Journal of Computer Virology and Hacking Techniques
https://doi.org/10.1007/s11416-021-00392-0

ORIGINAL PAPER

A framework for formal analysis and simulative evaluation of security attacks in wireless sensor networks

Cinzia Bernardeschi[1] · Gianluca Dini[1] · Maurizio Palmieri[1] · Francesco Racciatti[2]

Received: 2 August 2020 / Accepted: 10 July 2021

Abstract

When designing Wireless Sensor Networks it is important to analyze their security risks and provide adequate solutions for protecting them from malicious attacks. Unfortunately, perfect security cannot be achieved, for performance reasons. Therefore, designers have to devise security priorities, and select security mechanisms accordingly. However, in the early stages of the design process, the concrete effects of security attacks on the system may not be clearly identified. In this paper, we propose a framework that integrates formal verification and network simulation for enabling designers to evaluate the effects of attacks, identify possible security mechanisms, and evaluate their effectiveness, since design time. Formal methods are used to build the abstract model of the application, together with a set of attacks, and to state properties of general validity. The simulator measures the impact of the attacks in terms of common network parameters, like energy consumption or computational effort. Such information can be used to select adequate security mechanisms, then the initial abstract model can be refined to adopt them, and finally prove that former system properties are still verified. The framework relies on UPPAAL for formal modeling and verification and uses the Attack Simulation Framework on top of Castalia as a network simulator. As proof of concept, a case study is shown.

Keywords Security · Wireless sensor networks · Simulation · Attack evaluation · Formal verification

1 Introduction

Wireless Sensor Networks (WSNs) are particularly vulnerable to security threats, both on the physical and logical plane. In fact, WSN nodes operate unsupervised, even in hostile environments, without a physical line-of-defense. Moreover, WSN nodes are resource-scarce devices, so they cannot address all the attacks to which they are exposed to [10].

A WSN should provide information security (e.g., data integrity, and authenticity) as well as robustness to most common attacks like denial-of-service, routing attack, and replacement of sensors with malicious nodes, which can, for example, ignore messages to be transmitted. The analysis of security-related issues in WSNs has been largely studied [7,15].

A number of development methodologies have been proposed for secure application development by integrating a set of security-related activities through the Software Development Life-Cycle (SDL) [20]. Such activities, like the clear definition of the security requirements, the accurate knowledge of the assets of the system, the misuse cases definition, the threat modeling, the application of risk analysis, as well as the definition of security attacks countermeasures, may help designers to detect security flaws and solve security issues earlier in the system life-cycle. At later stages of development, other practices are suggested to improve the overall security, like secure coding, which allows typical programming mistakes to be avoided, and simulation of malicious

This research was partially supported by the Italian Ministry of Education and Research (MIUR) in the framework of the CrossLab project (Departments of Excellence).

✉ Maurizio Palmieri
maurizio.palmieri@ing.unipi.it

Cinzia Bernardeschi
cinzia.bernardeschi@unipi.it

Gianluca Dini
gianluca.dini@unipi.it

Francesco Racciatti
francesco.racciatti@unifi.it

[1] Department of Information Engineering, University of Pisa, Pisa, Italy

[2] Department of Information Engineering, University of Florence, Florence, Italy

Published online: 06 August 2021

attacks at runtime, exploiting for example the environment configuration. Similarly, penetration testing could also be used to identify vulnerabilities.
Many of the proposed approaches apply risk analysis and threat modeling activities in the design phase, in order to obtain a metric-based risk assessment of threats [14,18]. For example, in the Microsoft SDL framework, the STRIDE tool [23] can be used to identify threats and the risk related to each threat can be numerically estimated through some score-based methods, e.g. the DREAD method.

The approach proposed in this paper relies on model based design, and provides objective measurements of the impact of threats on the system, since the design phase, by exploiting both simulation and formal analysis. The possibility to model the WSN (or the application) at a high level through formal languages, and run such a model on a network simulator, makes it possible to collect real data about attacks and to rank security risks accordingly. This may help designers to better select the most adequate countermeasures to implement and reach a good trade-off between security, costs, and performances.

Generally, the process of designing and prototyping WSN protocols and applications exploits tools like network simulators [19]. Example of simulators are OMNeT++ [24], NS3 [17], Castalia [2] and Ptolemy [9].

These simulators provide practical measurements of network nodes quantities like communication latency, energy consumption, and computational effort. However, the scope of network simulators is only limited to the simulated scenarios at hand, and cannot be used to prove general properties of the system.

Conversely, formal models of WSN protocols and applications, designed via mathematical methods, allow designers to state properties of general validity and prove them through automated tools. On the other hand, the inclusion of fully detailed physical properties inside formal models may lead to an infeasible solution search.

Formal methods have been extensively applied in the literature for modeling and analyzing sensor networks. For example, in [8] key properties of a popular routing protocol are analyzed, in [21] performance of protocols are evaluated, in [6] formal methods are exploited for validating simulation results. A combined approach for both simulation and formal verification of WSN protocols has been proposed in [4,5]. Such an approach explores logic as a formal specification language, executable theories for simulation, and theorem proving for formal proofs. However, this approach only considers an abstract description of the communication protocol at the network layer.

The main contribution of this work is the definition and the implementation of a framework that enables designers, since the early stage of design, to (i) automatically turn a formal abstract model of a WSN system into a network model; (ii) describe the effects of cyber-physical attacks on the system; (iii) measure the effects of such attacks on the network and the application, and identify adequate countermeasures; (iv) refine the initial model with the adopted countermeasures, and test them against the attacks. In detail, starting from abstract models described in UPPAAL [3] using the Timed Automata formalism [1], such a framework produces concrete network models to be simulated against attack scenarios through the Attack Simulation Framework (ASF) [11]. The exploitation of simulators downstream of formal methods can also provide designers with useful insights on the physical aspects of the system, such as energy consumption or computational effort of nodes.

As proof of concept, the framework is used to study the flooding protocol [16], assuming the following security issues: (i) a compromised node drops a packet; (ii) a compromised node tampers a packet before sending it; and (iii) an external malicious node injects fake packets into the network. Though the system shows robustness against the drop and the tampering attacks, a critical issue related to the excessive energy consumption of a certain node is found when the packet injection attack occurs.

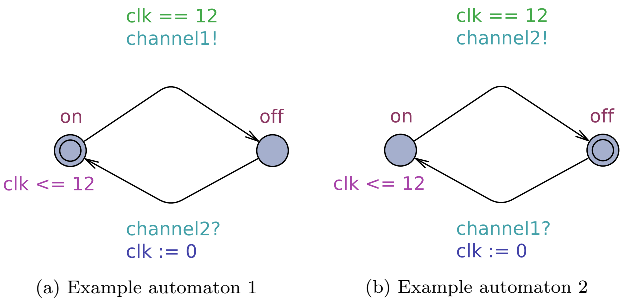
The paper is organized as follows: Sect. 2 provides background on the UPPAAL formal modeling framework and the ASF. Section 3 describes the proposed approach and the developed framework. Section 4 shows a case study and Sect. 5 concludes with a discussion on future work.

2 Background

The section that follows introduces the basic concepts of the tool UPPAAL, and the Attack Simulation Framework.

2.1 UPPAAL

UPPAAL is a tool for modeling and analyzing systems described by Timed Automata [1]. A *timed automaton* is a graph characterized by



(a) Example automaton 1
(b) Example automaton 2

Fig. 1 Example of UPPAAL automata

- a set of nodes (named *locations*);
- one *initial location*;
- a set of *invariant* conditions, labeling locations;
- a set of *edges* between locations;
- a set of *actions*, labeling edges;
- a set of *clocks*;
- a set of *constraints*, labeling edges (guards).

The values of the clocks and the current location represent the current *state* of a timed automaton. Location changes occur as a consequence of execution of edges, together with the changes explicitly written as actions. Instead, when the system remains in a certain location, the time progression is represented by the increasing of the clocks values, which happens at the same rate for all of them. As long as the current location in the state of the timed automaton has an invariant condition, this condition must always be verified. If an invariant is no longer satisfied an edge that changes accordingly the current `state` must be executed (for example an edge that changes the variable involved in the invariant or an edge that changes the current location); if such an edge is not executable the timed automaton ends up in a *deadlock* meaning that the system is not well defined.

Such timed automata can be connected and synchronized to each other for modeling complex *networks of timed automata*. Connections between timed automata can be implemented using communication channels, through which synchronization actions can be executed. Synchronization between automata can be realized through edges, one for each automaton to be synchronized. Such edges have to be labeled with *complementary actions*, namely *input* and *output*, which are represented through question marks `(?)` for input synchronizations, and exclamation marks `(!)` for output synchronizations, respectively. A timed automaton executing an output action, synchronizes with one or more timed automata executing input actions, and vice versa. Synchronizations can be blocking or not blocking, depending on the type of channel connecting the automata. For example, *broadcast channels* are not blocking and the output action can synchronize with many input ones.

An exemplary UPPAAL model is shown in Fig. 1 where two automata are shown. Both automata in Fig. 1a and b have two different locations, `on` and `off`, a local clock variable `clk`, used to specify the invariant of the `on` location and the guard `clk <= 12` in the upper edge. Finally the two automata synchronize on two different channels `channel1` and `channel2` (for example `channel1` is the output channel for the automaton 1 and the input channel for the automaton 2) . The initial locations of the two automata are complementary, i.e. automaton 1 is initially in the `on` location and automaton 2 is initially in the `off`, and they simultaneously switch from one location to the other every 12 time units, resetting the local clock `clk` every time the automaton moves from `off` to `on`.

For additional details regarding Timed Automata, and their possible usages, the readers can refer to [1] .

Automated formal verification is supported by the UPPAAL Model Checker. UPPAAL has been chosen in this work because of its intuitive user interface and for the successful example of uses for wireless communication modeling (e.g. [13,22]).

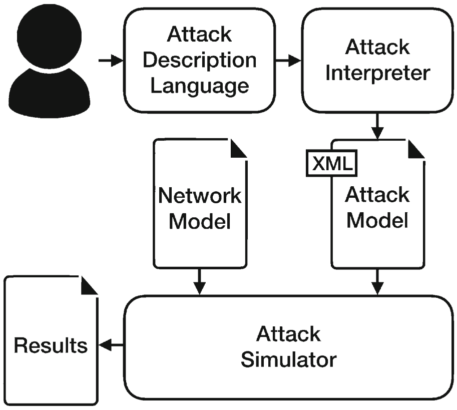


Fig. 2 Overview of the ASF workflow

2.2 Attack simulation framework

The ASF can be applied to any Commercial-Off-The-Shelf (COTS) network simulators for enabling the simulation of the effects of cyber-physical attacks against the network and the application [11].

Figure 2 shows an overview of the ASF workflow. ASF provides i) an *Attack Description Language* (ADL), which is a high-level language for describing the effects of the attacks against the network and the application; ii) an *Attack Interpreter*, that converts the attack description into an Attack Model; and iii) an *Attack Simulator*, that simulates the effects of the attacks on the given Network Model, and provides simulation results.

As first step, the user builds an attack scenario, i.e. the effects of the attacks, through the ADL are described. Then, the Attack Interpreter turns the description of the attacks into an XML file that contains the model of the attack scenario. Eventually, the Attack Simulator takes both the Network Model and the Attack Model as inputs, to simulate the effects of the attacks and measure their impact against the attack-free scenario. The impact of the attacks can be measured by using many metrics. Common metrics refer to nodes' and network's quantities and measure the impact of the attacks in terms of i) degradation in performance; ii) increment in energy consumption; and iii) reduction of network throughput; among others [12]. Instead, more specialized metrics, like security metrics [10], measure the effects of the attacks on system' security.

Generally speaking, ASF is simulator agnostic, i.e. it can work on top of any network simulator. However, the underlying simulator has to implement the features that make it possible to simulate the effects of the attacks described in the XML file containing the Attack Model. It is worth noting that the user is not required to modify and re-build the Attack Simulator for implementing attacks. The Attack Simulation Language is flexible enough to let the user simulate a large set of cyber-physical attacks, even complex ones.

2.2.1 Attack Description Language

The ADL provides a collection of primitives that allow the user to describe an attack as a sequence of atomic events that take place neatly. The primitives can be grouped into two sets i) node primitives, and ii) message primitives.

Node Primitives Node primitives account for physical attacks and allow the user to describe events that alter the physical behavior of nodes. In detail, node primitives are:

- `destroy(nodeID, t)`, removes node `nodeID` from the network at time `t`.
- `disable(nodeID, moduleID, t)`, disables the module `moduleID` of the node `nodeID` at time `t`.
- `deceive(nodeID, sensorID, t, val)`, imposes value `val` to all the readings of sensor `sensorID` of node `nodeID`, starting from time `t`.
- `move(nodeID, t, pos)`, moves the node `nodeID` to position `pos` at time `t`.

Node primitives can be used for simulating physical attacks against target nodes like i) capture and move; or ii) physical disruption; or iii) malfunctioning of internal components; among others.

Message Primitives Message primitives account for cyber attacks and allow the user to describe actions on network packets. In detail, message primitives are:

- `retrive(dst, pkt, pktFld)`, copies the content of field `pktFld` of packet `pkt` into the variable `dst`.
- `change(pktFld, pkt, src)`, copies the content of variable `src` into field `pktFld` of packet `pkt`.
- `drop(pkt)`, discards the packet `pkt`.
- `create(pkt, type, fld, val, ...)`, creates a new packet `pkt` of type `type`, and fills the field `fld` with value `val`. The type is in the format `layer.protocol`, e.g. `LINK.TMAC`, that specifies a link-layer TMAC packet. The user can specify the content of multiple fields of the packet.
- `clone(dstPkt, srcPkt)`, creates the packet `dstPkt` as exact copy of the packet `srcPkt`.
- `put(pkt, dstNode, TX|RX)`, puts packet `pkt` either in the transmission (TX) or reception (RX) buffer of nodes `dstNodes`.

Message primitives can be used for simulating cyber attacks like i) packet eavesdropping; or ii) packet dropping; or iii) packet injection; or iv) packet altering; or even v) wormholes; among others.

Loop statements The ASL provides loop statements for specifying the periodic occurrence of a list of message primitives. For instance, the statement:

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

describes the periodic occurrence of the list of events, with period `P`, starting from time `T`.

Conditional statements Furthermore, the ASL provides conditional statements for specifying the conditional occurrence of message primitives, which takes place depending on the specified condition evaluated at runtime by nodes. For instance, the statement:

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

describes the conditional occurrence of a list of events on the declared list of nodes. In detail, starting from time `T`, each target node in the list applies the `filter` condition on all the packets flowing through its communication stack. If the packet satisfies the `filter` condition, the list of events takes place.

As a practical example, let us consider a node reprogram attack in which, starting from time 50 s, the target node 2 is captured and reprogrammed for tampering the payload of the application packets received from Node 1. In particular, the original payload is decremented by one.

```
from 50 s nodes = "2" do {
 filter( packet.APP.source == 1
         AND packet.APP.type == DATA ) {
   var value = packet.APP.payload - 1;
   change(packet.APP.payload, value);
 }
}
```

As shown above, the dot notation `packet.layer.field` is used to access the field `field`, on layer `layer`, of packet `packet`. As a consequence, the user has to be aware of both i) the network protocols running on each communication layer; and ii) the structures of such protocols' packets.

Once the attack scenario is built by means of the ASL, it can be interpreted by the Attack Interpreter for turning it into the XML Attack Model. Eventually, the Attack Simulator gets the XML Attack Model for simulating the attacks on

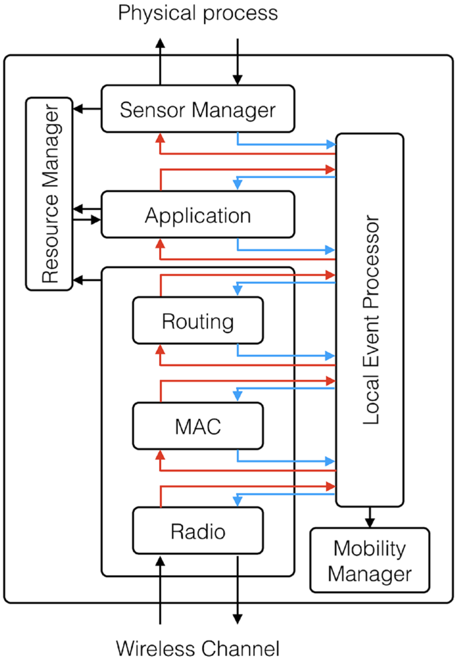


Fig. 3 Enhanced Castalia node architecture

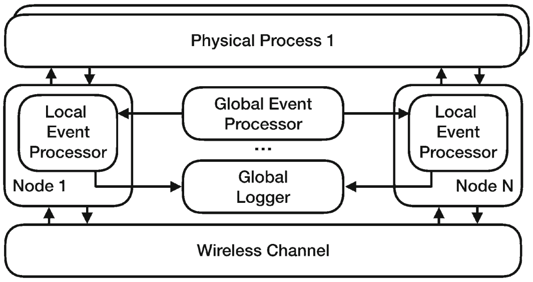


Fig. 4 Enhanced Castalia network architecture

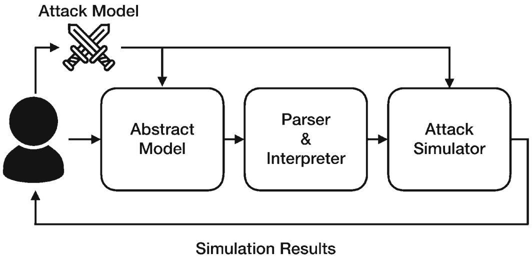


Fig. 5 Overview of the proposed approach for evaluating the impact of the attacks from the early stage of design

the current Network Model. In the following, we refer to the implementation of ASF on top of the WSN simulator Castalia.

2.2.2 Attack Simulator

The Attack Simulator showed in Figure 2 is obtained by enhancing the WSN simulator Castalia. In detail, the node's architecture and the network's architecture are modified to parse the Attack Model file and simulate the effects of the attacks on the WSN, while the simulation runs.

Enhanced Castalia node Figure 3 shows the architecture of an *enhanced Castalia* node. The node is provided with an additional module, namely the Local Event Processor (LEP) module, which operates transparently with respect to other modules. The LEP module manages the events related to the attacks, i.e. it performs the operations for simulating the occurrence of attacks described through the ASL's primitives.

Thanks to its particularly assembly, it interposes between all the modules implementing the node's communication stack. In this way, it intercepts all the packets flowing through the node's stack. Therefore, depending on the attack to be simulated, the LEP module can perform several operations like i) inspect packets; ii) alter the content of them; iii) discard certain packets; and iv) create new packets and inject them in any layer of the node's stack. Moreover, the LEP module can act on internal node's components for altering their behavior or for disabling them.

Enhanced Castalia network At network level, ASF provides an additional module, namely the Global Event Processor (GEP) module, which is directly connected with all the LEPs, as shown in Figure 4. The GEP module coordinates the operations of the LEPs to perform complex attacks, like packets injection and wormholes. Eventually, the LEP modules are connected with the Global Logger that collects the simulation data, which will be used for calculating the metrics of interest in post-simulation.

3 Proposed approach and related framework

The approach we propose binds formal methods and simulation, with the aim of quantitatively evaluating the effects of security attacks, and measuring the effectiveness of the adopted countermeasures (if needed), since the early stage of design. This approach develops on four main steps, as shown in Figure 5. As a first step, WSN designers can use formal methods to easily build an abstract model of the protocol or application, to study and prove its general properties. The abstract model can also be extended for including the attacks against which the application has to be tested. In this way, it is possible to prove the general properties of the system when attacks occur, also. Then, as a second step, a network model is generated starting from the original abstract model, namely the abstract model that does not contain the attacks, via a dedicated Interpreter. As a third step, the Attack Simulator simulates the network model both against attack-free and attack scenarios. When the simulation ends, the simulation results are used to i) validate the behavior of the network model; ii) evaluate the impact of the attacks on the net-

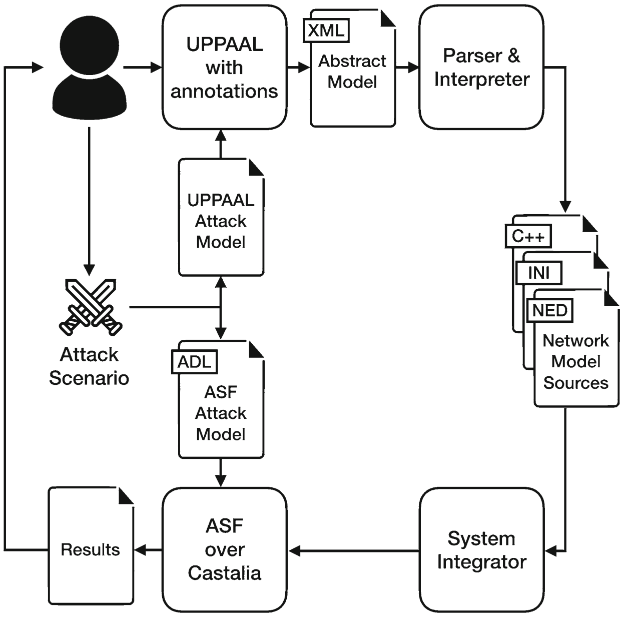


Fig. 6 Detailed workflow for binding UPPAAL with ASF on top of Castalia

work and the application; and iii) identify countermeasures, if needed. Eventually, as a fourth step, the abstract model can be *refined* for including countermeasures. Such a workflow can be repeated many times until the refined abstract model behave as expected against the considered attack scenarios.

In the following, it is shown the framework implementing this approach. Then, the design of an application-layer protocol is presented as a case study.

3.1 UPPAAL/ASF integration workflow

Figure 6 shows the workflow we propose for integrating UPPAAL with ASF over Castalia.

As a first step, the UPPAAL model has to be enriched with information regarding the physical features of the simulation environment, e.g. i) the size of the simulation field; ii) the position of nodes in the simulation field; iii) the latency of the channels; among others. UPPAAL does not take into account the physical features at all. However, such information are mandatory for performing network simulation.

The enrichment of the UPPAAL model takes place in the related XML file, precisely in the `system` tag, using annotations, i.e. comments having the format: `//@<Annotation>`. UPPAAL does not take into account such annotations at all, since they are written as comments in system description source code. As an example, the following code shows the annotation of the position of Node 1.

```
//@Position(20, 50, 10) node1 := relay(1);
```

In detail, the position of Node 1 is ($x = 20$, $y = 50$, $z = 10$) from the origin in the Cartesian coordinate system, which is used by Castalia as a coordinate system.

As a second step, the XML file produced by UPPAAL is parsed by the Parser for building a standard object-oriented model of the UPPAAL timed automata. Next, such an object-oriented model is processed by the Interpreter, which produces a set of source files, accordingly to the underlying Simulator. In particular, the Interpreter produces a set of C++ and INI files that implement the actual behavior of the network. Then, the System Integrator bundles the files produced by the Interpreter with the Simulator.

Eventually, as a fourth step, the Simulator runs the simulation and produces results. The analysis of such results may underline some problematic aspects that had not been taken into consideration while building the initial abstract model. If so, the abstract model is refined for addressing such issues. The whole process can be repeated until the system starts to behave as expected.

Additionally, the user can study the behavior of the system against attack scenarios of interest. Such attack scenarios can be included either at UPPAAL level, or at Simulator level, or both. By introducing the attack scenarios in UPPAAL, the user can check if the system's properties are still satisfied despite the attacks. For doing so, the attacks have to be actually integrated in the original abstract model, namely the abstract model without attacks. Instead, when performing the attacks on the Simulator, the user can precisely measure the impact of them on the network and the application. For doing so, the attacks have to be modeled through the Attack Description Language provided by ASF. Then, the analysis of the simulation results may lead to the adoption of countermeasures. As a consequence, the initial abstract model is refined and the whole process repeats for evaluating the effectiveness of such countermeasures. It is worth noting that the framework refers to the original abstract model for obtaining the related network model, and not to the abstract model that includes the attacks.

3.2 Implementation of the integration framework

Castalia comes with a set of ready-to-use components that cover the entire communication stack. Such components are fully tunable and customizable and can be combined with each other for implementing the desired behavior on each layer of the communication stack. The prototype we propose exploits such ready-to-use components. In fact, it interprets the UPPAAL model for generating the application-layer module from scratch. Then, it combines the generated application-layer module with ready-to-use modules that implement bottom layers. The section that follows describes key design elements of the framework for integrating UPPAAL with ASF over Castalia.

3.2.1 UPPAAL model annotations

The integration framework requires the user to annotate the position of nodes, only. In fact:

- the position is the only parameter that cannot be automatically inferred from the UPPAAL abstract model;
- the UPPAAL model involves the application-layer only, bottom layers are omitted;
- Castalia provides a set of ready-to-use modules for implementing the whole communication stack;
- modules' physical parameters of bottom layers can be tuned after integration.

In the following, it is shown the tag `system` of the enriched XML file related to the system topology depicted in Fig. 8. Nodes are positioned accordingly to Castalia's coordinate reference system.

```
// @Position(10, 0, 0)
sourcenode := source();

// @Position(0, 10, 0)
node1 := relay(1);

// @Position(20, 10, 0)
node2 := relay(2);

// @Position(10, 20, 0)
node3 := relay(3);

// @Position(30, 20, 0)
node4 := relay(4);

system sourcenode, node1, node2, node3, node4;
```

3.2.2 Parser and Interpreter

The Parser parses the annotated XML file produced by UPPAAL and builds an object-oriented model of the timed automata. Such an object-oriented model is independent of the underlying Simulator. Conversely, the Interpreter is strictly coupled with the Simulator, since it produces the source files that will be bundled with it. In the following, we focus on the files produced by the Interpreter.

Network Configuration The overall network configuration is contained in the file `omnetpp.ini`, which is extracted from the XML tag `system`. Such a file defines the number of nodes, the positioning of them, and the applications running on each layer of their communication stacks. Moreover, it contains all the network's physical parameters, like the latency of the channel, the transmission power of nodes' antennas, the nodes' internal clock, and many others. By tuning such parameters, it is possible to generate several different configurations of the same network, without re-building the Simulator.

Global data structures and type aliasing Global C/C++ headers containing global data structures and type aliasing are extracted from the global XML tag `declaration`. Such data and types are stored in the file `UppaalGlobal.h`, which will be imported by all the classes using global data or types.

Application-layer packet The structure of the application-layer packet, used by all the network nodes, is obtained from the definition of the UPPAAL communication channel, which is contained in the text of the global XML tag `declaration`. In detail, the Interpreter stores the NED description of the packet structure in the file `UppaalPacket.msg`. Such a file will be used during the build of the simulator, for producing the files `UppaalPacket_m.h` and `UppaalPacket_m.cc`, which contains the C++ model of the packet itself. The header `UppaalPacket_m.h` will be imported by all the classes sending and receiving application-layer packets.

Nodes' applications The Interpreter produces one application for each XML tag `template`. Each node of the network runs a certain application, namely template, in its application-layer simple module. From a general point of view, an application is made by:

- one NED description of the simple module executing the application;
- a set of C/C++ files implementing the application itself.

In detail, each application is provided with a Finite State Machine (FSM), that implements the behavior described by the XML template. Referring to the content of the XML tag `template`:

- each `location` accounts for one FSM' state;
- each `transition` accounts for one FSM's transition.

Moreover, the application is provided with the FSM's transition map, used to let the FSM evolve. All of FMS's transitions implements the following abstract functions:

```
bool
AbstractTransition::checkGuard();

bool
AbstractTransition::doSynchronization();

std::string
AbstractTransition::doAssignments();
```

The functions `checkGuard`, `doSynchronization`, and `doAssignments` implement the UPPAAL transition's guard, synchronization and assignments, respectively.

The FSM evolves according to the node's clock, through the execution of transitions, namely performing the UPPAAL transition's assignments. At each clock tick, the application retrieves all the outgoing transitions for the current node from the FSM transition map. Then, it executes the transition that satisfies both the guard and the synchronization conditions. If no transition is possible, the FSM does not evolve in the current clock frame. Conversely, if more transitions can be performed, the application executes one of them randomly.

Nodes synchronization In Castalia, nodes asynchronously communicate with each other. Moreover, when transmitting, nodes broadcast packets. UPPAAL transmission and reception synchronizations are supported in Castalia in two different ways.

Transitions containing a transmission synchronization can be always executed if the related guard is satisfied. A UPPAAL transmission synchronization, for example on channel 0, i.e. `message[0]!`, results in the broadcast of a UppaalApplication packet, as shown in the following code:

```
UppaalPacket* uppaalPacket;
uppaalPacekt = new UppaalPacket(nodeid);
toNewtorkLayer(uppaalPacket, BROADCAST);
```

After the packet is broadcasted, it is received by all nodes positioned inside the sender's transmission range.

To support the UPPAAL reception synchronization, each node is provided with a reception buffer on the application-layer. Such a buffer follows the FIFO policy. Nodes store UppaalPackets into the reception buffer as soon as they are received.

Then, when a reception transition is executed, for example on channel 0, i.e. `message[0]?`, it results in the scanning of the reception buffer, looking for the first UppaalPacket received from Node 0. If the target UppaalPacket is found, then the transition is executed. Otherwise, the FSM does not evolve in the current clock frame.

4 Case study

This section shows a case study with the application of the proposed framework to the *flooding* protocol [16], which is used to forward messages from a source node to all the members of the network, using each node as a relay of the message. Flooding can be used, for example, to distribute code updates to all nodes.

4.1 Application model in UPPAAL

The applications running on network nodes are modeled via parametric timed automata, by exploiting the UPPAAL template system. Such timed automata are parametric with respect to the node `id`. In general, leaving out initialization and final states, network nodes have only one main state from which a set of outgoing transitions starts. Such transitions account for the reception/transmission of packets from/to other network nodes.

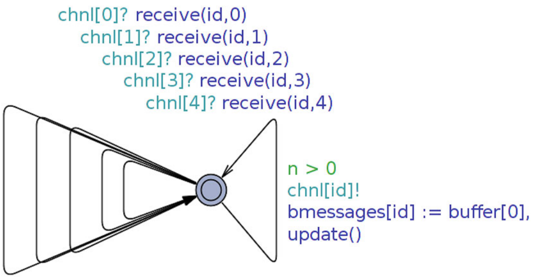


Fig. 7 Abstract model of a generic relay node

Moreover, network nodes are connected through a not blocking broadcast channel. It is worth noting that the transmission of messages results in the broadcast of them. Such a broadcast channel is modeled through the global array `chnl` indexed by the type `node_t`, which defines the size of the array itself. Similarly, we model the messages exchanged between network nodes through the global array `bmessages` indexed by `node_t`. Moreover, we assume that the timestamp uniquely identifies every message. Therefore, we use the global declarations that follow, where the variable `NODES` represents the number of network nodes:

```
const int NODES = ...;
typedef int[0,NODES-1] node_t;
typedef int timestamp;
chan chnl[node_t]:
timestamp bmessages[node_t];
```

In the following, we refer to a WSN made up of five nodes i) one source node; and ii) four relay nodes. The detailed model of such nodes is described in Sect. 4.1.1.

Figure 7 represents the template of a generic node. In detail, the model of a node is made by one location, one edge for the action of sending a message and `NODES` edges for receiving a message from every other node. In the specific topology considered in this case study there are:

– five edges execute the input actions `chnl[0]?` ··· `chnl[4]?`;
– one edge executes the output action `chnl[id]!`.

The input action `chnl[i]?` is the action executed by a node for receiving a message from the node `i`. Similarly, the output action `chnl[i]!bmessages[i]` represents the action executed by node `i` to broadcast the message `bmessages[i]`.

As depicted by Fig. 7, the output action `chnl[id]!` is enabled only if the global variable `bmessages[id]` contains at least one message to send, i.e. if `n > 0`. It is relevant to point out that `n` represents the number of messages to be forwarded, whereas `buffer[0]` represents the messages stored in the head of the local buffer of the node `id`, which is the FIFO buffer that contains all the messages to be forwarded. When a message is sent, it is stored into the global array `bmessages`, i.e. `bmessages[id] = buffer[0]`, then the function `update()` is executed for updating node's local data structures, as described in Sect. 4.1.1.

Conversely, the input action `chnl[i]?` is always enabled. However, messages broadcasted by node `i` are received by node `id` only if nodes `i` and `id` are neighbors. The function `receive(id, i)` implements the receiving of messages from neighbors, as described in Sect. 4.1.1.

4.1.1 Flooding protocol

Flooding [16] is a *one-to-many* routing protocol, in which a dedicated node (the base station) needs to communicate general information to all the nodes of the network. As an example, flooding can be applied for dynamic route discovery. A simple version of flooding behaves as follows: whenever a network node receives a message, it is forwarded to all its neighbors only if it has not already been forwarded; otherwise, it is dropped. Moreover, nodes drop old messages also, when received. In the following, an interesting property of the flooding protocol is stated.

Property P: every node receives all the messages sent by the base station, and every message that was received is then forwarded only once.

Relay nodes Referring to the abstract model of a generic relay node depicted in Fig. 7, nodes are provided with the local data structures that follow, in order to support the flooding protocol.

```
const int MSGS = ...;
typedef int [0,MSGS-1] checker_t;
const int DIM = ...;
typedef in [0,DIM-1] size_t;
timestamp TS;
timestamp buffer[DIM];
int n;
timestamp logger[DIM];
int m;
```

`MSGS` represents the number of messages sent by the base station; `TS` stores the most recent timestamp of received messages; `buffer` is a FIFO buffer that stores messages waiting to be forwarded; and, finally, the buffer `logger` stores the already broadcasted messages. The latter buffer will be used to check the *Property P*.

Moreover, for implementing the flooding algorithm on relay nodes, the functions `receive` and `update` of the timed automaton depicted in Fig. 7 can be specialized as follows.

```
void receive(int j) {
   if ( neighbor(id,j)
        && bmessages[j] > TS
        && n < DIM-1) ) {
       buffer[n] = bmessages[j];
       TS = bmessages[j];
       n++;
   }
}

void update() {
   n--;
   for( i : size_t ) {
      buffer[i-1]=buffer[i];
   }
   if ( m < DIM-1 ) {
       logger[m] = bmessages[id];
       m++;
   }
}
```

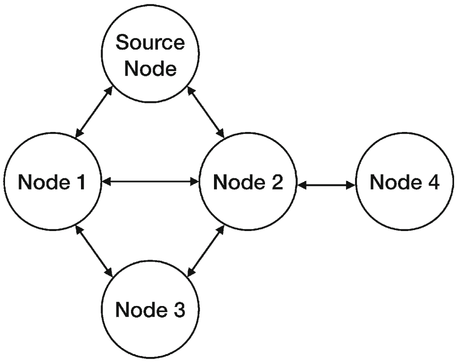


Fig. 8 Network topology

Where `n` and `m` represent the current number of elements stored in `buffer` and `logger`, respectively. In detail, the test `bmessages[j] > TS` in the function `receive` evaluates `false` when the node `id` receives either an old or an already received message. In such cases, the received message is not stored into `buffer` and is dropped. Moreover, after broadcasting a message, the function `update()` executes a backward one-position shift of `buffer`.

Figure 8 shows the network topology we consider. Moreover, we assume the communication range between nodes is one hop. As a consequence, the function `receive(id, i)` of relay nodes is tailored on such network parameters, and returns `true` if the nodes `id` and `i` are one-hop neighbors, `false` otherwise. As an example, `receive(4, 2)` returns `true`, since Node 2 is a one-hop neighbor of Node 4. Conversely, `receive(4, 3)` returns `false`, since Node 3 is two-hops far away from Node 4.

Source node Fig. 9 shows the UPPAAL template that models the base station, named Source node, of the flooding algorithm. The Source node broadcasts a brand new message every `clk` units. Messages sent by Source node are incrementally timestamped, from `1` to `MSGS`, which is the last message sent. After sending `MSGS` messages, the Source node stops transmitting.

Network According to the network topology (Fig. 8), the UPPAAL network is specified as follows.

```
sourcenode := source();
node1 := relay(1);
```

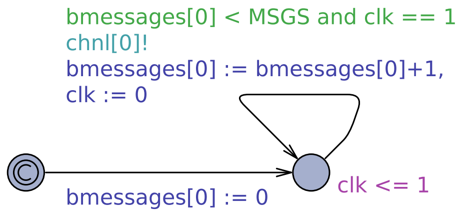

bmessages[0] < MSGS and clk == 1
chnl[0]!
bmessages[0] := bmessages[0]+1,
clk := 0
clk <= 1
bmessages[0] := 0

Fig. 9 Abstract model of the Source node

```
node2 := relay(2);
node3 := relay(3);
node4 := relay(4);
system sourcenode, node1, node2, node3, node4;
```

Where `source` and `relay(id)` represent the templates for the Source node and the Relay nodes, respectively. The *Property P* of the flooding protocol, which was previously described, can be checked via UPPAAL by exploiting the formulas that follow.

```
A<>( forall ( i:checker_t )
       node1.logger[i] == i + 1 )

A<>( forall ( i:checker_t )
       node2.logger[i] == i + 1 )

A<>( forall ( i:checker_t )
       node3.logger[i] == i + 1 )

A<>( forall ( i:checker_t )
       node4.logger[i] == i + 1 )
```

In detail, we test the *Property P* against the content of the buffer `logger` of all relay nodes. For each node, the buffer `logger` stores all the messages forwarded by it. Referring to the formulas above, `logger[i]` represents the `(i+1)`-th message that was forwarded by a certain node.

The *Property P* is proved to be true if, for each relay node, for each `i` such as `i` ∈ `[0, MSGS)`, `logger[i]` stores the timestamp `i+1`. In this case, all nodes forwarded only once all the messages they received.

It is worth noting that UPPAAL tests the formulas above against all the possible execution paths of the protocol. In particular, such formulas have been proved to be true in our attack free simulation scenario.

4.1.2 Modeling attacks

Referring to the network topology shown in Fig. 8, we consider three attacks (i) Packet dropping attack; (ii) Packet tampering attack; and (iii) Packet injection attack.

Packet dropping attack In the first attack, at a random time, the compromised node drops exactly one packet that, instead, should have been sent to its neighbors. Figure 10 shows the model of the compromised node that drops the packet.

Simulations done via UPPAAL show the results that follow.

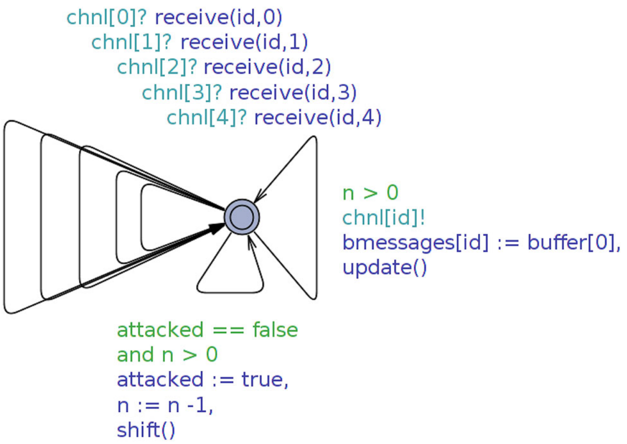


Fig. 10 UPPAAL template modeling the packet dropping attack

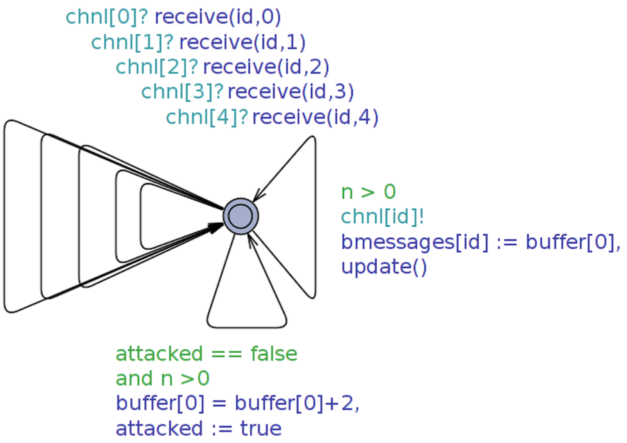


Fig. 11 Template modeling the packet tampering attack

- when the adversary compromises Node 1, the *Property P* described in 4.1.1 is still satisfied, since Node 3 receives a copy of the dropped packet from Node 2, thanks to link redundancy;
- when the adversary compromises Node 2, the *Property P* is not satisfied anymore because Node 4 does not receive a copy of the dropped packet since there is no redundancy on links connecting Node 4 with other network nodes.

Packet tampering attack In the second attack, the compromised node sends exactly one fake packet to its neighbors. Such a packet contains a fake timestamp, which is ahead in time compared to the current time. The reception of the fake packet causes, on the recipient, the discarding of all the genuine packets that carry a timestamp older than the fake one. Figure 11 shows the model of the compromised node that tampers the packet.

Simulations and model checking done via UPPAAL show the following results.

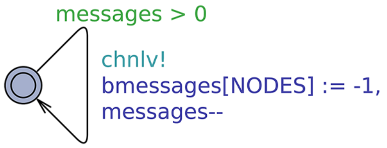


Fig. 12 Template modeling the injection attack

- When the adversary compromises Node 1, the *Property P* is not satisfied if Node 3 receives the fake packet from Node 1 before the genuine packets carrying timestamps older than the fake one from Node 2. Conversely, the *Property P* is satisfied if Node 3 receives from Node 2 all the genuine packets carrying timestamps older than the fake one before the fake packet from Node 1.
- When the adversary compromises Node 2, the *Property P* is not satisfied anymore because Node 4, after receiving the fake packet, discards all the subsequent genuine packets received from Node 4 that carry timestamps older than the fake one.

As shown in Figs. 10 and 11, both the attacks are modeled by adding exactly one transition to the model of the relay node shown in Fig. 7. Both attacks occur at random time and execute only once.

Packet injection attack In the packet injection attack, at random times, an external malicious node sends one message to a victim node. Such a message contains a timestamp that does not interfere with the protocol, as the malicious node is external to the network. Figure 12 shows the UPPAAL model of the malicious node that implements the packet injection attack. Since it is not possible to specify an infinite number in UPPAAL, the malicious node only sends a fixed number of messages, `messages`, using a newly introduced broadcast channel `chnlv`. All the genuine nodes of the network synchronize on `chnlv` but only the designated victim actually receives the message. Simulations done via UPPAAL show that the *Property P* is never affected by the packet injection attack.

4.1.3 Flooding protocol network simulation

The System Integrator bundles the files produced by the Interpreter with ASF on top of Castalia, then builds the simulator. After that, simulations can be performed. It is worth noting that Castalia makes it possible to have several different configurations for the same WSN, without re-build the simulator from scratch. A certain WSN model may have several different configurations that differ from each other due to i) the positioning of the nodes; ii) the latency of the channel; iii) the bottom layers protocols; iv) or other tuning parameters. Batch processing can be used for simulating a large number of different configurations for a certain WSN model, with the aim of validating a large number of different scenarios through simulation.

In the following, a sample of a simulation report provided by ASF over Castalia, when simulating the attack-free scenario, is shown. In detail, it lists the packets received by Node 4 during a simulation run. The clock period of all nodes is 100 ms.

```
...
Node4> at time 0.74 sec received value 5 from
       node 2
Node4> at time 0.84 sec received value 6 from
       node 2
Node4> at time 0.94 sec received value 7 from
       node 2
...
```

In the attack-free scenario, Node 4 receives all the packets sent by the Source node, through Node 2, without repeated messages coming from the same node.

Packet dropping attack The packet dropping attack is described through the Attack Description Language as follows:

```
from 0 s nodes = "2" do {
   var target = RND * 10;
   filter(packet.APP.type == FLOODING
   AND packet.APP.data == target ){
       drop(packet);
   }
}
```

Starting from the beginning, i.e. at time zero, the target node 2 is captured and reprogrammed for dropping the application packet of type `FLOODING` carrying the value `target`. The attack triggers when the packet filter intercepts a packet carrying a certain target value. The target value is chosen as a random integer value in `[0, 10)`, to enforce the random occurrence time of the attack. The monotonic growing integer value contained in the application packet payload ensures that only one packet is dropped, i.e. the packet intercepted by the packet filter that carries the random target value.

When simulating the packet dropping attack, Node 4 never receive the packet dropped by node 2, accordingly to the results provided by UPPAAL. For example, the result of a simulation run of the packet dropping attack is:

```
...
Node4> at time 0.74 sec received value 5 from
       node 2
Node4> at time 0.84 sec received value 6 from
       node 2
Node4> at time 1.04 sec received value 8 from
       node 2
...
```

In this case, when the attack occurs, Node 4 does not receive the packet 7 from Node 2. Conversely, when attacking Node 1, the system always shows robustness against the packet dropping attack, thanks to the redundancy provided by Node 2.

Packet tampering attack The packet tampering attack is described through the Attack Description Language as follows:

```
from 0 s nodes = "2" do {
   var target = RND * 10;
   var fakeValue = target + RND * 10;
   filter(packet.APP.type == FLOODING
   AND packet.APP.data == target ) {
   change(packet.APP.data, fakeValue);
   }
}
```

Like in the packet dropping attack, the random occurrence time is enforced by choosing a random target value, and the monotonic growing value carried by the application packets ensures that only one packet is tampered. When the attack triggers, the intercepted packet's data payload is tampered with the value `fakeValue`, which is ahead in time with respect to the value `target`.

When simulating the packet tampering attack, after receiving the fake packet containing a value ahead in time, Node 4 discards all the subsequent packets carrying seemingly old values, accordingly to the results provided by UPPAAL. For instance, the result of a simulation run of the packet tampering attack, in which `target = 7` and `fakeValue = 12` is:

```
...
Node4> at time 0.75 sec received value 6 from
       node 2
Node4> at time 0.85 sec received value 12 from
       node 2
Node4> at time 0.95 sec DISCARDED value 8, from
       node 2
Node4> at time 1.05 sec DISCARDED value 9, from
       node 2
Node4> at time 1.15 sec DISCARDED value 10,from
       node 2
Node4> at time 1.25 sec DISCARDED value 11 from
       node 2
Node4> at time 1.35 sec DISCARDED value 12 from
       node 2
Node4> at time 1.45 sec received value 13 from
       node 2
Node4> at time 1.55 sec received value 14 from
       node 2
...
```

In this case, when the attack occurs, at 0.85 sec Node 4 receives the packet carrying the fake value 12 (instead of 7). As a consequence, Node 4 discards all the subsequent packets, until the value reaches 13, which is accepted, at 1.45 sec. Like in the packet dropping attack, the system is robust against the packet tampering attack when attacking Node 1, thanks to the redundancy provided by Node 2.

A possible solution to cope with both the drop and the packet tampering attack may be the insertion of redundant paths connecting Node 4 with other network nodes, like Node 3 and Source Node.

Packet injection attack The packet injection attack is described through the Attack Description Language as follows:

```
from 0 s every 200 ms do {
   packet pkt;
   var target = [2];

   create(pkt, APP.Flooding);
   change(pkt.source, 0);
   change(pkt.value, 0);
   put(pkt, target, RX, 0);
}
```

The packet injection attack takes place from the beginning, with an injection rate of one packet every 200ms, i.e. 5 fake packets per second. First, it creates an application-layer packet `pkt` of type `Flooding`, which is intended to be processed by the application that runs on the application-layer of network nodes. The fake packet appears to be sent by Source Node, and carries zero as `value`, which should not interfere with the system operations, accordingly to UPPAAL. Then, the packet, which is meant to be received by Node 2, is injected in the network by an external device. In detail, the packet does not flow through the entire communication stack of Node2, but it is directly stored in the `RX` buffer of its application-layer module. However, for simulating the effects of the reception of such a packet, the Resource Manager module of Node 2 is triggered as if a genuine packet had been received by the Radio module, then decreasing the remaining battery charge, accordingly.

Once received by Node 2, all fake packets are discarded, as shown in the following execution sample:

```
...
Node2> at time 0.20 sec DISCARDED value 0 from
       node 0
Node2> at time 0.21 sec received value 2 from
       node 0
Node2> at time 0.31 sec received value 3 from
       node 0
Node2> at time 0.40 sec DISCARDED value 0 from
       node 0
Node2> at time 0.41 sec received value 4 from
       node 0
Node2> at time 0.51 sec received value 5 from
       node 0
Node2> at time 0.60 sec DISCARDED value 0 from
       node 0
...
```

In this case, Node 2 receives and discards a fake packet every 200 ms, while genuine packets are correctly received and processed by the application. However, the power consumption

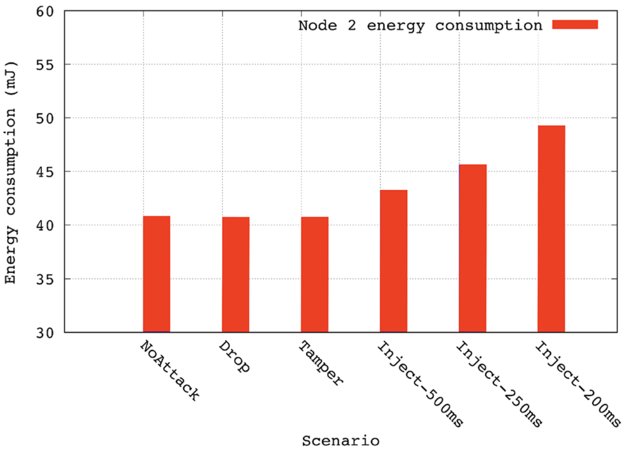


Fig. 13 Energy consumption of Node 2 in attack-free and attack scenarios

Table 1 Energy consumption of Node 2 for each simulation scenario

	Energy consumption(mJ)
No Attack	40.79
Drop	40.68
Tamper	40.71
Injection 500 ms	43.22
Injection 250 ms	45.61
Injection 200 ms	49.24

caused by the reception of fake packets may drain the battery of Node 2, resulting in the breakdown of the network. In such a case, Node 4 will not connect with other nodes anymore, and will remain isolated from the network.

Then, even if the packet injection attack does not interfere with the flooding protocol at application level, it results in draining the battery of the target node. Consequently, it could cause the interruption of the service due to the breakdown of the network earlier than expected.

Power consumption Figure 13 shows the impact of attacks on the power consumption of Node 2. Results were obtained by means of 20 simulation runs, whose length was 600 seconds each. Nodes are equipped with the CC2320 radio chipset. For the packet injection attack, we considered three injection rates (i) 500 ms, i.e. 2 fake packets per second; (ii) 250 ms, i.e. 4 fake packets per second; and (iii) 200 ms, i.e. 5 fake packets per seconds.

As expected, the power consumption of Node 2 among the attack-free scenario, the packet dropping attack scenario and the packet tampering attack scenario, is nearly unchanged. When the packet dropping attack occurs, the Node 2 saves the transmission of exactly one packet. Conversely, when the packet injection attack occurs, the energy consumption of Node 2 grows as the injection rate grows.

Table 1 shows the energy consumption of Node 2 for each simulation scenario. In particular, the energy consumption of Node 2 when the packet injection attack occurs at its maximum extent grows of 20.7%.

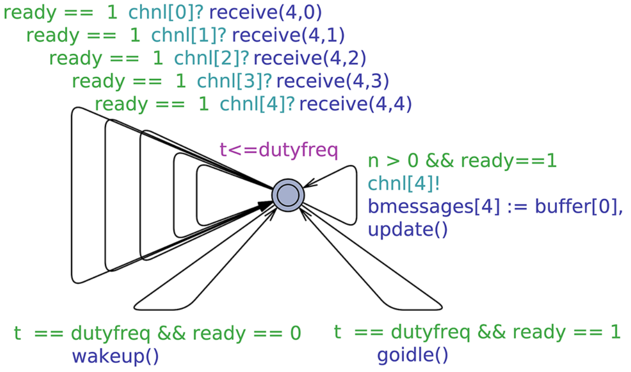

ready == 1 chnl[0]? receive(4,0)
ready == 1 chnl[1]? receive(4,1)
ready == 1 chnl[2]? receive(4,2)
ready == 1 chnl[3]? receive(4,3)
ready == 1 chnl[4]? receive(4,4)
t<=dutyfreq
n > 0 && ready==1
chnl[4]!
bmessages[4] := buffer[0],
update()
t == dutyfreq && ready == 0
wakeup()
t == dutyfreq && ready == 1
goidle()

Fig. 14 Model for the relay node 4 with duty cycle

A possible solution for mitigating the impact of the packet injection attack may be represented by synchronizing nodes activity by means of a duty cycle.

4.2 Abstract model refinement

This section shows the UPPAAL implementation of a duty cycle as a possible countermeasure for mitigating the power drain issue.

Duty cycle The UPPAAL template of a node of the network has been extended with a duty cycle: the node periodically turns on/off with a periodicity of `dutyfreq` time units (see Figure 14 for the resulting model of the node with `id` 4). The template of the node is extended with a variable named `ready` which represents the state of the node: `ready == 1` implies that the node is on while `ready == 0` implies that the node is off. The node can receive and send messages only if it is on (condition `ready == 1` on all the guards of the transitions with a synchronization on `chnl`) and the functions `wakeup()` and `goidle()` change the value of `ready` accordingly. The invariant `t<=dutyfreq` together with the guards on the `wakeup()` and `goidle()` transitions forces the periodic transition of the `ready` variable.

It is important to notice that the value of `dutyfreq` is not a constant value: the functions `wakeup()` and `goidle()` can modify its value in order to obtain different balancing between radio on and radio off. For example if `wakeup()` sets the value of `dutyfreq` to be three times greater than the value set by the `goidle()` than it is possible to achieve a 75% duty cycle.

4.2.1 Attacks on refined model

Under the assumption of a correct synchronization between all the nodes of the network the *Property P* is still satisfied in

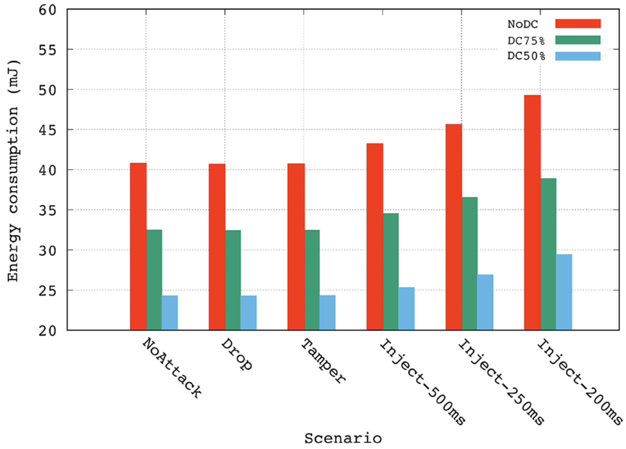


Fig. 15 Energy consumption of Node 2 with different duty-cycles in the considered simulation scenarios

the attack free scenario and the results of the analysis depicted in Sect. 4.1.2 still hold.

The UPPAAL template of a node has been extended with a variable named `receivedmessages`, which is a counter of the messages received, and a variable named `links` which represents the number of reachable genuine nodes; these new variables can be used to write the following formula, where

```
( node1.links+
 node2.links+
 node3.links+
 node4.links)*MSGS
```

is the number of expected messages sent among the genuine nodes:

```
A[](node1.receivedmessages +
    node2.receivedmessages +
    node3.receivedmessages +
    node4.receivedmessages
              > ( node1.links+
                  node2.links+
                  node3.links+
                  node4.links)*MSGS
  )
```

The formula states that for all possible execution trace, in all the states of the system, the sum of the messages received by each node is less that or equal to the number of expected messages. For the system without the packet injection attack this formula is true, while the introduction of the packet injection attack falsifies it.

4.2.2 Refined model network simulation

The refined model is simulated against the previous scenarios. The latter simulation results fit with the former results, described in Sect. 4.1.3, for both the dropping attack and the tampering attack scenarios, as well as the attack-free scenario. Conversely, the implementation of a duty cycle on nodes helps to reduce the impact of the packet injection attack, but does not nullify its effects, since Node 2 continues to receive fake packets in the time frames in which it is fully operative. It is worth considering that the objective of the injection attack is to drain nodes' batteries without being detected. Then, a massive injection of fake packets, for exploiting the time frames in which the target node is operative, would result in a very high, if not prohibitive, transmission rate. Furthermore, such an attack would become actually a Denial of Service attack and would be promptly detected. In fact, the chances of detecting such an attack grows as the injection rate grows.

Table 2 Overview of energy consumption of Node 2 with various duty cycles

	Energy consumption (mJ)		
	No DC	DC 75%	DC 50%
No attack	40.79	32.49	24.31
Drop	40.68	32.43	24.30
Tamper	40.71	32.45	24.33
Injection 500 ms	43.22	34.52	25.31
Injection 250 ms	45.61	36.55	26.90
Injection 200 ms	49.24	38.89	29.42

In detail, the latter simulations consider Duty Cycles (DC) of 75% and 50%, respectively. As shown in Fig. 15, power consume decrease as the duty cycle decreases. On the other hand, too long downtimes may adversely affect the network throughput, and ultimately be incompatible with the application.

Table 2 shows the energy consumption of Node 2 for all the simulation scenarios. In particular, when the packet injection attack occurs at its maximum extent, the energy consumption of Node 2 decreases of (i) 4.7% if using a duty cycle of 75%; and (ii) 27.9% if using a duty cycle of 50%; with respect to the attack free scenario with no duty cycle. So, a duty cycle of 50% is an effective solution for mitigating the effects of the packet injection attack performed at his maximum extent.

5 Conclusions

This paper presents ongoing work on a security-aware design approach for WSN applications and protocols. Such an approach exploits the integration between formal methods and network simulators enhanced for reproducing security attacks. This enables WSN designers to gather valuable insights on the realistic behavior of the abstract model since design time, thus helping them to recognize design flaws and security-related issues, and then select and validate appropriate countermeasures.

As a proof of concept, we have built a framework that integrates the model checker UPPAAL with the network and attack simulator ASF. Our framework has been used to study the robustness of a flooding application-layer protocol against a set of predefined attacks and highlighted a design issue related to power consumption when a packet injection attack has occurred. Then, a countermeasure has been implemented and its effectiveness evaluated. The case study demonstrated that it is fundamental to obtain concrete measurements about the behavior of the system when attacks occur.

As further work, new evaluation metrics based on security aspects can be introduced. Such metrics can be used to conduct a deeper analysis of the effects of the attacks on the system, and to rank them in order to select the most significant countermeasures. Moreover, our framework can be integrated with additional off-the-shelf network simulators, e.g. INET and Simulink, for extending its application domain to more complex systems, like integrated clinical environments, or smart grids. Finally, additional formal method tools, such as theorem provers can be considered, to overcome the problem of state space explosion, which is usually raised by model checkers when modeling complex systems.

Acknowledgements The authors would like to thank the anonymous referees for their useful comments and suggestions.

Funding Open access funding provided by Università di Pisa within the CRUI-CARE Agreement.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**(2), 183–235 (1994)
2. Boulis, A., Pediaditakis, D.: Castalia - A simulator for Wireless Sensor Networks and Body Area Networks, version 3.3, User's Manual (2013). https://github.com/boulis/Castalia/blob/master/Castalia
3. Behrmann, G., David, A., Larsen, K.G.: A Tutorial on UPPAAL 4.0 (2006). http://www.it.uu.se/research/group/darts/papers/texts/new-tutorial.pdf
4. Bernardeschi, C., Masci, P., Pfeifer, H.: Early prototyping of wireless sensor network algorithms in PVS. In: Computer Safety, Reliability, and Security, 27th International Conference, pp. 346–359 (2008)
5. Bernardeschi, C., Masci, P., Pfeifer, H.: Analysis of wireless sensor network protocols in dynamic scenarios. In: Stabilization, Safety, and Security of Distributed Systems, 11th International Symposium, pp. 105–119 (2009)
6. Bhargavan, K., Gunter, C., Lee, I., Sokolsky, O., Kim, M., Obradovic, D., Viswanathan, M.: Verisim: formal analysis of network simulations. IEEE Trans. Softw. Eng. **28**(2), 129–145 (2002)
7. Bhushan, B., Sahoo, G.: Recent advances in attacks, technical challenges, vulnerabilities and their countermeasures in wireless sensor networks. Wireless Personal Commun. **98**(2), 2037–2077 (2018)
8. Bolton, C., Lowe, G.: Analyses of the reverse path forwarding routing algorithm. In: Intl. Conf. on Dependable Systems and Networks Proceedings, pp. 485–494. IEEE Computer Society (2004)
9. Buck, J., Ha, S., Lee, E.A., Messerschmitt, D.G.: Ptolemy: a framework for simulating and prototyping heterogeneous systems. Int. J. Comput. Simulat. **4**, 155–182 (1994)
10. Cardenas, A.A., Roosta, T., Sastry, S.: Rethinking security properties, threat models, and the design space in sensor networks: a case study in scada systems. Ad Hoc Netw. **7**(8), 1434–1447 (2009)
11. Dini, G., Tiloca, M.: Asf: An attack simulation framework for wireless sensor networks. 2012 IEEE 8th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob) pp. 203–210 (2012)
12. 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 (2013)
13. Fatima, T., Saghar, K., Ihsan, A.: Evaluation of model checkers SPIN and UPPAAL for testing wireless sensor network routing protocols. In: 2015 12th International Bhurban Conference on Applied Sciences and Technology (IBCAST), pp. 263–267 (2015)
14. Fonseca, J., Vieira, M.: A survey on secure software development lifecycles. In: Software Development Techniques for Constructive Information Systems Design, pp. 57–73. IGI Global (2013)
15. Healy, M., Newe, T., Lewis, E.: Security for wireless sensor networks: A review. In: 2009 IEEE Sensors Applications Symposium, pp. 80–85 (2009)
16. Heinzelman, W., Kulik, J., Balakrishnan, H.: Adaptive protocols for information dissemination in wireless sensor networks. In: Proceedings of International Conference on Mobile Computing and Networking, pp. 174–185. ACM (1999)
17. Henderson, T., Riley, G., Floyd, S., Roy, S., et al.: The NS Manual (2019)
18. Hudaib, A., Alshraideh, M., Surakhi, O., Alkhanafseh, M.: A survey on design methods for secure software development. Int. J. Comput. Technol. **16**, 7047–7064 (2017)
19. Lazarescu, M.T., Lavagno, L.: Wireless Sensor Networks, pp. 1–42. Springer, Netherlands, Dordrecht (2017)
20. Mohammad, A., Alqatawna, J., Abushariah, M.: Secure software engineering: Evaluation of emerging trends. In: 2017 8th International Conference on Information Technology (ICIT), pp. 814–818 (2017)
21. Nair, S., Cardell-Oliver, R.: Formal specification and analysis of performance variation in sensor network diffusion protocols. In: Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems Proceedings, pp. 170–173. ACM (2004)
22. Rashid, A., Hasan, O., Saghar, K.: Formal analysis of a ZigBee-based routing protocol for smart grids using UPPAAL. In: 2015 12th International Conference on High-capacity Optical Networks and Enabling/Emerging Technologies (HONET), pp. 1–5 (2015)
23. Shostack, A.: Threat Modeling: Designing for Security. Wiley, Hoboken (2014)
24. Varga, A.: OMNeT++ (2014). https://doc.omnetpp.org/omnetpp4/manual