Chapter 7

SEA++: A Framework for Evaluating the Impact of Security Attacks in OMNeT++/INET

Marco Tiloca, Gianluca Dini, Francesco Racciatti, and Alexandra Stagkopoulos

Abstract This chapter presents SEA++, a simulation framework that extends OMNeT++/INET for evaluating the impact of security attacks on networks and applications in a flexible and user-friendly way. To this end, SEA++ relies on two fundamental building blocks. First, the user describes the attacks to be evaluated by using a high-level Attack Specification Language (ASL). In particular, only the final effects of such attacks are described, rather than their actual performance. Second, the Attack Simulation Engine (ASE) takes these high-level descriptions as input and accordingly injects attack events at runtime, by means of additional software modules that seamlessly and transparently operate with the other INET modules. This allows the user to quantitatively assess the impact of cyber/physical attacks in simulated network scenarios, and hence rank them according to their severity as a support to risk assessment and selection of countermeasures. As a further advantage, the user is not required to alter any software module or application, or to implement any adversary model for the actual execution of security attacks. Finally, this chapter also includes a step-by-step explicative example showing how to setup and use SEA++ for describing attacks and assessing their impact.

Marco Tiloca (✉)

Security Lab - RISE SICS, Isafjordsgatan 22, Kista, Sweden, e-mail: marco.tiloca@ri.se

Gianluca Dini

Department of Information Engineering - University of Pisa, Via Girolamo Caruso 16, Pisa, Italy, e-mail: gianluca.dini@unipi.it

Francesco Racciatti

Department of Information Engineering - University of Pisa, Via Girolamo Caruso 16, Pisa, Italy, e-mail: racciatti.francesco@gmail.com

Alexandra Stagkopoulos

Security Lab - RISE SICS, Isafjordsgatan 22, Kista, Sweden, e-mail: alstagop@gmail.com

7.1 Introduction

Computer networks are a fundamental component for a wide range of systems and applications, including sensor networks, smart environments, and critical infrastructures. These networked (systems of) systems are also exposed to several cyber and physical security attacks against their infrastructure and the communication between their components. Conducting a risk assessment process is therefore vital to identify potential threats and risks against the system, and to provide indications on how to mitigate them to an acceptable level already at design time.

However, it is not viable to fully address all possible attacks to the maximum possible extent, i.e., achieving “perfect” security, due to technical, performance, and economical reasons. Therefore, it is especially important to clearly identify the attacks with the most severe effects on the system and network infrastructure. In particular, it is important to have a clear understanding on the expected effects of successful attacks, in order to rank them based on their severity and then accordingly prioritize security countermeasures to adopt.

One approach to achieve this goal is the use of software simulation tools such as OMNeT++/INET. This is particularly convenient as it avoids the impractical alternative of conducting security experiments on real (large scale) systems, possibly already operating. In addition, simulative analysis does not require the deployment of a real networked system at all, thus enabling a thorough assessment of attack effects already at the design phase. Furthermore, simulation is much more feasible to use for studying large-scale and real systems, with respect to alternative approaches based on analytical models or testbeds.

But when it comes to evaluating security attacks in networked systems, available simulation tools are often limited and inflexible. That is, the user is typically provided with a limited set of pre-defined attacks to mount against the network scenario, or instead one has to actually *implement* the desired attack execution and the adversary behavior as additional/extended modules in the simulation tool. This evidently undermines usability as well as flexibility and requires to rebuild the simulation tool in order to evaluate different attacks or even just different attack configurations.

This chapter presents SEA++, a simulation framework based on OMNeT++/INET that quantitatively evaluates the effects of security attacks against communication network scenarios in a user-friendly and flexible way. To this end, SEA++ reproduces the final actual effects of successful attacks, regardless of how they have been specifically carried out. That is, the specific way such attacks have been mounted is out of the scope of SEA++, as instead related to a separate earlier assessment of attack feasibility and likelihood.

A considerable advantage of SEA++ is that the user describes the security attacks to evaluate by using a high-level specification language. Thus, the user is not required to implement the actual adversary behavior or the actual attack execution, nor to extend or add software modules in OMNeT++/INET. Instead, the high-level attack description is interpreted and translated into an Extensible Markup Language (XML) attack configuration file, which is finally provided as input to the SEA++ Attack Simulation Engine (ASE). The latter reproduces the effects of the attacks under

evaluation, by seamlessly injecting events at simulation runtime according to the user's high-level description.

SEA++ is particularly convenient to be adopted at the last stage of the risk assessment process, in order to quantitatively evaluate the effects of successful security attacks in networked systems, provided that they have been properly modeled as simulation scenarios. This provides valuable insights on current attack activities, i.e., a tangible and understandable impact on performance and outcome of the target network, expressed through the same metric-based approach used in OMNeT++/INET. This information helps to assess the overall effects of security attacks on networks and applications. In particular, a quantitative evaluation of the effects of security attacks makes it possible to rank them according to their severity, and thus helps in selecting and tuning the most appropriate security controls to address them.

The rest of this chapter is organized as follows. Section 7.2 discusses the security risk assessment process, and briefly overviews existing supporting tools adopting a simulative approach. Section 7.3 presents our SEA++ framework in terms of its overall approach as well as its core components. Section 7.4 provides a walk-through example showing how to set up SEA++ and how to use it in a simple illustrative network scenario. Finally, Sect. 7.5 draws the conclusive remarks for this chapter and discusses potential future research directions for this topic.

7.2 Evaluation of Attack Impact Through Simulation

There are a lot of possible security attacks against networked computing systems. They can have different objectives, be performed at different communication layers or directly against physical nodes, and result in different effects. However, providing a security countermeasure for every possible attack is prohibitive in terms of impact on performance and cost. It is therefore vital to properly evaluate attacks to decide how to address them, and whether to mitigate, eliminate, transfer, or accept them.

This evaluation activity is referred to as *risk assessment*. It comprises the identification and evaluation of risks and risk impacts, and the recommendation of risk-reducing measures [8]. Risk assessment defines and separates the notions of threat, vulnerability, and attack. A *vulnerability* is a flaw or weakness in system security that could result in a security breach, while a *threat* represents the potential for a *threat-source* (i.e., an attacker) to exercise a specific vulnerability. An *attack* is defined as the entire process allowing a threat-source to realize a threat. Therefore, the risk assessment process practically consists of identifying and evaluating the *risks* to system security. In particular, a risk is defined as a function of (i) the *likelihood* of a given threat-source's exercising a particular potential vulnerability; and (ii) the resulting *impact* of that adverse event. Of course, in order to perform attack impact analysis, a thorough *threat model* must have been previously defined. As an example, let us consider a set of attacks $S = \{A_1, A_2, \dots, A_n\}$. Also, let us assume that a given attack A_i is successfully performed with probability P_i and results in an impact E_i . Then, the risk R_i related to that attack can be computed as $R_i = P_i \times E_i$.

Analysis of the impact of an attack can be conducted either *qualitatively* or *quantitatively*. The qualitative approach makes it possible to easily prioritize the considered risks, as well as to identify what areas deserve more attention in addressing vulnerabilities. On the other hand, such a method fails to provide specific quantifiable measurements of impact magnitude. This, in turn, complicates the cost-benefit analysis of any possible security countermeasure. Conversely, the quantitative approach provides an actual measurement of the impact magnitude, and such an information can be effectively used during the cost-benefit analysis of security solutions. However, such a method might make the actual meaning of impact analysis unclear, which would force to interpret results in a qualitative way. In any case, evaluating the impact magnitude should take additional factors into account, such as attack frequency, cost associated to a single attack occurrence, as well as a subjective analysis of specific attack impact.

The presented simulation framework SEA++ supports a quantitative analysis of the impact of cyber-physical security attacks. That is, for any given attack A_i , it focuses on E_i but does not take into account the feasibility likelihood P_i , which is therefore out of scope of this chapter. In particular, SEA++ considers a worst-case scenario and it assumes that all the n attacks under evaluation have been successfully performed, i.e., $P_i = 1 \forall i = \{1, 2, \dots, n\}$. Therefore, a distinctive feature of SEA++ is that it simulates the *effects* of security attacks by reproducing the events that such attacks generate.

SEA++ is not the only tool that strives to evaluate the impact of security attacks, and a number of different approaches have been presented so far. For instance, several alternative tools [13, 17, 19] strive to define analytical models aimed at detecting and contrasting attacks, and only afterwards rely on simulation to validate their own correctness and efficiency. Note that producing an effective analytical model of complex systems can be a considerable challenge in the first place, unless one recurs to practical simplifications that hence risk to misrepresent the network scenario and are additionally hard to be reused. Genge et al. presented AMICI [1], an assessment/analysis platform for multiple interdependent critical infrastructures. In particular, AMICI relies on simulation for the physical system components and an emulation testbed based on Emulab to recreate cyber components [4].

Wang and Bagrodia proposed *SenSec* [18], a framework that simulates the occurrence of security attacks in Wireless Sensor Networks (WSNs) by injecting events into real application simulators. The framework *NETA* [11] is based on OMNeT++/INET and relies on implementing *attacker nodes*, which can strike attacks when triggered at runtime through dedicated control messages. Queiroz *et al.* presented *SCADASim* [3], a simulation tool to test the effect of attacks in Supervisory Control and Data Acquisition (SCADA) systems.

Although it displays similarities with SenSec, NETA, and SCADASim, SEA++ is easier as well as more flexible to use and it displays a number of distinctive features. First, it is based on an off-the-shelf network simulator that was extended, but not modified, by integrating components for the processing of attack events. Good simulator tools are always the result of a large effort, and therefore any modification is preferably avoided. Furthermore, as also mentioned above, SEA++ actually focuses

on the effects of security attacks rather than on their practical execution. Finally, SEA++ does not require the user to implement or customize any component of the simulation platform. As discussed in detail in Sect. 7.3, this is particularly important because it allows us to simply use simulator components, and it does not require to modify them or create new ones.

7.3 The SEA++ Framework

This section describes the SEA++ attack simulation framework. It provides a functional overview of its core idea as well as of its main components. In particular, Sect. 7.3.1 discusses the goals of SEA++ and the approach it adopts to achieve them. Section 7.3.2 to Sect. 7.3.4 describe the three framework components. SEA++ is freely available¹ as open-source software together with a user documentation [15].

7.3.1 Goals and Benefits

SEA++ is an attack simulation framework built on OMNeT++/INET. It enables a flexible and user-friendly quantitative evaluation of cyber and physical security attacks, as to their effects against networks and applications in terms of typical and customizable network performance indicators, such as network delay and throughput. SEA++ supports the design of secure network scenarios, as it facilitates the ranking of security attacks according to the severity of their effects, hence helping to wisely select proper security countermeasures to adopt. The framework is especially intended to expert security and network architects as end users.

The SEA++ framework focuses on the *effects* of security attacks. That is, it does not consider the likelihood of successfully mounted attacks, i.e., the likelihood of a given threat to be exploited, which is expected to be covered as an early attack feasibility assessment in the broader risk assessment process. As a consequence, SEA++ assumes that attacks to be evaluated are successfully carried out, and thus abstracts away from the specific way they are mounted. Consistently, the user is required only to model security attacks in terms of their final and practical effects against the network and application scenario.

As an example, let us consider a deception attack such as the injection of bogus network messages. Then, it is not relevant for SEA++ how an adversary becomes able to inject fake messages in the system, nor the way the actual message injection occurs. SEA++ instead focuses on assessing and quantitatively evaluating what are the final effects of these messages on the network and application after they have been successfully injected, e.g., in terms of delay and throughput variations, or unfulfilled deadlines and unreliable information gathering on the applications.

¹ SEA++ Github repository: https://github.com/seapp/seapp_stable

In order to achieve its goal, SEA++ relies on three fundamental steps which are also summarized in the flowchart depicted in Fig. 7.1. As a first step, the user produces a high-level description of the security attacks to evaluate. To this end, SEA++ provides an Attack Specification Language (ASL), featuring a set of statements and primitive functions to model security attacks (see Sect. 7.3.2). The attack description results in a single *adl* file, which can be possibly stored in a separate attack database for future retrieval and reuse. As a second step, the user provides the *adl* file to the Attack Specification Interpreter (ASI), i.e., a Python script that converts the high-level ASL attack description into a structured XML configuration file (see Sect. 7.3.3). In the final step, the user runs a simulation of the network scenario of interest, which has been previously modeled as an attack-free simulation scenario for OMNeT++/INET. The simulation results in a number of (raw) metric indicators, defined as expected in OMNeT++/INET and to be separately processed with analysis tools.

This third simulation step is driven by the ASE that SEA++ integrates in the INET Framework (see Sect. 7.3.4). Intuitively, the ASE parses the XML configuration file received as input, and accordingly injects additional simulation events at runtime. Such additional events reproduce the effects of the security attacks specified by the user in the original attack description. The ASE achieves this by means of two fundamental modules. On one hand, a Local Event Processor (LEP) module is instantiated on each network node. Every LEP module is able to: (i) intercept network messages flowing through the communication stack of the related node, possibly altering their content; (ii) inject new network messages or duplicate existing ones; and (iii) influence the physical behavior of the network node. On the other hand, a single Global Event Processor (GEP) module is instantiated in order to interconnect the different LEP modules and enable the evaluation of complex distributed attacks.

Note that the user can indeed use vanilla versions of OMNeT++ and the INET Framework as an alternative in order to quantitatively assess the impact of security attacks. However, this would be much less flexible and convenient. In fact, the user would have to implement *ad-hoc* the actual attack performance and effects. Moreover, this would require to rebuild the whole framework before running a simulation. Instead, SEA++ allows the user to conduct the very same task, while more flexibly

Fig. 7.1 Overview of the SEA++ framework flowchart.

and easily describing events through the high-level ASL. Consequently, the user is not required to write source code or to rebuild the simulation framework. At the same time, the conducted simulations still produce metric-based results, i.e., just like if vanilla OMNeT++/INET was used, the impact of security attacks is measurable and processable with well-established means, and hence usable for attack ranking based on severity.

To summarize, the SEA++ framework displays a number of benefits. First, the user is not required to *implement* security attacks and their actual execution to any extent, but rather he/she simply describes their final effects by using the high-level ASL. Second, the user is not required to modify or extend the SEA++ framework either, with particular reference to the ASE module, nor the application modules and the communication modules composing the whole simulation framework. Third, the user can simply take preexisting network and application scenarios as a starting point, considering them as the attack-free baseline for the attack assessment process. Fourth, attack descriptions in the ASL are portable and combinable, as well as store- and retrievable for future reuse. Finally, the overall approach is in principle portable to any simulation environment based on discrete events, at the cost of porting the ASE to the specific target platform.

This approach for simulative attack assessment was initially introduced for an alternative attack simulator built on *Castalia* [2] as described in [5, 6]. It was later extended for OMNeT++/INET towards SEA++ as described in [14, 16].

As a final note, it is worth clarifying that SEA++ is not an omni-comprehensive tool. That is, it should not be intended to fully cover all steps and aspects of the risk assessment process. Instead, it provides a non-exhaustive set of indications and highlights to *support* the final stages of the risk assessment. Such indications come as metric-based results produced and collected through basic means available in OMNeT++/INET, although, as discussed before, achievable in a more flexible and convenient way. Consequently, SEA++ should not be intended to provide a full connection with the overall full spectrum of security consequences on the networked system, or with final measures of the handled damage. To this end, either specific extensions to OMNeT++/INET or alternative tools are required, providing modules that implement, among other things, physical and control processes, billing and accounting processes, as well as specialized control and monitoring protocols. At the time of writing, this kind of support and extensions are out of the scope of SEA++, and the risk assessment process as a whole clearly requires to be supported by additional complementary tools.

7.3.2 Attack Specification Language (ASL)

The high-level ASL allows the user to describe attacks to be evaluated in terms of their final effects. That is, the user assumes that attacks are successfully performed, regardless how an adversary can specifically mount and execute them.

Consistently with the approach adopted by SEA++ and discussed in Sect. 7.3.1, the user describes the successful attacks as a sequence of *events* that atomically take place during the network simulation. In order to specify such events, the ASL provides a collection of primitives organized into two different sets, namely *node primitives* and *message primitives*, which account physical attacks and cyber attacks, respectively. Node primitives are described in Sect. 7.3.2.1, while message primitives are described in Sect. 7.3.2.2.

Besides, the ASL provides statements to specify the occurrence of a *list* of events described by means of message primitives. Specifically, *conditional attacks* and *unconditional attacks* define the execution of a list of message primitives, based on either the evaluation of a condition or a periodic repetition of events, respectively. Conditional attacks are described in Sect. 7.3.2.3, while unconditional attacks are described in Sect. 7.3.2.4. Finally, Sect. 7.3.2.5 discusses syntax contentions relevant for the attack description and the correct reproduction of their effects.

7.3.2.1 Node Primitives

This set includes primitives used to describe physical attacks performed against network nodes. The three primitives in this category are:

- `move (nodeID, t, x, y, z)`
Changes the position of node `nodeID` to the new position `(x,y,z)` at time `t`.
- `destroy (nodeID, t)`
Disconnects the node `nodeID` at time `t`. After that, the node discards all incoming and outgoing network packets, and thus stops taking part in the network communication. Unlike when using the `disable()` primitive, the node remains part of the network scenario, i.e., it continues to run its application(s).
- `disable (nodeID, t)`
Removes the node `nodeID` from the simulation scenario at time `t`. After that, the node cannot take part in the network communication any longer and becomes totally inoperative, i.e., it stops running its application(s).

7.3.2.2 Message Primitives

This set includes primitives used to describe cyber attacks on network packets, including eavesdropping, content altering, data injection, and packet dropping. The seven primitives in this category are:

- `create (packet, field, content ...)`
Creates a new packet `packet` and fill its field `field` with content `content`. A single invocation makes it possible to specify the content of multiple fields.

- `change(packet, field, newContent)`
Changes the content of field `field` of packet `packet` with the `newContent`. `change()` primitive always follows the `create()` action in order to initialize the fields of the newly created packet.
- `clone(srcPacket, dstPacket)`
Clones the packet `srcPacket` into packet `dstPacket`.
- `retrieve(packet, field, variable)`
Retrieves the value of field `field` of packet `packet` and assigns it to the variable `variable`.
- `drop(packet)`
Discards the packet `packet`.
- `send(packet, forwardingDelay)`
Schedules the transmission of a selected packet `packet` produced by `clone()` or `create()` to the bottom layer after a delay of `forwardingDelay`.
- `put(packet, recipientNodes, direction, updateStats, forwardingDelay)`
Puts the packet `packet` based on the `direction` argument either in the transmission (TX) or the reception (RX) buffer of all nodes in the `recipientNodes` list after a delay `forwardingDelay`, bypassing the communication channel.

7.3.2.3 Conditional Attacks

This class of statements enables the execution of the list of events described through message primitives only if the specified `filter` condition is evaluated as `TRUE`. The filter condition is applied on packets intercepted at every layer of the communication stack of each node in the `list of nodes` argument. A conditional attack is described according to the following syntax, with keywords highlighted in bold.

```
from T nodes in <list of nodes> do {
    filter(<condition>)
    <list of events>
}
```

An example of a conditional attack is reported below in Listing 7.1.

Listing 7.1 Example of a conditional attack in the ASL.

```
1 list targetList = {1,2,5}
2 from 200 nodes in targetList do {
3     filter("TRA.sourcePort" == "1025" and "TRA.destinationPort" == "2000")
4     drop(original)
5 }
```

At simulation time $t = 200$ s, the involved nodes in `targetList` start intercepting packets traversing their communication stack. When a packet produces a positive

match against the `filter` conditions, namely a transport-layer packet (TRA) with source port 1025 and destination port 2000, the attack simulation engine discards that packet as specified by the message primitive `drop`. The adopted “dot notation” is further discussed in Sect. 7.3.2.5.

7.3.2.4 Unconditional Attacks

This class of statements enables the periodical execution of the list of events described through message primitives, performed starting from an initial occurrence time '`T`' and then repeated according to a period '`P`'. Unlike conditional attacks, unconditional attacks are not related to the interception of packets by network nodes, but rather enforce the creation or duplication of network packets and their injection in the network. In particular, 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. A unconditional attack is described according to the following syntax, with keywords highlighted in bold.

```
from T every P do {
  <list of events>
}
```

An example of an unconditional attack is reported below in Listing 7.2.

Listing 7.2 Example of an unconditional attack in the ASL.

```
1 list dstList = {1,2,5}
2 from 200 every 1 {
3   packet fakePacket
4   create(fakePacket, "APP.type", 1000)
5   change(fakePacket, "APP.name", "myFakePacket")
6   # Set TCP-related control info
7   change(fakePacket, "controlInfo.connId", 22)
8   change(fakePacket, "controlInfo.userId", 0)
9   # Set the layer where to inject the new packet
10  change(fakePacket, "sending.outputGate", "app_tcp_inf$0[0]")
11  put(fakePacket, dstList, TX, false, 0)
12 }
```

The attack starts at simulation time $t = 200$ s and is repeated every 1 second (line 2). A packet `fakePacket` is first declared as variable (line 3) and its creation as an application-layer packet (line 4). All the packets have a specific type representing a simulation-based message. In the considered example, created malicious packets have type 1000 and only one information element, i.e., the field `name` (line 5). Furthermore, these newly created packets should be correctly filled in order to represent a complete OMNeT++ message (see Sect. 7.3.2.5). When invoking the `put()` primitive (line 11), the user specifies that the packet has to be injected in the transmission buffer TX of the nodes in the `dstList` defined at line 1, without any additional delay.

7.3.2.5 Syntax Conventions

SEA++ is intended to cover all the layers in the communication stack, except for the physical layer. This means that the attack simulator can interact with and affect the application, transport (TCP/UDP protocol), network, and data link layer. The abbreviations used to refer the communication layers are listed in Table 7.1.

Additional reserved words used as keywords are: `RANDOM_IP`, `RANDOM_MAC`, `RANDOM_INT`, `RANDOM_SHORT`. They are used with the primitive `change()` to instruct the assignment of random values based on their type.

With reference to the examples in Sect. 7.3.2.3 and Sect. 7.3.2.4, note the use of the “dot notation” `packet.layer.field`, in order to specify the field of the packet in the header of a layer (e.g., `layer.field` is translated as `TRA.sourcePort`). This implies that the user must be aware of the actual specific network protocols adopted at each communication layer. Also, for each of them, the user must be aware of the packet header structure and fields, and the specific capabilities possibly offered by the simulation platform. In particular, SEA++ relies on the objects `descriptors` provided by OMNeT++/INET in order to handle packets of a given communication layer and conveniently access and possibly alter their header fields.

In OMNeT++/INET, communication between protocol layers requires to specify additional information embedded into network packets, by means of `controlInfo` objects. Then, in accordance with the considered “dot notation”, the keyword `controlInfo.field` is used as the argument `field` in the `change()` primitive, in order to fill the field of `controlInfo` objects. In particular, `controlInfo.payload` can be used in order to change the size of the packet specified as first argument in the `change()` primitive. With particular reference to the unconditional attack provided as an example in Sect. 7.3.2.4, the user must provide the specific information required by the Transmission Control Protocol (TCP) to correctly handle application-layer packets. The keyword `controlInfo` is used to provide such information (cp. line 7–line 8).

Finally, there are two further keywords used with unconditional attacks, namely `sending.outputGate` and `attackInfo.fromGlobalFilter`. Both are used with the primitive `change()` for the argument `field`. The value provided for `sending.outputGate` for the argument `newContent` specifies in which layer the packet has to be injected. On the other hand, the value for `attackInfo.fromGlobalFilter` provided with the argument `newContent`

Table 7.1 ASL abbreviations for different communication layers.

Layer	Abbreviation
Application	APP
Transport	TRA
Network	NET
Data Link	MAC

is set to 1 for packets created specifically by the GEP (see Sect. 7.3.4). With further reference to the unconditional attack provided as example in Sect. 7.3.2.4, the value specified for `sending.outputGate` indicates that the packet `fakePacket` is injected between the application and the transport layer as an outgoing packet, i.e., the argument `newContent` is set to `app_tcp_inf$0` (cp. line 10).

7.3.3 Attack Specification Interpreter (ASI)

The ASI is a Python script that converts a *adl* file including the high-level attack description in ASL into an XML configuration file. The latter is then provided as input to the attack simulator, to be ultimately processed by the ASL. To ensure a successful translation of the *adl* file, the user must follow the syntax rules defined by the ASL and adopted by the ASI, with particular reference to the “dot notation” `packet.layer.field` discussed in Sect. 7.3.2.5.

The resulting XML configuration file is split into three distinct sections:

- The first section lists all the described physical attacks, each of which is composed by a single node primitive. That is, each physical attack indicates the involved node and the simulation time when the attack takes place.
- The second section lists all the cyber attacks described as conditional attacks. Each conditional attack indicates:
 1. the set of involved nodes,
 2. the simulation time when the attack starts,
 3. the `filter` condition, and
 4. the list of message primitives modeling the attack events.
- The third section lists all the cyber attacks described as unconditional attacks. Each unconditional attack indicates:
 1. the simulation time when the attack starts,
 2. the time period according to which the attack is reproduced, and
 3. the list of message primitives modeling the attack events.

Listing 7.3 shows an example of a XML configuration file, produced by the ASI. It includes one attack for each of the three sections described above.

Listing 7.3 Example of a XML configuration file describing three attacks.

```

1 <?xml version="1.0"?>
2 <configuration>
3   <Physical>
4     <Attack>
5       <start_time>20</start_time>
6       <node>5</node>
7       <action>
8         <name>Disable</name>
9       </action>
10    </Attack>
11  </Physical>
12  <Conditional>
```

```

13 <Attack>
14 <start_time>200</start_time>
15 <node>1:2:5</node>
16 <filter>
17 [:TRA.sourcePort::=:1025:][:TRA.destinationPort::=:2000:]AND
18 </filter>
19 <action>
20 <name>Drop</name>
21 <parameters>packetName:original:threshold:0</parameters>
22 </action>
23 </Attack>
24 </Conditional>
25 <Unconditional>
26 <Attack>
27 <start_time>60</start_time>
28 <frequency>0.1</frequency>
29 <var>
30 <name>"myFakePacket"</name>
31 <value>myFakePacket</value>
32 <type>STRING</type>
33 </var>
34 <var>
35 <name>123</name>
36 <value>123</value>
37 <type>NUMBER</type>
38 </var>
39 <var>
40 <name>4</name>
41 <value>4</value>
42 <type>NUMBER</type>
43 </var>
44 <var>
45 <name>1250</name>
46 <value>1250</value>
47 <type>NUMBER</type>
48 </var>
49 <var>
50 <name>0</name>
51 <value>0</value>
52 <type>NUMBER</type>
53 </var>
54 <var>
55 <name>"10.0.0.3"</name>
56 <value>10.0.0.3</value>
57 <type>STRING</type>
58 </var>
59 <var>
60 <name>"app_udp_inf$o[0]"</name>
61 <value>app_udp_inf$o[0]</value>
62 <type>STRING</type>
63 </var>
64 <action>
65 <name>Create</name>
66 <parameters>packetName:fakePacket:APP.type:1001</parameters>
67 </action>
68 <action>
69 <name>Change</name>
70 <parameters>
71 packetName:fakePacket:field_name:APP.name:value:"myFakePacket"
72 </parameters>
73 </action>
74 <action>
75 <name>Change</name>
76 <parameters>
77 packetName:fakePacket:field_name:controlInfo.packetSize:value:1250
78 </parameters>
79 </action>

```

```

80 <action>
81   <name>Change</name>
82   <parameters>
83     packetName:fakePacket:field_name:controlInfo.destAddr:value:"10.0.0.3"
84   </parameters>
85 </action>
86 <action>
87   <name>Change</name>
88   <parameters>
89     packetName:fakePacket:field_name:controlInfo.destPort:value:123
90   </parameters>
91 </action>
92 <action>
93   <name>Change</name>
94   <parameters>
95     packetName:fakePacket:field_name:controlInfo.sockId:value:4
96   </parameters>
97 </action>
98 <action>
99   <name>Change</name>
100  <parameters>
101    packetName:fakePacket:field_name:controlInfo.interfaceId:value:0
102  </parameters>
103 </action>
104 <action>
105   <name>Change</name>
106   <parameters>
107     packetName:fakePacket:field_name:sending.outputGate:value:"app_udp_inf$0
108       [0]"
109   </parameters>
110 </action>
111 <action>
112   <name>Put</name>
113   <parameters>
114     packetName:fakePacket:nodes:4:direction:TX:throughWC:false:delay:0
115   </parameters>
116 </action>
117 </Attack>
118 </Unconditional>
119 </configuration>

```

The XML configuration file from Listing 7.3 includes information for the ASE to reproduce the effects of the three different attacks described below using the ASL, namely a physical attack (see Listing 7.4), a conditional attack (see Listing 7.5), and an unconditional attack (see Listing 7.6).

In particular, the considered physical attack is described using the ASL in Listing 7.4, and results in line 3–line 11 in the physical section of Listing 7.3. These include: (i) the simulation time when the attack occurs (line 5); (ii) the involved node (line 6); and (iii) the node primitive composing the attack (line 7).

Listing 7.4 Physical attack described using the ASL.

```

1 disable (5, 20)

```

The considered conditional attack is described using the ASL in Listing 7.5, and results in line 12–line 24 in the conditional section of Listing 7.3. These include: (i) the simulation time when the attack starts (line 14); (ii) the involved nodes (line 15); (iii) the filter condition (line 16); and (iv) the action composing the attack (line 19).

Listing 7.5 Conditional attack described using the ASL.

```

1 list targetList = {1,2,5}
2 from 200 nodes in targetList do {
3     filter("TRA.sourcePort" == "1025" and "TRA.destinationPort" == "2000")
4     drop(original,0)
5 }

```

The considered unconditional attack is described using the ASL in Listing 7.6, and results in line 25–line 117 in the unconditional section of Listing 7.3. These comprise the following:

1. the simulation time when the attack starts (line 27),
2. the frequency according to which the actions of the attack are reproduced (line 28),
3. the declaration of the variables used in the attack description (line 29–line 63),
4. and the actions composing the attack (line 64–line 115).

Listing 7.6 Unconditional attack described using the ASL.

```

1 list attackNodesList = {4}
2 from 60 every 0.1 do {
3
4     # declare a packet
5     packet fakePacket
6
7     # create a new packet
8     create(fakePacket, "APP.type", "1001")
9
10    # fill the new packet properly
11    change(fakePacket, "APP.name", "myFakePacket")
12
13    change(fakePacket, "controlInfo.packetSize", 1250)
14
15    change(fakePacket, "controlInfo.destAddr", "10.0.0.3")
16    change(fakePacket, "controlInfo.destPort", 123)
17    change(fakePacket, "controlInfo.sockId", 4)
18    change(fakePacket, "controlInfo.interfaceId", 0)
19
20    change(fakePacket, "sending.outputGate", "app_udp_inf$0[0]")
21
22    put(fakePacket, attackNodesList, TX, FALSE, 0)
23 }

```

7.3.4 Attack Simulation Engine (ASE)

The ASE considers every node in the simulation scenario as an *Enhanced Network Node* module. The latter is in turn composed of: (i) an application module, that possibly includes multiple sub-modules modeling the actual application(s) running on the node; (ii) an arbitrarily complex collection of communication protocols composing the node's stack; and, finally, (iii) a LEP module. All sub-modules apart from the LEP can be off-the-shelf.

The LEP module is responsible for managing events related to physical and conditional attacks. In particular, the LEP module intercepts all application and network packets traversing the communication stack of a network node, as interposed

between each pair of layers in the node's stack. Based on the attack description, the LEP can alter the packets' content, generate and inject new packets, or even discard them. Also, it can change the node's behavior, i.e., change its position, neutralize the node by making it inactive, or completely remove the node from the network, according to the specific events in the attack description.

Furthermore, SEA++ enables the simulation of complex attacks by means of a single GEP module. In particular, the GEP module takes care of unconditional attacks, by repeatedly executing the list of events specified in their description. The GEP module is connected to the LEP module of each *Enhanced Network Node*, hence enabling the reproduction and evaluation of complex distributed security attacks, for instance wormhole attacks.

Figure 7.2 shows the overall architecture of the ASE, with particular reference to two interconnected network nodes. In each *Enhanced Network Node*, the LEP is interposed between each pair of layers in the communication stack, hence acting as gate-bypass between each pair of modules providing the respective layers. Finally, the GEP is connected to the LEP modules of the two *Enhanced Network Nodes*.

SEA++ provides also support for Software Defined Networking (SDN) environments [7, 10]. In particular, it considers the de facto standard *OpenFlow* [12] implemented in commercial SDN controllers and switches, and it builds on the simulation model initially proposed in [9].

Specifically, the LEP module has also been integrated into INET nodes modeling *OpenFlow* switches to enable the simulation of SDN-related security attacks. The separation of the *Control Plane* and *Data Plane* that is typical for SDN has also been modeled in the INET Framework. That is, the LEP interposes itself between the two planes, namely the data plane and the southbound *OpenFlow* interface. In this architecture, the LEP is responsible for intercepting network packets traversing the data plane, as well as *OpenFlow* messages which are exchanged between the SDN controller and the switch through the southbound *OpenFlow* interface. Figure 7.3 shows the overall architecture of an *Enhanced OpenFlow Switch Node*. Like the *Enhanced Network Nodes*, the LEP of the *Enhanced OpenFlow Switch Node* also communicates with the single GEP module. Note that SDN controllers can instead be simply represented as an *Enhanced Network Node* including a full communication stack. Further details on the SDN support in SEA++ are available in [16].

7.3.4.1 Reproduction of Attack Effects

As discussed in Sect. 7.3.3, the ASI converts the high-level attack description in ASL from an *adl* file to an XML configuration file, which consists of three distinct sections covering physical attacks, conditional attacks, and unconditional attacks. At simulation start up, the ASE takes the XML file as input, parses it, and initializes three data structures separately for the three types of attack. In detail, the ASE creates the following lists, whose list elements are all chronologically ordered to reflect the respective attack's occurrence time.

Fig. 7.2 ASE architecture with two enhanced network nodes.

Fig. 7.3 ASE architecture with an enhanced OpenFlow switch.

- ASE creates a list LP_n for each network node n involved in a physical attack. In every list, each element includes the description of a physical attack.
- ASE creates a list LC_n for each network node n involved in a conditional attack. Again, each element in every list includes the description of a conditional attack.
- ASE creates a list LU_n for each network node n involved in an unconditional attack. Just like before, each element in every list includes the description of an unconditional attack.

After that, the ASE instantiates a set of timers for each network node n involved in at least one of the lists defined above. In particular, each timer is associated to a specific attack involving that node. Then, the ASE starts all the timers in order to schedule the respective attack's (first) occurrence. That is, upon the expiration of a timer associated to a node n , the ASE retrieves the associated attack A from the list where it is included, and performs the following actions:

- In case A is a physical attack, the ASE executes the associated node primitive, by means of the LEP module of node n . Then, it removes A from the attack list LP_n .
- In case A is a conditional attack, node n starts intercepting packets traversing its communication stack by means of its own LEP. In particular, node n filters the intercepted packets according to the `filter` condition specified in A . Then, for each packet producing a positive match against the filter condition, node n executes the message primitives listed in A . Note that the actual execution of such primitives may involve the GEP module, as well as the LEP module of other nodes than n .
- In case A is an unconditional attack, the ASE starts executing the message primitives listed in A . Thereafter, the GEP repeatedly performs A , according to the specified occurrence period. This is practically achieved by resetting the timer associated to node n and attack A at the end of each repetition. Note that it is the prerogative of the GEP module to start the reproduction of unconditional attacks, then involving the LEP module of individual network nodes.

7.4 Explicative Example – Injection of Fake Packets

This section describes how to correctly setup the SEA++ framework. It also presents a step-by-step example on how to use the framework to evaluate the attack impact on a simple network scenario. The example presented in this section intends to provide a basic usage tutorial. It has no ambition to show how a comprehensive evaluation of attack impact can be conducted in a real-world networked system, nor to produce an attack ranking as part of a full risk assessment process.

In the considered example, we refer to a network composed of two client nodes and one server, and we specifically go through the steps to describe and reproduce an unconditional attack, in order to quantitatively evaluate its impact.

7.4.1 Framework Setup

The current version of SEA++ relies on:

- a compiler for C++ 11,
- an interpreter for Python 2.7.6,
- the *libxml* library, and
- INET 2.6, which is based on OMNeT++ 4.x.

SEA++ has been developed and tested under Ubuntu Linux 14.04 LTS, while it is usable in older or newer versions after applying basic configuration changes. In the following, we assume that the installation and configuration of OMNeT++ 4.6 and INET 2.6 has been previously completed, and we focus on the setup of the actual SEA++ framework. The user installs and configures SEA++ according to the following steps when he/she finished installing and configuring OMNeT++/INET. Additional details on the framework setup and configuration are available in the SEA++ user manual [15].

1. Obtain and install the *libxml* library, available at <http://www.xmlsoft.org>.
2. Download the SEA++ sources from the official Git repository at https://github.com/seapp/seapp_stable.
3. Extract the source archive and rename the main folder to *seapp_stable*.
4. Move to the SEA++ main folder and build the framework by running `make` followed by `makefile`.

To build SEA++ on older Ubuntu versions, it is recommended to additionally:

1. upgrade the Python interpreter to version 2.7.6.,
2. upgrade the C++ compiler to version 4.7 and set it as default, and
3. change the SEA++ *makefile* to meet the requirements of the specific system's architecture.

After a successful build, the user can include additional network scenarios as a subfolder of */examples/seapp*.

7.4.2 Evaluation of Attack Impact

We refer to a simple network topology consisting of two clients and one UDP server connected through an Ethernet switch. In particular, we assume that an adversary has successfully compromised *client1* and uses it to inject newly generated bogus packets addressed to the server. This section describes how to: (i) describe the attack using the ASL; (ii) define the network scenario in the presence of SEA++ components; and (iii) evaluate the attack impact through simulation.

7.4.2.1 Attack Description

As a first step, the user describes the attack mentioned above by using the ASL. As shown in Listing 7.7, the attack is described as an *unconditional attack*, where new bogus packets are injected in the transmission buffer of the compromised *client1*, that then forwards them to the victim server node. Unlike with the direct injection in the reception buffer of the victim server, this makes it also possible to reproduce the actual transmission and reception of such packets in the network. Note that, as an *unconditional attack*, it is the GEP module of the ASE that takes care of the single attack events at simulation runtime (see Sect. 7.3.4.1).

Listing 7.7 Attack description using the ASL.

```

1 list attackNodesList = {4}
2
3 from 60 every 0.1 do {
4     #declare a packet
5     packet fakePacket
6
7     #create a new packet
8     create(fakePacket, "APP.type", "1001")
9
10    #fill the new packet properly
11    change(fakePacket, "APP.name", "myFakePacket")
12    change(fakePacket, "controlInfo.packetSize", 1250)
13    change(fakePacket, "controlInfo.destAddr", "10.0.0.3")
14    change(fakePacket, "controlInfo.destPort", 123)
15    change(fakePacket, "controlInfo.sockId", 4)
16    change(fakePacket, "controlInfo.interfaceId", 0)
17
18    #instructions to Global Filter
19    change(fakePacket, "sending.outputGate", "app_udp_inf$0[0]")
20    put(fakePacket, attackNodesList, TX, FALSE, 0)
21 }
```

In detail, the considered attack is carried out through the node with ID 4, namely *client1* in the simulation scenario. To this end, the list `attackNodesList` in line 1 includes the identifier of *client1*. Then, the attack starts at simulation time $T = 60$ s, and the events listed in its body are periodically repeated every 0.1 s (line 3–line 21). In particular, at each loop repetition, the GEP performs the following actions.

First, the `'fakePacket'` variable of type `'packet'` is declared (line 5), and then used as first argument of the `change()` primitive to actually initialize the new packet (line 8). In particular, the new packet is specified as an application-level packet, by setting its `APP.type` field to the predefined message type 1001. This makes it possible to correctly create a `cPacket` message representing `fakePacket`.

After that, the primitive `change()` is further invoked to fill (meta-)information related to `fakePacket` and allow OMNeT++/INET to correctly handle the associated `cPacket` message (line 11–line 16). In particular, the application name `APP.name` is set to `myFakePacket`, essentially to enable selective packet tracing (line 11). Also, information required to correctly dispatch the packet are specified as values of `controlInfo` subfields (line 12–line 16). In particular, the packet size is set to 1250 bytes (line 12), i.e., the same size of legitimate application packets in the considered network scenario. Besides, the `controlInfo` subfields are filled with

information related to the UDP transport layer, as the one immediately traversed by the outgoing (application-level) packet (line 13–line 16).

Finally and consistent with the syntax conventions discussed in Sect. 7.3.2.5, a further indication is provided to the GEP module through the `change()` primitive (line 19). That is, the user specifies the stack layer in *client1* where `fakePacket` has to be injected. Then, the `put()` primitive instructs the GEP module to inject `fakePacket` in the transmission buffer of *client1* (line 20).

The described attack would continue for the whole simulation duration. This results in *client1* sending 10 additional bogus packets per second to the victim server, the intended *final effect* of the described attack.

Before proceeding with the network simulation in the presence of the described attack, the user has to convert the *adl* attack description in ASL into an XML file providing a parsable collection of attack events. To this end, the user simply runs the ASI as shown in Listing 7.8, i.e., providing the *adl* file as input. The resulting output is the XML attack configuration file, which is provided as input to the ASE through the description of the network scenario in the *omnetpp.ini* file (see Sect. 7.4.2.2). The ASI is available as a Python script in the */seapp_stable/interpreter* directory.

Listing 7.8 Usage of the ASI to convert the *adl* description file into the XML configuration file.

```
../../../../interpreter/interpreter/interpreter.py -i simple_attack.adl -o
simple_attack.xml
```

7.4.2.2 Description of the Network Scenario

The user is now requested to define the topology of the network scenario. This is done by filling an OMNeT++/INET Network Topology Description (NED) file and storing it in the */examples/seapp* directory. A NED file example is shown in Listing 7.9. Note that an already existing NED file can be considered as a starting point from previous simulation scenarios and then be extended accordingly.

Listing 7.9 Network scenario as defined in the NED file.

```
1 package inet.examples.seapp.simpleTopo;
2
3 import inet.nodes.inet.StandardHost;
4 import inet.nodes.ethernet.EtherSwitch;
5 import inet.networklayer.autorouting.ipv4.IPv4NetworkConfigurator;
6 import inet.util.ThruputMeteringChannel;
7 import inet.globalfilter.GlobalFilter;
8
9 network Scenario
10 {
11     parameters:
12         string attackConfigurationFile = default("none");
13         @display("bgb=600,300");
14     types:
15         channel ethernetline extends ThruputMeteringChannel
16         {
17             delay = 1us;
18             datarate = 100Mbps;
19             thruputDisplayFormat = "u";
20         }
```

```

21 submodules:
22 configurator: IPv4NetworkConfigurator
23 {
24     @display("p=40,40");
25 }
26 globalFilter: GlobalFilter
27 {
28     @display("p=350,50");
29 }
30 client1: StandardHost
31 {
32     @display("p=200,100;i=device/laptop");
33 }
34 client2: StandardHost
35 {
36     @display("p=200,200;i=device/laptop");
37 }
38 server: StandardHost
39 {
40     @display("p=500,150;i=device/server");
41 }
42 switch: EtherSwitch
43 {
44     @display("p=350,150");
45 }
46
47 connections allowunconnected:
48 switch.ethg++ <--> ethernetline <--> client1.ethg++;
49 switch.ethg++ <--> ethernetline <--> client2.ethg++;
50 switch.ethg++ <--> ethernetline <--> server.ethg++;
51
52 globalFilter.nodes++ <--> client1.global_filter;
53 globalFilter.nodes++ <--> client2.global_filter;
54 globalFilter.nodes++ <--> server.global_filter;
55 }

```

In particular, it is important to observe the following steps:

- Include the string parameter *attackConfigurationFile* to the network (line 12). This is initialized to *none*, but it will be overwritten later on within the *omnetpp.ini* file, when specifying the *xml* file including the attack description.
- Import the *GlobalFilter* class (line 7) providing the GEP module.
- Declare the actual GEP submodule (line 26).
- Connect the GEP module to every network node (line 52–line 54).

The next step consists of defining the traffic model in the network scenario by filling the OMNeT++ file *omnetpp.ini*. An example of such a *omnetpp.ini* file is shown in Listing 7.10. During the simulation, each client node runs the application *UDPSink* available in *INET* and sends 5 packets per second to the server node. The server node runs the application *UDPSink* available in *INET*, without replying back to the clients. According to this traffic model, the server receives 10 packets per second overall.

Listing 7.10 Content of the *omnetpp.ini* file.

```

1 [General]
2 debug-on-errors = true
3 sim-time-limit = 120s
4
5 network = Scenario
6

```



```

7 *.configurator.networkAddress = "192.168.2.0"
8
9 #Traffic Configuration
10 *.server.numUdpApps = 1
11 *.server.udpApp[0].typename = "UDPSink"
12 *.server.udpApp[0].localPort = 123
13
14 *.client.numUdpApps = 1
15 *.client.udpApp[0].typename = "UDPBasicApp"
16 *.client.udpApp[0].localPort = 100
17 *.client.udpApp[0].destPort = 123
18 *.client.udpApp[0].messageLength = 1250 bytes
19 *.client.udpApp[0].sendInterval = 0.2s
20 *.client.udpApp[0].destAddresses = "server"
21
22 [Config Simple_attack]
23 extends = General
24 *.attackConfigurationFile = "simple_attack.xml"

```

In particular, it is important to include the configuration `Simple_attack` (line 22–line 24) to specify the XML file that includes the attack description (see Sect. 7.4.2.1). This configuration extends the traffic behavior described in the `General` section by including the attack description as a parsable XML file.

7.4.2.3 Simulation of Attack Effects

Once the attack and scenario description are completed, the user can finally run the actual simulation in SEA++, just as it is usually done in OMNeT++/INET and typically through the native Graphical User Interface (GUI).

Figure 7.4 shows the output during the simulation runtime. This also includes the events related to the attack under evaluation, such as the injection of fake packets `myFakePacket` from the GEP module to `client1` through the wrapper message `PutReq`, as well as the following transmission to the server node as the final destination. The application `UDPSink` running on the server node additionally measures the number of received packets per second. Once the simulation is completed, the user can collect the statistics and perform post-analysis on simulation results.

For instance, Fig. 7.5 shows the packet reception rate on the server node both in the baseline attack-free scenario (10 packets per second) and in the presence of the described attack (20 packets per second after $T = 60$ s). The graph also includes results produced by assessing the same attack, while considering higher attack injection rates, for a total reception rate on the server node of 25 of 30 packets per second. The attack injection rate can be easily changed in the attack description, by specifying a different time period in the `from` loop of the modeled unconditional attack (see Sect. 7.4.2.1, Listing 7.7 – line 3).

Fig. 7.4 Simulation output on the OMNeT++/INET GUI during the injection attack.

Fig. 7.5 Packet reception rate on the server node.

7.5 Conclusion

This chapter presented SEA++, a simulation framework for assessing the impact of cyber-physical security attacks through simulative evaluation, in a way which is flexible and user-friendly. The framework is designed and implemented over OMNeT++/INET and allows network security architects to quantitatively evaluate the effects of security attacks against networks and applications in Information and Communication Technology (ICT) networked infrastructures, including support for SDN architectures. Therefore, SEA++ is especially convenient at the last stage of the security risk assessment process, as it makes it possible to clearly understand the effects of successful attacks, rank them based on their severity, and accordingly prioritize security countermeasures to adopt.

The key advantage offered by SEA++ is the ability to reproduce the final actual effects of successful attacks, regardless how they have been carried out. To this end, attacks to be evaluated are described through a high-level specification language. As a consequence, the user is not required to implement the actual adversary behavior or the actual attack execution, nor to modify any other software module. Besides, other than specific SEA++ modules seamlessly integrated into OMNeT++/INET, all other software modules can be off-the-shelf. The SEA++ framework is available as open-source software, and its core approach makes it portable to alternative simulation environments based on discrete events.

Finally, we foresee a number of potential interesting developments related to the SEA++ framework. First, feedback and requirements from the OMNeT++/INET community and use-case providers would be helpful to improve the Attack Specification Language as well as the core Attack Simulation Engine of SEA++. At the same time, it would be convenient to provide an end-user GUI, as well as tools for attack assessment in batches and for automatic analysis of simulation output.

Second, it would be good to harmoniously interconnect SEA++ with alternative simulation tools focused on different environments, technologies, or assessments. This would aim at a single and comprehensive simulation tool for the assessment of security attacks in complex and heterogeneous network environments. Such a tool would be composed of different synchronized sub-frameworks, and hence be able to perform more complex assessments through a single composite simulation. A first good candidate to consider is the ASF++ simulation framework, which is built on OMNeT++ Castalia and is based on the same rationale and approach of SEA++.

Third, SEA++ would greatly benefit from having security protocols available in OMNeT++/INET as off-the-shelf software modules, which would be selectable when composing communication stacks of network nodes. This includes implementations of protocols such as Transport Layer Security (TLS), Datagram Transport Layer Security (DTLS), Object Security for Constrained RESTful Environments (OSCORE), Internet Protocol security (IPsec), and Internet Key Exchange Version 2 (IKEv2). Similarly, SEA++ would greatly benefit from available software modules in OMNeT++/INET providing application protocols typical of Internet of Things (IoT) environments, such as Constrained Application Protocol (CoAP) and Lightweight Machine-to-Machine (LWM2M).

Acknowledgements The authors sincerely thank the anonymous reviewers as well as the editors Antonio Virdis and Michael Kirsche for their constructive feedback and comments.

References

- [1] B. Genge, C. Siaterlis and M. Hohenadel: AMICI: An Assessment Platform for Multi-domain Security Experimentation on Critical Infrastructures. In: *Critical Information Infrastructures Security, Lecture Notes in Computer Science*, vol. 7722, pp. 228–239. Springer Berlin Heidelberg (2013)
- [2] Boulis, T.: Castalia. [online] <https://github.com/boulis/Castalia> (2018)
- [3] C. Queiroz, A. Mahmood and Z. Tari: SCADASim—A Framework for Building SCADA Simulations. *IEEE Trans. on Smart Grid* **2**(4), 589–597 (2011)
- [4] C. Siaterlis, A.P. Garcia and B. Genge: On the Use of Emulab Testbeds for Scientifically Rigorous Experiments. *IEEE Communications Surveys Tutorials* **15**(2), 929–942 (2013)
- [5] Dini, G., Tiloca, M.: ASF: an Attack Simulation Framework for wireless sensor networks. In: *The 8th IEEE Intl. Conf. on Wireless and Mobile Computing, Networking and Communications (WiMob 2012)*, pp. 203–210. IEEE (2012)
- [6] Dini, G., Tiloca, M.: On simulative analysis of attack impact in Wireless Sensor Networks. In: *2013 IEEE 18th Conference on Emerging Technologies & Factory Automation (ETFA)*, pp. 1–8. IEEE (2013)
- [7] Foundation, O.N.: *Software-Defined Networking: The New Norm for Networks* (2012)
- [8] G. Stoneburner and A. Goguen and A. Feringa: *Risk Management Guide for Information Technology Systems - Recommendations of the National Institute of Standards and Technology*. Tech. rep., National Institute of Standards and Technologies (2002). URL <http://csrc.nist.gov/publications/nistpubs/800-30/sp800-30.pdf>
- [9] Klein, D., Jarschel, M.: An OpenFlow Extension for the OMNeT++ INET Framework. In: *6th International ICST Conference on Simulation Tools and Techniques (SimuTools '13)*, pp. 322–329 (2013)
- [10] Kreutz, D., Ramos, F.M.V., Veríssimo, P.E., Rothenberg, C.E., Azodolmolky, S., Uhlig, S.: *Software-Defined Networking: A Comprehensive Survey*. *Proceedings of the IEEE* **103**(1), 14–76 (2015)
- [11] L. Sánchez-Casado, R.A. Rodríguez-Gómez, R. Magán-Carrión and G. Maciá-Fernández: NETA: Evaluating the Effects of NETWORK Attacks. MANETs as a Case Study. In: *Advances in Security of Information and Communication Networks, Communications in Computer and Information Science*, vol. 381, pp. 1–10. Springer Berlin Heidelberg (2013)
- [12] Pitt, D.: Open Networking Foundation. [online] <http://opennetworking.org> (2012)

- [13] T. Bonaci, L. Bushnell and R. Poovendran: Node capture attacks in wireless sensor networks: A system theoretic approach. In: The 49th IEEE Conference on Decision and Control (CDC 2010), pp. 6765–6772 (2010)
- [14] Tiloca, M., Racciatti, F., Dini, G.: Simulative Evaluation of Security Attacks in Networked Critical Infrastructures. In: 2nd International Workshop on Reliability and Security Aspects for Critical Infrastructure Protection (ReSA4CI 2015), published in Lecture Notes in Computer Science, LNCS 9338, pp. 314–323. Springer International Publishing (2015)
- [15] Tiloca, M., Racciatti, F., Stagkopoulou, A., Dini, G.: SEA++, a tool for Simulative Evaluation of Attacks. [online] https://github.com/seapp/seapp_stable (2017)
- [16] Tiloca, M., Stagkopoulou, A., Dini, G.: Performance and Security Evaluation of SDN Networks in OMNeT++/INET. In: OMNeT++ Community Summit 2016, pp. 9–14 (2016)
- [17] Y.-L. Huang, A. A. Cárdenas, S. Amin, Z.-S. Lin, H.-Yi Tsai and S. Sastry: Understanding the physical and economic consequences of attacks on control systems. *International Journal of Critical Infrastructure Protection* **2**(3), 73–83 (2009)
- [18] Y.-T. Wang and R. Bagrodia: SenSec: A Scalable and Accurate Framework for Wireless Sensor Network Security Evaluation. In: The 31st Intl. Conf. on Distributed Computing Systems Workshops (ICDCSW 2011), pp. 230–239 (2011)
- [19] Y. Xu, G. Chen, J. Ford and F. Makedon: Detecting Wormhole Attacks in Wireless Sensor Networks. In: Eric Goetz and Sujeet Shenoi (ed.) *Critical Infrastructure Protection, Post-Proceedings of the First Annual IFIP Working Group 11.10 International Conference on Critical Infrastructure Protection, IFIP*, vol. 253, pp. 267–279. Springer (2007)