch.gate_controller
            ++;

        globalFilter.nodes++ <--> client.global_filter;
        globalFilter.nodes++ <--> server.global_filter;
        globalFilter.nodes++ <--> open_flow_switch.global_filter;
        globalFilter.nodes++ <--> controller.global_filter;
}
```

In this step, is fundamental to:

- add the the string parameter `attackConfigurationFile` to the network;
- import the GlobalFilter class, declare a GlobalFilter submodule and connect it to all the other nodes;
- import the ExMachina class and declare an ExMachina submodule if *disable* action is performed. The module does not need to be connected to the rest of the nodes;
- import and declare an **Open_Flow_Switch_SEA** submodule. This is the extended version of a simple open flow switch including SEA++ mechanism;
- import and declare an **Open_Flow_Controller_SEA** submodule. This is the extended version of SDN controller node including the SEA++ mechanism;
- use the *FlatNetworkConfigurator* to assign IP addresses to the nodes. The usage of IPv4NetworkConfigurator in SDN architectures is under development.

All the SDN-related components will be found under the */openflow* folder.

3rd step - edit omnetpp.ini As in INET, the 3rd step is to edit the `omnetpp.ini` file. In this step is fundamental to bind the configuration(s) with the ACF(s), by overwriting the name of the network parameter `attackConfigurationFile` with the name of a particular ACF.

```
[General]
network = scenario
sim-time-limit = 250s

// General settings
```

```
// ...

// Config(s) specific settings

[Config simple_attack]
**.attackConfigurationFile = simple_attack.xml

[Config disable]
**.attackConfigurationFile = disable.xml

// ...
```

4th step - add the ACF The 4th and last step is to add the ACF(s) in the folder. Write the description of the attack in the `.adl` file. Compile the file using the `interpreter.py` in the `inet_sdn/interpreter/interpreter` folder. The output will be the ACF file in `.xml` format.

```
~/seapp_stable/examples/inet_sdn/scenario ../../../interpreter/interpreter/
    interpreter.py -i simple_attack.adl -o simple_attack.xml
```

Run the simulation The simulation is ready to run. In the terminal, run the bash script `run.sh` which calls `run_inet` in the `src` folder:

```
~/seapp_stable/examples/inet_sdn/scenario ./run.sh
```

It will start the simulation supported by the GUI, as shown in figure 4.2.

To run the simulation in express mode without using the GUI, type the following by specifying the configuration:

```
~/seapp_stable/examples/inet_sdn/scenario ../../../src/run_inet -u Cmdenv -c
    general
```

or :

```
~/seapp_stable/examples/inet_sdn/scenario ../../../src/run_inet -u Cmdenv -c
    simple_attack
```

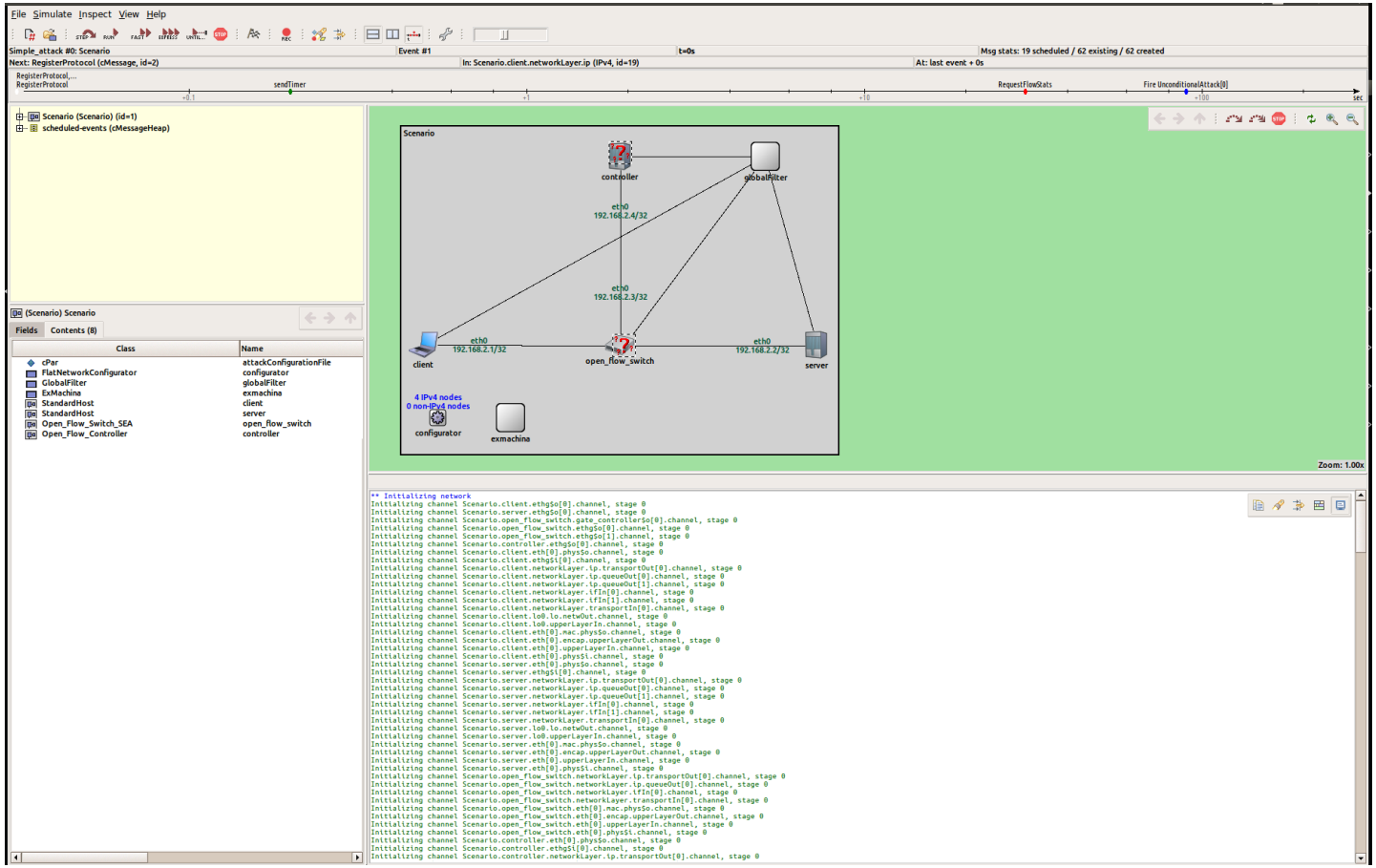

Figure 4.2: Simulation SDN scenario

Chapter 5

Create and fill packets

This chapter shows informations that may help the user to create and fill a packet properly. To build a packet from scratch that belongs to a certain layer, the user has to know its structure and the protocol that runs on the layer below or above. The user has also to know the output gate of the local filter (or local filters) to which forward the packet.

5.1 Create new packet

To create a new packet, the user has to define the ADL type of the packet in the *create* action. The table 5.1 presents the assigned ADL types to the existing application packets and the control info objects which are attached to them.

In case of a new type of packet which is not supported by SEA++, the user has to extend the implementation of SEA++ by adding the new type. To do so, the *Create.h*, *Create.cc* and the *seapputils.cc* files has to be modified. More in details:

- Create a new application packet (*.msg*)
- Extend the *type_t* enum class in the */actions/create/Create.h* file by writing the name of the new packet;
- Extend the *buildNewPacket(cPacket** packet, int layer, type_t type)* method of */actions/create/Create.cc* file to create a new application packet;
- Extend the *getPacketLayer(cPacket* packet)* method of */util/seapputils/seapputils.cc* to return the layer of the new packet.

5.2 Handle ControlInfo object

After the creation of a packet, the user has to fill its header by using the action change. In some cases the user has also to fill the fields contained in the `ControlInfo` object appended to packets. The `ControlInfo` object contains commands and informations that are used by the recipient layer to handle properly the incoming packets.

The tables below show the packets that the user can create. Some packets are associated with the related ControlInfo object by default.

Example If the user wants to create a generic packet of layer 5 and send it to the bottom layer, he must know the protocol that runs on the layer 4, for example UDP. By analyzing the table 5.1, which specifies the structure of the ControlInfo object, the user finds the field which has to fill: `sockId`, `destAddr`, `destPort`, `srcAddr`, `interfaceId`.

5.3 Handle output gate

When the user creates a new packet, it has to specify the output gate in the local filter by using the action *change* and the keyword **sending.outputGate**. For example, if the user creates an application packet that flows in reception direction, it has to specify the gate `app_udp_sup$o[0]`.

Listing 5.1: Handle output gate example

```
create(newPacket, ...)
...
change(newPacket, "sending.outputGate", "app_udp_sup$o[0]")
```

The picture below indicates the gates of the local filter:

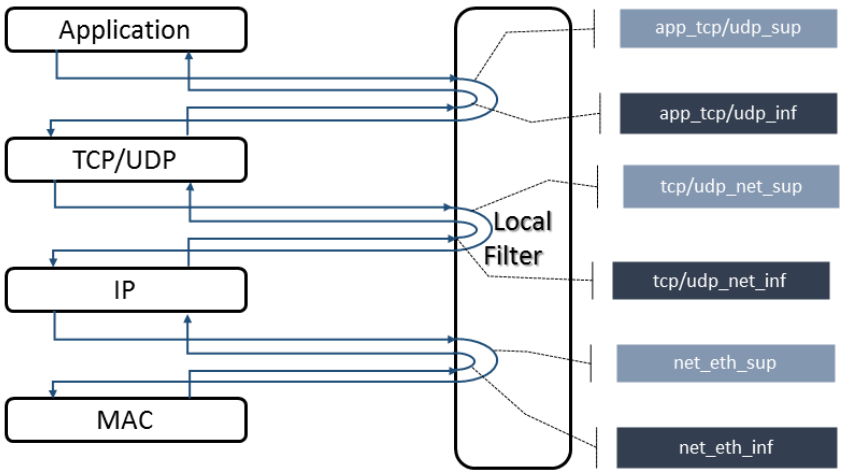

Figure 5.1: Gates of Local Filter

Table 5.1: ControlInfo object structure

From layer 5 to layer 4			
ASL type	**Packet type**	**ControlInfo object**	**fields**
APP.0000	cPacket	UDPSendCommand	sockId destAddr destPort srcAddr interfaceId
APP.0100	cPacket	TCPSendCommand	connId userId
APP.0201	TimingReport	UDPSendCommand	sockId destAddr destPort srcAddr interfaceId
APP.0301	TimingCommand	UDPSendCommand	sockId destAddr destPort srcAddr interfaceId

Table 5.2: ControlInfo object structure

From layer 4 to layer 5			
ASL type	**Packet type**	**ControlInfo object**	**fields**
APP.0000	cPacket	UDPDataIndication	sockId srcAddr destAddr srcPort destPort ttl interfaceId typeOfService
APP.0100	cPacket	TCPCommand	connId userId

Table 5.3: ControlInfo object structure

From layer 4 to layer 3			
ASL type	**Packet type**	**ControlInfo object**	**fields**
TRA.0000	UDPPacket	IPv4ControlInfo	destAddr srcAddr interfaceId multicastLoop protocol typeOfService timeToLive dontFragments nextHopAddr moreFragments macSrc macDest diffServCodePoint explicitCongestionNotification
TRA.0010	TCPSegment	IPv4ControlInfo	destAddr srcAddr interfaceId multicastLoop protocol typeOfService timeToLive dontFragments nextHopAddr moreFragments macSrc macDest diffServCodePoint explicitCongestionNotification

Table 5.4: ControlInfo object structure

From layer 3 to layer 4			
ASL type	**Packet type**	**ControlInfo object**	**fields**
TRA.0000	UDPPacket	IPv4ControlInfo	destAddr srcAddr interfaceId multicastLoop protocol typeOfService timeToLive dontFragments nextHopAddr moreFragments macSrc macDest diffServCodePoint explicitCongestionNotification
TRA.0010	TCPSegment	IPv4ControlInfo	destAddr srcAddr interfaceId multicastLoop protocol typeOfService timeToLive dontFragments nextHopAddr moreFragments macSrc macDest diffServCodePoint explicitCongestionNotification

Table 5.5: ControlInfo object structure

From layer 3 to layer 2			
ASL type	**Packet type**	**ControlInfo object**	**fields**
NET.0000	IPv4Datagram	none	none
NET.0010	IPv4Datagram	Ieee802Ctrl	src dest etherType interfaceId switchPort ssap dsap pauseUnits

Table 5.6: ControlInfo object structure

From layer 2 to layer 3			
ASL type	**Packet type**	**ControlInfo object**	**fields**
NET.0000	IPv4Datagram	-	-
NET.0010	IPv4Datagram	Ieee802Ctrl	src dest etherType interfaceId switchPort ssap dsap pauseUnits

Table 5.7: ControlInfo object structure

From layer 2 to layer 1			
ASL type	**Packet type**	**ControlInfo object**	**fields**
MAC.0000	PPPFrame	-	-
MAC.0010	EthernetFrame	-	-
MAC.0020	IdealAirFrame	-	-
MAC.0030	AirFrame	-	-

Table 5.8: ControlInfo object structure

From layer 1 to layer 2			
ASL type	**Packet type**	**ControlInfo object**	**fields**
MAC.0000	PPPFrame	-	-
MAC.0010	EthernetFrame	-	-
MAC.0020	IdealAirFrame	-	-
MAC.0030	AirFrame	-	-