ASF++ 1.1 User Manual

Marco Tiloca[2]
Francesco Racciatti[1]
Alessandro Pischedda[1]
Gianluca Dini[1]

September 15, 2016

1 Dept. of Computer Engineering, University of Pisa, Largo Lazzarino 1, 56100 Pisa, Italy

2 SICS Swedish ICT AB, Security Lab, Isafjordsgatan 22, SE-164 40 Kista, Sweden

Contacts:

marco@sics.se · write to Marco Tiloca

racciatti.francesco@gmail.com · write to Francesco Racciatti

alessandro.pischedda@gmail.com · write to Alessandro Pischedda

gianluca.dini@ing.unipi.it · write to Gianluca Dini

Abstract

This document is the official ASF++ user manual. ASF++ a tool to simulate the effects of cyber-physical attacks in WSNs and BANs.

Contents

1 Installing ASF++

ASF++ is based on OMNeT++ 4.x and Castalia 3.x and it is designed to operate on Linux distributions based on Debian. It also requires the library `libxml`.

ASF++ was succesfully tested under the following conditions:

- Ubuntu 14.04 LTS;
- Python 2.7.6;
- g++ compiler 4.9.2;
- OMNeT++ 4.6;
- Castalia 3.2;

This manual refers to an environment based on the conditions listed above.

The steps to install ASF++ are the following:

1. satisfy the prerequisites;
2. get and install the library `libxml`;
3. get and install the framework OMNeT++ ;
4. get and install ASF++ .

1.1 Prerequisites

ASF++ requires at least Python 2.7.6 and g++ 4.9.2. Ubuntu 14.04 comes with them.

Anyway, you can install the package build-essential by following commands in terminal:

```
~$ sudo add-apt-repository "deb http://archive.ubuntu.com/ubuntu $(lsb_
    release -sc) main universe"
~$ sudo apt-get update
~$ sudo apt-get install build-essential
```

1.2 Library `libxml`

The instructions to install the library `libxml` can be found at `http://libxmlplusplus.sourceforge.net/docs/manual/html/index.html`. The sources are available at `http://libxmlplusplus.sourceforge.net/`.

You can install the library `libxml` by following command in terminal:

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

1.3 OMNeT++

ASF++ is built on Castalia 3.2 and OMNeT++ 4.6. It is advisable to install at least the version 4.6 of OMNeT++ . You can obtain the `tgz` file at `https://omnetpp.org/omnetpp/summary/30-omnet-releases/2290-omnet-4-6-source-ide-tgz` (you have to pass the Captcha protection test). The download will take a while.

Untar and unzip the downloaded file:

```
~$ tar xvfz omnetpp-4.6-src.tgz
```

It creates the folder `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 two export commands to `.bash_profile` file.

Now you can build OMNeT++ 4.6:

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

It will take a while.

If something goes wrong, use the OMNeT++ 4.6 manual as a reference. You can get the OMNeT++ 4.6 manual at `http://www.omnetpp.org/documentation`.

1.4 ASF++ (and Castalia)

ASF++ is bundled with Castalia 3.2.

You can get ASF++ sources at `https://github.com/asfpp/asfpp`

Unzip it in the folder `asfpp`:

```
~$ unzip asfpp-master.zip -d asfpp
```

Now you can build it:

```
~$ cd asfpp
~/asfpp$ ./makemake
~/asfpp$ make
```

It will take a while.

At last, check if all files in the folder `bin` have the execution permission. If not set it by following in terminal:

```
~/asfpp$ cd bin
~/asfpp/bin$ chmod u+x Castalia
~/asfpp/bin$ chmod u+x CastaliaPlot
~/asfpp/bin$ chmod u+x CastaliaResult
```

For a better user experience, we suggest to export the environment variable `/asfpp/bin`:

```
~$ export PATH=$PATH:~/asfpp/bin
```

Add the command above to the `.bash_profile` file:

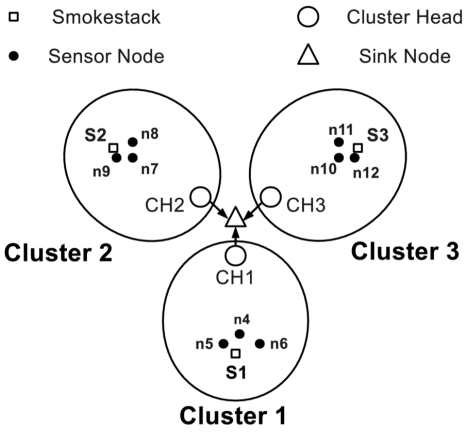

Figure 1: Chimney scenario

2 Using ASF++

This section describes how to succesfully perform simulations of cyber-physical attacks. Before running simulations, you have to build ASF++ succesfully (see section 1).

2.1 Run your first attack

ASF++ provides a set of ready-to-use simulation scenarios. We will use `chimneys-3smokes`:

```
~/asfpp $ cd Simulations/chimneys-3smokes
```

Folder The folder `chimneys-3smokes` contains the following files and folders:

```
~/asfpp/Simulations/chimneys-3smokes ls
attacks chimneys-3smokes.pdf omnetpp.ini results run.sh scripts
```

Files:

- `chimneys-3smokes.pdf` shows the simulation environment;
- `omnetpp.ini` is the `ini` configuration file;
- `run.sh` is a useful script that automatize the simulation process and the subsequent calculations.

Folders:

- `attacks` contains the attacks to permorm, it is provided with a set of redy-to-use attacks (`xml` with related `adl` files);
- `scripts` contains the scripts used to perform the calculations to produce the values in which we are interested.
- `results` contains the simulative results, it is provided with a set of already-executed simulations's results.

Scenario Figure 1 shows the referred scenario. It consists of an industrial area in which three independent plants release pollutant into the air, through smokestack S1, S2 and S2 respectively. In the field there is also

a WSN. It monitors pollution levels and detects any plants's abnormal behavior.

The WSN is composed by one sink and three clusters. Each cluster is associated with one smokestack and consists of one *cluster head* and three *sensor nodes*.

Each sensor node periodically senses the pollution in its own cluster and sends reports to its own cluster head. Periodically, each cluster head computes the average pollution level according to the received reports and delivers it to the sink node. Finally, the sink node checks whether any report exceeds a given threshold.

Configurations You can see all the possible configurations by following command: [1]

```
~/asfpp/Simulations/chimneys-3smokes $ Castalia

List of available input files and configurations:

* omnetpp.ini
  General
  misread
  misplace
  injection
```

There are 4 different configurations. `General` is the Castalia's standard configuration, i.e. a configuration without cyber-physical attacks. `misread`, `misplace` and `injection` refer different attacks.

To automate the plotting of simulations's results, it is provided a bash script `run.sh`.

2.1.1 General (no attack)

The first run refers to the configuration `General`. If you are interested in the correctness of data, you can reapeat the simulation for N times, for example 5:

```
~/asfpp/Simulations/chimneys-3smokes$ ./run.sh

List of available input files and configurations:

* omnetpp.ini
  General
  misread
  misplace
  injection

Using input file 'omnetpp.ini'
Enter the 'Config' to run: General
Using configuration 'General'
Enter the number of repetitions: 5
```

The script `run.sh` moves all the simulative results into a new folder, for example `General-20151014-172931`. The name of the new folder is that of the configuration that was used to which it is appended a timestamp.

Figure 2 (stored into the folder `plot`) shows the results of the simulation just performed. The threshold (dashed red line) is fixed at 50 $\frac{\mu g}{m^3}$. The graph shows that emissions from S1 are exceeding the threshold.

[1] If you have not exported the environment variable `/asfpp/bin` as explained in the section 1.4, you have to specify the entire relative path `../../bin/Castalia`.

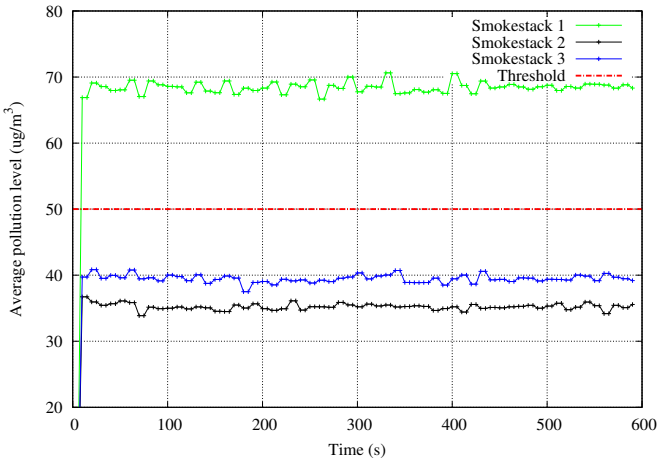

Figure 2: Results of configuration `General`

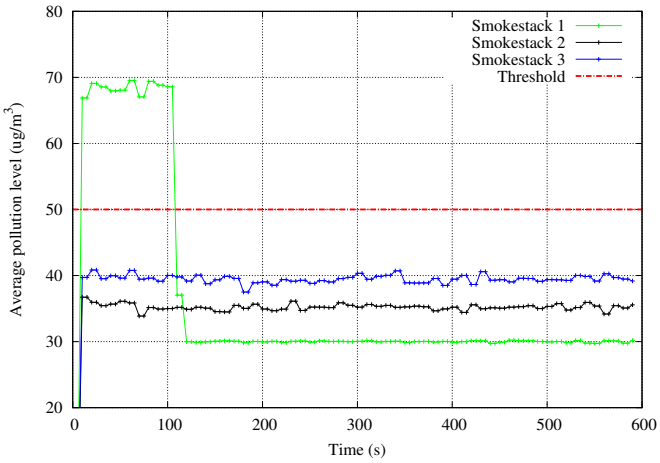

Figure 3: Results of configuration `misread`

2.1.2 misread

This is a *physical* attack where sensor readings of 4, 5 and 6 tampered with the aim to lowering the value of their measurements.

```
~/asfpp/Simulations/chimneys-3smokes$ ./run.sh

List of available input files and configurations:

* omnetpp.ini
  General
  misread
  misplace
  injection

Using input file 'omnetpp.ini'
Enter the 'Config' to run: misread
Using configuration 'General'
Enter the number of repetitions: 5
```

Figure 3 shows the results of the simulation just performed. By performing the attack `misread`, the emissions from S1 appear to be regular.

2.1.3 misplace and injection

You can run other ready-to-use attacks as explained above.

2.2 Run your own attack

Before running your own attack, you must follow the following steps:

1. edit an `adl` file by using any text editor;
2. interpret the `adl` file by using the Attack Description Interpreter (ADI);
3. edit the file `omnetpp.ini` to include a configuration that refers the new attack.

Write your own `adl` file To write your own attack you must know the Attack Description Language (ADL), see section 3. You can write a new attack by editing editing a new `adl` file. For example, we can create the file `destroy.asl` that contains the description of an attack that destroys nodes 4, 5 and 6.

```
# destroy attack
destroy(4, 100)
destroy(5, 100)
destroy(6, 100)
```

Interpret the `adl` file After creating the file that contains the description of the attack, you have to intepret it by using the ADI. The general sintax to invoke the ADI is the following:

```
python path/interpreter.py -i <file.adl> -o <file.xml>
```

The input file is mandatory. You can specify the name of the `xml` file, that is the output of the ADI. If the file `destroy.adl` is stored in `/asfpp/Simulations/chimneys-3smokes/attacks`, to interpret it type:

```
~/asfpp/Simulations/chimneys-3smokes/attacks$ python ../../../interpreter
    /interpreter/interpreter.py -i destroy.adl
Using default output filename 'destroy.xml'
```

The output file is `destroy.xml`.

Edit the file `omnetpp.ini` The last step is to edit the configuration file `omnetpp.ini`. It is composed of several parts, each of which describes a particular configuration. The firsts part is always referred to the configuration `General`, which is mandatory. After the configuration `General`, you can describe other configurations.

To describe a new configuration that includes the file `destroy.xml`, you have to append to the configuration file the following lines:

```
[Config destroy]
SN.configurationFile ="attacks/destroy.xml"
```

For a better understanding about the configuration file, see the user manuals of OMNeT++ and Castalia.

Run the simulation Finally you can run the simulation that includes the attack destroy.

3 Attack Description Language

The Attack Description Language (ADL) is a very simple and flexible high-level language. It provides to users a set to primitives that can be combined to describe cyber-physical attacks.

3.1 Attacks

ASF++ conceives an attack as a ordered sequence of events. Events are described by using the primitives provided by the ADL. Each primitive produces a particular effect. Primitives are organized into two sets, as described below.

i) *node primitives*, that account for physical attacks, and allow users to describe alterations in node behavior. In particular, the node primitives are:
   - `destroy`, destroys a node from the sensor network;
   - `disable`, disables a node's NIC;
   - `move`, misplaces a node;
   - `fakeread`, tampers a node's sensor readings.

ii) *message primitives*, that account for cyber attacks, and allow users to describe actions on network messages. In particular, the message primitives are:
   - `drop`, discards a certain packet;
   - `create`, creates from scratch a new packet of a certain type;
   - `clone`, creates a perfect copy of a given packet;
   - `retrieve`, inspects a certain field of a given packet;
   - `change`, changes the content of a particular field of a given packet;
   - `send`, sends a packet (one created from scratch) to the bottom layer in the communication stack;
   - `put`, sends a certain packet to a list of recipient nodes.

The user can combine the primitives to describe complex attacks. By using node primitives, the user can describe *physical attacks*, e.g. destruction of nodes or alteration of sensed values. By using message primitives, the user can describe *cyber attacks*, e.g. dropping, alteration injection or eavesdropping of messages. In addition to primitives, the ADL provides some support structures to users as filter statements to intercept certain messages, loop statements to build cyclic operations, lists, variables, expression solver engines, etc.

3.2 Node primitives

Node primitives act on the nodes and are used to describe physical attacks.

3.2.1 destroy

The primitive `destroy` removes a target node from the simulation field, preventing it from doing any operation.

```
destroy(node_id, occurrence_time)
```

Parameters:

- `node_id` is the id of the target node;
- `occurrence_time` is the time in which the primitive is performed.

3.2.2 disable

The primitive `disable` disables the NIC of a target node, preventing it from taking part in further communications. The node can still continue to perform all the other operations.

```
disable(node_id, occurrence_time)
```

Parameters:

- `node_id` is the id of the target node;
- `occurrence_time` is the time in which the primitive is performed.

3.2.3 move

The primitive `move` misplaces a target node.

```
move(node_id, occurrence_time, coord_x, coord_y, coord_z)
```

Parameters:

- `node_id` is the id of the target node;
- `occurrence_time` is the time in which the primitive is performed;
- `coord_x` is the target x coordinate;
- `coord_y` is the target y coordinate;
- `coord_z` is the target z coordinate.

3.2.4 fakeread

The primitive `fakeread` tampers the sensor readings of a target node.

```
fakeread(node_id, occurrence_time, sensor_id, function)
```

Parameters:

- `node_id` is the id of the target node;
- `occurrence_time` is the time in which the primitive is performed;
- `sensor_id` is the id of sensor (of the target node);
- `function` is the mathematical function used to compute the tampered value.

3.2.5 Message primitives

Message primitives act on messages and are used to describe cyber attacks.

3.2.6 drop

The primitive `drop` discards a target packet.

```
drop(pck, threshold)
```

Parameters:

- `pck` is the target packet;
- `threshold` is the threshold that models the probability of dropping the target packet.

3.2.7 create

The primitive `creates` creates a new packet from scratch. The new packet can contain other encapsulated packets. A single invocation makes it possible to specify the types of all the packets, from the outer one to the inner one.

```
create(pck, layer_n, type_n, layer_n-1, type_n-1, ...)
```

Parameters:

- `pck` is the new packet;
- `layer_n, type_n` is the couple that defines the type of the outer packet that belongs to the layer n;
- `layer_n-1, type_n-1` is the couple that defines the type of the inner packet that belongs to the layer n-1.

3.2.8 clone

The primitive `clone` clones a target packet, i.e. it creates a perfect copy of the target packet.

```
clone(src_pkt, dst_pck)
```

Parameters:

- `src_pck` is the source packet (i.e. the packet to clone);
- `dst_pck` is the destination packet (i.e. the cloned one).

3.2.9 retrieve

The primitive `retrieve` copies the value of a target field (of a target packet) into a variable.

```
retrieve(pck, field_name, variable_name)
```

Parameters:

- `pck` is the packet to inspect;
- `field_name` is the name of the target field;
- `variable_name` is the name of the variable in which to store the retrieved value.

3.2.10 change

The primitive `change` changes the value of a target field (of a target packet).

```
change(pck, field_name, variable_name)
```

Parameters:

- `pck` is the target packet;
- `field_name` is the name of the target field;
- `variable_name` is the name of the variable which stores the new value.

3.2.11 send

The primitive `send` sends a packet that was created from scratch to the next layer in the communication stack . It must be used with packet created by usign the actions `create` or `clone`.

```
send(pck, delay)
```

Parameters:

- `pck` is the packet to send;
- `delay` is the forwarding delay.

3.2.12 put

The primitive `put` puts a certain packet in the RX or TX buffer of a set of nodes.

```
put(pck, dst_nodes, TX|RX, TRUE|FALSE, delay)
```

Parameters:

- `pck` is the packet to put;
- `dst_nodes` is the list of nodes;
- `TX|RX` defines in which buffer to put the packet (in the TX buffer or in the RX buffer);
- `TRUE|FALSE` defines in use or not the wireless channel as communication channel;
- `delay` is the forwarding delay.

3.3 Support structures

The ADL provides a set of support structures to allow the user to describe even complex attacks.

3.3.1 Variables, lists and packets

Variables The ADL is dynamically typed. It provides the keyword `var` to declare and initialize variables.

```
var foo
```

Variables must be declared before use.

If a variable is declared and initialized inside a loop statement, the initialization is carried out only in the first iteration.

```
<loop_statement_begin>
var foo = 0
...
foo += 1
<loop_statement_end>
```

Tha var foo is initialized to 1. At the end of the 1st iteration its value is 1, at the end of the 2nd iteration its value is 2, at the end of the 3rd iteration its value is 3, and so on.

Lists The ADL provides the keyword `list` to declare lists of values.

```
list nodes_list = {1, 2, 3, 4, 5}
```

Lists must be declared before use.

Packets The ADL provides the keyword `packet` to declare a reference to packets.

```
packet my_pck
```

3.3.2 Functions

The ADL provides the keyword `function` to declare mathematical functions to be evaluated by nodes at runtime. The user can use 2 independent variables:

- `t` is the simulation time, evaluated at runtime;
- `s` is the sensor reading value, evaluated at run time.

At the moment, functions can be used only if combined with the primitive `fakeread`.

```
function f1 = "sin(2*pi*t) + 0.75*s"
fakeread(1, 100, 0, f1)
```

The independet variable `s` refers the sensor reading value of the sensor 0 of the node 1.

3.3.3 Expressions

An expression is an ordered collection of operands, namely numbers, strings or variables, and operators. Figure 1 shows the ADL expression table.

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

operator	numbers	strings
=	supported	supported
+ =	supported	not supported
− =	supported	not supported
× =	supported	not supported
/ =	supported	not supported
÷ =	supported	not supported

(a) Assignment operators

operator	numbers	strings
+	supported	supported
−	supported	not supported
×	supported	not supported
/	supported	not supported
÷	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 1: ADL expression table

3.3.4 Loop statements

The ADL provides a *loop statement* that allows the user to specify the occurrence of a list of events described through message primitives. Events occurrence may depend or not on the evaluation of a conditional statement. It is possible to use two types of loop statemens:

i) *conditional loop statement*, if the execution of the message primitives contained in the loop statement depends on the evaluation of a conditional statements;

ii) *unconditional loop statement*, if the message primitives contained in the loop statement are executed unconditionally.

Conditional loop statement A conditional loop statement has the structure that follows.

```
from T nodes in <nodes_list> do {
<packet_filter>
    <List of message primitives>
}
```

The execution of the message primitives contained inside a conditional loop statement depends on the evaluation of a *packet filter*.

Packet filter The packet filter is a set of simple boolean conditions c1, c2, ..., cN joined by logic operators, i.e. AND, OR, and NOT. Each condition is related to a certain field of a certain type of packet. Accordingly, the packet filter is evaluated by nodes at runtime. Nodes inspect packets flowing through their communication stack. If a packet matches the packet filter, the message primitives contained in the conditional loop statements are executed.

The ADL provides the keyword `filter` to describe the packet filter.

```
filter (layer5.source == 10 AND layer4.sourcePort == 1000)
```

In the example above, all packets belonging (at least) to the layer 4 of the TCP/IP stack having the field `source` in the layer 5 setted to 10 and the field `sourcePort` of the layer 4 setted to 1000 match the packet filter. The ASF++ provides the keyword `original` to refer the original packet intercepted by the packet filter.

What follows is an example of conditional loop statement. Starting from time 200 s, nodes 1, 2, and 7 intercept all packets travelling through their communication stack having the the field `sourcePort` of layer 4 setted to 1000. Then, such nodes check if each intercepted packet satisfies the packet filter. In case of a positive match, the message primitives contained in the conditional loop statements are executed.

```
list nodes_list = {1, 2, 7}
...
from 200 nodes in nodes_list do {
filter(layer4.sourcePort == 1000)
    <List of message primitives>
}
```

Note that the execution frequency of the conditional loop statements can't be setted by the user but it depends on the dynamic of the simulation (i.e. the frequency with which packets are generated) and on the evaluation at runtime of the packet filter.

3.3.5 Unconditional loop statements

Unlike conditional loop statements, the unconditional are not related to interception of packets by network nodes. However, new packets can be created, and possibly cloned, in order to be naughtly injected into the network. The user must specify the time `T` starting from which the attack takes place, and the occurrence period `P` according to which the attack has to be repeatedly reproduced over time.

The unconditional loop statements concern the network nodes but do not run directly from them. ASF++ provides a particular node, namely the Global Filter, which is not part of the network and has the task to execute the attacks based on the unconditional loop statements.

An unconditional loop statement has the structure that follows.

```
from T every P do {
    <List of message primitives>
}
```

What follows is an example of unconditional loop statement. Starting from time 200 s, every 0.1 s (10 Hz), the Global Filter executes the list of message primitives.

```
from 200 every 0.1 do {
    <List of message primitives>
}
```

Note that the user can set the execution frequency (even to 0 to obtain a one-time execution) of the unconditional loop statements.

3.4 Attack types

By using the primitives and the support structures provided by the ADL (in particular the loop statements), the user can describe three different types of attacks:

i) *physical attacks*, that consist of a single node primitive;

ii) *conditional attacks*, that consist of a set of message primitives contained in conditional loop statements;

iii) *unconditional attacks*, that consis of a set of message primitives contained in unconditional loop statements.

Physical and conditional attacks are performed directly by nodes, unconditional ones instead are performed by a particular node provided by ASF++ which does is not part of the network.

3.4.1 Physical attacks

A physical attack consists of a single node primitive and is triggered directly by the node indicated in the primitive.

```
destroy(1, 150)
```

In the example above, the node 1 is destroyed at time 150. In particular, it is the node itself that triggers its own destruction.

3.4.2 Conditional attacks

A conditional attack consists of a set of message primitives contained in a conditional loop statement. Conditional attacks are performed directly by nodes specified in the conditional loop statement. As specified above, a node performs the primitives contained in the loop statement only if the current packet matches the packet filter.

```
list nodes_list = {1, 3, 7}
...
from 150 nodes in nodes_list do {
filter (layer4.sourcePort == 1000)
    var sourcePort = 1500
    change(original, layer4.sourcePort, sourcePort)
}
```

In the example above, from time 150, nodes 1, 3 and 7 change the value of the field `sourcePort` of the layer 4 from 1000 to 1500, only if the current packet matches the packet filter. The current packet for nodes 1, 3 and 7, i.e. the packet passing through the communication stack of nodes 1, 3 and 7 respectively, matches the packet filter if the value of the field `sourcePort` of the layer 4 is 1000. The keyword `original` refers the original packet intercepted by the packet filter.

3.4.3 Unconditional attacks

An unconditional attack consists of a set of message primitives contained in an unconditional loop statement. Unconditional attacks are not performed directly by nodes but are executed by the Global Filter, a particular node provided by ASF++ which is not part ot the newtork.

```
list nodes_list = {1, 3, 7}
...
from 150 every 0.1 do{
    # declare a packet
    packet my_packet
    # declare and initialize variables
    var sensorId = 0
    var value = 50
    var seqNumb = 1000
    # create a new packet from scratch
    create(my_packet, layer5, 0001)
    # set values in the packet fields
    change(my_packet, layer5.sensorId, sensorId)
    change(my_packet, layer5.seqNumb, seqNumb)
    change(my_packet, layer5.value, value)
    # put the new packet into the RX buffer of nodes 1,3,7, without
        forwarding delay
    put(my_packet, nodes_list, RX, 0)
    # update seqNumb
    seqNumb += 1
}
```

In the example above, from time 150, the Global Filter creates a new packet, sets its fields appropriately and puts the packet in the RX buffer of nodes 1, 3 and 7.

3.4.4 Expected knowledge

In general, to evaluate the effects of fairly simple attacks, ASF++ requires to the user only the knowledge about the network topology. Instead, to perform more complex attacks that involves the creation of packets and

the alteration of their fields (even the creation of protocols's signaling messages), ASF++ requires to the user the knowledge of the:

- communication stack (encapsulation/decapsulation mechanisms);
- structure of packets (headers and payload);
- protocols running on each layer.

However, given that the majority of attacks (except those special ones) does not require any in-depth knowledge, ASF++ is a very user friendly tool.

3.5 Attack description

3.5.1 ADI output

The following examples show how the Attack Description Interpreter interprets physical, conditional and unconditional attacks.

Interpreted physical attack The listing below describes a physical attack in which the readings of the sensor 0 of the node 2 are tampered from time 150 by using the function f1.

```
function f1 = "sin(2*pi*t) + 0.75*s"
fakeread(2, 150, 0, f1)
```

The physical attack above is interpreted by the ADI as follows:

```
<configuration>
    <Physical>
        <Attack>
            <start_time>150</start_time>
            <node>2</node>
            <action>
                <name>Fakeread</name>
                <parameters>sensor_id:0:function:sin(2*pi*t) + 0.75*s</
                    parameters>
            </action>
        </Attack>
    </Physical>
</configuration>
```

Interpreted conditional attack The listing below describes a conditional attack in which, from time 150, nodes 1, 3 and 7 change the value of the field `sourcePort` of the layer 4 from 1000 to 1500, only if the current packet matches the packet filter.

```
list nodes_list = {1, 3, 7}
from 150 nodes in nodes_list do {
filter (layer4.sourcePort == 1000)
    var sourcePort = 1500
    change(original, layer4.sourcePort, sourcePort)
}
```

The conditional attack above is interpreted by the ADI as follows:

```
<configuration>
    <Conditional>
        <Attack>
            <start_time>150</start_time>
            <node>1:3:7</node>
            <var><name>sourcePort</name><value>1500</value><type>NUMBER</
                type></var>
```

```
            <filter>[:layer4.sourcePort:==:1000:]</filter>
            <action>
                <name>Change</name>
                <parameters>packetName:original:field_name:layer4.
                    sourcePort:value:sourcePort</parameters>
            </action>
        </Attack>
    </Conditional>
</configuration>
```

Interpreted unconditional attack The listing below describes an unconditional attack in which, from time 150, the Global Filter creates a new packet, sets its fields appropriately and puts the packet in the RX buffer of nodes 1, 3 and 7.

```
list nodes_list = {1, 3, 7}
from 150 every 0.1 do{
    # declare a packet
    packet my_packet
    # declare and initialize variables
    var sensorId = 0
    var value = 50
    var seqNumb = 1000
    # create a new packet from scratch
    create(my_packet, layer5, 0001)
    # set values in the packet fields
    change(my_packet, layer5.sensorId, sensorId)
    change(my_packet, layer5.seqNumb, seqNumb)
    change(my_packet, layer5.value, value)
    # put the new packet into the RX buffer of nodes 1,3,7, without
        forwarding delay
    put(my_packet, nodes_list, RX, 0)
    # update seqNumb
    seqNumb += 1
}
```

The conditional attack above is interpreted by the ADI as follows:

```
<configuration>
    <Unconditional>
        <Attack>
            <start_time>150</start_time>
            <frequency>0.1</frequency>
            <var><name>seqNumb</name><value>1000</value><type>NUMBER</type
                ></var>
            <var><name>sensorId</name><value>0</value><type>NUMBER</type></
                var>
            <var><name>1</name><value>1</value><type>NUMBER</type></var>
            <var><name>value</name><value>50</value><type>NUMBER</type></
                var>
            <action>
                <name>Create</name>
                <parameters>packetName:my_packet:layer5..type:1</parameters
                    >
            </action>
            <action>
                <name>Change</name>
                <parameters>packetName:my_packet:field_name:layer5.sensorId
                    :value:sensorId</parameters>
            </action>
            <action>
                <name>Change</name>
                <parameters>packetName:my_packet:field_name:layer5.seqNumb:
                    value:seqNumb</parameters>
            </action>
            <action>
                <name>Change</name>
```

```
            <parameters>packetName:my_packet:field_name:layer5..value:
                value:value</parameters>
        </action>
        <action>
            <name>Put</name>
            <parameters>packetName:my_packet:nodes:1|3|7:direction:RX:
                false:delay:0</parameters>
        </action>
        <action>
            <name>Expression</name>
            <item>1</item>
            <item>+=</item>
            <item>seqNumb</item>
        </action>
    </Attack>
  </Unconditional>
</configuration>
```

Note that expressions are handled by ASF++ as additional events.