WiNeSim: A Wireless Network Simulation Tool

Matthew Emerson
Institute for Software Integrated Systems
Vanderbilt University
Nashville, TN, 37203
mjemerson@isis.vanderbilt.edu

Janos Mathe
Institute for Software Integrated Systems
Vanderbilt University
Nashville, TN, 37203
jmathe@isis.vanderbilt.edu

Sean Duncavage
Institute for Software Integrated Systems
Vanderbilt University
Nashville, TN, 37203
sduncavage@isis.vanderbilt.edu

Abstract

We provide an overview of WiNeSim, a highly extensible wireless network modeling and simulation tool with a network attack-modeling component. WiNeSim provides a high-level graphical modeling interface for the rapid declarative specification of network configurations, including the selection of node hardware, MAC protocols, and routing protocols. Furthermore, it enables users to specify certain network nodes to act as "smart" attackers who adapt their behavior and attack styles based on perceived network conditions in accordance with user-specified algorithms given as timed automata. WiNeSim is designed to easily integrate new network protocols and attack styles.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: Simulation—*General*

General Terms

Wireless networking, Simulation, Security

Keywords

Domain-Specific Modeling, Model-Integrated Computing

1 Introduction

Security is an ever-growing concern in the arena of wireless networking, especially for embedded wireless sensor networks with limited resources available for implementing security measures. The typical approaches for the analysis of wireless networks and protocols are simulation-based, with the expectation that high-fidelity protocol and system models can yield insights into network performance and security. These simulations may be used to gather relative performance metrics for specific algorithms, or explore the performance of an algorithm relative to network topology or node characteristics (computing power or mobility). They may even examine the effects of malicious attacks or node failures. Network simulation frameworks such as ns2 [ns2], or OMNeT++ [IKH$^+$] can aid in the implementation of simulations; however, there are few widely-available libraries or modules capturing network protocols, hardware platforms, and network attack styles. Consequently, these simulation efforts largely represent a tremendous duplication of work. Because of the significant amount of time that must be expended to continually re-develop such basic simulation modules, network simulations are often limited to the examination of point scenarios, for example the process and effects of a single attack or style of attack against a particular network configuration.

There is a great need in the wireless networking arena for a way to rapidly define and examine more complex, dynamic network attack scenarios. It would be especially useful to be able to capture the behavior of "smart" attackers who carry out a well-developed plan of attack based on known network vulnerabilities, alter or tailor the attack plan based on detectable dynamic network conditions, and even coordinate with other attackers. But more than the attacker behavior must be specified; the network characteristics such as the details of the node hardware, the protocols used, and the messaging patterns still need to be captured too. However, the thought of capturing such a large number of details in source code using a framework such as OMNeT++ is not the most appealing. Completely hand-coding a simulation of the above-described level of complexity to examine a single networking scenario may produce a buggy, brittle, tightly-coupled end result that is difficult to maintain and reuse. This is especially true if the researcher is not already intimately familiar with ns2 or OMNeT++.

It would be useful to have a higher-level interface for harnessing the power of network simulation frameworks without being exposed to all of their complexity. Given the proper level of abstraction, the definition of complex network scenarios can be rapid and simple. Model Integrated Computing (MIC) is a powerful model-based approach which advocates the use of graphical *domain-specific modeling languages* for the design and analysis of complex systems. DSMLs are tailored to the concepts, relationships, needs, and constraints of their particular system domains. Past efforts in the Model Integrated Computing initiative have focused on the generation of custom simulations from domain-specific models of embedded systems. Specifically, [LDNA03] describes the MILAN framework. MILAN is a single unified domain-specific modeling environment for capturing a broad category of embedded systems, and input to multiple simulators can be generated from a MILAN embedded system model. MILAN facilitates the rapid evaluation of performance characteristics such as power use, latency, and throughput;

however, because it abstracts away many of the implementation details of modeled systems, it is easier and more rapid to capture an embedded system using MILAN than than a lower-level design tool.

Motivated by the above issues and continuing in the tradition of MILAN, we propose a new tool for wireless network simulation, WiNeSim. WiNeSim includes a domain-specific modeling language for capturing simulation scenarios for wireless embedded systems, including security attack scenarios. WiNeSim models translate into executable OMNeT++ simulations, but users are not exposed to the full functionality of OMNeT++ when modeling with WiNeSim. This allows users to rapidly evolve simulation experiments that focus on a few key aspects of wireless network design: MAC and routing protocol selection, network topology, and attacker behavior. Like MILAN models, WiNeSim models will enforce correctness-by-construction. In this way, WiNeSim can build upon an existing network simulation framework to take a step toward easing the analysis of new protocols, network configurations, attack patterns, and security defense measures. The WiNeSim tool chain is designed to be highly modular and can easily be extended to support additional node types, protocols, and network attack styles. WiNeSim is currently a work-in-progress.

2 WiNeSim Design

This section describes WiNeSim's modeling interface and the toolchain that enables the simulation of WiNeSim models. WiNeSim itself consists of a graphical modeling language, a set of pre-implemented OMNeT++ components, and a code-generating model interpreter that builds up simulation configurations and OMNeT++ modules implementing network attacker behavior from the WiNeSim models. WiNeSim builds on previous work in network simulaion, most notably the OMNeT++ simulation framework and the SENSIM platform, a set of OMNeT++ interfaces for simulating sensor networks [MES$^+$05][SEN]. The overall structure of the WiNeSim toolchain is given in Figure 1

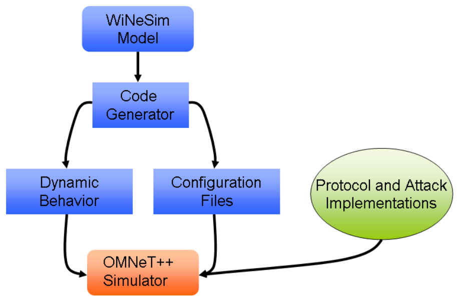


Figure 1. WiNeSim Toolchain

2.1 Modeling with WiNeSim

WiNeSim includes a high-level modeling interface implemented as a domain-specific modeling language for the Generic Modeling Environment [LMB$^+$01], the graphical MIC modeling platform. This language consists of a number of intuitive entities and relationships. The abstract syntax for the WiNeSim modeling langauge is specified in GME using a graphical UML metamodel (Figure 2). A simple example WiNeSim model is given in Figure 3, and the WiNeSim modeling constructs are described below:

- **Nodes** capture various classes of embedded hardware platforms. In simulation, the differences between various node types manifests at the physical layer in terms of available computational, radio, and energy resources. Nodes also have physical coordinates for determining network reachability, and are refined in a scenario to capture the networking protocols, messaging patterns, and network attacks manifested by the node. It is possible to extend the the analysis capabilities of WiNeSim by adding additional node types to the modeling language. In figure 3, **baseStation** is a node of the generic BaseStation class. All BaseStation-class Nodes share a common set values for various physical properties, including CPU and radio power ratings, radio radius, the amount of energy consumed by the radio while transmitting, receiving, or idle, and the battery capacity.
- **Wireless Channels** represent logically discrete communication channels in the shared media used for wireless communication. All of the Nodes connected to a Wireless Channel object may communicate using that channel if they are in range of one another. Communications through different wireless channels do not interfere with one another. The delay and error rate due to physical disruptions associated with particular channel are included as parameters.
- **MAC and Network Layers** are used to configure the protocol stack of the Node which contains them. Each node must contain one and only one MAC Layer object as well as one and only one Network Layer object. The analysis capabilities of WiNeSim can be extended by adding new MAC and Network layer configuration options.
- **Application Layers** permit the user to characterize patterns of message generation, include the rate of generation and the message size. These factors are given using uniform distribution curves. Like the MAC and Network Layer models, Application Layers are sub-models of Nodes.
- **Modes** give the dynamic behavior of a node at the MAC and Network protocol layers. Each MAC or Network layer contains a timed automata with Modes as states, where each Mode indicates either a step in a network attack or simply the normal operation of the protocol. It is possible to extend the the analysis capabilities of WiNeSim by adding new attack types. Note that the automata do not specify the details of the normal operation of the protocols or attack steps – rather, single Modes are used to abstract entire protocols and attack steps. In Figure 3, the MAC layer of the attackingSensor can operate in normal, passive-listening, or MAC-flooding modes.
- **Mode Transitions** dictate the dynamic evolution of Node behavior during a simulation. Each transition can specify a time constraint, a constraint based on the success or failure of the current Mode, and a clock manipulation action. So, with WiNeSim users can model attacker behavior that varies based on timing and on the success or failure of previous attacks In Figure 3, the operation of the MAC layer of the attackingSensor changes dynamically at simulation run-time. The attackingSensor executes the normal S-MAC protocol for the first minute of simulation, then begins to try and determine the largest recipient of messages on the network. This is an instance of the Passive Listening attack. If it fails to find any node within three minutes that leads significantly in messages receives, it will continue to passivly listen. However, if it does find a primary recipient, it will execute a MAC Flooding attack versus that node for four minutes, then go back to passively listening. The attackingSensor's attack behavior varies as time passes and as it observes the network.

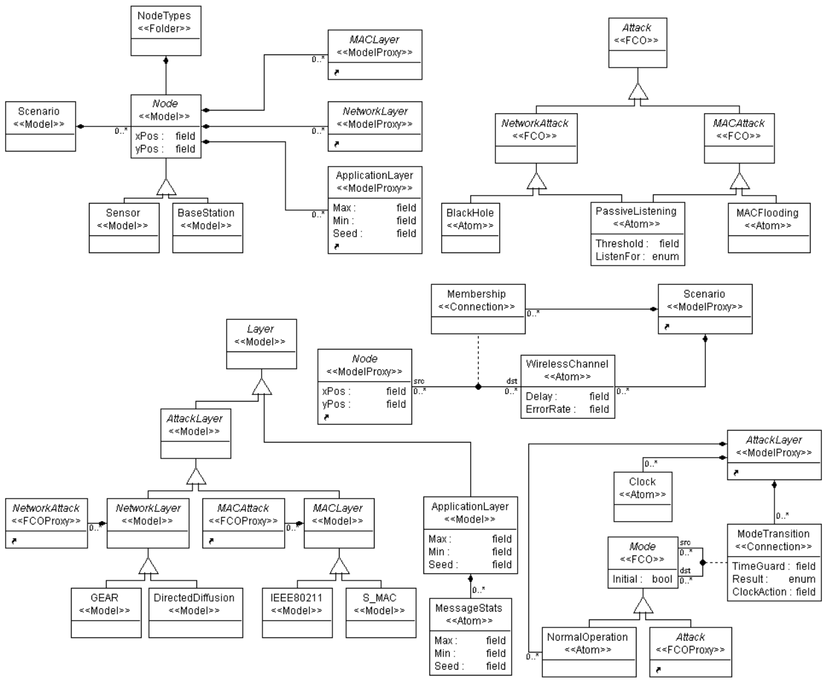


Figure 2. WiNeSim DSML Metamodel

2.2 OMNeT++ and WiNeSim

WiNeSim builds on our reimplementation of the SENSIM project. SENSIM extended the standard OMNeT++ framework with additional capabilities for modeling wireless sensor networks. It provides base classes and inheritable interfaces for MAC protocols, routing protocols, physical components (including CPU, radio, and battery), and wireless channels. By extending these base classes, it is possible to create executable simulations for wireless networks involving different platform types using a highly modular approach. Consequently, WiNeSim is designed to integrate pre-impelemented modules written in C++ which extend the SENSIM framework. For each MAC protocol, network protocol, and network attack style, an appropriate SENSIM/OMNeT++ module must be implemented. However, the WiNeSim Code Generator automatically generates the code for network attacker behaviors and also generates configurations to parameterize the hardware of each simulated node with the proper characteristics. The structure of the OMNeT++ simulation code for each Node in the network is given in Figure 4. The vertical arrows show the propagation of messages through the network stack. The horizontal arrows show the request and management of platform resources through a special module called the **CoOrdinator**. The CoOrdinator essentially plays the role of a simple operating systems for managing the hardware resources of a Node on behalf of the networking software and applications.

As stated, WiNeSim allows users to select from a list of network attack primitives to build up complex attack behaviors and test network performance under adverse conditions. Past efforts in the categorization of wireless attacks typically group them by the type of property compromised: confidentiality, integrity, and availability [Lou01]. An attack on confidentiality allows a malicious user access to private information, such as the body of a text or a communication session. Integrity attacks weaken the ability of the network to function appropriately and may result in slow response times or other improper network operation. If availability is compromised, the node loses its ability to retain connectivity in the network. WiNeSim takes a different approach to the categorization of network attacks. In WiNeSim, we classify attacks according to the layers of the network stack through which they operate. This choice manifests itself in the way attacker behavior is modeled: the attack primitives, such as Passive Listening and MAC Flooding, are always scoped to a particular layer of the protocol stack. The categorization is an important design decision which is intended to support WiNeSim's modularity and extensibility, and also to help us decide which attacks are appropriate for analysis with WiNeSim. Figure 5 summarizes the categorizations of some common attacks [DS03][Lou01]. Note that these attacks are assumed to be perpetrated by trusted nodes within the network.

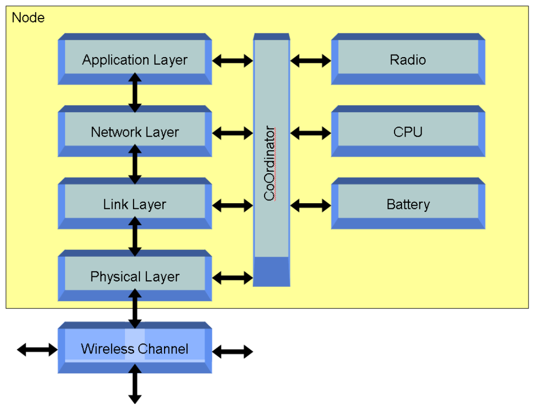


Figure 4. Code Structure for Each Wireless Node

In order to implement a network attack primitive, an appropriate variation of one or more of the normal protocol layer modules must be implemented to carry out the attack. For example, a node using a

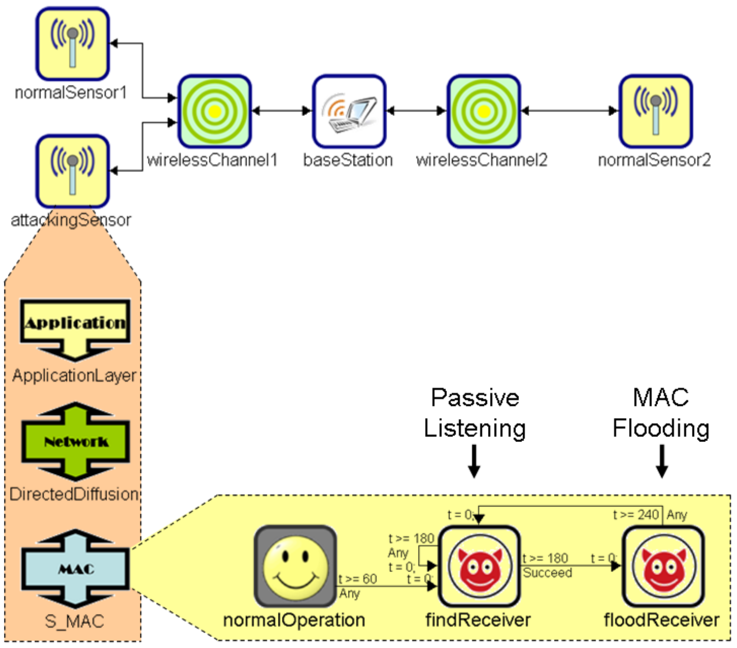


Figure 3. WiNeSim Example Model

Black Hole attack really only manifests that behavior at the routing layer – the MAC and physical layers behave normally. So, we must implement "Black Hole versions" of each implemented networking protocol in order to fully support that style of attack. Then, whether the user wants to create a node that routes data using Directed Diffusion, GEAR, or some other routing protocol, a Black Hole variant will be available if the user wants the node to exhibit that style of attack as part of its dynamic behavior.

Because the primitive modules (protocol layers, network channels, attack styles, and hardware components) are pre-implemented, the task of the WiNeSim domain-specific modeling language is to capture the configurations that organize these modules into simulations. First the nodes should be configured with protocols that capture their behavior, then the nodes and wireless channels may be composed into full network configurations. Because the protocols and primitive attack steps are captured in independent modules, they can be freely combined in a plug-and-play manner by users to explore different network set-ups.

2.3 The Code Generator

A key part of the WiNeSim toolchain is the model interpreter that generates OMNeT++ NED files, INI files, and network attacker behavior from WiNeSim graphical models. NED files capture the network composition and topology for a simulation. In terms of WiNeSim, this means that NED files describe the nodes, the protocols they use, the attack styles they exhibit, and the connectivity of nodes through wireless channels. The purpose of NED files is to organize both the pre-implemented and code-generated SENSIM-based OMNeT++ modules into node types and build communication paths between them. The OMNeT++ core then takes the NED file and the modules that implement the protocols, attacks, and node behaviors and integrates them into an executable simulation (Figure 1). In all, the Code Generator produces one simulation INI file, one NED file for the top-level scenario, a NED file for each node type, A NED file, source file, and header file for each MAC and routing layer type, and a NED file for each MAC and routing mode (protocol or attack style). The header and source files for the modes must be pre-implemented.

Figure 6 shows the NED file code-generated for the MAC layer of the **attackingSensor**. Here, the submodules which correspond to the different MAC layer modes. Figures 7 and 8 show respectively the automatically-generated header and source files for the C++ class that implements the OMNeT++ module referenced in the NED file. The timed automata captured in the example model has been transformed into C++ code that switches the module providing the module providing the MAC layer implementation according to the conditions specified in the mode transitions in the model. The *start, stop*, and *processMessage* methods belong to the WiNeSim-LayerImpl interface. *start* and *stop* are used to govern the activity of implementations which operate continuously, such as the module for MAC flooding. *processMessage* is used to have a layer manipulate a network packet.

3 Extending WiNeSim

WiNeSim was designed with extensibility in mind, in no small part because it natively includes only a very limited set of protocols, attacks, and node types. Adding support for a new MAC protocol (for example) consists of only four steps:

1. Define a new modeling object in the WiNeSim language to represent the protocol. This is easily done by subtyping the abstract MACLayer class in the WiNeSim GME metamodel (Figure 2).
2. Re-interpret the WiNeSim GME metamodel to obtain an updated WiNeSim graphical modeling environment including the new protocol. This is an automatic process that takes approximately ten seconds.
3. Extend the WiNeSimLayerImpl C++ class provided by the our reimplementation of the SENSIM framework with an im-

Attack Name	Compromises	Type / Explanation	Layers Attacked
Frequency jamming	Availability	Denial of Service / Continuously broadcast at the transmission frequency	Physical
Node theft or destruction	Availability	Making the node unusable	Physical
Traffic Analysis / Wiretapping	Confidentiality	Passive Listener / listen to gain logistic information b/w adjacent nodes (frequency, location, time of communications)	Link
MAC flooding	Availability	Denial of Service	Link
Replay attack	Integrity / Confidentiality	Spoofing, Impersonation / Session hijacking, address spoofing, forge origin of data	Link + Network
Random or targeted packet dropping	Integrity	Packet Dropping and Message Blocking	Link
Discover MAC address	Confidentiality	Encryption Attack	Link
Kill the battery of nodes within range with unnecessary activity [13][14]	Availability	Energy Consumption	Link
Routing Table Overflow (route requests) [13],	Availability	Denial of Service, Data flooding	Network
Dropping packets to deliberately slow the network	Integrity	Packet Dropping and Targeted Message Blocking	Network
Black Hole	Availability	Advertise shortest path, drop all packets [13]	Network
Routing information discovery (known plaintext attack [15])	Confidentiality	Encryption Attack	Network
Kill the battery of a chosen node with	Availability	Energy Consumption	Network

Figure 5. Categorizations of Network Attacks

plementation of the MAC protocol, making sure to conform to a simple set of naming conventions assumed by the Code Generator.

4. Create a NED file for the simple module that corresponds to the MAC protocol implementation, again following a set of simple naming conventions.

So, in extending WiNeSim a developer could concentrate primarily on the implementation of their protocol for simulation purposes and be assured that the module they create will "plug-and-play" with the rest of the framework. Developers need not worry about altering the WiNeSim Code Generator or any of the source code files included with WiNeSim.

3.1 Conclusion and Future Goals

WiNeSim is a promising solution for wireless network modeling and simulation. It offers users an intuitive, high-level interface for building simulatable wireless network models consisting of multiple node hardware types and utilizing multiple routing and MAC protocols. Furthermore, it includes a capability for rapidly building in complex dynamic network attacks to test protocol and network security versus "smart" attackers. WiNeSim encourages the development of modular, independent protocol and network attack simulation implementations which can be reused and composed to rapidly test new protocols.

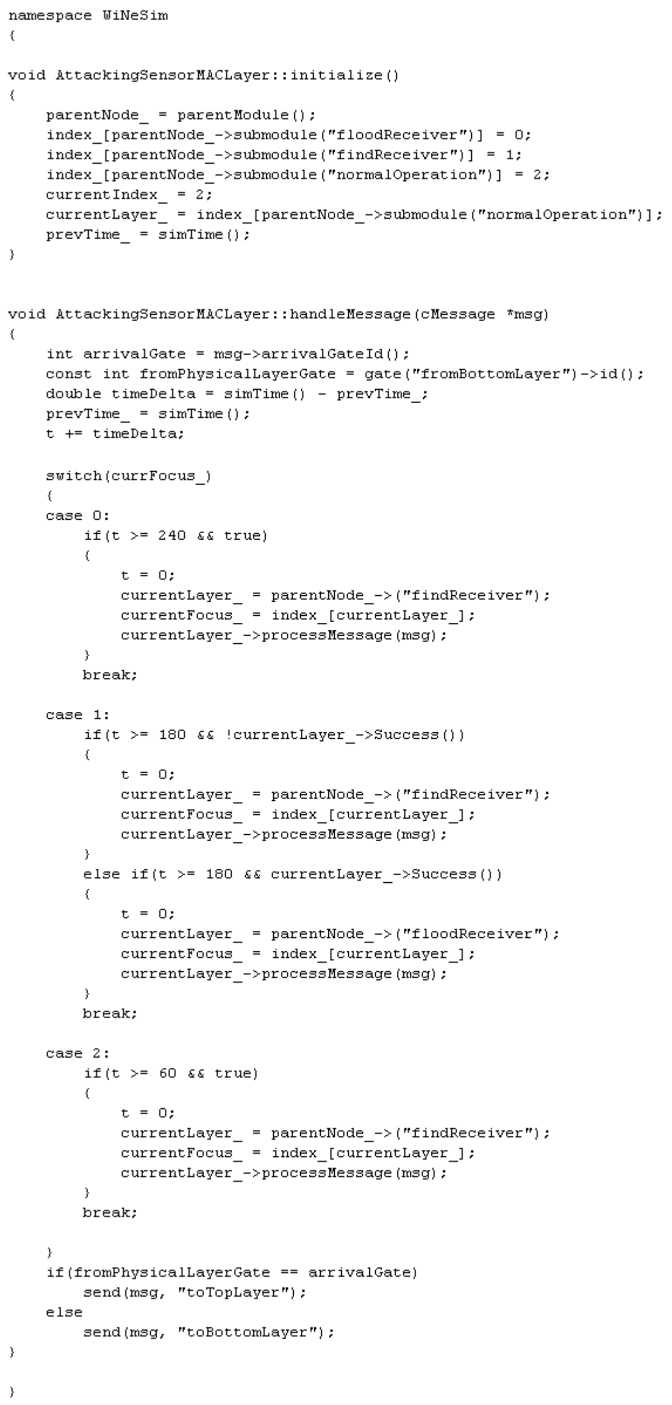
WiNeSim is currently a work-in-progress. Predictably, the piece most lacking is a significant useful number of protocol implementations and network attack styles that can be incorporated into the framework. Currently we have completed the graphical WiNeSim modeling environment, the code generator which produces OMNeT++ NED files as well as C++ header and source files for the

```
import "S_MAC.ned";
import "MACFlooding.ned";
import "PassiveListening.ned";

module AttackingSensorMACLayer
    gates:

        in:
            fromTopLayer,
            fromBottomLayer;
        out:
            toTopLayer,
            toTopLayer;

    submodules:

        normalOperation : S_MAC;

        floodReceiver : MACFlooding;

        findReceiver : PassiveListening;
            parameters:
                Threshold = 0.000000,
                ListenFor = "Receiver";


endmodule
```

Figure 6. attackingSensor MAC Layer NED File

```
#ifndef ___AttackingSensorMACLayer_H___
#define ___AttackingSensorMACLayer_H___

#include <map>
#include "CommonIncludes.h"
#include "WiNeSimLayer.h"

namespace WiNeSim
{

class AttackingSensorMACLayer
{
protected:
    std::map<WiNeSimLayerImpl*, int> index_;
    cModule *parentNode_;
    int currentIndex_;
    double prevTime_;
    double t;

public:
    AttackingSensorMACLayer(const char *name,
       cModule *parentModule, unsigned stacksize);

};

}

#endif
```

Figure 7. attackingSensor MAC Layer Header File



```
namespace WiNeSim
{

void AttackingSensorMACLayer::initialize()
{
    parentNode_ = parentModule();
    index_[parentNode_->submodule("floodReceiver")] = 0;
    index_[parentNode_->submodule("findReceiver")] = 1;
    index_[parentNode_->submodule("normalOperation")] = 2;
    currentIndex_ = 2;
    currentLayer_ = index_[parentNode_->submodule("normalOperation")];
    prevTime_ = simTime();
}


void AttackingSensorMACLayer::handleMessage(cMessage *msg)
{
    int arrivalGate = msg->arrivalGateId();
    const int fromPhysicalLayerGate = gate("fromBottomLayer")->id();
    double timeDelta = simTime() - prevTime_;
    prevTime_ = simTime();
    t += timeDelta;

    switch(currFocus_)
    {
    case 0:
        if(t >= 240 && true)
        {
            t = 0;
            currentLayer_ = parentNode_->("findReceiver");
            currentFocus_ = index_[currentLayer_];
            currentLayer_->processMessage(msg);
        }
        break;

    case 1:
        if(t >= 180 && !currentLayer_->Success())
        {
            t = 0;
            currentLayer_ = parentNode_->("findReceiver");
            currentFocus_ = index_[currentLayer_];
            currentLayer_->processMessage(msg);
        }
        else if(t >= 180 && currentLayer_->Success())
        {
            t = 0;
            currentLayer_ = parentNode_->("floodReceiver");
            currentFocus_ = index_[currentLayer_];
            currentLayer_->processMessage(msg);
        }
        break;

    case 2:
        if(t >= 60 && true)
        {
            t = 0;
            currentLayer_ = parentNode_->("findReceiver");
            currentFocus_ = index_[currentLayer_];
            currentLayer_->processMessage(msg);
        }
        break;

    }
    if(fromPhysicalLayerGate == arrivalGate)
        send(msg, "toTopLayer");
    else
        send(msg, "toBottomLayer");
}

}
```

Figure 8. attackingSensor MAC Layer Source File

layer-specific timed automata. We have also completed much of the generic configurable node and network structures, including the battery, CPU, coordinator, physical layer, application layer, and wireless channel implementations (see Figure 4). The scope of WiNeSim may be increased in the future; for example, the dynamic network node behavior modeling capability could be taken much further. In addition to enabling the modeling "smart" attackers, we could support the dynamic modeling of "smart" defenders, who work to detect and circumvent network attacks. It would also be useful to provide a much wider set of network properties the user can work with when defining the mode transition guards to allow even more complex behavioral modeling. We also plan to add multiple options for node mobility into WiNeSim.

4 Additional Authors

Additional authors: Janos Sztipanovits (Institute for Software Integrated Systems, email: `sztipaj@isis.vanderbilt.edu`).

5 References

[DS03] Welch D. and Lathrop S. Wireless security threat taxonomy. In *Information Assurance Workshop, 2003. IEEE Systems, Man and Cybernetics Society*, pages 76–83, June 2003.

[IKH+] S. Imre, Cs. Keszei, D. Holls, P. Barta, and Cs. Kujbus. Simulation environment for ad-hoc networks in omnet++. Available from: `http://www.omnetpp.org/links.php`.

[LDNA03] Akos Ledeczi, James Davis, Sandeep Neema, and Aditya Agrawal. Modeling methodology for integrated simulation of embedded systems. *ACM Trans. Model. Comput. Simul.*, 13(1):82–103, 2003.

[LMB+01] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason IV, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The generic modeling environment. In *Workshop on Intelligent Signal Processing*, Budapest, Hungary, May 2001.

[Lou01] Daniel Lowry Lough. *A taxonomy of computer attacks with applications to wireless networks*. PhD thesis, 2001. Chairman-Nathaniel J. Davis, IV.

[MES+05] C. Mallanda, S. Else, A. Suri, V. Kunchkarra, S.S. Iyengar, R. Kannan, , and A. Durresi. Simulating wireless sensor networks with omnet++. *IEEE Computer*, 2005.

[ns2] ns2. Available from: `http://www.isi.edu/nsnam/ns/`.

[SEN] SENSIM. Available from: `http://bit.csc.lsu.edu/sensor_web/sensim.html`.