Detection of Wormhole Attack in Wireless Networks

Aninda Sarker Rahul
ID : 1304103

October, 2018

Bachelor of Science in Computer Science and Engineering

Detection of Wormhole Attack in Wireless Networks

Aninda Sarker Rahul
ID : 1304103

October, 2018

Department of Computer Science & Engineering
Chittagong University of Engineering & Technology
Chittagong-4349, Bangladesh.

Detection of Wormhole Attack in Wireless Networks

This thesis is submitted in partial fulfillment of the requirement for the degree of Bachelor of Science in Computer Science & Engineering.

Aninda Sarker Rahul

ID : 1304103

Supervised by
Mir Md. Saki Kowsar
Assistant Professor
Department of Computer Science & Engineering (CSE)
Chittagong University of Engineering & Technology (CUET)

Department of Computer Science & Engineering
Chittagong University of Engineering & Technology
Chittagong-4349, Bangladesh.

The thesis titled **"Detection of Wormhole Attack in Wireless Networks"** submitted by Roll No. 1304103, Session 2016-2017 has been accepted as satisfactory in fulfillment of the requirement for the degree of Bachelor of Science in Computer Science & Engineering (CSE) as B.Sc. Engineering to be awarded by the Chittagong University of Engineering & Technology (CUET).

Board of Examiners

1. ______________________ Chairman
Mir Md. Saki Kowsar
Assistant Professor
Department of Computer Science & Engineering (CSE)
Chittagong University of Engineering & Technology (CUET)

2. ___________________________________ Member
Dr. Mohammad Shamsul Arefin (Ex-officio)
Head
Department of Computer Science & Engineering (CSE)
Chittagong University of Engineering & Technology (CUET)

3. ______________________ Member
Dr. Md. Mokammel Haque (External)
Associate Professor
Department of Computer Science & Engineering (CSE)
Chittagong University of Engineering & Technology (CUET)

Statement of Originality

It is hereby declared that the contents of this thesis is original and any part of it has not been submitted elsewhere for the award of any degree or diploma.

Signature of the Supervisor	**Signature of the Candidate**
Date:	**Date**:

Acknowledgment

First of all thanks to God for his great blessings on me to complete this project successfully. There after I am expressing my gratitude to my honorable project Supervisor Mir Md. Saki Kowsar, Assistant Professor, Department of Computer Science and Engineering, Chittagong University of Engineering and Technology, for his valuable suggestion, constructive advice, encouragement and sincere guidance in my entire project.

It was a real opportunity to work with him. I learned a lot from him. I am grateful that he invested so much of his precious time with us. It was really an amazing experience.

I am also expressing my gratitude to Professor Dr. Mohammed Moshiul Hoque, honorable head of the Department of Computer Science and Engineering for his kind support. I am highly indebted to all of my teachers for their valuable effort in teaching us for last four and a half years.

This thesis would not be possible without the support of our faculty staffs especially Mr. Osman Billah, Mr. Shafikul Islam, Mr. Liton Kar, Mr. Provatosh and others.

I would like to give thanks to my family, friends, seniors, juniors, batchmates for their constant support and motivation. I thank them for giving me confidence and drive for pursuing my degree.

Special thanks to Moinul Islam Bappi,Shourav Sinha Klington and Raihan Roman for working day and night for the completion of the thesis. I am looking forward to work with them in future again.

Abstract

Wireless Sensor Networks (WSNs) provide flexible infrastructures for numerous applications like healthcare,industry automation,surveillance and defence. Wormhole attack is one of the most dangerous attack which can distabilize or disable wireless sensor networks. In order to provide a stable and uninterrupted packet sending experience to a network, Wormhole attack detection is required to maintain the network connection stable. Ideally, Wormhole attack detection should be completely transparent to legitimate users in a network. However the current IEEE 802.11 standards do not detect the Wormhole attack well. In this thesis current network layer attacks scheme is analyzed and an efficient method is proposed. Finally, it is implemented in network simulator 3 and analyzed. The analysis shows that the proposed scheme reduces the packet loss and improves the overall network performance.

Existing solutions on reducing the packet loss in WSNs ignore one important factor for the long handoff delay. Data can be lost during the multihop transmission resulting in increase in packet loss and the data collection becomes incomplete[17]. Studies have revealed that standard hand-off on IEEE 802.11 WLANs increase a latency of the order of hundreds of milliseconds to several seconds. Moreover the discovery step in the handoff process accounts for more than 99% of this latency.

Ad-hoc wireless network signals are not really strong as compared to the wireless connections which uses routers to function properly. On discovery steps, multiple paths between source and destination is discovered using AODV routing protocol. Discovering multipath comes with a number of disadvantages like link failure, congestion error etc. To overcome those disadvantages, the AODV protocol is modified to select the main path for data transmission based on the time of routing establishment.

The feasibility of the proposed scheme to support fast handoff in WSNs has been demonstrated through computer simulations under different network conditions. The results from the simulations show that the latency associated with handoff can be reduced by using this technique. This scheme can improve the overall performance by increasing packet delivery fraction and throughput and reducing ETE delay.

In conclusion, it can be said that the latency in the link layer is reduced by introducing an efficient and powerful technique which also improves the overall performance.

Contents

List of Figures

List of Tables

List of Abbreviations

WSN	Wireless Sensor Network
SN	Sensor Node
MR	Mesh Router
MC	Mesh Client
AP	Access Point
AODV	Ad-hoc On-Demand Distance Vector
NS-3	Network Simulator 3

Chapter 1

Introduction

Wireless sensor networks(WSNs) consist of many interconnected self-controlled device(i.e sensor nodes) that are used in a collective manner to monitor and/or contorl environmental phenomena in local or remote environments[8]. Sensor nodes which are spacially distributed communicate with their peers in order to send aggregated data to the base station efficiently. WSN is a spacial kind of Ad-hoc wireless network that has gained popularity for its versatile application in military and civil domains such as battlefield monitoring, tracking objects, healthcare and home automation. Due to the broadcast nature of the transmission medium and fact that sensor nodes often operate in hostile environments. WSN are vulnerable to variety of security attacks[1]. This chapter contais some introductory information on wormhole attack in wireless sensor network, motivation of our work, challanges of implementing our work and our objectives.

1.1 Wormhole Attack in Wireless Sensor Network

The wormhole attack is recognized as one of the most dangerous security threats for WSNs. This attack has one or more malicious node and a tunnel between them. The attacking nodes capture the packets from one location and transfers them to other distant location node which distributes them locally[1]. The tunnel can be established in many ways e.g in-band and out-of-band channel.Routing mechanisms which rely on the knowledge about distance between nodes can get confuse because because wormhole nodes fake a route that is shorter than the original one within the network[1].

Wireless sensor networks are susceptible to wide range of security attacks due to the multi-hop nature of the transmission medium. Also, wireless sensor networks have an additional vulnerability because nodes are generally deployed in a hostile or unprotected environment. From [2] a summarization of possible attacks in different layers with respect to ISO-OSI model are shown in the Table 1.1

Layer	Attacks
Physical Layer	**Denial of Service, Tampering**
Data Link Layer	**Jamming, Collision, Traffic manipulation**
Routing/Network Layer	**Wormhole, Sinkhole, Flooding**

Table 1.1: Different types of Layering based attacks

For the nature wireless transmission the attacker can create a wormhole even for packets not addressed to itself, since it can overhear them in wireless transmission and tunnel them in wireless transmission and tunnel them to the colluding attacker at the opposite end of wormhole[7].

However, in wireless sensor networks, mobility, limited bandwidth, routing functionalities etc. associated with each node, present many new opportunities for launching a Wormhole attack. Wormhole attack is classified into four models[3] :-

Encapsulation: Here a malicious node at one part of the network overhears the RREQ packet. It is then tunnel through a low latency link with the help of normal node, to the second colluding malicious node at a distance near to the destination node. Once this packet is received by the second malicious code, the legitimate neighbour of the node drops any further legitimate requests from a legitimate neighbour node. This result to the routes between the source and the destination go through the wormhole link, because it has broadcast itself has the fastest route. It prevents legitimate nodes from discovering legitimate paths more than two hops away.

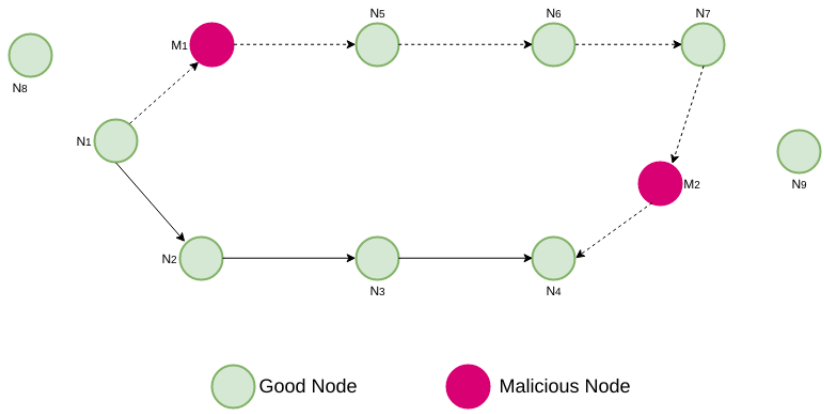

Figure 1.1: Encapsulation Wormhole

Packet Relay: This is another type of wormhole attack where malicious node relays packet between source and destination nodes. Unlike encapsulation, this type of wormhole attack can be launched using only one malicious node.

Out-of-band Channel: As the name suggest is a type of worm-hole attack that uses a long range directional wireless link or a wired link. It is a very difficult attack to launch because its needs a specialized hardware.

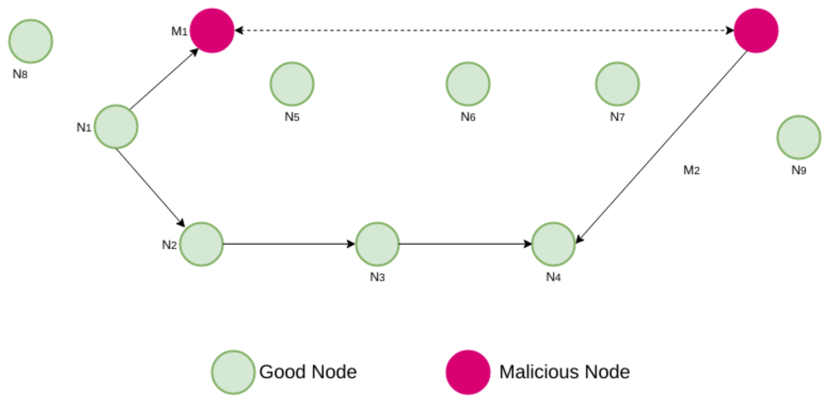

Figure 1.2: Out-of-band Wormhole

High Power Transmission: In this mode of attack, a single ma- licious node can create a wormhole without colluding node. when this single malicious node received a RREQ, it rebroadcasts the request at a very high power level capability compared to normal node, thereby attracting normal nodes to overhear this RREQ and further on broadcast the packet towards destination.

1.2 Background or Previous works

Security in WSNs is still in its infancy as very little attention has been devoted thus so far to this topic by the research community and so it has become more vulnerable to various types of attacks.

In [11] an efficient method for detecting wormhole attacks against the routing functionality of network is propounded. The author proposed an algorithm which is meant to secure each link. In this algorithm, each node considers the distance. The Distance separates it from its direct neighbors. Using a radio interferometry, this estimate is performed by using an exchange of message simultaneously. Then each node exchanges the information of calculated distances. If these data are exchanged, Then each node runs a set of geometric tests on the local data to detect false links present due to the Wormhole attack. The disadvantage of this approach is that each node is needed to be equipped with a second ultrasound radio in order to allow the estimation of distances between neighboring nodes.

In [12], a statistical approach is proposed, known as SWAN, in which each sensor collects a recent number of neighbors. A wormhole attack is detected if the current number of neighbors exhibits an unusual increase, compared to the previous neighborhood counts which are taken outside of the wormhole zones. This is a distributed approach. Unlike a centralized approach, it doesn't cause any overhead. However, this schemes has been designed for and perform better in a uniformly distributed network, but its performance is in question for non-uniformly distributed sensor networks.

In [6], Hu and Evans propose a solution to wormhole attacks for ad hoc networks in which all nodes are equipped with directional antennas. Nodes use specific 'sectors' of their antennas in order to communicate with each other using bidirectional antenna. Therefore, a node receiving a message from its neighbor has some information about the location of that neighbor, for which it knows the relative orientation of the neighbor. This extra bit information makes wormhole discovery much easier than in networks. This approach does not require either location information or clock synchronization. This approach is more efficient with energy. They consider the packet arrival direction to defend the attacks by using directional antenna. They use the neighbor verification methods and verified neighbors are really neighbors and only accept messages from verified neighbors. But it has the drawback that the the directional antenna is not possible for sensor networks.

HU et al [4] describe a defense based on the leashes of the packet. In this approach, each message keeps a timestamp and a location of its transmitter where the distance of a message route is limited. The receiver compares this information with its own location and timestamp in order to check if the intervals of transmissions are exceeded or not. However, this proposal presents two disadvantages: It requires a coordinate system such as the GPS in order to obtain the geographic information about each node and It requires a precise synchronization of clocks between different nodes in order to use timely data.

In WSNs are vulnerable to several kinds of attacks because of their inherent attributes such as the open communication medium. Malicious sensor devices can launch attacks to disrupt the network routing operations, then putting the entire sensor network at risk. Many techniques have been suggested for Wormhole attack detection and characterization. Most of these techniques have one or more limitations. Network visualization method can be effective against Wormhole attack but it requires central coordination and the mobility is not studied for this method.

1.3 Present state and Contribution

The objective of this thesis is to develop a Procedure to detect Wormhole attack in WSNs. After a study and analysis of the wireless sensor network Wormhole attack detection procedure, the detection process was divided into two phases: neighborhood sensing and malicious node detection. A fast Wormhole attack detection scheme have been developed to provide a novel use of the channel in wireless sensor network. This detection scheme may be implemented by upgrading the protocol of wireless sensor network, no hardware upgrade is required. NS-3 simulations were used in order to verify the feasibility of the proposed scheme. The results presented in chapter 5 indicate that the latency associated with Wormhole attack can be efficiently detect by using the proposed technique. The performance of the proposed scheme was also analyzed. The results show that the scheme continued to successfully operate under different network conditions.

1.4 Motivation

Security in wireless sensor network system is one of the main concerns to provide protected communication between mobile nodes in strange environment. Unlike the wired line networks, the unique characteristics of WSN create a number of nontrivial challenges to security design like open peer-to-peer network architecture, shared wireless medium, inflexible resources constraints and highly dynamic network topology.

Guarding against Wormhole attack is a critical component of any WSN security system. Security services in WSNs are needed to protect from attacks and to ensure the security of the information. The wireless channel is accessible to both intended and unintended users. There is no well-defined place where traffic monitoring or access control mechanisms can be brought into life. As a result, there is no clear boundary that separates the inside network from the outside world.

1.5 Prospects of the problem

Our main prospects of this project is to detect the Wormhole attack in wireless sensor network and improve the overall performance.

1.6 Organization of the Project

The remainder of the report is organized as follows. In the next chapter, an overview of our project related terminologies are given and contains brief discussion on previous works that is already implemented with their limitations. Chapter three describes the working procedure of our proposed system. In Chapter 4, we have illustrated our implementation of the project in details. Chapter 5 focuses on the experimental result of the proposed system. The thesis concludes with a summary of research contributions and future plan of our work in chapter 6. This thesis contains an appendix intended for persons who wish to explore the source code.

Chapter 2

Literature Review

Wormhole attack detection is an essential issue to ensure continuous communications in wireless sensor networks (WSNs). The Wormhole attack performance in WSNs can be largely degraded by the packet loss which dropped the valuable information by neighborhood sensing at each sensor node, especially when the backbone traffic volume is high. In this chapter, we present studies on the terminologies related to the project which are important to understand. This chapter also contains brief discussion on previous works that is already implemented along with their limitations.

2.1 Wireless Sensor Network

A Wireless Sensor Network is one kind of wireless network includes a large number of circulating, self-directed, low powered devices named sensor nodes. These networks certainly cover a huge number of spatially distributed, little, battery-operated, embedded devices that are networked to caringly collect, process, and transfer data to the operators, and it has controlled the capabilities of computing & processing. Nodes are the tiny computers, which work jointly to form the networks.

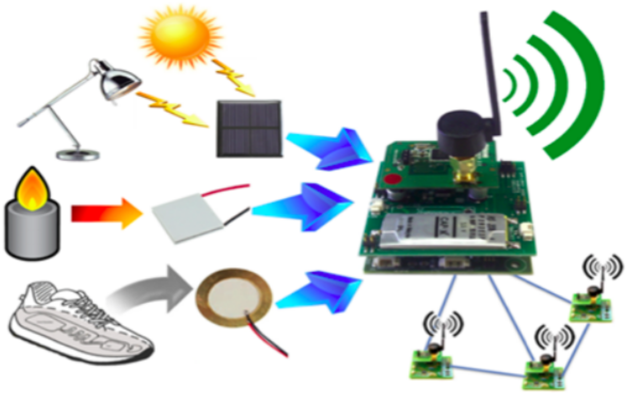

Figure 2.1: Wireless Sensor Network

The sensor node is a multi-functional, energy efficient wireless device. The applications in industrial sectors are widespread. A collection of sensor nodes collects the data from the surroundings to achieve specific application objectives. The communication between nodes can be done with each other using transceivers. In a wireless sensor network, the number of nodes can be in the order of hundreds/ even thousands. In contrast with sensor nodes, Ad Hoc networks will have fewer nodes without any structure.

2.1.1 Network Architecture

The most common WSN architecture follows the OSI architecture Model. The architecture of the WSN includes five layers and three cross layers. Mostly in sensor n/w we require five layers, namely application, transport, n/w, data link & physical layer. The three cross planes are namely power management, mobility management, and task management. These layers of the WSN are used to accomplish the n/w and make the sensors work together in order to raise the complete efficiency of the network.

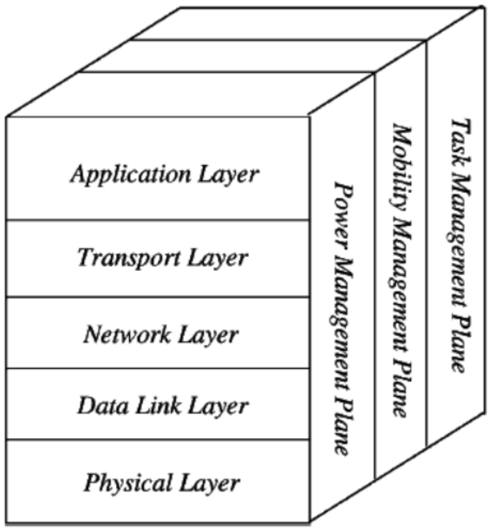

Figure 2.2: Wireless Sensor Network Architecture

Application Layer: The application layer is liable for traffic management and offers software for numerous applications that convert the data in a clear form to find positive information. Sensor networks arranged in numerous applications in different fields such as agricultural, military, environment, medical, etc.

Transport Layer: The function of the transport layer is to deliver congestion avoidance and reliability where a lot of protocols intended to offer this function are either practical on the upstream. These protocols use dissimilar mechanisms for loss recognition and loss recovery. The transport layer is exactly needed when a system is planned to contact other networks.

Network Layer: The main function of the network layer is routing, it has a lot of tasks based on the application, but actually, the main tasks are in the power conserving, autentication, authorization and identity certification.

Data Link Layer: The data link layer is liable for multiplexing data frame detection, data streams, MAC, & error control, confirm the reliability of point–point (or) point– multipoint.

Physical Layer: The physical layer provides an edge for transferring a stream of bits above physical medium. This layer is responsible for the selection of frequency, generation of a carrier frequency, signal detection, modulation & data encryption.

WSN Network Topologies

For radio communication networks, the structure of a WSN includes various topologies like the ones given below.

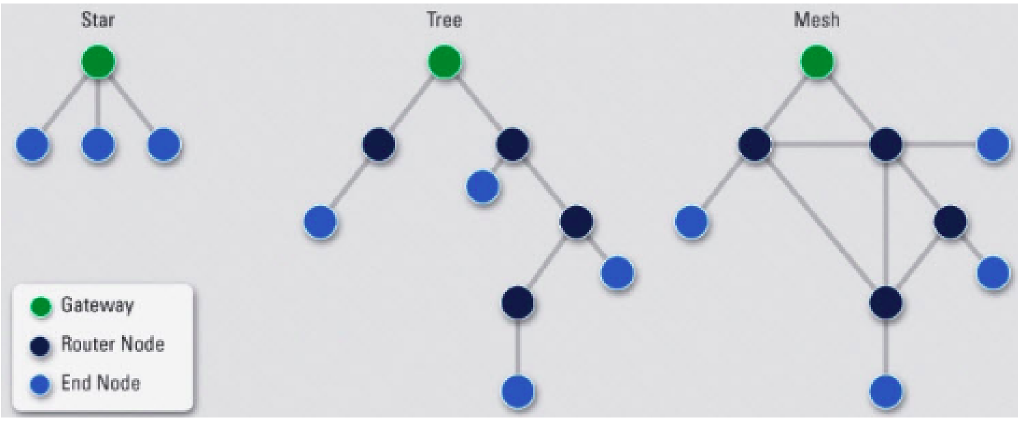

Figure 2.3: Wireless Sensor Network Topologies

Star Topologies: Star topology is a communication topology, where each node connects directly to a gateway. A single gateway sends or receives a message to a number of remote nodes. The nodes are not permitted to send messages to each other in star topologies. This allows low-latency communications between the remote node and the gateway (base station).

Tree Topologies: In tree topology, which is also called as cascaded star topology, each node connects to a node that is placed higher in the tree, and then to the gateway. The advantage of the tree topology is that the expansion of a network can be easily possible, and also error detection becomes easy. The disadvantage with this network is that depends heavily on the bus cable. If the bus cable breaks, all the network will collapse.

Mesh Topologies: The Mesh topologies allow transmission of data from one node to another, which is within its radio transmission range. If a node, which is out of radio communication range wants to send a message to another node, it needs an intermediate node to forward the message to the desired node. The advantage with this mesh topology is that it includes easy isolation and detection of faults in the network. The disadvantage is that the network is large and requires huge investment.

2.1.2 Characteristics

There have been several characteristics of WSNs. Some of characteristics are explained as follows:

- **Low cost:** In WSN, if we want to measure any physical environment, hundreds or thousands of sensor nodes are deployed normally. In order to reduce the overall cost of the whole network the cost of the sensor node must be kept as low as possible.

- **Energy efficient:** Energy in WSN is used for different purpose such as computation, communication and storage. Sensor node consumes more energy compare to any other for communication. If they run out of the power they often become invalid as we do not have any option to recharge. So, the protocols and algorithm development should consider the power consumption in the design phase.

- **Computational power:** Normally the node has limited computational capabilities as the cost and energy need to be considered.

- **Communication Capabilities:** WSN typically communicate using radio waves over a wireless channel. It has the property of communicating in short range, with narrow and dynamic bandwidth. The communication channel can be either bidirectional or unidirectional. It is difficult to run WSN smoothly with the unattended and hostile operational environment . So, the hardware and software for communication must have to consider the robustness, security and resiliency.

- **Distributed sensing and processing:** the large number of sensor node is distributed uniformly or randomly. WSNs each node is capable of collecting, sorting, processing, aggregating and sending the data to the sink. Therefore the distributed sensing provides the robustness of the system.

- **Dynamic network topology:** In general WSN are dynamic network. The sensor node can fail for battery exhaustion or other circumstances, communication channel can be disrupted as well as the additional sensor node may be added to the network that result the frequent changes in the network topology. Thus, the WSN nodes have to be embedded with the function of reconfiguration, self adjustment.

- **Multi-hop communication:** A large number of sensor nodes are deployed in WSN. So, the feasible way to communicate with the sinker or base station is to take the help of a intermediate node through routing path. If one need to communicate with the other node or base station which is beyond its radio frequency it must me through the multi-hop route by intermediate node.
- **Application oriented:** WSN is different from the conventional network due to its nature. It is highly dependent on the application ranges from military, environmental as well as health sector. The nodes are deployed randomly and spanned depending on the type of use.
- **Robust Operations:** Since the sensors are going to be deployed over a large and sometimes hostile environment. So, the sensor nodes have to be fault and error tolerant. Therefore, sensor nodes need the ability to self-test, self-calibrate, and self repair.
- **Security and Privacy:** Each sensor node should have sufficient security mechanisms in order to prevent unauthorized access, attacks, and unintentional damage of the information inside of the sensor node. Furthermore, additional privacy mechanisms must also be included.
- **Small physical size:** sensor nodes are generally small in size with the restricted range. Due to its size its energy is limited which makes the communication capability low.

2.1.3 Advantages of Wireless Mesh Network

The advantages of WSN over other networks are very significant and have great deal of importance. There are some unique features compared to other network in WSN. These features are explained below:

- Network arrangements can be carried out without immovable infrastructure.
- Apt for the non-reachable places like mountains, over the sea, rural areas and deep forests.
- Flexible if there is a casual situation when an additional workstation is required.
- Execution pricing is inexpensive.
- It avoids plenty of wiring.
- The data processing is pretty fast.
- Easy installation and uninstall.
- It might provide accommodations for the new devices at any time.
- It can be opened by using a centralized monitoring.

Building a network without any wire brings a great deal of advantage. In most the case bigger network don't really use any wire. The internet we use in our daily life is a realistic example for this. It is seen that most of the network are inter connected with each other wirelessly creating a mesh topology which is also called seamlessly. It is cheap as it don't use any wire. In WSN the nodes automatically adjust themselves according to the situation, so there is no need for network administrator if there is a problem regarding nodes or the network. WSN nodes can communicate with their neighboring nodes as well without going back to the central device, which increases its data processing speed. According to the requirement WSN nodes can be installed or uninstalled. Like all other wireless networks standards, WSN also uses one of those standards. Being a new technology it does not require new Wi-Fi standard. WSNs are very much tolerant to faults, if couple of nodes in a network fails, the communication will always keep on going.

2.1.4 Application

Wireless sensor networks may comprise of numerous different types of sensors like low sampling rate, seismic, magnetic, thermal, visual, infrared, radar, and acoustic, which are clever to monitor a wide range of ambient situations. Sensor nodes are used for constant sensing, event ID, event detection & local control of actuators. The applications of wireless sensor network mainly include health, military, environmental, home, & other commercial areas.

- Military Applications
- Health Applications
- Environmental Applications
- Home Applications
- Commercial Applications
- Area monitoring
- Health care monitoring
- Environmental/Earth sensings
- Air pollution monitoring
- Forest fire detection
- Landslide detection
- Water quality monitoring
- Industrial monitoring

In addition to the above applications, these systems has low power consumption, low cost and is a convenient way to control real-time monitoring for unprotected agriculture and habitat. Moreover, it can also be applied to indoor living monitoring, greenhouse monitoring, climate monitoring and forest monitoring. These approaches have been proved to be an alternative way to replace the conventional method that use men force to monitor the environment and improves the performance, robustness, and provides efficiency in the monitoring system.

2.2 IEEE 802.11s

In this section a detailed explanation of how the IEEE 802.11s works is presented.

2.2.1 Network design

In 802.11, an extended service set (ESS) consists of multiple basic service sets (BSSs) connected through a distributed system (DS) and integrated with wired LANs. The DS service (DSS) is provided by the DS for transporting MAC service data units (MSDUs) between APs, between APs and portals, and between stations within the same BSS 5 that choose to involve DSS. The portal is a logical point for letting MSDUs from a non-802.11 LAN to enter the DS. The ESS appears as single BSS to the logical link control layer at any station associated with one of the BSSs. As is explained in, the 802.11 standard has pointed out the difference between independent basic service set (IBSS) and ESS. IBSS actually has one BSS and does not contain a portal or an integrated wired LANs since no physical DS is available. Thus, an IBSS cannot meet the needs of client support or Internet access, while the ESS architecture can. However, IBSS has its advantage of self-configuration and ad-hoc networking. Thus, it is a good strategy to develop schemes to combine the advantages of ESS and IBSS. The solution being specified by IEEE 802.11s is one of such schemes. In 802.11s, a meshed wireless LAN is formed via ESS mesh networking. In other words, BSSs in the DS do not need to be connected by wired LANs. Instead, they are connected via wireless mesh networking possibly with multiple hops in between. Portals are still needed to interconnect 802.11 wireless LANs and wired LANs.

2.2.2 WSN formation and management

There are four elements that characterize a wireless sensor network:

- Gateway
- Relay Node
- Sink Node
- Sensor

Together these four elements define a profile. If we compare the structure of Sensor Network on IEEE 802.11s with heterogeneous hierarchical wireless sensor net-work, we can find that they are very similar. Sensor Portal can work as a gateway and provide access to other networks, Sensor Portal is similar to the sink node of wireless sensor network and mobile client terminals are similar to the common nodes in wireless sensor network[17]. Most of the nodes of wireless sensor network are powered by batteries, so their node energy is restricted and the calculating capability and storage capacity are limited. So the network protocols of Wireless Sensor Network can only be used in the new generation of remote AMR network after being optimized. Since the default routing protocol of Wireless sensor is HWMP, we should reduce the energy consumption of nodes running with HWMP.

2.3 IEEE 802.11s model in NS-3

This section provides an explanation of how is implemented the 802.11s wireless sensor networking model in the Network Simulator 3 (NS-3) and which features or characteristics are supported and which not. First, NS3 is briefly explained to see why this simulator has been chosen. The model used has been the one developed by the Wireless Software R&D Group of IITP RAS and included in NS-3 from the release 3.6. Although it is based in the IEEE P802.11s/D3.0, for the aim of this research, the characteristics used and analyzed are not different from the ones present in the last draft of 802.11s.

2.3.1 Network Simulator 3

NS-3 is a discrete-event network simulator for Internet systems, targeted primarily for research and educational use. NS-3 is free software, licensed under the GNU GPLv2 license, and is publicly available for research, development, and use. It is a tool aligned with the simulation needs of modern networking research allowing researchers to study Internet protocols and large-scale systems in a controlled environment. The following trends is how Internet research is being conducted are responded by NS-3:

- **Extensible software core:** written in C++ with optional Python interface and an extensively documented API (doxygen).

- **Attention to realism:** model nodes more like a real computer and support key interfaces such as sockets API and IP/device driver interface (in Linux).

- **Software integration:** conforms to standard input/output formats (pcap trace output, NS-2 mobility scripts, etc.) and adds support for running implementation code.

- **Support for virtualization and testbeds:** Develops two modes of integration with real systems:
  -Virtual machines run on top of ns-3 devices and channels
  -NS-3 stacks run in emulation mode and emit/consume packets over real devices.

- **Flexible tracing and statistics:** decouples trace sources from trace sinks so we have customizable trace sinks.
- **Attribute system:** controls all simulation parameters for static objects, so you can dump and read them all in configuration files.
- **New models:** includes a mix of new and ported models.

To sum up, NS-3 tries to avoid some problems of its predecessor, NS-2, which is still being used by many researchers, but it has some important lacks such as: interoperability and coupling between models, lack of memory management, debugging of split language objects or lack of realism (in the creation of packets for example). Mainly, the new available high fidelity IEEE 802.11 MAC and PHY models together with real world design philosophy and concepts made NS-3 the choice for developing this 802.11s model as well as for carrying out this research.

2.3.2 Model design

To meet these requirements imposed by 802.11s of supporting multiple interfaces (wireless devices) and also different sensor networking protocol stacks, WS RD Group designed and implemented a runtime configurable multi-interface and multi-protocol mesh STA architecture.

Supported features

The most important features supported are the implementation of the Peering Management Protocol, the HWMP and the ALM. A part from the functionality described in section2.3, the PMP includes link close heuristics and beacon collision avoidance. HWMP includes proactive and on-demand modes, unicast/broadcast propagation of management traffic and, as an extra functionality not specified yet in the draft, multi-radio extensions. However, for the moment RANN mechanism is implemented but there is no support, so only the PREQ can be used.

Unsupported features

The most important feature not implemented is Mesh Coordinated Channel Access(MCCA). Internetworking using a Mesh Access Point or a Portal is not implemented neither, but this functionality is not needed to evaluate the performance in the creation of mesh networks. As other less relevant features not implemented we can point out the security, power safe mode and although multi-radio operation is supported, no channel assignment protocol is proposed.

2.3.3 Model implementation

The description of which modules are implemented in C++ and how they interconnect with each other is presented in appendix B. The explanation is in a high-level in order to see which modules and classes need to be accessed or created when designing a sensor network, but the low-level code structure is not described. For more information on each module and a more detailed low level explanation, please check NS-3 documentation under Doxygen [19]. First is explained the way the MAC-layer routing is implemented presenting the most important classes. Then are analyzed the class SensorHelper (used to create a 802.11s network easier). They provide some functions to configure the different parameters of the network and its devices, so the main parameters and the way to configure them is studied.

2.4 Wormhole attack in 802.11b WSN

Due to multihop routing in wireless sensor network, it is prone to various types of attacks. Wormhole attack in an IEEE 802.11b WSN occurs when a mobile STA sends data to the destination beyond the radio range and receives data by another mobile STA. During the data sending and receiving process, management of frames are exchanged between the STA and the SN. Consequently, there is a latency involved in the Wormhole attack process during which the STA is unable to send or receive traffic because of malicious attribute of a node in WSN which prevent service to the legitimate users and drop packets in Wormhole attack.

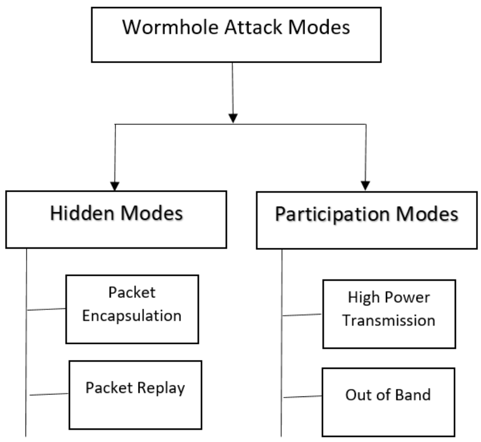

Figure 2.4: Taxonomy of Wormhole attack

2.4.1 Wormhole using Packet Encapsulation

Here several nodes exist between two malicious nodes and data packets are encapsulated between the malicious nodes. Hence it prevents nodes on way from incrementing hop counts. The packet is converted into original form by the second end point. This mode of wormhole attack is not difficult to launch since the two ends of wormhole do not need to have any cryptographic information, or special requirement such as high-power source or high bandwidth channel.

2.4.2 Wormhole using Packet Relay

One or more malicious nodes can launch packet-relay-based wormhole attacks. In this type of attack malicious node replays data packets between two far nodes and this way fake neighbours are created.

2.4.3 Wormhole using Out-of-Band Channel

This kind of wormhole approach has only one malicious node with much high transmission capability in the network that attracts the packets to follow path passing from it. The chances of malicious nodes present in the routes established between sender and receiver increases in this case.

2.4.4 Wormhole using High Power Transmission

In this mode of attack, a single ma- licious node can create a wormhole without colluding node. when this single malicious node received a RREQ, it rebroadcasts the request at a very high power level capability compared to normal node, thereby attracting normal nodes to overhear this RREQ and further on broadcast the packet towards destination.

2.5 Related Works

Many Wormhole attack detection schemes have been proposed in the researched literature for WSN networks. An efficient method for detecting wormhole attacks against the routing functionality of network is proposed in [11] in which the author propose an algorithm which is meant to secure each link.The disadvantage of this approach is that each node must be equipped with a second ultrasound radio, allowing the estimation of distances between neighboring nodes. In paper[12], a statistical approach is proposed, known as SWAN, in which each sensor collects a recent number of neighbors. In [6], Hu and Evans propose a solution to wormhole attacks for ad hoc networks in which all nodes are equipped with directional antennas. HU et al [4] describe a defense based on the leashes of the packet, where the distance of a message route is limited, each message having a timestamp and a location of its transmitter. But this proposal presents two disadvantages: It requires a coordinate system such as the GPS in order to obtain the geographic information about each node; It requires a precise synchronization of clocks between different nodes in order to use timely data.

2.6 Chapter Summary

This chapter has discussed what Wormhole attack is and why detection is important to WSNs and also outlines some fundamental aspects of the operation of IEEE

802.11 WLANs and WSNs. The chapter ended with discussion of related works. The following chapters will further outline the technical details of the proposed scheme and its implementation, as well as an analysis of its performance along with the comparing with the existing system.

Chapter 3

Methodology

In this chapter, a detailed description of the proposed Wormhole attack detection methodology is given. Besides an analysis of the existing wormhole attack detection procedure is provided.

3.1 Solution utilizing network redundancy

This is an improvement of the normal AODV. The solution proposes that the source node does not immediately start sending data packets after receiving a route reply. It waits to receive other route replies from nearby nodes to confirm that they contain the same next hop information. This solution uses an assumption that there are redundant routes that can be used to reach the destination node. When a RREP packet arrives at the source node, the full path is extracted and the source node waits for another RREP packet. The routes from other RREP packets are compared with the route extracted from the first RREP, and they must have shared hops. If there are no shared hops in the routes, the source node takes the routes to be untrustworthy and waits for more RREP packets until there are shared hops or until the expiration of the routing timer. Even though this solution assures a safe route, it increases the time delay and messages will never be forwarded to the destination node if there are no shared hops in the paths.

3.1.1 Malicious Node Detection

The malicious node detection technique is responsible for detecting the Wormhole nodes in the network. Initially, the Wormhole detector initializes the malicious node detection process. First, it broadcasts the spoofed RREQ packets. As discussed above, the spoofed RREQ packet contains the non existence source id and the TTL value set to 1. Then this spoofed RREQ packet is broadcast to all the other nodes in the network. The broadcasted Honeypot spoofed RREQ packet waits for the reply from the neighbor nodes. If any neighbor replies to this packet, those nodes are marked as Wormhole nodes in the routing table. The reason is, since the normal nodes which are not malicious will not reply to this spoofed RREQ packet. So the routing table updates this Wormhole node information by marking it as malicious.

3.1.2 Route Lookup in Network Layer

In order to resolve the route, the AODV calls the modified Route Lookup function. This algorithm is very important, because it detects the Wormhole attacks by checking the node id. If the malicious node replies that, it has the route towards the non existence node, then that vulnerable (Wormhole) node is marked as malicious. In order to find a Wormhole node, a detection flag is set on the routing table. If the detection flag is true then, it is observed that the malicious node id is marked. Thus, routing via the malicious node is avoided. The Route Lookup algorithm for the network layer is responsible for updating the reply from the neighbor nodes. The node which replies to the spoofed RREQ packet is identified as the Wormhole node. Then, the node is marked as malicious in RTF and this information is updated in the routing table. Hence the above route lookup algorithm is responsible which marks the malicious node ids in the routing table.

3.1.3 Isolation in Network Layer

The isolation technique is responsible for isolating the malicious node from the network. This technique is important, because it prevents broadcasting routes via the malicious node. A flag is set as malicious, and the nodes which reply to the non-existence node id are marked as malicious.

3.2 Existing Wormhole Attack Detection Procedures in WSNs

Several Detection procedure is used to detect wormhole attack. Most of them detect attacks in wsn modifying the AODV routing protocol. Those proposed procedures are descibed in the following subsections.

3.2.1 Multipath AODV (AOMDV)

In[1], they proposed AOMDV. In this scheme, AOMDV protocol is used which is an extention to AODV in order to discover multiple paths between the source and the destination in every route discovery. In AOMDV routing protocol the sender node checks in the route table whether a route is present or not for communication of any two nodes, if present it gives the routing information else it broadcasts the packet, if the route is not present then it broadcasts the RREQ packet to its neighbours which in turn checks whether a route is present to the required destination or not.

In the usual operation of AODV, the value of round trip time is checked in the routing table by the node that receives a RREP packet. By dividing round trip time with hop count we denote the value for that route. Averaging all the processed value of round trip time for all path, we get the threshold value. If the value of processed round trip time with hop count of that is smaller than the threhold value, then it is assumed that there is wormhole link on that route.

After wormhole link spotted in that route, sender detects first neighbour node as wormhole node and sends dummy RREQ packet through that route and corresponding neighbour. At the destination end receiver receives dummy RREQ packet from its neighbour and detects neighbour as wormhole node. Routing entries for those nodes are removed from the source node and broadcast to other nodes. Thus wormhole affected link is jammed and is no more used. So, that from the next time onwards whenever a source node needs a route to that destination, first it checks in the routing table in the route established phase for a route and it will come to know that, the route is having wormhole link and it will not take that route instead it will take another route from the routing list of the source node which is free from wormhole link if available.

3.2.2 Intrusion Detection System AODV (IDSAODV)

In[27], they proposed IDSAODV in order to make the wormhole attack detection process extended. This is achieved by altering the way normal AODV updates the routing process. The routing update process is modified by adding a procedure to disregard the route that is established first.

The tactic applied in this method is that the network that is attacked has many RREP packets from various paths, so is assumed that the first RREP packet is generated by a malicious node. The assumption is based on the fact that a wormhole node does not look up into its routing table before sending a RREP packet. Therefore, to avoid updating routing table with wrong route entry, the first RREP is ignored.

This method improves packet delivery but it has limitations that; the first RREP can be received from an intermediate node that has an updated route to the destination node, or if RREP message from a malicious node can arrive second at the source node, the method is not able to detect the attack.

3.2.3 Secure AODV (SAODV)

In[28], they proposed a secure routing protocol SAODV that addresses wormhole attack in AODV. The difference between AODV and SAODV is that in SAODV, there are random frequenceis that are used to verify the destination node. An extra verification packet is introduced in the route discovery process. After receiving a RREP packet, the source node stores it in the routing table and immediately sends a verification packet using reverse route of received RREP. The verification packet contains a random number generated by source node.

When two or more verification packets from the source node are received at the destination node, coming from different routes, the destination node stores them in its routing table and checks whether the contents contain the same random numbers. If the verification packets contain same random numbers along different paths, the destination node sends verification confirm packet to the source node which contains random number generated by destination node.

If verification confirm packet contains different random numbers, the source node will wait until at least two or more verification confirm packets contain same random numbers. When the source node receives two or more verification confirm packet with same random numbers, it will use the shortest route to send data to the destination node. The security mechanism in this protocol is that malicious node cannot pretend to be destination node and send correct verification confirm packet to the source node.

3.3 Proposed Methodology

The proposed methodology is based on calculation on the hop by hop to find out Wormhole attacks.

In order to detect Wormhole attack, a malicious node has been created in wireless sensor network which attributes is drop packets for Wormhole attack. The specific value of those thresholds can be calculated based probability of packet dropping for those attacks.

The Wormhole attack detection algorithm is divided into the several steps as shown in Figure 4:

Header addition: Add a 16-bit header in each packet for source id and destination id. First 8-bit is used for source id and last 8-bit is used for destination id.

Collection: Find the route for sending packet from source to destination in wireless sensor network by OSLR algorithm. Also collect the forwarding and received packet for all node in wireless sensor network.

Calculation: For forwarding node calculate the forwarding packet FP, for receiving node calculate the receiving packet RP and for calculating round trip time for all routes Ts.

Determination: Determine forwarded and received packet difference. If the difference is zero, then there is no types of attack. So, send the next packet.

Packet analysis: Analyse the packet.If the forwarded and received packet difference is less than threshold value which is for attacker, then send next packet.If the forwarded and received packet difference is greater than threshold value which is for channel and other losses, then save the node id, from routing table. Then,calculate the packet loss and also calculate the probability of attack. If the probability of attack is not greater than the probability of wormhole attack then check if the round trip time is smaller than the threshold value for round trip time. If the probability of attack is greater than the probability of Wormhole attack and the threshold of round trip time is greater than the present value, then notify that there is Wormhole attack and send message to all sensor nodes and update the attack table.

The specific value of those thresholds can be calculated based on probability of packet dropping for those attacks. At time threshold 1, the wormhole attack

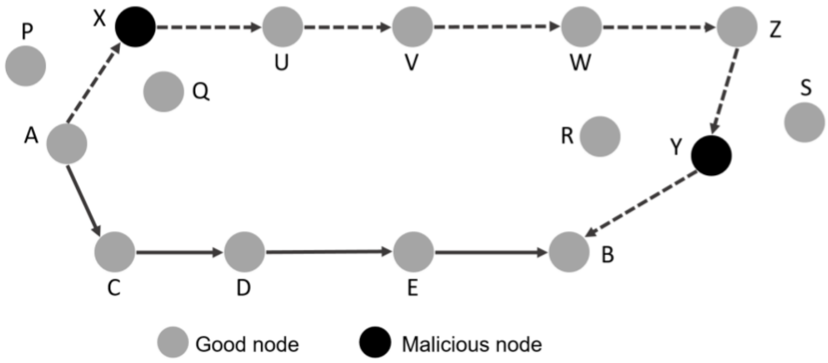

Figure 3.1: Wormhole attack in wireless sensor network

detection process will be triggered in advance in order to detect the wormhole attack.

In figure 3.1, For wormhole attack, source and destination is selected. Then add 16-bit header in each packet which routes from source to destination.

Then select the best route from source to destination by modified aodv routing algorithm. Calculate the round trip time for each routes. Also calculate the the all forwarding and received packets for all routes in wireless sensor network.

In figure 3.1, we need to send data from node A to Node B. We calculate the round trip time and find out the threshold value of average round trip time with respect to hop count. Then we calculate FP and RP of A to B. Also we calculate the threshold of packet loss based on channel loss and other issue.

If the packet loss exceeds the threshold value of packet loss then we check if the probability of attack is greater than the probability of wormhole attack. If yes, then we detect that data is passing to destination through wormhole nodes and there is a wormhole link on the network. Thus there is wormhole attack is detected.

Methodology :

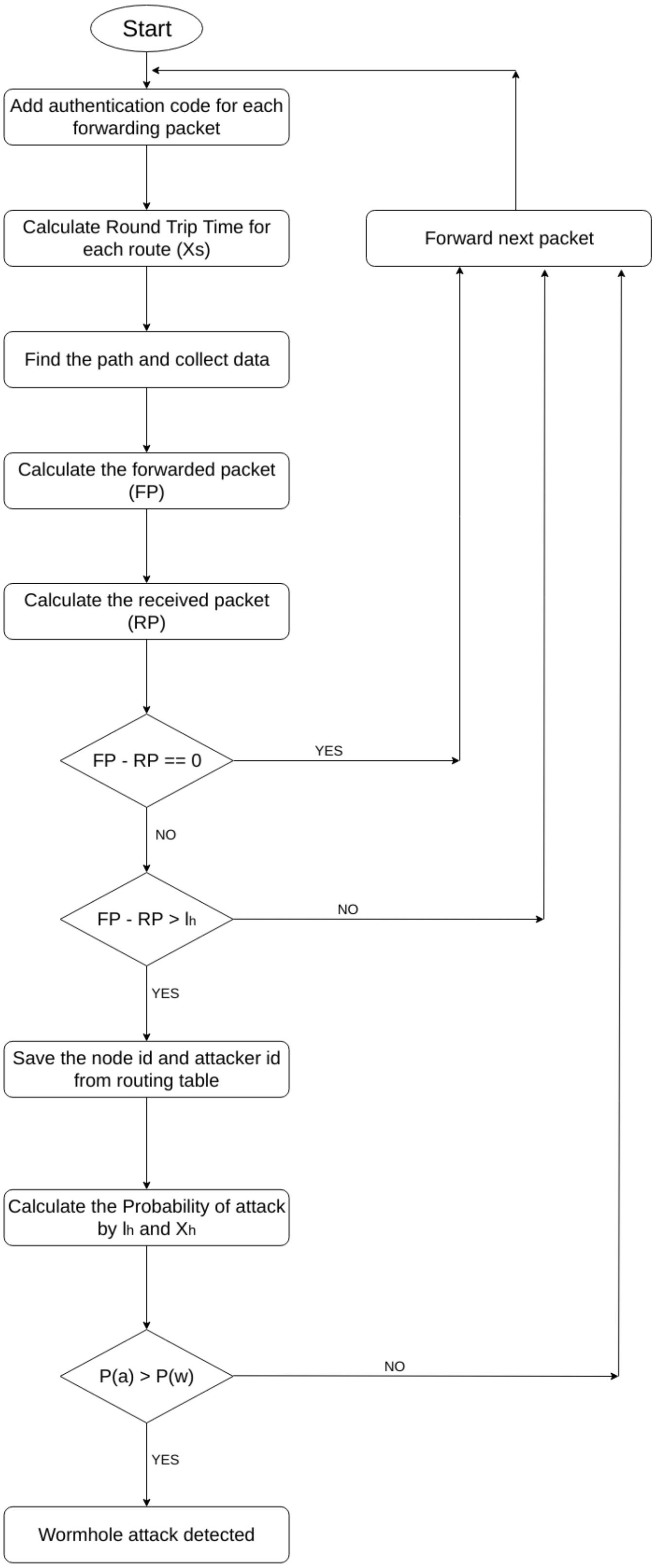

Figure 3.2: Flow chart for Wormhole attack detection mechanism.

3.4 Chapter Summary

This chapter has outlined the different phases of this study including Wormhole attack detection analysis and the operation of the proposed scheme. A detailed description of the scheme was given also. The next chapter will outline the implement of the scheme in NS-3

Chapter 4

Implementation

This study evaluated the Wormhole attack impact on performance of WSN using AODV protocols. It further compared the performance of AODV under black hole attack. The solutions that have been previously proposed to combat effects of Wormhole attack, which were tested using the base protocol AODV, were studied and the study tries to determine the solution that performs better than others. This is achieved by using network simulator version 3 (NS-3) to simulate Wireless sensor network scenarios that include Wormhole node. It can be very expensive to carry out a networking research by setting up an actual network with several computers and routers. Network simulators save a lot of money and time in accomplishing network research goals that is why a simulator-based approach has been chosen for this study. The disadvantage of simulation is that some factors have to be estimated because it is not possible to accurately duplicate the whole world inside a computer model. Thus simulation over simplifies real network scenarios. There exists a variety of network simulation tools that are used in research, but NS-3 has been selected for this study because the protocols under study (AODV) have not been implemented in NS-3. Also NS-3 is distributed freely and is an open source environment which allows the creation of new protocols, and modification of existing ones, so it is possible to introduce a black hole attack in NS-3 by modifying its source code. Moreover, NS-3 is well documented and user online support is provided. This chapter explains the implementation of the research study on NS-3 simulation tool stipulating in detail the parameters used in the simulation and outlining the changes made to the NS-3 source code to introduce Wormhole attack.

4.1 Implementation Tools

The necessary tools to implement this system can be divided in to two categories. Hardware & Software as described below:

- Hardware Requirements
  - Personal Computer with basic configuration
-

Software Tools

- Operating System:Ubuntu 16.04 LTS
- Network Simulator 3 version 3.24
- NetAnim
- PyViz
- Wireshark
- Flow Monitor

4.2 Implementation Details

As discussed in the previous chapter, a Wormhole attack detection scheme have been developed for reducing packet loss in wireless sensor network. It is implemented in NS-3 in order to analyze its performance through experiments. The basis of the scheme is to decrease the total packet loss by detecting the Wormhole attack.

4.3 Simulation Parameters

- Number of Nodes: 14
- Simulation Time: 100s
- Mobility Model: Constant Position Mobility Model, Random Walk Mobility Model
- Routing Protocol: AODV Routing Protocol
- Size of packets in UDP ping: 1024 bytes
- Packet interval: 0.1 sec
- Data Rate of Wireless links: 250Kbps
- Data Rate of Wireless sensor links: 250Kbps
- Data Rate of CSMA connection: 100Mbps
- Node distance: 50 meters

4.4 Simulation of wormhole attack

A malicious node is introduced to both AODV to implement a wormhole by modifying NS-3 C++ source code as shown below. A malicious node attracts the packets and discards them.

4.5 AODV Modifications

i. Declared malicious node variable in aodv.h file.
bool malicious;

ii.Initialized the variable to false in aodv.cc constructor function to show that initially all nodes are not malicious.
malicious= false;

iii.In AODV.cc route handling function, the following code was added to maliciously drop packets.

if(IsMalicious)//When malicious node receives packet it drops the packet.
std :: cout ¡¡"Wormhole attack detected !!
return false;

4.6 Simulation Visualization

The network model used in our simulation is shown in Figure 4.1. The sensor backbone size varies when sensors are added in the network. The link between Sensors and sink node are wireless.

The figure 4.1 shows the normal behavior of in the wireless sensor network and the figure 4.2 shows the malicious attribute in the wireless sensor network.

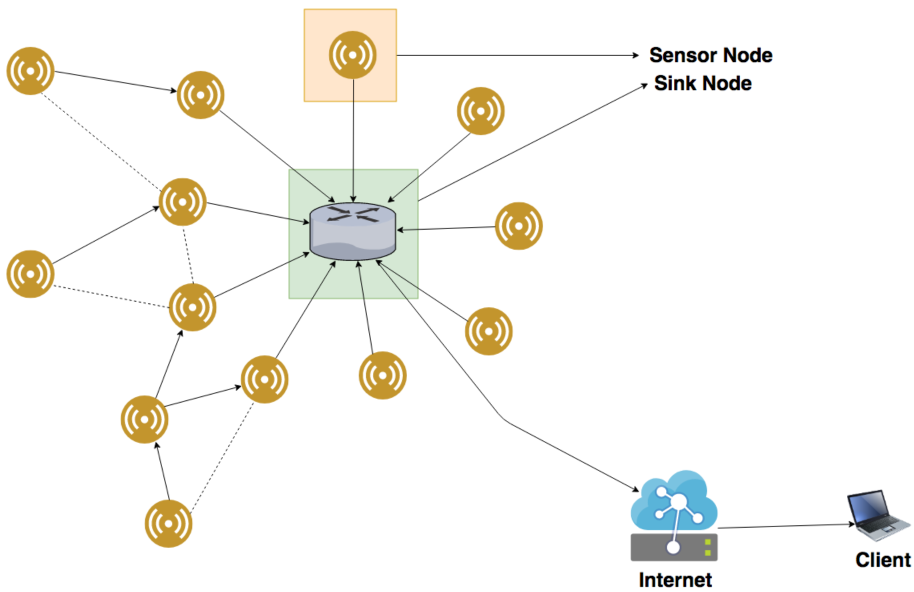

Figure 4.1: Normal Network Model for Simulation

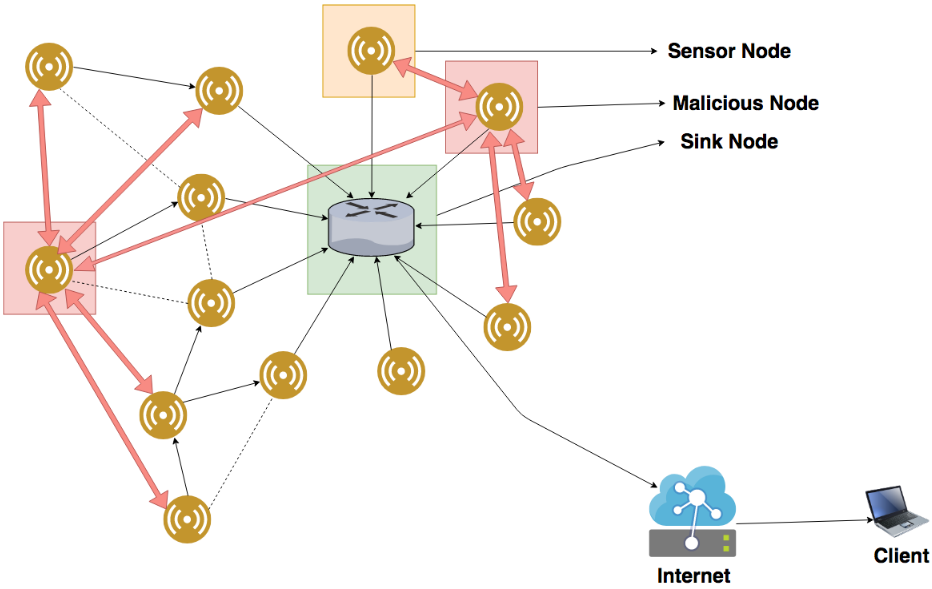

Figure 4.2: Malicious Network Model for Simulation

For the analysis and evaluation ns-3 (network simulator 3) and for the visualization NetAnim and PyViz is used. Wireshark is used to analyze the signaling packet and data packets.

The network model shown above is simulated in PyViz as below:

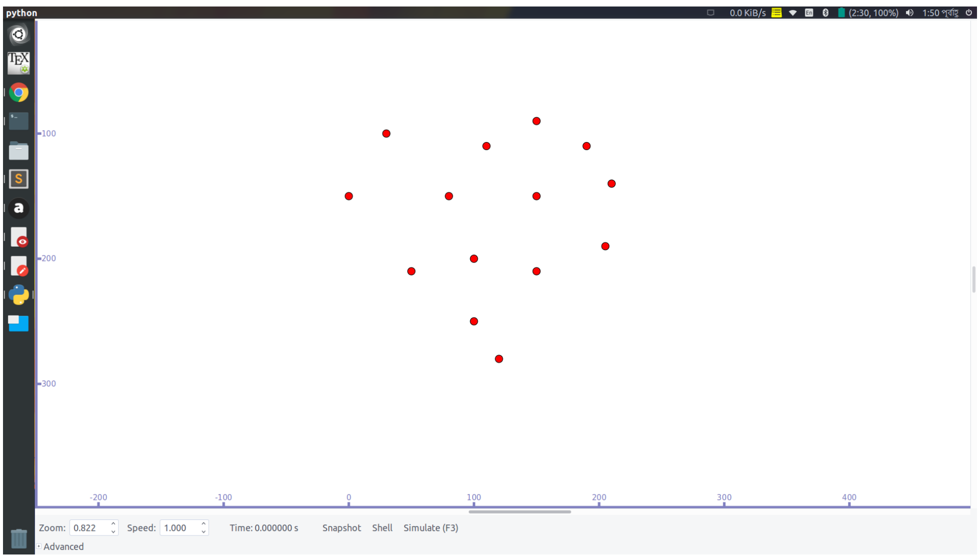

Figure 4.3: Network model

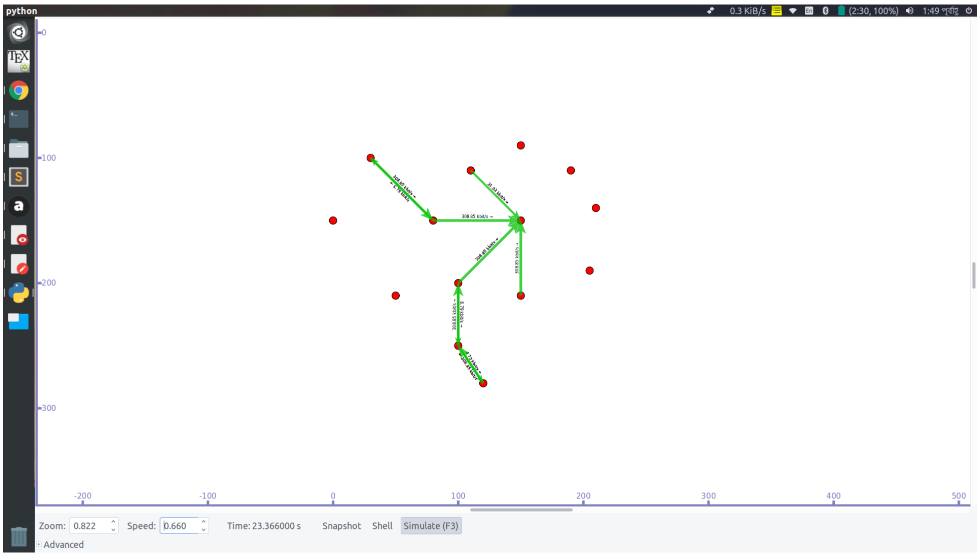

Figure 4.4: Active probing

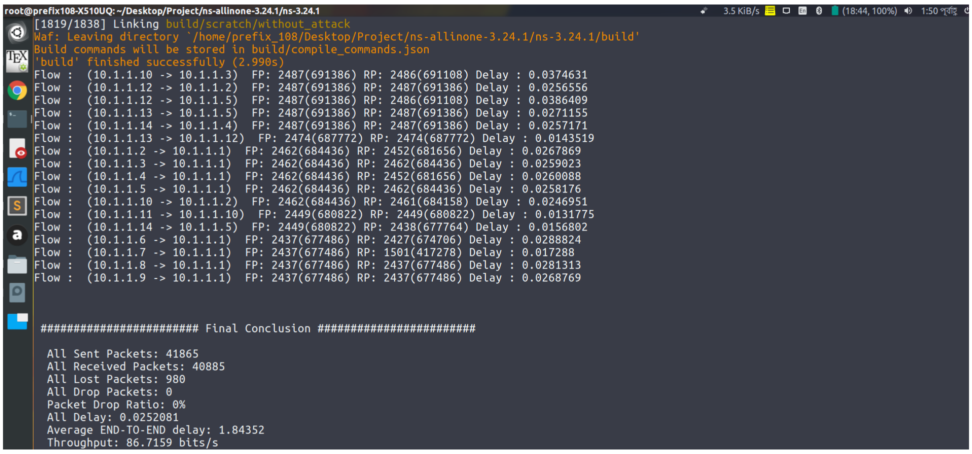

Figure 4.5: Transmission of Data Packets in normal mode

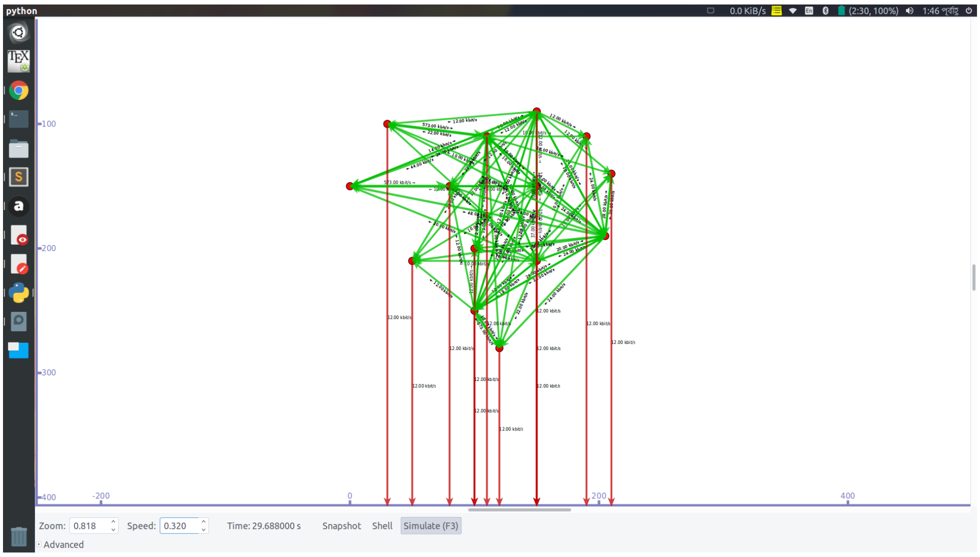

Figure 4.6: Active probing on wormhole attack

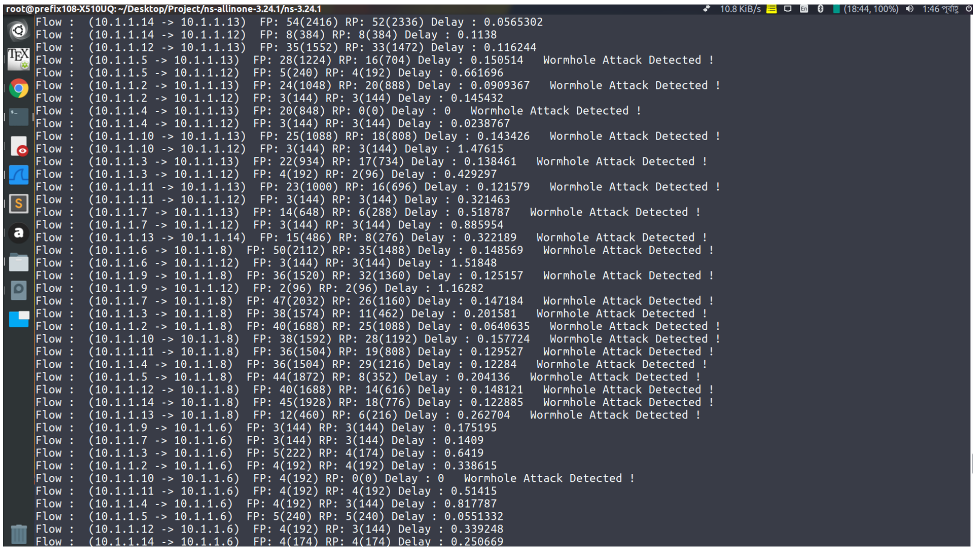

Figure 4.7: Transmission of data packets Under Blackhole attack

4.7 Chapter Summary

In this chapter, the details of the implementation of the scheme is given in NS-3.The next chapter will present the results generated from the simulations.

Chapter 5

Simulation Results and Analysis

In this section the simulation results are presented in detail and an explanation is provided.

5.1 Parameters for Evaluating Simulation Model

The following parameters are needed for evaluating our simulation.

Average Throughput: Number of bits received divided by the difference between the arrival time of the first packet and the last one.

$$Throughput = \frac{Bits\ Received}{timeLastRxPacket - timeFirstTxPacket}$$

Average Packet Delivery Fraction (PDF): Number of packets received divided by the number of packets transmitted.

$$PDF = \frac{No.\ of\ Packets\ Received}{No.\ of\ Packets\ Transmitted}$$

Average end-to-end Delay: The sum of the delay of all received packets divided by the number of received packets.

$$ETE\ Delay = \frac{\sum Delay\ of\ all\ received\ packets}{number\ of\ received\ packets}$$

5.2 Performance of Proposed Method

5.2.1 Average End to End Delay

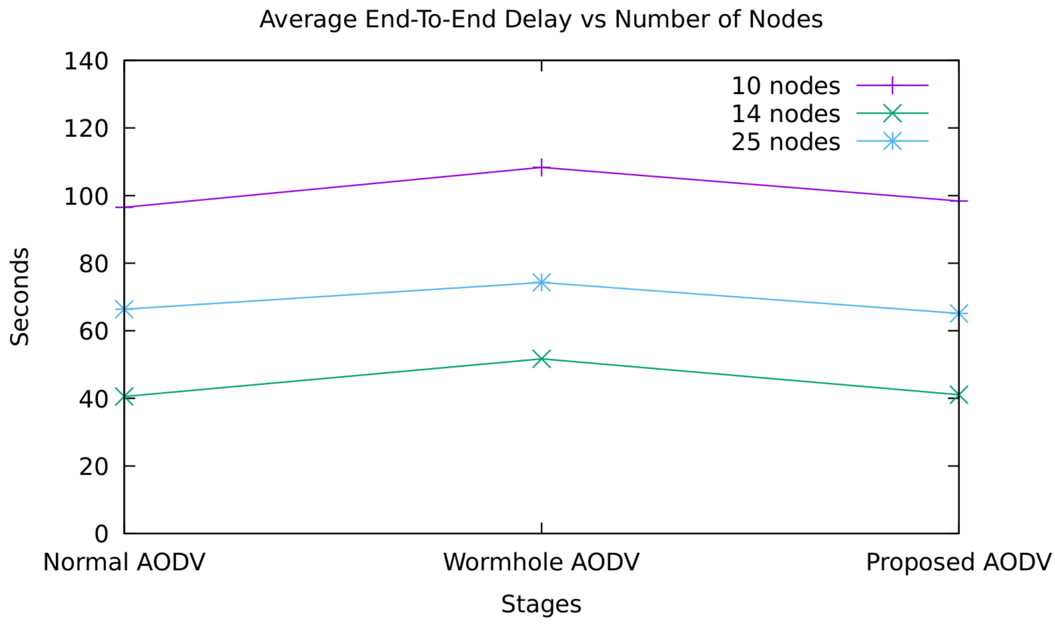

Figure 5.1: Avg. ETE Delay vs. Number of Hops

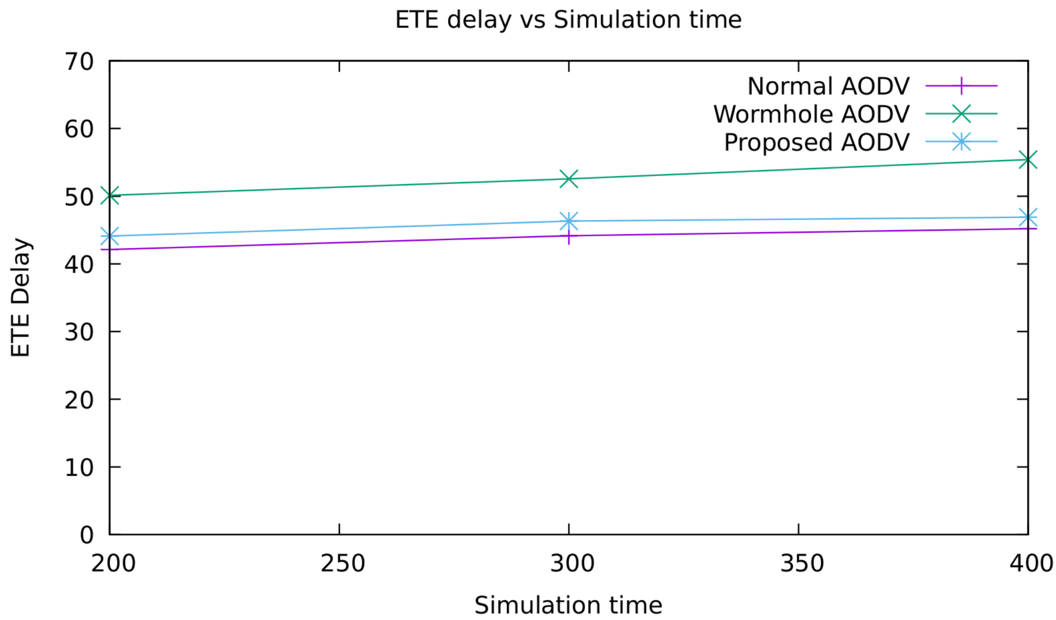

Figure 5.2: Avg. ETE Delay vs. Simulation time

5.2.2 Average Throughput

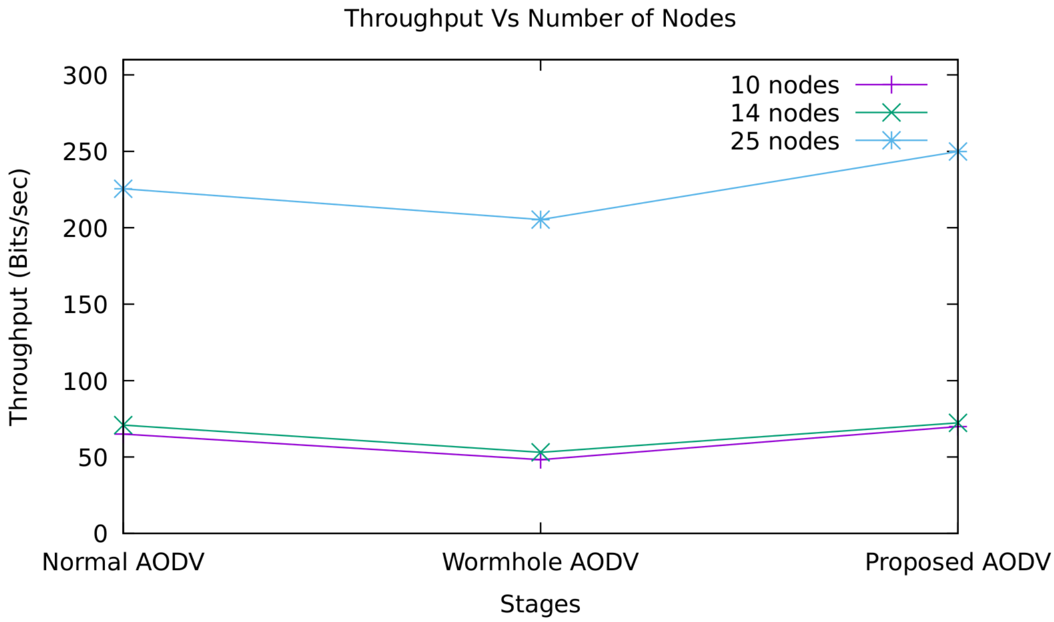

Figure 5.3: Avg. Throughput vs. Number of nodes

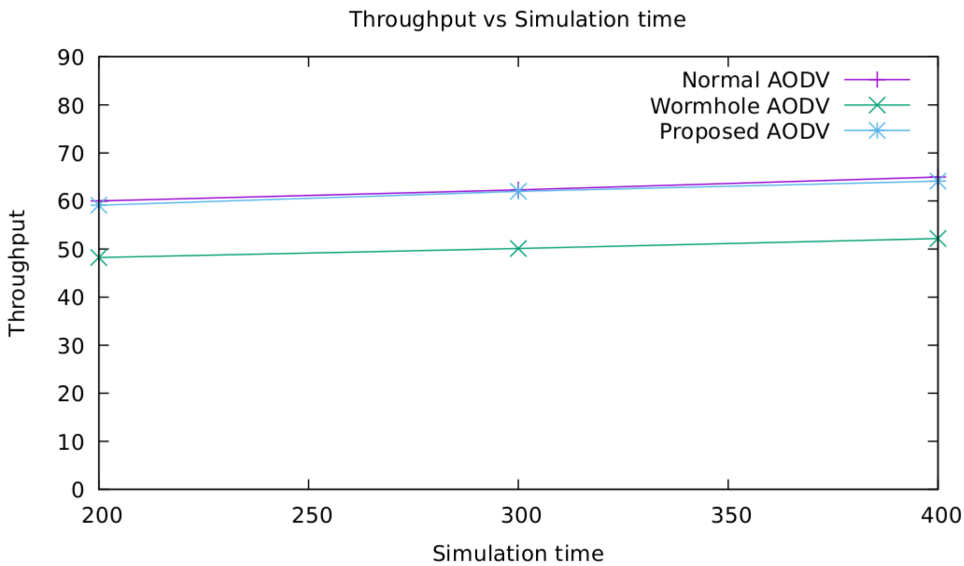

Figure 5.4: Avg. Throughput vs. Simulation time

5.2.3 Packet delivery fraction(PDF)

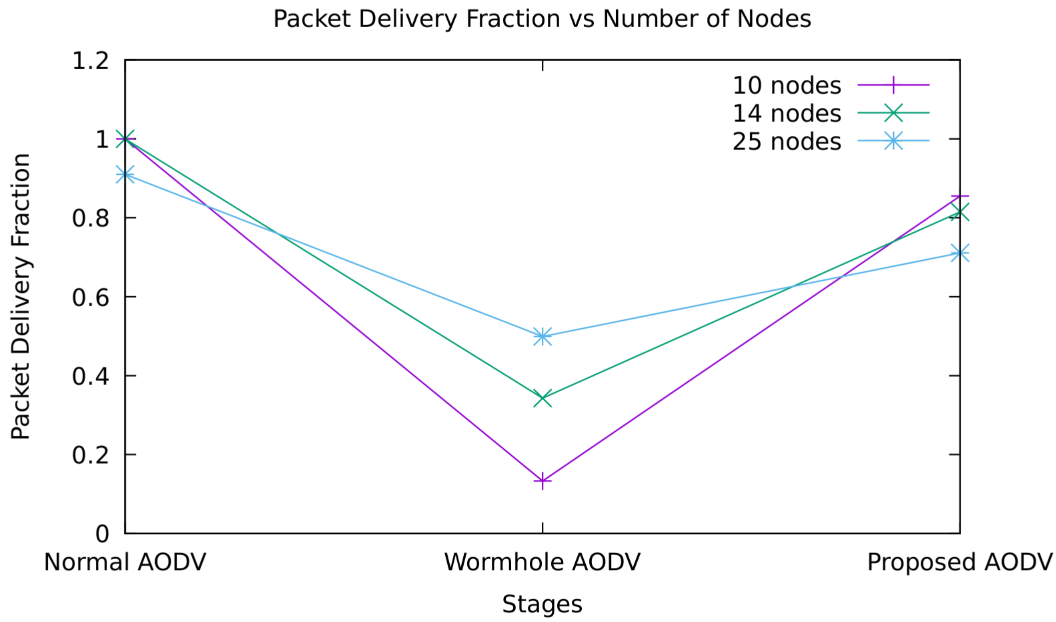

Figure 5.5: PDF vs. Number of Nodes

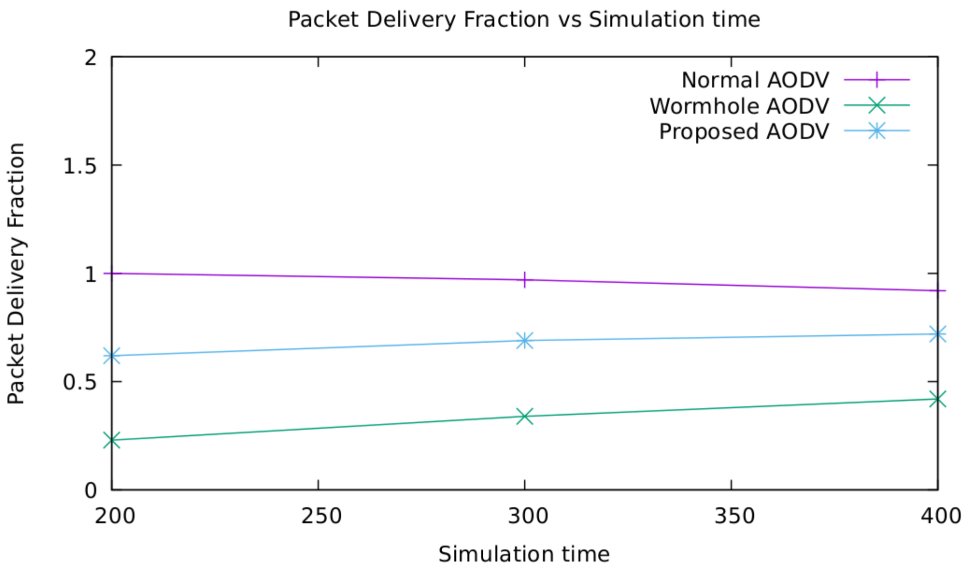

Figure 5.6: PDF vs. Simulation time

5.3 Overall Performance

No. of Nodes	Normal AODV	Wormhole AODV	Proposed AODV
10	67.043	55.534	75.564
14	70.91	53.06	72.38
25	225.53	205.32	249.87

Table 5.1: Average Throughput vs No. of Nodes

Simulation time	Normal AODV	Wormhole AODV	Proposed AODV
200	60.02	48.24	59.10
300	62.32	50.12	61.11
400	64.93	52.20	64.14

Table 5.2: Average Throughput vs Simulation time

No. of Nodes	Normal AODV	Wormhole AODV	Proposed AODV
10	96.54	108.33	98.37
14	40.54	51.72	41.02
25	66.37	70.29	65.11

Table 5.3: Average ETE Delay vs No. of Nodes

Simulation Time	Normal AODV	Wormhole AODV	Proposed AODV
200	42.12	50.12	44.11
300	44.15	52.55	46.32
400	45.20	55.40	46.10

Table 5.4: Average ETE Delay vs Simulation Time

No. of Nodes	Normal AODV	Wormhole AODV	Proposed AODV
10	1.000	0.2933	0.7994
14	1.000	0.3430	0.8150
25	0.9100	0.5227	0.7992

Table 5.5: Packet Delivery Fraction vs No. of Nodes

Simulation Time	Normal AODV	Wormhole AODV	Proposed AODV
200	1	0.23	0.62
300	1	0.34	0.69
400	0.96	0.42	0.72

Table 5.6: Packet Delivery Fraction vs Simulation Time

5.4 Chapter Summary

The objective for the simulation work was to verify the feasibility of the scheme and to compare its latency with the current standard. From the results presented above, the conclusion can be made that this scheme shows the better performance in finding the next AP for STA to associate with when handoff is required compared to other scan techniques.

Chapter 6

Conclusion

This chapter contains an overview of the system and its limitations with future recommendations.

6.1 Findings of the Work

In this thesis, a practical Wormhole attack management scheme have been developed, to manage the transmitted packet. Theoretically, this scheme can reduce the latency associated with Wormhole attack in a network. A set of simulation studies were conducted in order to investigate the performance of the scheme in an IEEE 802.11. In the computer simulations, NS-3 was used to implement the theoretical procedures of the scheme and to simulate the scheme under different network scenarios in order to verify the feasibility of the scheme. Over the course of simulation, the effectiveness of our scheme was demonstrated by comparing it to the IEEE 802.11 standard Wormhole attack and other schemes. The following main observations were made:

- The proposed scheme can reduce the ETE delay.
- It can improve the overall performance by increasing Packet Delivery Fraction. and Throughput.

6.2 Future Works

In this work a Wormhole attack scheme for Infrastructure WSNs has been developed and analyzed. Although this scheme has shown improved Wormhole attack latency in WSN, further analysis of the scheme under different network conditions could be performed. There are some limitations that should be pointed out which concern the experimental setup.

Bibliography

[1] P. Amish and V. Vaghela, "Detection and prevention of wormhole attack in wireless sensor network using aomdv protocol," *Procedia computer science*, vol. 79, pp. 700–707, 2016.

[2] P. Maidamwar and N. Chavhan, "A survey on security issues to detect wormhole attack in wireless sensor network," *International Journal on AdHoc Networking Systems (IJANS) Vol*, vol. 2, pp. 37–50, 2012.

[3] M. O. Johnson, A. Siddiqui, and A. Karami, "A wormhole attack detection and prevention technique in wireless sensor networks."

[4] Y.-C. Hu, A. Perrig, and D. B. Johnson, "Packet leashes: a defense against wormhole attacks in wireless networks," in *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*, vol. 3. IEEE, 2003, pp. 1976–1986.

[5] R. A. Prakash, W. Jeyaseelan, and T. Jayasankar, "Detection, prevention and mitigation of wormhole attack in wireless adhoc network by coordinator," *Appl. Math*, vol. 12, no. 1, pp. 233–237, 2018.

[6] L. Hu and D. Evans, "Using directional antennas to prevent wormhole attacks." in *NDSS*, vol. 4, 2004, pp. 241–245.

[7] Z. Tun and A. H. Maw, "Wormhole attack detection in wireless sensor networks," *World Academy of Science, Engineering and Technology*, vol. 46, p. 2008, 2008.

[8] M. N. A. Shaon and K. Ferens, "Wormhole attack detection in wireless sensor network using discrete wavelet transform," in *Proceedings of the International Conference on Wireless Networks (ICWN)*. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2016, p. 29.

[9] M. Bendjima and M. Feham, "Wormhole attack detection in wireless sensor networks," in *SAI Computing Conference (SAI), 2016*. IEEE, 2016, pp. 1319–1326.

[10] M. A. Matin and M. Islam, "Overview of wireless sensor network," in *Wireless Sensor Networks-Technology and Protocols*. InTech, 2012.

[11] R. Shokri, M. Poturalski, G. Ravot, P. Papadimitratos, and J.-P. Hubaux, "A low-cost secure neighbor verification protocol for wireless sensor networks," Tech. Rep., 2008.

[12] S. Song, H. Wu, and B.-Y. Choi, "Statistical wormhole detection for mobile sensor networks," in *Ubiquitous and Future Networks (ICUFN), 2012 Fourth International Conference on.* IEEE, 2012, pp. 322–327.

[13] V. Bhuse, A. Gupta, and L. Lilien, "Dpdsn: Detection of packet-dropping attacks for wireless sensor networks," in *Proc. Fourth Trusted Internet Workshop*, vol. 107. Citeseer, 2005.

[14] K. Srinivasan, P. Dutta, A. Tavakoli, and P. Levis, "Understanding the causes of packet delivery success and failure in dense wireless sensor networks," in *Proceedings of the 4th international conference on Embedded networked sensor systems.* ACM, 2006, pp. 419–420.

[15] M. Lakde and V. Deshpande, "Packet loss in wireless sensor network: A survey."

[16] O. Doxygen, "Version 2.3. x," 2015.

[17] L. Li, X. Hu, and B. Zhang, "A routing algorithm for wifi-based wireless sensor network and the application in automatic meter reading," *Mathematical Problems in Engineering*, vol. 2013, 2013.

[18] R. Haboub and M. Ouzzif, "Secure routing in wsn," *International Journal of Distributed and Parallel Systems*, vol. 2, no. 6, p. 291, 2011.

[19] J. Govindasamy and S. Punniakodi, "Energy efficient intrusion detection system for zigbee based wireless sensor networks."

Appendix A

Source Code

Header File:

```
#include "ns3/applications-module.h"
#include "ns3/core-module.h"

using namespace ns3;


class MyApp : public Application
{
public:

  MyApp ();
  virtual ~MyApp();

  void Setup (Ptr<Socket> socket, Address address, uint32_t packetSize
    , uint32_t nPackets, DataRate dataRate);

private:
  virtual void StartApplication (void);
  virtual void StopApplication (void);

  void ScheduleTx (void);
  void SendPacket (void);

  Ptr<Socket>     m_socket;
  Address         m_peer;
  uint32_t        m_packetSize;
  uint32_t        m_nPackets;
  DataRate        m_dataRate;
  EventId         m_sendEvent;
  bool            m_running;
  uint32_t        m_packetsSent;
};

MyApp::MyApp ()
  : m_socket (0),
    m_peer (),
    m_packetSize (0),
    m_nPackets (0),
    m_dataRate (0),
```

```
    m_sendEvent (),
    m_running (false),
    m_packetsSent (0)
{
}

MyApp::~MyApp()
{
  m_socket = 0;
}

void
MyApp::Setup (Ptr<Socket> socket, Address address, uint32_t packetSize
    , uint32_t nPackets, DataRate dataRate)
{
  m_socket = socket;
  m_peer = address;
  m_packetSize = packetSize;
  m_nPackets = nPackets;
  m_dataRate = dataRate;
}

void
MyApp::StartApplication (void)
{
  m_running = true;
  m_packetsSent = 0;
  m_socket->Bind ();
  m_socket->Connect (m_peer);
  SendPacket ();
}

void
MyApp::StopApplication (void)
{
  m_running = false;

  if (m_sendEvent.IsRunning ())
    {
      Simulator::Cancel (m_sendEvent);
    }

  if (m_socket)
    {
      m_socket->Close ();
    }
}

void
MyApp::SendPacket (void)
{
  Ptr<Packet> packet = Create<Packet> (m_packetSize);
  m_socket->Send (packet);

  if (++m_packetsSent < m_nPackets)
    {
```

```
      ScheduleTx ();
    }
}

void
MyApp::ScheduleTx (void)
{
  if (m_running)
    {
      Time tNext (Seconds (m_packetSize * 8 / static_cast<double> (m_dataRate.GetBitRate ())));
      m_sendEvent = Simulator::Schedule (tNext, &MyApp::SendPacket, this);
    }
}
```

Normal Mode Source Code:

```
#include "ns3/propagation-module.h"
#include "ns3/flow-monitor-module.h"
#include "ns3/netanim-module.h"

#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/mobility-module.h"
#include "ns3/config-store-module.h"
#include "ns3/wifi-module.h"
#include "ns3/internet-module.h"
#include "ns3/applications-module.h"
#include "ns3/ipv4-global-routing-helper.h"
#include "ns3/olsr-helper.h"
#include "ns3/point-to-point-module.h"
#include "ns3/inet-socket-address.h"
#include "ns3/csma-module.h"

#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <cassert>


NS_LOG_COMPONENT_DEFINE ("WifiSimpleAdhoc");

using namespace ns3;


int main (int argc, char *argv[])
{
  std::string phyMode ("DsssRate1Mbps");
  double     rss = -80;  // -dBm
  bool       verbose = false;
  //uint32_t  packet_size = 2500 ;


  // Set up some default values for the simulation.
  Config::SetDefault ("ns3::OnOffApplication::PacketSize", StringValue
```

```
  ("250kb/s"));
  Config::SetDefault ("ns3::OnOffApplication::DataRate", StringValue ("25kb/s"));


  CommandLine cmd;

  cmd.AddValue ("phyMode", "Wifi Phy mode", phyMode);
  cmd.AddValue ("rss", "received signal strength", rss);
  cmd.AddValue ("verbose", "turn on all WifiNetDevice log components", verbose);

  cmd.Parse (argc, argv);


// Convert to time object
  //Time interPacketInterval = Seconds (interval);



  // disable fragmentation for frames below 2200 bytes
  Config::SetDefault ("ns3::WifiRemoteStationManager::FragmentationThreshold", StringValue ("2500"));
  // turn off RTS/CTS for frames below 2200 bytes
  Config::SetDefault ("ns3::WifiRemoteStationManager::RtsCtsThreshold", StringValue ("2200"));
  // Fix non-unicast data rate to be the same as that of unicast
  Config::SetDefault ("ns3::WifiRemoteStationManager::NonUnicastMode", StringValue (phyMode));

// Here, we will create n nodes.
  NS_LOG_INFO ("Create nodes.");

  NodeContainer serverNode;
  NodeContainer clientNodes;
  serverNode.Create (1);
  clientNodes.Create (4);

  NodeContainer cn;
  cn.Create(4);

  NodeContainer cn_extra;
  cn_extra.Create(1);

  NodeContainer cn_extra_2;
  cn_extra_2.Create(4);

  NodeContainer allNodes = NodeContainer (serverNode, clientNodes, cn, cn_extra, cn_extra_2);

  // NodeContainer sn;
  // NodeContainer cn;
  // sn.Create(1);
  // cn.Create(4);
  // NodeContainer all = NodeContainer(sn,cn);

```

```
// NodeContainer sn1;
// NodeContainer cn1;
// sn1.Create(1);
// cn1.Create(4);
// NodeContainer all1 = NodeContainer(sn1,cn1);

// The below set of helpers will help us to put together the wifi
  NICs we want
WifiHelper wifi;
if (verbose)
  {
    wifi.EnableLogComponents (); // Turn on all Wifi logging
  }
wifi.SetStandard (WIFI_PHY_STANDARD_80211b);


YansWifiPhyHelper wifiPhy =  YansWifiPhyHelper::Default ();
// This is one parameter that matters when using FixedRssLossModel
// set it to zero; otherwise, gain will be added
wifiPhy.Set ("RxGain", DoubleValue (0) );
// ns-3 supports RadioTap and Prism tracing extensions for 802.11b
wifiPhy.SetPcapDataLinkType (YansWifiPhyHelper::DLT_IEEE802_11_RADIO
  );

YansWifiChannelHelper wifiChannel;
wifiChannel.SetPropagationDelay ("ns3::
  ConstantSpeedPropagationDelayModel");
// The below FixedRssLossModel will cause the rss to be fixed
  regardless
// of the distance between the two stations, and the transmit power

wifiChannel.AddPropagationLoss ("ns3::FixedRssLossModel","Rss",
  DoubleValue (rss));
wifiPhy.SetChannel (wifiChannel.Create ());

// Add a non-QoS upper mac, and disable rate control
NqosWifiMacHelper wifiMac = NqosWifiMacHelper::Default ();
wifi.SetRemoteStationManager ("ns3::ConstantRateWifiManager",
                              "DataMode",StringValue (phyMode),
                              "ControlMode",StringValue (phyMode));
// Set it to adhoc mode
wifiMac.SetType ("ns3::AdhocWifiMac");
NetDeviceContainer devices = wifi.Install (wifiPhy, wifiMac,
  allNodes);
// NetDeviceContainer devices1 = wifi.Install (wifiPhy, wifiMac, all
  );
// NetDeviceContainer devices2 = wifi.Install (wifiPhy, wifiMac,
  all1);

// Note that with FixedRssLossModel, the positions below are not
// used for received signal strength.
MobilityHelper mobility;
Ptr<ListPositionAllocator> positionAlloc = CreateObject<
  ListPositionAllocator> ();
positionAlloc->Add (Vector (150.0, 150.0, 150.0));
```

```
positionAlloc->Add (Vector (100.0, 200.0, 0.0));
positionAlloc->Add (Vector (150.0, 210.0, 0.0));
positionAlloc->Add (Vector (110.0, 110.0, 0.0));
positionAlloc->Add (Vector (80.0, 150.0, 0.0));

// positionAlloc->Add (Vector (160.0, 160.0, 160.0));
positionAlloc->Add (Vector (210.0, 140.0, 300.0));
positionAlloc->Add (Vector (150.0, 90.0, 300.0));
positionAlloc->Add (Vector (205.0, 190.0, 300.0));
positionAlloc->Add (Vector (190.0, 110.0, 300.0));

positionAlloc->Add (Vector (100.0, 250.0, 0.0));
positionAlloc->Add (Vector (120.0, 280.0, 0.0));
positionAlloc->Add (Vector (50.0, 210.0, 0.0));
positionAlloc->Add (Vector (0.0, 150.0, 0.0));
positionAlloc->Add (Vector (30.0, 100.0, 0.0));

//positionAlloc->Add (Vector (200.0, 240.0, 0.0));

// positionAlloc->Add (Vector (60.0, -80.0, 0.0));


// positionAlloc->Add (Vector (140.0, 170.0, 0.0));
// positionAlloc->Add (Vector (-100.0, 300.0, 300.0));
// positionAlloc->Add (Vector (-50.0, 350.0, 300.0));
// positionAlloc->Add (Vector (-0.0, 290.0, 300.0));
// positionAlloc->Add (Vector (-50.0, 250.0, 300.0));

mobility.SetPositionAllocator (positionAlloc);
mobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
mobility.Install (allNodes);
// mobility.Install (all);
// mobility.Install (all1);

InternetStackHelper internet;
internet.Install (allNodes);
// internet.Install (all);
// internet.Install (all1);

Ipv4AddressHelper ipv4;
NS_LOG_INFO ("Assign IP Addresses.");
ipv4.SetBase ("10.1.1.0", "255.255.255.0");
Ipv4InterfaceContainer i = ipv4.Assign (devices);
// Ipv4AddressHelper add;
// add.SetBase ("10.1.2.0", "255.255.255.0");
// Ipv4InterfaceContainer j = add.Assign (devices1);
// Ipv4AddressHelper add1;
// add1.SetBase ("10.1.3.0", "255.255.255.0");
// Ipv4InterfaceContainer k = add1.Assign (devices2);

// Create a packet sink on the star "hub" to receive these packets
uint16_t port = 5000;
Address sinkLocalAddress (InetSocketAddress (Ipv4Address::GetAny (),
  port));
PacketSinkHelper sinkHelper ("ns3::UdpSocketFactory",
 sinkLocalAddress);
```

```

  ApplicationContainer sinkApp = sinkHelper.Install (serverNode);
  // ApplicationContainer sinkApp_2 = sinkHelper.Install (sn);
  // ApplicationContainer sinkApp_3 = sinkHelper.Install (sn1);
  sinkApp.Start (Seconds (1.0));
  //sinkApp.Stop (Seconds (100.0));

  // sinkApp_2.Start(Seconds(1.0));
  // sinkApp_2.Stop (Seconds (100.0));

  // sinkApp_3.Start(Seconds(1.0));
  // sinkApp_3.Stop (Seconds (100.0));

  // Create the OnOff applications to send UDP to the server
  OnOffHelper clientHelper ("ns3::UdpSocketFactory", Address ());
  //Ptr<Socket> ns3UdpSocket = Socket::CreateSocket (cn_extra.Get (10), UdpSocketFactory::GetTypeId ());


  clientHelper.SetAttribute ("OnTime", StringValue ("ns3::ConstantRandomVariable[Constant=1]"));
  clientHelper.SetAttribute ("OffTime", StringValue ("ns3::ConstantRandomVariable[Constant=0]"));
//normally wouldn't need a loop here but the server IP address is different
  //on each p2p subnet
  ApplicationContainer clientApps;
  for(uint32_t j=0; j<clientNodes.GetN (); ++j)
    {
      AddressValue remoteAddress (InetSocketAddress (i.GetAddress (0), port));
      clientHelper.SetAttribute ("Remote", remoteAddress);
      clientApps.Add (clientHelper.Install (clientNodes.Get (j)));
    }
  clientApps.Start (Seconds (3.0));
  //clientApps.Stop (Seconds (100.0));


  ApplicationContainer clientApps_2;
  for(uint32_t k=0; k<cn.GetN (); ++k)
    {
      AddressValue remoteAddress (InetSocketAddress (i.GetAddress (0), port));
      clientHelper.SetAttribute ("Remote", remoteAddress);
      clientApps_2.Add (clientHelper.Install (cn.Get (k)));
    }
  clientApps_2.Start (Seconds (5.0));
  //clientApps_2.Stop (Seconds (100.0));


  ApplicationContainer extra_1;
  for(uint32_t k=0; k<cn_extra.GetN (); ++k)
    {
      AddressValue remoteAddress (InetSocketAddress (i.GetAddress (2), port));
      clientHelper.SetAttribute ("Remote", remoteAddress);
```

```
      extra_1.Add (clientHelper.Install (cn_extra.Get (k)));
    }
  extra_1.Start (Seconds (1.0));
  //extra_1.Stop (Seconds (100.0));


  ApplicationContainer extra_2;
  AddressValue remoteAddress (InetSocketAddress (i.GetAddress (9),
    port));
  clientHelper.SetAttribute ("Remote", remoteAddress);
  extra_2.Add (clientHelper.Install (cn_extra_2.Get (0)));
  extra_2.Start (Seconds (4.0));
  //extra_2.Stop (Seconds (100.0));


  ApplicationContainer extra_4;
  AddressValue remoteAddress1 (InetSocketAddress (i.GetAddress (1),
    port));
  clientHelper.SetAttribute ("Remote", remoteAddress1);
  extra_4.Add (clientHelper.Install (cn_extra_2.Get (1)));
  extra_4.Start (Seconds (1.0));
  //extra_4.Stop (Seconds (100.0));

  ApplicationContainer extra_5;
  AddressValue remoteAddress2 (InetSocketAddress (i.GetAddress (4),
    port));
  clientHelper.SetAttribute ("Remote", remoteAddress2);
  extra_5.Add (clientHelper.Install (cn_extra_2.Get (2)));
  extra_5.Start (Seconds (1.0));
  //extra_5.Stop (Seconds (100.0));

  ApplicationContainer extra_6;
  AddressValue remoteAddress3 (InetSocketAddress (i.GetAddress (3),
    port));
  clientHelper.SetAttribute ("Remote", remoteAddress3);
  extra_6.Add (clientHelper.Install (cn_extra_2.Get (3)));
  extra_6.Start (Seconds (1.0));
  //extra_6.Stop (Seconds (100.0));

  ApplicationContainer extra_7;
  AddressValue remoteAddress4 (InetSocketAddress (i.GetAddress (4),
    port));
  clientHelper.SetAttribute ("Remote", remoteAddress4);
  extra_7.Add (clientHelper.Install (cn_extra_2.Get (3)));
  extra_7.Start (Seconds (4.0));
  //extra_7.Stop (Seconds (100.0));


  ApplicationContainer extra_8;
  AddressValue remoteAddress5 (InetSocketAddress (i.GetAddress (4),
    port));
  clientHelper.SetAttribute ("Remote", remoteAddress5);
  extra_8.Add (clientHelper.Install (cn_extra_2.Get (1)));
  extra_8.Start (Seconds (1.0));
  //extra_8.Stop (Seconds (100.0));

```

```
ApplicationContainer extra_9;
AddressValue remoteAddress6 (InetSocketAddress (i.GetAddress (11),
  port));
clientHelper.SetAttribute ("Remote", remoteAddress6);
extra_9.Add (clientHelper.Install (cn_extra_2.Get (2)));
extra_9.Start (Seconds (2.0));
//extra_9.Stop (Seconds (100.0));


ApplicationContainer extra_10;
AddressValue remoteAddress7 (InetSocketAddress (i.GetAddress (1),
  port));
clientHelper.SetAttribute ("Remote", remoteAddress7);
extra_10.Add (clientHelper.Install (cn_extra.Get (0)));
extra_10.Start (Seconds (3.0));
//extra_10.Stop (Seconds (100.0));

//configure tracing
AsciiTraceHelper ascii;
wifiPhy.EnableAsciiAll (ascii.CreateFileStream ("udp-single-hop.tr")
  );
wifiPhy.EnablePcapAll ("udp-single-hop");

// Install FlowMonitor on all nodes
FlowMonitorHelper flowmon;
Ptr<FlowMonitor> monitor = flowmon.InstallAll ();


// AnimationInterface anim ("without_attack.xml"); // Mandatory
// AnimationInterface::SetConstantPosition (serverNode.Get (0),
  500,500);
// AnimationInterface::SetConstantPosition (clientNodes.Get (0),
  750,500);
// AnimationInterface::SetConstantPosition (clientNodes.Get (1),
  750, 750);
// AnimationInterface::SetConstantPosition (clientNodes.Get (2),
  500, 750);
// AnimationInterface::SetConstantPosition (clientNodes.Get (3),
  250, 750);
// AnimationInterface::SetConstantPosition (clientNodes.Get (4),
  250, 500);
// AnimationInterface::SetConstantPosition (clientNodes.Get (5),
  250, 250);
// AnimationInterface::SetConstantPosition (clientNodes.Get (6),
  500, 250);
// AnimationInterface::SetConstantPosition (clientNodes.Get (7),
  750, 250);
// anim.EnablePacketMetadata(true);


// Run simulation
Simulator::Stop (Seconds (200));
Simulator::Run ();

// Print per flow statistics
```

```
  monitor->CheckForLostPackets ();

  Ptr<Ipv4FlowClassifier> classifier = DynamicCast<Ipv4FlowClassifier> (flowmon.GetClassifier ());
  std::map<FlowId, FlowMonitor::FlowStats> stats = monitor->GetFlowStats ();


         uint32_t txPacketsum = 0;
         uint32_t rxPacketsum = 0;
         uint32_t rxBytesum = 0;
         double DropPacketsum = 0;
         uint32_t LostPacketsum = 0;
         double Delaysum = 0;


for (std::map<FlowId, FlowMonitor::FlowStats>::const_iterator i = stats.begin (); i != stats.end (); ++i)
    {
           Ipv4FlowClassifier::FiveTuple t = classifier->FindFlow (i->first);
           std::cout << "Flow : " << " (" << t.sourceAddress << " -> " << t.destinationAddress << ")  ";

                  txPacketsum += i->second.txPackets;
                  rxPacketsum += i->second.rxPackets;
                  rxBytesum += i->second.rxBytes;
                  LostPacketsum += i->second.lostPackets;
                  DropPacketsum += i->second.packetsDropped.size();
                  Delaysum += i->second.delaySum.GetSeconds();

           std::cout << "FP: " << i->second.txPackets << "(" << i->second.txBytes<< ")  ";
           std::cout << "RP: " << i->second.rxPackets << "(" << i->second.rxBytes<< ")  ";
           //std::cout << "  Lost Packets:   " << i->second.lostPackets << "\n";
           //std::cout << "  Drop Packets:   " << i->second.packetsDropped.size() << "\n";
           //std::cout << "  Packets Delivery Ratio: " << ((rxPacketsum * 100) /txPacketsum) << "%" << "\n";
           //std::cout << "  Packets Lost Ratio: " << ((LostPacketsum * 100) /txPacketsum) << "%" << "\n";
           //std::cout <<"Throughput: " << i->second.rxBytes * 8.0 / 10.0 / 1024 / 1024  << " Mbps ";
           std::cout << "Delay : " << (i->second.delaySum.GetSeconds() / i->second.txPackets) ;
           std::cout << "\n";
    }
         std::cout << "\n\n\n ######################### Final Conclusion #########################" << "\n\n";
         std::cout << "  All Sent Packets: " << txPacketsum <<"
                \n" << "  All Received Packets: " << rxPacketsum << "\n";
         std::cout << "  All Lost Packets: " << (txPacketsum - rxPacketsum) <<"                    \n" << "  All Drop Packets: " <<
```

```
DropPacketsum << "\n";
        std::cout << "  Packet Drop Ratio: " <<(double)((DropPacketsum /txPacketsum) *100 ) << "% \n";
        //std::cout << "  Packets Delivery Ratio: " << ((rxPacketsum * 100) /txPacketsum) << "%" << "            \n" << "  Packets Lost Ratio: " << (double)((LostPacketsum * 100) /txPacketsum) << "%" << "\n";
        std::cout << "  All Delay: " << Delaysum / txPacketsum << "\n";
        std::cout << "  Average END-TO-END delay: " << ((Delaysum / txPacketsum) / rxPacketsum) * 2990000 << "\n" ;
        std::cout << "  Throughput: " << (rxBytesum * 8.0 / 10.0 / 1024 / 1024)*10 << " bits/s\n";

      // //hop count
      // Ipv4Header header;
      // packet->PeekHeader (&header);
      // uint8_t ttl = header.GetTtl();
      // std::cout<<"Hop Count: "<<ttl<<"\n";
      // //end of hop count



  Simulator::Destroy ();

  return 0;
}
```

Wormhole Attack Mode Source Code:

```
#include "ns3/propagation-module.h"
#include "ns3/flow-monitor-module.h"

#include "ns3/aodv-module.h"
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/mobility-module.h"
#include "ns3/config-store-module.h"
#include "ns3/wifi-module.h"
#include "ns3/internet-module.h"
#include "ns3/applications-module.h"
#include "ns3/ipv4-global-routing-helper.h"
#include "myapp.h"

#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <cassert>



NS_LOG_COMPONENT_DEFINE ("WifiSimpleAdhoc");

using namespace ns3;


```

```
int main (int argc, char *argv[])
{
  std::string phyMode ("DsssRate1Mbps");
  double rss = -80;  // -dBm
  bool verbose = false;

  // Set up some default values for the simulation.
  Config::SetDefault ("ns3::OnOffApplication::PacketSize", StringValue
    ("2500kb/s"));
  Config::SetDefault ("ns3::OnOffApplication::DataRate", StringValue (
   "25kb/s"));


  CommandLine cmd;

  cmd.AddValue ("phyMode", "Wifi Phy mode", phyMode);
  cmd.AddValue ("rss", "received signal strength", rss);
  cmd.AddValue ("verbose", "turn on all WifiNetDevice log components",
    verbose);

  cmd.Parse (argc, argv);


// Convert to time object
  //Time interPacketInterval = Seconds (interval);



  // disable fragmentation for frames below 2200 bytes
  Config::SetDefault ("ns3::WifiRemoteStationManager::
   FragmentationThreshold", StringValue ("2200"));
  // turn off RTS/CTS for frames below 2200 bytes
  Config::SetDefault ("ns3::WifiRemoteStationManager::RtsCtsThreshold"
   , StringValue ("2200"));
  // Fix non-unicast data rate to be the same as that of unicast
  Config::SetDefault ("ns3::WifiRemoteStationManager::NonUnicastMode",
    StringValue (phyMode));

// Here, we will create nodes.
  NS_LOG_INFO ("Create nodes.");
  NodeContainer malicious;
  NodeContainer not_malicious;
  NodeContainer serverNode;
  NodeContainer clientNodes;
  serverNode.Create (1);
  clientNodes.Create (4);

  NodeContainer cn;
  cn.Create(4);

  NodeContainer cn_extra;
  cn_extra.Create(1);

  NodeContainer cn_extra_2;
  cn_extra_2.Create(4);

```

```
  NodeContainer allNodes = NodeContainer (serverNode, clientNodes, cn,
    cn_extra, cn_extra_2);

  not_malicious.Add(serverNode.Get(0));
  not_malicious.Add(clientNodes.Get(0));
  not_malicious.Add(clientNodes.Get(1));
  not_malicious.Add(clientNodes.Get(2));
  not_malicious.Add(clientNodes.Get(3));

  not_malicious.Add(cn.Get(0));
  not_malicious.Add(cn.Get(1));
  malicious.Add(cn.Get(2));
  not_malicious.Add(cn.Get(3));

  not_malicious.Add(cn_extra.Get(0));

  not_malicious.Add(cn_extra_2.Get(0));
  not_malicious.Add(cn_extra_2.Get(1));
  malicious.Add(cn_extra_2.Get(2));
  not_malicious.Add(cn_extra_2.Get(3));

  // The below set of helpers will help us to put together the wifi
    NICs we want
  WifiHelper wifi;
  if (verbose)
    {
      wifi.EnableLogComponents ();  // Turn on all Wifi logging
    }
  wifi.SetStandard (WIFI_PHY_STANDARD_80211b);


  YansWifiPhyHelper wifiPhy =  YansWifiPhyHelper::Default ();
  // This is one parameter that matters when using FixedRssLossModel
  // set it to zero; otherwise, gain will be added
  wifiPhy.Set ("RxGain", DoubleValue (0) );
  // ns-3 supports RadioTap and Prism tracing extensions for 802.11b
  wifiPhy.SetPcapDataLinkType (YansWifiPhyHelper::DLT_IEEE802_11_RADIO
    );

  YansWifiChannelHelper wifiChannel;
  wifiChannel.SetPropagationDelay ("ns3::
    ConstantSpeedPropagationDelayModel");
  // The below FixedRssLossModel will cause the rss to be fixed
    regardless
  // of the distance between the two stations, and the transmit power

  wifiChannel.AddPropagationLoss ("ns3::FixedRssLossModel","Rss",
    DoubleValue (rss));
  wifiPhy.SetChannel (wifiChannel.Create ());

  // Add a non-QoS upper mac, and disable rate control
  NqosWifiMacHelper wifiMac = NqosWifiMacHelper::Default ();
  wifi.SetRemoteStationManager ("ns3::ConstantRateWifiManager",
                                "DataMode",StringValue (phyMode),
                                "ControlMode",StringValue (phyMode));
  // Set it to adhoc mode
```

```
wifiMac.SetType ("ns3::AdhocWifiMac");
NetDeviceContainer devices = wifi.Install (wifiPhy, wifiMac, allNodes);
NetDeviceContainer mal_devices = wifi.Install(wifiPhy, wifiMac, malicious);

//  Enable AODV
AodvHelper aodv;
AodvHelper malicious_aodv;


// Note that with FixedRssLossModel, the positions below are not
// used for received signal strength.
MobilityHelper mobility;
Ptr<ListPositionAllocator> positionAlloc = CreateObject<ListPositionAllocator> ();
positionAlloc->Add (Vector (150.0, 150.0, 150.0));
positionAlloc->Add (Vector (100.0, 200.0, 0.0));
positionAlloc->Add (Vector (150.0, 210.0, 0.0));
positionAlloc->Add (Vector (110.0, 110.0, 0.0));
positionAlloc->Add (Vector (80.0, 150.0, 0.0));

// positionAlloc->Add (Vector (160.0, 160.0, 160.0));
positionAlloc->Add (Vector (210.0, 140.0, 300.0));
positionAlloc->Add (Vector (150.0, 90.0, 300.0));
positionAlloc->Add (Vector (205.0, 190.0, 300.0));
positionAlloc->Add (Vector (190.0, 110.0, 300.0));

positionAlloc->Add (Vector (100.0, 250.0, 0.0));
positionAlloc->Add (Vector (120.0, 280.0, 0.0));
positionAlloc->Add (Vector (50.0, 210.0, 0.0));
positionAlloc->Add (Vector (0.0, 150.0, 0.0));
positionAlloc->Add (Vector (30.0, 100.0, 0.0));

mobility.SetPositionAllocator (positionAlloc);
mobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
mobility.Install (allNodes);

InternetStackHelper internet;
internet.SetRoutingHelper (aodv);
internet.Install (not_malicious);

malicious_aodv.Set("EnableWrmAttack",BooleanValue(true)); // putting *false* instead of *true* would disable the malicious behavior of the node

malicious_aodv.Set("FirstEndWifiWormTunnel",Ipv4AddressValue("10.0.1.1"));
malicious_aodv.Set("FirstEndWifiWormTunnel",Ipv4AddressValue("10.0.1.2"));

internet.SetRoutingHelper (malicious_aodv);
internet.Install (malicious);

Ipv4AddressHelper ipv4;
NS_LOG_INFO ("Assign IP Addresses.");
```

```
  ipv4.SetBase ("10.1.1.0", "255.255.255.0");
  Ipv4InterfaceContainer i = ipv4.Assign (devices);


  ipv4.SetBase ("10.1.2.0", "255.255.255.0");
  Ipv4InterfaceContainer mal_ifcont = ipv4.Assign (mal_devices);

  // Create a packet sink on the star "hub" to receive these packets
  uint16_t port = 50000;
  Address sinkLocalAddress (InetSocketAddress (Ipv4Address::GetAny (),
     port));
  PacketSinkHelper sinkHelper ("ns3::UdpSocketFactory",
   sinkLocalAddress);

  ApplicationContainer sinkApp = sinkHelper.Install (serverNode);
  sinkApp.Start (Seconds (1.0));
  //sinkApp.Stop (Seconds (100.0));

  // Create the OnOff applications to send UDP to the server
  OnOffHelper clientHelper ("ns3::UdpSocketFactory", Address ());

 clientHelper.SetAttribute ("OnTime", StringValue ("ns3::
   ConstantRandomVariable[Constant=1]"));
  clientHelper.SetAttribute ("OffTime", StringValue ("ns3::
   ConstantRandomVariable[Constant=0]"));
//normally wouldn't need a loop here but the server IP address is
   different
  //on each p2p subnet
  ApplicationContainer clientApps;
  for(uint32_t j=0; j<clientNodes.GetN (); ++j)
    {
      AddressValue remoteAddress (InetSocketAddress (i.GetAddress (0),
    port));
      clientHelper.SetAttribute ("Remote", remoteAddress);
      clientApps.Add (clientHelper.Install (clientNodes.Get (j)));
    }
  clientApps.Start (Seconds (3.0));
  //clientApps.Stop (Seconds (100.0));


  ApplicationContainer clientApps_2;
  for(uint32_t k=0; k<cn.GetN (); ++k)
    {
      AddressValue remoteAddress (InetSocketAddress (i.GetAddress (0),
    port));
      clientHelper.SetAttribute ("Remote", remoteAddress);
      clientApps_2.Add (clientHelper.Install (cn.Get (k)));
    }
  clientApps_2.Start (Seconds (5.0));
  //clientApps_2.Stop (Seconds (100.0));


  ApplicationContainer extra_1;
  for(uint32_t k=0; k<cn_extra.GetN (); ++k)
    {
      AddressValue remoteAddress (InetSocketAddress (i.GetAddress (2),
```

```
      port));
        clientHelper.SetAttribute ("Remote", remoteAddress);
        extra_1.Add (clientHelper.Install (cn_extra.Get (k)));
      }
    extra_1.Start (Seconds (7.0));
    //extra_1.Stop (Seconds (100.0));


    ApplicationContainer extra_2;
    AddressValue remoteAddress (InetSocketAddress (i.GetAddress (9), port));
    clientHelper.SetAttribute ("Remote", remoteAddress);
    extra_2.Add (clientHelper.Install (cn_extra_2.Get (0)));
    extra_2.Start (Seconds (4.0));
    //extra_2.Stop (Seconds (100.0));

    // ApplicationContainer extra_3;
    // AddressValue remoteAddress0 (InetSocketAddress (i.GetAddress (2), port));
    // clientHelper.SetAttribute ("Remote", remoteAddress0);
    // extra_3.Add (clientHelper.Install (cn_extra_2.Get (0)));

    ApplicationContainer extra_4;
    AddressValue remoteAddress1 (InetSocketAddress (i.GetAddress (1), port));
    clientHelper.SetAttribute ("Remote", remoteAddress1);
    extra_4.Add (clientHelper.Install (cn_extra_2.Get (1)));

    ApplicationContainer extra_5;
    AddressValue remoteAddress2 (InetSocketAddress (i.GetAddress (4), port));
    clientHelper.SetAttribute ("Remote", remoteAddress2);
    extra_5.Add (clientHelper.Install (cn_extra_2.Get (2)));

    ApplicationContainer extra_6;
    AddressValue remoteAddress3 (InetSocketAddress (i.GetAddress (3), port));
    clientHelper.SetAttribute ("Remote", remoteAddress3);
    extra_6.Add (clientHelper.Install (cn_extra_2.Get (3)));

    //configure tracing
    AsciiTraceHelper ascii;
    wifiPhy.EnableAsciiAll (ascii.CreateFileStream ("udp-single-hop.tr"));
    wifiPhy.EnablePcapAll ("udp-single-hop");

    // Install FlowMonitor on all nodes
    FlowMonitorHelper flowmon;
    Ptr<FlowMonitor> monitor = flowmon.InstallAll ();

    // Run simulation for 10 seconds
    Simulator::Stop (Seconds (20));
    Simulator::Run ();

    // Print per flow statistics
    monitor->CheckForLostPackets ();
```

```

Ptr<Ipv4FlowClassifier> classifier = DynamicCast<Ipv4FlowClassifier>(flowmon.GetClassifier ());
std::map<FlowId, FlowMonitor::FlowStats> stats = monitor->GetFlowStats ();


        uint32_t txPacketsum = 0;
        uint32_t rxPacketsum = 0;
        uint32_t rxBytesum = 0;
        double DropPacketsum = 0;
        uint32_t LostPacketsum = 0;
        double packet_loss_threshold = 2.0;
        double delay_threshold = 1.5;
        double Delaysum = 0;
        //double etf = 40 ;


for (std::map<FlowId, FlowMonitor::FlowStats>::const_iterator i = stats.begin (); i != stats.end (); ++i)
    {
          Ipv4FlowClassifier::FiveTuple t = classifier->FindFlow (i->first);

                //for counting the total result
                txPacketsum += i->second.txPackets;
                rxPacketsum += i->second.rxPackets;
                rxBytesum += i->second.rxBytes;
                LostPacketsum += i->second.lostPackets;
                //DropPacketsum += i->second.packetsDropped.size();
                 for (uint32_t j=0; j < i->second.packetsDropped.size() ; j++){
                  DropPacketsum += i->second.packetsDropped[j];
                 }

                Delaysum += i->second.delaySum.GetSeconds();
                //end of counting the total result

          if( t.sourceAddress != "10.1.1.1" ){
          std::cout << "Flow : " << " (" << t.sourceAddress << " -> " << t.destinationAddress << ")  ";
          std::cout << "FP: " << i->second.txPackets << "(" << i->second.txBytes<< ") ";
          std::cout << "RP: " << i->second.rxPackets << "(" << i->second.rxBytes<< ") ";
          //std::cout << "  Lost Packets:   " << i->second.lostPackets << "\n";
          //std::cout << "  Drop Packets:   " << i->second.packetsDropped.size() << "\n";
          //std::cout << "  Packets Delivery Ratio: " << ((rxPacketsum * 100) /txPacketsum) << "%" << "\n";
          //std::cout << "  Packets Lost Ratio: " << ((LostPacketsum * 100) /txPacketsum) << "%" << "\n";
          //std::cout << "Throughput: " << i->second.rxBytes * 8.0 / 10.0 / 1024 / 1024  << " Mbps ";
          std::cout << "Delay : " << (i->second.delaySum.GetSeconds()
```

```
/ i->second.txPackets) ;
          std::cout << "  ";
          if( (i->second.txPackets - i->second.rxPackets) != 0){
            if( (i->second.txPackets - i->second.rxPackets) > packet_loss_threshold ){
              if( (i->second.delaySum.GetSeconds() / i->second.txPackets) < delay_threshold ){
                std::cout << " Wormhole Attack Detected ! \n";
              }
              else{
                std::cout<<"\n";
              }
            }
            else{
              std::cout<<"\n";
            }
          }
          else{
            std::cout<<"\n";
          }
        }
    }
        std::cout << "\n\n\n ########################## Final Conclusion ##########################" << "\n\n";
        std::cout << "  All Sent Packets: " << txPacketsum <<"
              \n" << "  All Received Packets: " << rxPacketsum << "\n";
        std::cout << "  All Lost Packets: " << LostPacketsum <<"
              \n" << "  All Drop Packets: " << DropPacketsum << "\n";
        std::cout << "  Packet Drop Ratio: " <<(double)((DropPacketsum/txPacketsum) *100 ) << "% \n";
        //std::cout << "  Packets Delivery Ratio: " << ((rxPacketsum * 100) /txPacketsum) << "%" << "          \n" << "  Packets Lost Ratio: " << ((LostPacketsum * 100) /txPacketsum) << "%" << "\n";
        std::cout << "  All Delay: " << Delaysum / txPacketsum << "\n";
        std::cout << "  Average END-TO-END delay: " << ((Delaysum / txPacketsum) / rxPacketsum) * 1000000  << "\n" ;
        std::cout << "  Throughput: " << (rxBytesum * 8.0 / 10.0 / 1024 / 1024) * 10  << " bits/s\n";




  Simulator::Destroy ();

  return 0;
}
```

Modified AODV Routing Protocol Source Code:

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * Copyright (c) 2009 IITP RAS
 *
 * This program is free software; you can redistribute it and/or modify
```

```
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation;
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307  USA
 *
 * Based on
 *      NS-2 AODV model developed by the CMU/MONARCH group and optimized and
 *      tuned by Samir Das and Mahesh Marina, University of Cincinnati;
 *
 *      AODV-UU implementation by Erik Nordström of Uppsala University
 *      http://core.it.uu.se/core/index.php/AODV-UU
 *
 * Authors: Elena Buchatskaia <borovkovaes@iitp.ru>
 *          Pavel Boyko <boyko@iitp.ru>
 */
#define NS_LOG_APPEND_CONTEXT                                   \
  if (m_ipv4) { std::clog << "[node " << m_ipv4->GetObject<Node> ()->GetId () << "] "; }

#include "aodv-routing-protocol.h"
#include "ns3/log.h"
#include "ns3/boolean.h"
#include "ns3/random-variable-stream.h"
#include "ns3/inet-socket-address.h"
#include "ns3/trace-source-accessor.h"
#include "ns3/udp-socket-factory.h"
#include "ns3/wifi-net-device.h"
#include "ns3/adhoc-wifi-mac.h"
#include "ns3/string.h"
#include "ns3/pointer.h"
#include <algorithm>
#include <limits>

using namespace std;

namespace ns3
{

NS_LOG_COMPONENT_DEFINE ("AodvRoutingProtocol");

namespace aodv
{
NS_OBJECT_ENSURE_REGISTERED (RoutingProtocol);

/// UDP Port for AODV control traffic
```

```
const uint32_t RoutingProtocol::AODV_PORT = 654;

//
-----------------------------------------------------------------------------
/// Tag used by AODV implementation

class DeferredRouteOutputTag : public Tag
{

public:
  DeferredRouteOutputTag (int32_t o = -1) : Tag (), m_oif (o) {}

  static TypeId GetTypeId ()
  {
    static TypeId tid = TypeId ("ns3::aodv::DeferredRouteOutputTag")
      .SetParent<Tag> ()
      .SetGroupName("Aodv")
      .AddConstructor<DeferredRouteOutputTag> ()
    ;  //SetGroupName is new CNLAB
    return tid;
  }

  TypeId  GetInstanceTypeId () const
  {
    return GetTypeId ();
  }

  int32_t GetInterface() const
  {
    return m_oif;
  }

  void SetInterface(int32_t oif)
  {
    m_oif = oif;
  }

  uint32_t GetSerializedSize () const
  {
    return sizeof(int32_t);
  }

  void  Serialize (TagBuffer i) const
  {
    i.WriteU32 (m_oif);
  }

  void  Deserialize (TagBuffer i)
  {
    m_oif = i.ReadU32 ();
  }

  void  Print (std::ostream &os) const
  {
    os << "DeferredRouteOutputTag: output interface = " << m_oif;
```

```
  }

private:
  /// Positive if output device is fixed in RouteOutput
  int32_t m_oif;
};

NS_OBJECT_ENSURE_REGISTERED (DeferredRouteOutputTag);


//
  -----------------------------------------------------------------------------
RoutingProtocol::RoutingProtocol () :
  RreqRetries (2),
  RreqRateLimit (10),
  RerrRateLimit (10),
  ActiveRouteTimeout (Seconds (3)),
  NetDiameter (35),
  NodeTraversalTime (MilliSeconds (40)),
  NetTraversalTime (Time ((2 * NetDiameter) * NodeTraversalTime)),
  PathDiscoveryTime ( Time (2 * NetTraversalTime)),
  MyRouteTimeout (Time (2 * std::max (PathDiscoveryTime,
    ActiveRouteTimeout))),
  HelloInterval (Seconds (1)),
  AllowedHelloLoss (2),
  DeletePeriod (Time (5 * std::max (ActiveRouteTimeout, HelloInterval)
    )),
  NextHopWait (NodeTraversalTime + MilliSeconds (10)),
  BlackListTimeout (Time (RreqRetries * NetTraversalTime)),
  MaxQueueLen (64),
  MaxQueueTime (Seconds (30)),
  DestinationOnly (false),
  GratuitousReply (true),
  EnableHello (false),
  m_routingTable (DeletePeriod),
  m_queue (MaxQueueLen, MaxQueueTime),
  m_requestId (0),
  m_seqNo (0),
  m_rreqIdCache (PathDiscoveryTime),
  m_dpd (PathDiscoveryTime),
  m_nb (HelloInterval),
  m_rreqCount (0),
  m_rerrCount (0),
  m_htimer (Timer::CANCEL_ON_DESTROY),
  m_rreqRateLimitTimer (Timer::CANCEL_ON_DESTROY),
  m_rerrRateLimitTimer (Timer::CANCEL_ON_DESTROY),
  m_lastBcastTime (Seconds (0))
{
  m_nb.SetCallback (MakeCallback (&RoutingProtocol::
    SendRerrWhenBreaksLinkToNextHop, this));
}

TypeId
RoutingProtocol::GetTypeId (void)
{
```

```
        //groupname is new
    static TypeId tid = TypeId ("ns3::aodv::RoutingProtocol")
     .SetParent<Ipv4RoutingProtocol> ()
     .SetGroupName("Aodv")
     .AddConstructor<RoutingProtocol> ()
     .AddAttribute ("HelloInterval", "HELLO messages emission interval.
    ",
                    TimeValue (Seconds (1)),
                    MakeTimeAccessor (&RoutingProtocol::HelloInterval),
                    MakeTimeChecker ())
     .AddAttribute ("RreqRetries", "Maximum number of retransmissions
    of RREQ to discover a route",
                    UintegerValue (2),
                    MakeUintegerAccessor (&RoutingProtocol::RreqRetries
    ),
                    MakeUintegerChecker<uint32_t> ())
     .AddAttribute ("RreqRateLimit", "Maximum number of RREQ per second
    .",
                    UintegerValue (10),
                    MakeUintegerAccessor (&RoutingProtocol::
    RreqRateLimit),
                    MakeUintegerChecker<uint32_t> ())
     .AddAttribute ("RerrRateLimit", "Maximum number of RERR per second
    .",
                    UintegerValue (10),
                    MakeUintegerAccessor (&RoutingProtocol::
    RerrRateLimit),
                    MakeUintegerChecker<uint32_t> ())
     .AddAttribute ("NodeTraversalTime", "Conservative estimate of the
    average one hop traversal time for packets and should include "
                    "queuing delays, interrupt processing times and
    transfer times.",
                    TimeValue (MilliSeconds (40)),
                    MakeTimeAccessor (&RoutingProtocol::
    NodeTraversalTime),
                    MakeTimeChecker ())
     .AddAttribute ("NextHopWait", "Period of our waiting for the
    neighbour's RREP_ACK = 10 ms + NodeTraversalTime",
                    TimeValue (MilliSeconds (50)),
                    MakeTimeAccessor (&RoutingProtocol::NextHopWait),
                    MakeTimeChecker ())
     .AddAttribute ("ActiveRouteTimeout", "Period of time during which
    the route is considered to be valid",
                    TimeValue (Seconds (3)),
                    MakeTimeAccessor (&RoutingProtocol::
    ActiveRouteTimeout),
                    MakeTimeChecker ())
     .AddAttribute ("MyRouteTimeout", "Value of lifetime field in RREP
    generating by this node = 2 * max(ActiveRouteTimeout,
    PathDiscoveryTime)",
                    TimeValue (Seconds (11.2)),
                    MakeTimeAccessor (&RoutingProtocol::MyRouteTimeout)
    ,
                    MakeTimeChecker ())
     .AddAttribute ("BlackListTimeout", "Time for which the node is put
     into the blacklist = RreqRetries * NetTraversalTime",
```

```
                    TimeValue (Seconds (5.6)),
                    MakeTimeAccessor (&RoutingProtocol::BlackListTimeout),
                    MakeTimeChecker ())
    .AddAttribute ("DeletePeriod", "DeletePeriod is intended to provide an upper bound on the time for which an upstream node A "
                    "can have a neighbor B as an active next hop for destination D, while B has invalidated the route to D."
                    " = 5 * max (HelloInterval, ActiveRouteTimeout)",
                    TimeValue (Seconds (15)),
                    MakeTimeAccessor (&RoutingProtocol::DeletePeriod),
                    MakeTimeChecker ())
    .AddAttribute ("NetDiameter", "Net diameter measures the maximum possible number of hops between two nodes in the network",
                    UintegerValue (35),
                    MakeUintegerAccessor (&RoutingProtocol::NetDiameter),
                    MakeUintegerChecker<uint32_t> ())
    .AddAttribute ("NetTraversalTime", "Estimate of the average net traversal time = 2 * NodeTraversalTime * NetDiameter",
                    TimeValue (Seconds (2.8)),
                    MakeTimeAccessor (&RoutingProtocol::NetTraversalTime),
                    MakeTimeChecker ())
    .AddAttribute ("PathDiscoveryTime", "Estimate of maximum time needed to find route in network = 2 * NetTraversalTime",
                    TimeValue (Seconds (5.6)),
                    MakeTimeAccessor (&RoutingProtocol::PathDiscoveryTime),
                    MakeTimeChecker ())
    .AddAttribute ("MaxQueueLen", "Maximum number of packets that we allow a routing protocol to buffer.",
                    UintegerValue (64),
                    MakeUintegerAccessor (&RoutingProtocol::SetMaxQueueLen,
                                          &RoutingProtocol::GetMaxQueueLen),
                    MakeUintegerChecker<uint32_t> ())
    .AddAttribute ("MaxQueueTime", "Maximum time packets can be queued (in seconds)",
                    TimeValue (Seconds (30)),
                    MakeTimeAccessor (&RoutingProtocol::SetMaxQueueTime,
                                      &RoutingProtocol::GetMaxQueueTime),
                    MakeTimeChecker ())
    .AddAttribute ("AllowedHelloLoss", "Number of hello messages which may be loss for valid link.",
                    UintegerValue (2),
                    MakeUintegerAccessor (&RoutingProtocol::AllowedHelloLoss),
                    MakeUintegerChecker<uint16_t> ())
    .AddAttribute ("GratuitousReply", "Indicates whether a gratuitous RREP should be unicast to the node originated route discovery.",
                    BooleanValue (true),
                    MakeBooleanAccessor (&RoutingProtocol::
```

```
SetGratuitousReplyFlag ,
                                        &RoutingProtocol::
GetGratuitousReplyFlag ),
                    MakeBooleanChecker ())
    .AddAttribute ("DestinationOnly", "Indicates only the destination
may respond to this RREQ.",
                    BooleanValue (false),
                    MakeBooleanAccessor (&RoutingProtocol::
SetDesinationOnlyFlag ,
                                        &RoutingProtocol::
GetDesinationOnlyFlag ),
                    MakeBooleanChecker ())
    .AddAttribute ("EnableHello", "Indicates whether a hello messages
enable.",
                    BooleanValue (true),
                    MakeBooleanAccessor (&RoutingProtocol::
SetHelloEnable ,
                                        &RoutingProtocol::
GetHelloEnable ),
                    MakeBooleanChecker ())
    .AddAttribute ("EnableBroadcast", "Indicates whether a broadcast
data packets forwarding enable.",
                    BooleanValue (true),
                    MakeBooleanAccessor (&RoutingProtocol::
SetBroadcastEnable ,
                                        &RoutingProtocol::
GetBroadcastEnable ),
                    MakeBooleanChecker ())
    .AddAttribute ("UniformRv",
                    "Access to the underlying UniformRandomVariable",
                    StringValue ("ns3::UniformRandomVariable"),
                    MakePointerAccessor (&RoutingProtocol::
m_uniformRandomVariable ),
                    MakePointerChecker<UniformRandomVariable> ())
    .AddAttribute ("IsMalicious", "Is the node malicious",
                    BooleanValue (false),
                    MakeBooleanAccessor (&RoutingProtocol::
SetMaliciousEnable ,
                                        &RoutingProtocol::
GetMaliciousEnable ),
                    MakeBooleanChecker ())

/*Introduction of attributes to enable the wormhole attack feature*/
//CNLAB
    .AddAttribute("EnableWrmAttack",
        "Indicates whether a Wormhole Attack is enabled or not.",
            BooleanValue(false),
            MakeBooleanAccessor (&RoutingProtocol::SetWrmAttackEnable ,
                                 &RoutingProtocol::GetWrmAttackEnable ),
                                 MakeBooleanChecker())
    .AddAttribute("FirstEndOfWormTunnel",
        "Indicates the first end of the Wormhole tunnel.",
        Ipv4AddressValue("10.1.2.1"),
        MakeIpv4AddressAccessor (&RoutingProtocol::
FirstEndOfWormTunnel ),
        MakeIpv4AddressChecker ())
```

```
      .AddAttribute ("SecondEndOfWormTunnel",
          "Indicates the second end of the Wormhole tunnel",
          Ipv4AddressValue ("10.1.2.2"),
          MakeIpv4AddressAccessor(&RoutingProtocol::
    SecondEndOfWormTunnel),
          MakeIpv4AddressChecker ())
      .AddAttribute ("FirstEndWifiWormTunnel",
          "Indicates the wifi interface of the first end of the Wormhole
     tunnel",
          Ipv4AddressValue ("10.0.1.37"),
          MakeIpv4AddressAccessor(&RoutingProtocol::
    FirstEndWifiWormTunnel),
          MakeIpv4AddressChecker ())
      .AddAttribute ("SecondEndWifiWormTunnel",
          "Indicates the wifi interface of the second end of the
    Wormhole tunnel",
          Ipv4AddressValue ("10.0.1.38"),
          MakeIpv4AddressAccessor(&RoutingProtocol::
    SecondEndWifiWormTunnel),
          MakeIpv4AddressChecker ());

  ;
  return tid;
}

void
RoutingProtocol::SetMaxQueueLen (uint32_t len)
{
  MaxQueueLen = len;
  m_queue.SetMaxQueueLen (len);
}
void
RoutingProtocol::SetMaxQueueTime (Time t)
{
  MaxQueueTime = t;
  m_queue.SetQueueTimeout (t);
}

RoutingProtocol::~RoutingProtocol ()
{
}

void
RoutingProtocol::DoDispose ()
{
  m_ipv4 = 0;
  for (std::map<Ptr<Socket>, Ipv4InterfaceAddress>::iterator iter =
         m_socketAddresses.begin (); iter != m_socketAddresses.end ();
     iter++)
    {
      iter->first->Close ();
    }
  m_socketAddresses.clear ();
        //for and broadcast added. CNLAB
  for (std::map<Ptr<Socket>, Ipv4InterfaceAddress>::iterator iter =
         m_socketSubnetBroadcastAddresses.begin (); iter !=
```

```
     m_socketSubnetBroadcastAddresses.end (); iter++)
    {
      iter->first->Close ();
    }
  m_socketSubnetBroadcastAddresses.clear ();
  Ipv4RoutingProtocol::DoDispose ();
}

void
RoutingProtocol::PrintRoutingTable (Ptr<OutputStreamWrapper> stream) const
{
  *stream->GetStream () << "Node: " << m_ipv4->GetObject<Node> ()->GetId () << " Time: " << Simulator::Now ().GetSeconds () << "s ";
  m_routingTable.Print (stream);
}

int64_t
RoutingProtocol::AssignStreams (int64_t stream)
{
  NS_LOG_FUNCTION (this << stream);
  m_uniformRandomVariable->SetStream (stream);
  return 1;
}

void
RoutingProtocol::Start ()
{
  NS_LOG_FUNCTION (this);
  if (EnableHello)
    {
      m_nb.ScheduleTimer ();
    }
  m_rreqRateLimitTimer.SetFunction (&RoutingProtocol::RreqRateLimitTimerExpire,
                                    this);
  m_rreqRateLimitTimer.Schedule (Seconds (1));

  m_rerrRateLimitTimer.SetFunction (&RoutingProtocol::RerrRateLimitTimerExpire,
                                    this);
  m_rerrRateLimitTimer.Schedule (Seconds (1));

}

Ptr<Ipv4Route>
RoutingProtocol::RouteOutput (Ptr<Packet> p, const Ipv4Header &header,
                              Ptr<NetDevice> oif, Socket::SocketErrno &sockerr)
{
  NS_LOG_FUNCTION (this << header << (oif ? oif->GetIfIndex () : 0));
  if (!p)
    {
      NS_LOG_DEBUG("Packet is == 0");
      return LoopbackRoute (header, oif); // later
    }
```

```
  if (m_socketAddresses.empty ())
    {
      sockerr = Socket::ERROR_NOROUTETOHOST;
      NS_LOG_LOGIC ("No aodv interfaces");
      Ptr<Ipv4Route> route;
      return route;
    }
  sockerr = Socket::ERROR_NOTERROR;
  Ptr<Ipv4Route> route;
  Ipv4Address dst = header.GetDestination ();
  RoutingTableEntry rt;
  if (m_routingTable.LookupValidRoute (dst, rt))
    {
      route = rt.GetRoute ();
      NS_ASSERT (route != 0);
      NS_LOG_INFO ("Exist route to " << route->GetDestination () << "
 from interface " << route->GetSource ());
      if (oif != 0 && route->GetOutputDevice () != oif)
        {
          NS_LOG_DEBUG ("Output device doesn't match. Dropped.");
          sockerr = Socket::ERROR_NOROUTETOHOST;
          return Ptr<Ipv4Route> ();
        }
      UpdateRouteLifeTime (dst, ActiveRouteTimeout);
      UpdateRouteLifeTime (route->GetGateway (), ActiveRouteTimeout);
      return route;
    }

  // Valid route not found, in this case we return loopback.
  // Actual route request will be deferred until packet will be fully
 formed,
  // routed to loopback, received from loopback and passed to
 RouteInput (see below)
  uint32_t iif = (oif ? m_ipv4->GetInterfaceForDevice (oif) : -1);
  DeferredRouteOutputTag tag (iif);
  NS_LOG_DEBUG ("Valid Route not found");
  if (!p->PeekPacketTag (tag))
    {
      p->AddPacketTag (tag);
    }
  return LoopbackRoute (header, oif);
}

void
RoutingProtocol::DeferredRouteOutput (Ptr<const Packet> p, const
 Ipv4Header & header,
                                      UnicastForwardCallback ucb,
 ErrorCallback ecb)
{
  NS_LOG_FUNCTION (this << p << header);
  NS_ASSERT (p != 0 && p != Ptr<Packet> ());

  QueueEntry newEntry (p, header, ucb, ecb);
  bool result = m_queue.Enqueue (newEntry);
  if (result)
    {
```

```
      NS_LOG_LOGIC ("Add packet " << p->GetUid () << " to queue. Protocol " << (uint16_t) header.GetProtocol ());
      RoutingTableEntry rt;
      bool result = m_routingTable.LookupRoute (header.GetDestination (), rt);
      if(!result || ((rt.GetFlag () != IN_SEARCH) && result))
        {
          NS_LOG_LOGIC ("Send new RREQ for outbound packet to " << header.GetDestination ());
          SendRequest (header.GetDestination ());
        }
    }
}

bool
RoutingProtocol::RouteInput (Ptr<const Packet> p, const Ipv4Header &header,
                             Ptr<const NetDevice> idev, UnicastForwardCallback ucb,
                             MulticastForwardCallback mcb, LocalDeliverCallback lcb, ErrorCallback ecb)
{

  NS_LOG_FUNCTION (this << p->GetUid () << header.GetDestination () << idev->GetAddress ());
  if (m_socketAddresses.empty ())
    {
      NS_LOG_LOGIC ("No aodv interfaces");
      return false;
    }
  NS_ASSERT (m_ipv4 != 0);
  NS_ASSERT (p != 0);
  // Check if input device supports IP
  NS_ASSERT (m_ipv4->GetInterfaceForDevice (idev) >= 0);
  int32_t iif = m_ipv4->GetInterfaceForDevice (idev);

  Ipv4Address dst = header.GetDestination ();
  Ipv4Address origin = header.GetSource ();

  // Deferred route request
  if (idev == m_lo)
    {
      DeferredRouteOutputTag tag;
      if (p->PeekPacketTag (tag))
        {
          DeferredRouteOutput (p, header, ucb, ecb);
          return true;
        }
    }

  // Duplicate of own packet
  if (IsMyOwnAddress (origin))
    return true;

  // AODV is not a multicast routing protocol
  if (dst.IsMulticast ())
```

```
    {
      return false;
    }

  // Broadcast local delivery/forwarding
  for (std::map<Ptr<Socket>, Ipv4InterfaceAddress>::const_iterator j =
         m_socketAddresses.begin (); j != m_socketAddresses.end (); ++j)
    {
      Ipv4InterfaceAddress iface = j->second;
      if (m_ipv4->GetInterfaceForAddress (iface.GetLocal ()) == iif)
        if (dst == iface.GetBroadcast () || dst.IsBroadcast ())
          {
            if (m_dpd.IsDuplicate (p, header))
              {
                NS_LOG_DEBUG ("Duplicated packet " << p->GetUid () << " from " << origin << ". Drop.");
                return true;
              }
            UpdateRouteLifeTime (origin, ActiveRouteTimeout);
            Ptr<Packet> packet = p->Copy ();
            if (lcb.IsNull () == false)
              {
                NS_LOG_LOGIC ("Broadcast local delivery to " << iface.GetLocal ());
                lcb (p, header, iif);
                // Fall through to additional processing
              }
            else
              {
                NS_LOG_ERROR ("Unable to deliver packet locally due to null callback " << p->GetUid () << " from " << origin);
                ecb (p, header, Socket::ERROR_NOROUTETOHOST);
              }
            if (!EnableBroadcast)
              {
                return true;
              }
            if (header.GetTtl () > 1)
              {
                NS_LOG_LOGIC ("Forward broadcast. TTL " << (uint16_t) header.GetTtl ());
                RoutingTableEntry toBroadcast;
                if (m_routingTable.LookupRoute (dst, toBroadcast))
                  {
                    Ptr<Ipv4Route> route = toBroadcast.GetRoute ();
                    ucb (route, packet, header);
                  }
                else
                  {
                    NS_LOG_DEBUG ("No route to forward broadcast. Drop packet " << p->GetUid ());
                  }
              }
            else
              {
```

```
                    NS_LOG_DEBUG ("TTL exceeded. Drop packet " << p->
      GetUid ());
                  }

              return true;
            }


      }

  // Unicast local delivery
  if (m_ipv4->IsDestinationAddress (dst, iif))
    {
      UpdateRouteLifeTime (origin, ActiveRouteTimeout);
      RoutingTableEntry toOrigin;
      if (m_routingTable.LookupValidRoute (origin, toOrigin))
        {
          UpdateRouteLifeTime (toOrigin.GetNextHop (),
      ActiveRouteTimeout);
          m_nb.Update (toOrigin.GetNextHop (), ActiveRouteTimeout);
        }

      if (lcb.IsNull () == false)
        {
          NS_LOG_INFO ("Unicast local delivery to " << dst);
          //CNLAB

          if(EnableWrmAttack)
                {


                  if(dst==FirstEndOfWormTunnel || dst==
      SecondEndOfWormTunnel)
                  {
                    iif=1;
                  }
                }
          lcb (p, header, iif);
        }
      else
        {
          NS_LOG_ERROR ("Unable to deliver packet locally due to null
      callback " << p->GetUid () << " from " << origin);
          ecb (p, header, Socket::ERROR_NOROUTETOHOST);
        }
      return true;
    }

  // Forwarding
  return Forwarding (p, header, ucb, ecb);
}

bool
RoutingProtocol::Forwarding (Ptr<const Packet> p, const Ipv4Header &
      header,
                             UnicastForwardCallback ucb, ErrorCallback
```

```
    ecb)
{
  NS_LOG_FUNCTION (this);
  Ipv4Address dst = header.GetDestination ();
  Ipv4Address origin = header.GetSource ();
  m_routingTable.Purge ();
  RoutingTableEntry toDst;
 /* Code added by Shalini Satre, Wireless Information Networking Group
    (WiNG), NITK Surathkal for simulating Blackhole Attack */
 /* Check if the node is suppose to behave maliciously */
        if(IsMalicious)
          {//When malicious node receives packet it drops the packet.
                std :: cout <<"Launching Blackhole Attack! Packet
    dropped . . . \n";
                return false;
          }
 /* Code for Blackhole attack simulation ends here */
  if (m_routingTable.LookupRoute (dst, toDst))
    {
      if (toDst.GetFlag () == VALID)
        {
          Ptr<Ipv4Route> route = toDst.GetRoute ();
          NS_LOG_LOGIC (route->GetSource ()<<" forwarding to " << dst
    << " from " << origin << " packet " << p->GetUid ());

          /*
           * Each time a route is used to forward a data packet, its
    Active Route
           * Lifetime field of the source, destination and the next
    hop on the
           * path to the destination is updated to be no less than
    the current
           * time plus ActiveRouteTimeout.
           */
          UpdateRouteLifeTime (origin, ActiveRouteTimeout);
          UpdateRouteLifeTime (dst, ActiveRouteTimeout);
          UpdateRouteLifeTime (route->GetGateway (),
    ActiveRouteTimeout);
          /*
           * Since the route between each originator and destination
    pair is expected to be symmetric, the
           * Active Route Lifetime for the previous hop, along the
    reverse path back to the IP source, is also updated
           * to be no less than the current time plus
    ActiveRouteTimeout
           */
          RoutingTableEntry toOrigin;
          m_routingTable.LookupRoute (origin, toOrigin);
          UpdateRouteLifeTime (toOrigin.GetNextHop (),
    ActiveRouteTimeout);

          m_nb.Update (route->GetGateway (), ActiveRouteTimeout);
          m_nb.Update (toOrigin.GetNextHop (), ActiveRouteTimeout);

          ucb (route, p, header);
          return true;
```

```
        }
      else
        {
          if (toDst.GetValidSeqNo ())
            {
              SendRerrWhenNoRouteToForward (dst, toDst.GetSeqNo (), origin);
              NS_LOG_DEBUG ("Drop packet " << p->GetUid () << " because no route to forward it.");
              return false;
            }
        }
    }
  NS_LOG_LOGIC ("route not found to "<< dst << ". Send RERR message.");
  NS_LOG_DEBUG ("Drop packet " << p->GetUid () << " because no route to forward it.");
  SendRerrWhenNoRouteToForward (dst, 0, origin);
  return false;
}

void
RoutingProtocol::SetIpv4 (Ptr<Ipv4> ipv4)
{
  NS_ASSERT (ipv4 != 0);
  NS_ASSERT (m_ipv4 == 0);

  //setting HELLO timer expiry missing. CNLAB

  m_ipv4 = ipv4;

  // Create lo route. It is asserted that the only one interface up for now is loopback
  NS_ASSERT (m_ipv4->GetNInterfaces () == 1 && m_ipv4->GetAddress (0, 0).GetLocal () == Ipv4Address ("127.0.0.1"));
  m_lo = m_ipv4->GetNetDevice (0);
  NS_ASSERT (m_lo != 0);
  // Remember lo route
  RoutingTableEntry rt (/*device=*/ m_lo, /*dst=*/ Ipv4Address::GetLoopback (), /*know seqno=*/ true, /*seqno=*/ 0,
                                    /*iface=*/ Ipv4InterfaceAddress (Ipv4Address::GetLoopback (), Ipv4Mask ("255.0.0.0")),
                                    /*hops=*/ 1, /*next hop=*/ Ipv4Address::GetLoopback (),
                                    /*lifetime=*/ Simulator::GetMaximumSimulationTime ());
  m_routingTable.AddRoute (rt);

  Simulator::ScheduleNow (&RoutingProtocol::Start, this);
}

void
RoutingProtocol::NotifyInterfaceUp (uint32_t i)
{
  NS_LOG_FUNCTION (this << m_ipv4->GetAddress (i, 0).GetLocal ());
  Ptr<Ipv4L3Protocol> l3 = m_ipv4->GetObject<Ipv4L3Protocol> ();
```

```
if (l3->GetNAddresses (i) > 1)
  {
    NS_LOG_WARN ("AODV does not work with more then one address per
  each interface.");
  }
Ipv4InterfaceAddress iface = l3->GetAddress (i, 0);
if (iface.GetLocal () == Ipv4Address ("127.0.0.1"))
  return;

// Create a socket to listen only on this interface
Ptr<Socket> socket = Socket::CreateSocket (GetObject<Node> (),
                                           UdpSocketFactory::
  GetTypeId ());
NS_ASSERT (socket != 0);
socket->SetRecvCallback (MakeCallback (&RoutingProtocol::RecvAodv,
  this));
socket->Bind (InetSocketAddress (Ipv4Address::GetAny (), AODV_PORT))
  ;
socket->BindToNetDevice (l3->GetNetDevice (i));
socket->SetAllowBroadcast (true);
socket->SetAttribute ("IpTtl", UintegerValue (1));
m_socketAddresses.insert (std::make_pair (socket, iface));

// create also a subnet broadcast socket -> added. CNLAB
socket = Socket::CreateSocket (GetObject<Node> (),
                               UdpSocketFactory::GetTypeId ());
NS_ASSERT (socket != 0);
socket->SetRecvCallback (MakeCallback (&RoutingProtocol::RecvAodv,
  this));
socket->Bind (InetSocketAddress (iface.GetBroadcast (), AODV_PORT));
socket->BindToNetDevice (l3->GetNetDevice (i));
socket->SetAllowBroadcast (true);
socket->SetAttribute ("IpTtl", UintegerValue (1));
m_socketSubnetBroadcastAddresses.insert (std::make_pair (socket,
  iface));

// Add local broadcast record to the routing table
Ptr<NetDevice> dev = m_ipv4->GetNetDevice (m_ipv4->
  GetInterfaceForAddress (iface.GetLocal ()));
RoutingTableEntry rt (/*device=*/ dev, /*dst=*/ iface.GetBroadcast
  (), /*know seqno=*/ true, /*seqno=*/ 0, /*iface=*/ iface,
                                  /*hops=*/ 1, /*next hop=*/ iface.
  GetBroadcast (), /*lifetime=*/ Simulator::GetMaximumSimulationTime
   ());
m_routingTable.AddRoute (rt);

//if is new. CNLAB
if (l3->GetInterface (i)->GetArpCache ())
  {
    m_nb.AddArpCache (l3->GetInterface (i)->GetArpCache ());
  }

// Allow neighbor manager use this interface for layer 2 feedback if
   possible
Ptr<WifiNetDevice> wifi = dev->GetObject<WifiNetDevice> ();
if (wifi == 0)
```

```
    return;
  Ptr<WifiMac> mac = wifi->GetMac ();
  if (mac == 0)
    return;

  mac->TraceConnectWithoutContext ("TxErrHeader", m_nb.GetTxErrorCallback ());
  //no ARP Cache. CNLAB
}

void
RoutingProtocol::NotifyInterfaceDown (uint32_t i)
{
  NS_LOG_FUNCTION (this << m_ipv4->GetAddress (i, 0).GetLocal ());

  // Disable layer 2 link state monitoring (if possible)
  Ptr<Ipv4L3Protocol> l3 = m_ipv4->GetObject<Ipv4L3Protocol> ();
  Ptr<NetDevice> dev = l3->GetNetDevice (i);
  Ptr<WifiNetDevice> wifi = dev->GetObject<WifiNetDevice> ();
  if (wifi != 0)
    {
      Ptr<WifiMac> mac = wifi->GetMac ()->GetObject<AdhocWifiMac> ();
      if (mac != 0)
        {
          mac->TraceDisconnectWithoutContext ("TxErrHeader",
                                              m_nb.GetTxErrorCallback ());
          m_nb.DelArpCache (l3->GetInterface (i)->GetArpCache ());
        }
    }

  //changed. CNLAB
  // Close socket
  Ptr<Socket> socket = FindSocketWithInterfaceAddress (m_ipv4->GetAddress (i, 0));
  NS_ASSERT (socket);
  socket->Close ();
  m_socketAddresses.erase (socket);

  // Close socket
  socket = FindSubnetBroadcastSocketWithInterfaceAddress (m_ipv4->GetAddress (i, 0));
  NS_ASSERT (socket);
  socket->Close ();
  m_socketSubnetBroadcastAddresses.erase (socket);

  if (m_socketAddresses.empty ())
    {
      NS_LOG_LOGIC ("No aodv interfaces");
      m_htimer.Cancel ();
      m_nb.Clear ();
      m_routingTable.Clear ();
      return;
    }
  m_routingTable.DeleteAllRoutesFromInterface (m_ipv4->GetAddress (i, 0));
```

```
}

void
RoutingProtocol::NotifyAddAddress (uint32_t i, Ipv4InterfaceAddress address)
{
  NS_LOG_FUNCTION (this << " interface " << i << " address " << address);
  Ptr<Ipv4L3Protocol> l3 = m_ipv4->GetObject<Ipv4L3Protocol> ();
  if (!l3->IsUp (i))
    return;
  if (l3->GetNAddresses (i) == 1)
    {
      Ipv4InterfaceAddress iface = l3->GetAddress (i, 0);
      Ptr<Socket> socket = FindSocketWithInterfaceAddress (iface);
      if (!socket)
        {
          if (iface.GetLocal () == Ipv4Address ("127.0.0.1"))
            return;
          // Create a socket to listen only on this interface
          Ptr<Socket> socket = Socket::CreateSocket (GetObject<Node> (),
                                                     UdpSocketFactory::GetTypeId ());
          NS_ASSERT (socket != 0);
          socket->SetRecvCallback (MakeCallback (&RoutingProtocol::RecvAodv,this));
          socket->Bind (InetSocketAddress (iface.GetLocal (), AODV_PORT));
          socket->BindToNetDevice (l3->GetNetDevice (i));
          socket->SetAllowBroadcast (true);
          m_socketAddresses.insert (std::make_pair (socket, iface));

          // create also a subnet directed broadcast socket. Added CNLAB
          socket = Socket::CreateSocket (GetObject<Node> (),
                                         UdpSocketFactory::GetTypeId ());
          NS_ASSERT (socket != 0);
          socket->SetRecvCallback (MakeCallback (&RoutingProtocol::RecvAodv, this));
          socket->Bind (InetSocketAddress (iface.GetBroadcast (), AODV_PORT));
          socket->BindToNetDevice (l3->GetNetDevice (i));
          socket->SetAllowBroadcast (true);
          socket->SetAttribute ("IpTtl", UintegerValue (1));
          m_socketSubnetBroadcastAddresses.insert (std::make_pair (socket, iface));

          // Add local broadcast record to the routing table
          Ptr<NetDevice> dev = m_ipv4->GetNetDevice (
              m_ipv4->GetInterfaceForAddress (iface.GetLocal ()));
          RoutingTableEntry rt (/*device=*/ dev, /*dst=*/ iface.GetBroadcast (), /*know seqno=*/ true,
                                /*seqno=*/ 0, /*iface=*/ iface, /*hops=*/ 1,
```

```
                                              /*next hop=*/ iface.GetBroadcast (), /*lifetime=*/ Simulator::GetMaximumSimulationTime ());
          m_routingTable.AddRoute (rt);
        }
    }
  else
    {
      NS_LOG_LOGIC ("AODV does not work with more then one address per each interface. Ignore added address");
    }
}

void
RoutingProtocol::NotifyRemoveAddress (uint32_t i, Ipv4InterfaceAddress address)
{
  NS_LOG_FUNCTION (this);
  Ptr<Socket> socket = FindSocketWithInterfaceAddress (address);
  if (socket)
    {
      m_routingTable.DeleteAllRoutesFromInterface (address);
      socket->Close ();
      m_socketAddresses.erase (socket);
      //added: CNLAB
      Ptr<Socket> unicastSocket = FindSubnetBroadcastSocketWithInterfaceAddress (address);
      if (unicastSocket)
        {
          unicastSocket->Close ();
          m_socketAddresses.erase (unicastSocket);
        }

      Ptr<Ipv4L3Protocol> l3 = m_ipv4->GetObject<Ipv4L3Protocol> ();
      if (l3->GetNAddresses (i))
        {
          Ipv4InterfaceAddress iface = l3->GetAddress (i, 0);
          // Create a socket to listen only on this interface
          Ptr<Socket> socket = Socket::CreateSocket (GetObject<Node> (),
                                                     UdpSocketFactory::GetTypeId ());
          NS_ASSERT (socket != 0);
          socket->SetRecvCallback (MakeCallback (&RoutingProtocol::RecvAodv, this));
          // Bind to any IP address so that broadcasts can be received. TTL Added. CNLAB
          socket->Bind (InetSocketAddress (iface.GetLocal (), AODV_PORT));
          socket->BindToNetDevice (l3->GetNetDevice (i));
          socket->SetAllowBroadcast (true);
          socket->SetAttribute ("IpTtl", UintegerValue (1));
          m_socketAddresses.insert (std::make_pair (socket, iface));

          // create also a unicast socket. Added. CNLAB.
          socket = Socket::CreateSocket (GetObject<Node> (),
```

```
      UdpSocketFactory::GetTypeId ());
          NS_ASSERT (socket != 0);
          socket->SetRecvCallback (MakeCallback (&RoutingProtocol::RecvAodv, this));
          socket->Bind (InetSocketAddress (iface.GetBroadcast (), AODV_PORT));
          socket->BindToNetDevice (l3->GetNetDevice (i));
          socket->SetAllowBroadcast (true);
          socket->SetAttribute ("IpTtl", UintegerValue (1));
          m_socketSubnetBroadcastAddresses.insert (std::make_pair (socket, iface));

          // Add local broadcast record to the routing table
          Ptr<NetDevice> dev = m_ipv4->GetNetDevice (m_ipv4->GetInterfaceForAddress (iface.GetLocal ()));
          RoutingTableEntry rt (/*device=*/ dev, /*dst=*/ iface.GetBroadcast (), /*know seqno=*/ true, /*seqno=*/ 0, /*iface=*/ iface,
                                                    /*hops=*/ 1, /*next hop=*/ iface.GetBroadcast (), /*lifetime=*/ Simulator::GetMaximumSimulationTime ());
          m_routingTable.AddRoute (rt);
        }
      if (m_socketAddresses.empty ())
        {
          NS_LOG_LOGIC ("No aodv interfaces");
          m_htimer.Cancel ();
          m_nb.Clear ();
          m_routingTable.Clear ();
          return;
        }
    }
  else
    {
      NS_LOG_LOGIC ("Remove address not participating in AODV operation");
    }
}

bool
RoutingProtocol::IsMyOwnAddress (Ipv4Address src)
{
  NS_LOG_FUNCTION (this << src);
  for (std::map<Ptr<Socket>, Ipv4InterfaceAddress>::const_iterator j =
         m_socketAddresses.begin (); j != m_socketAddresses.end (); ++j)
    {
      Ipv4InterfaceAddress iface = j->second;
      if (src == iface.GetLocal ())
        {
          return true;
        }
    }
  return false;
}
```

```
Ptr<Ipv4Route>
RoutingProtocol::LoopbackRoute (const Ipv4Header & hdr, Ptr<NetDevice> oif) const
{
  NS_LOG_FUNCTION (this << hdr);
  NS_ASSERT (m_lo != 0);
  Ptr<Ipv4Route> rt = Create<Ipv4Route> ();
  rt->SetDestination (hdr.GetDestination ());
  //
  // Source address selection here is tricky.  The loopback route is
  // returned when AODV does not have a route; this causes the packet
  // to be looped back and handled (cached) in RouteInput() method
  // while a route is found. However, connection-oriented protocols
  // like TCP need to create an endpoint four-tuple (src, src port,
  // dst, dst port) and create a pseudo-header for checksumming.  So,
  // AODV needs to guess correctly what the eventual source address
  // will be.
  //
  // For single interface, single address nodes, this is not a problem.
  // When there are possibly multiple outgoing interfaces, the policy
  // implemented here is to pick the first available AODV interface.
  // If RouteOutput() caller specified an outgoing interface, that
  // further constrains the selection of source address
  //
  std::map<Ptr<Socket>, Ipv4InterfaceAddress>::const_iterator j = m_socketAddresses.begin ();
  if (oif)
    {
      // Iterate to find an address on the oif device
      for (j = m_socketAddresses.begin (); j != m_socketAddresses.end (); ++j)
        {
          Ipv4Address addr = j->second.GetLocal ();
          int32_t interface = m_ipv4->GetInterfaceForAddress (addr);
          if (oif == m_ipv4->GetNetDevice (static_cast<uint32_t> (interface)))
            {
              rt->SetSource (addr);
              break;
            }
        }
    }
  else
    {
      rt->SetSource (j->second.GetLocal ());
    }
  NS_ASSERT_MSG (rt->GetSource () != Ipv4Address (), "Valid AODV source address not found");
  rt->SetGateway (Ipv4Address ("127.0.0.1"));
  rt->SetOutputDevice (m_lo);
  return rt;
}

void
```

```
RoutingProtocol::SendRequest (Ipv4Address dst)
{
  NS_LOG_FUNCTION ( this << dst);
  // A node SHOULD NOT originate more than RREQ_RATELIMIT RREQ messages per second.
  if (m_rreqCount == RreqRateLimit)
    {
      Simulator::Schedule (m_rreqRateLimitTimer.GetDelayLeft () + MicroSeconds (100),
                           &RoutingProtocol::SendRequest, this, dst);
      return;
    }
  else
    m_rreqCount++;
  // Create RREQ header
  RreqHeader rreqHeader;
  rreqHeader.SetDst (dst);

  RoutingTableEntry rt;
  if (m_routingTable.LookupRoute (dst, rt))
    {
      rreqHeader.SetHopCount (rt.GetHop ());
      if (rt.GetValidSeqNo ())
        rreqHeader.SetDstSeqno (rt.GetSeqNo ());
      else
        rreqHeader.SetUnknownSeqno (true);
      rt.SetFlag (IN_SEARCH);
      m_routingTable.Update (rt);
    }
  else
    {
      rreqHeader.SetUnknownSeqno (true);
      Ptr<NetDevice> dev = 0;
      RoutingTableEntry newEntry (/*device=*/ dev, /*dst=*/ dst, /*validSeqNo=*/ false, /*seqno=*/ 0,
                                  /*iface=*/ Ipv4InterfaceAddress (),/*hop=*/ 0,
                                  /*nextHop=*/ Ipv4Address (), /*lifeTime=*/ Seconds (0));
      newEntry.SetFlag (IN_SEARCH);
      m_routingTable.AddRoute (newEntry);
    }

  if (GratuitousReply)
    rreqHeader.SetGratiousRrep (true);
  if (DestinationOnly)
    rreqHeader.SetDestinationOnly (true);

  m_seqNo++;
  rreqHeader.SetOriginSeqno (m_seqNo);
  m_requestId++;
  rreqHeader.SetId (m_requestId);
  rreqHeader.SetHopCount (0);

  // Send RREQ as subnet directed broadcast from each interface used by aodv
```

```
  for (std::map<Ptr<Socket>, Ipv4InterfaceAddress>::const_iterator j =
         m_socketAddresses.begin (); j != m_socketAddresses.end (); ++
    j)
    {
      Ptr<Socket> socket = j->first;
      Ipv4InterfaceAddress iface = j->second;

      rreqHeader.SetOrigin (iface.GetLocal ());
      m_rreqIdCache.IsDuplicate (iface.GetLocal (), m_requestId);

      Ptr<Packet> packet = Create<Packet> ();
      packet->AddHeader (rreqHeader);
      TypeHeader tHeader (AODVTYPE_RREQ);
      packet->AddHeader (tHeader);
      // Send to all-hosts broadcast if on /32 addr, subnet-directed
    otherwise
      Ipv4Address destination;
      if (iface.GetMask () == Ipv4Mask::GetOnes ())
        {
          destination = Ipv4Address ("255.255.255.255");
        }
      else
        {
          destination = iface.GetBroadcast ();
        }
      NS_LOG_DEBUG ("Send RREQ with id " << rreqHeader.GetId () << "
    to socket");
      m_lastBcastTime = Simulator::Now ();
      Simulator::Schedule (Time (MilliSeconds (m_uniformRandomVariable
    ->GetInteger (0, 10))), &RoutingProtocol::SendTo, this, socket,
    packet, destination);
    }
  ScheduleRreqRetry (dst);
}

void
RoutingProtocol::SendTo (Ptr<Socket> socket, Ptr<Packet> packet,
    Ipv4Address destination)
{
    socket->SendTo (packet, 0, InetSocketAddress (destination,
    AODV_PORT));

}
void
RoutingProtocol::ScheduleRreqRetry (Ipv4Address dst)
{
  NS_LOG_FUNCTION (this << dst);
  if (m_addressReqTimer.find (dst) == m_addressReqTimer.end ())
    {
      Timer timer (Timer::CANCEL_ON_DESTROY);
      m_addressReqTimer[dst] = timer;
    }
  m_addressReqTimer[dst].SetFunction (&RoutingProtocol::
    RouteRequestTimerExpire, this);
  m_addressReqTimer[dst].Remove ();
  m_addressReqTimer[dst].SetArguments (dst);
```

```
  RoutingTableEntry rt;
  m_routingTable.LookupRoute (dst, rt);
  rt.IncrementRreqCnt ();
  m_routingTable.Update (rt);
  m_addressReqTimer[dst].Schedule (Time (rt.GetRreqCnt () *
   NetTraversalTime));
  NS_LOG_LOGIC ("Scheduled RREQ retry in " << Time (rt.GetRreqCnt () *
     NetTraversalTime).GetSeconds () << " seconds");
}

void
RoutingProtocol::RecvAodv (Ptr<Socket> socket)
{
  NS_LOG_FUNCTION (this << socket);
  Address sourceAddress;
  Ptr<Packet> packet = socket->RecvFrom (sourceAddress);
  InetSocketAddress inetSourceAddr = InetSocketAddress::ConvertFrom (
   sourceAddress);
  Ipv4Address sender = inetSourceAddr.GetIpv4 ();
  Ipv4Address receiver;
  //added till debug. CNLAB
  if (m_socketAddresses.find (socket) != m_socketAddresses.end ())
    {
      receiver = m_socketAddresses[socket].GetLocal ();
    }
  else if(m_socketSubnetBroadcastAddresses.find (socket) !=
   m_socketSubnetBroadcastAddresses.end ())
    {
      receiver = m_socketSubnetBroadcastAddresses[socket].GetLocal ();
    }
  else
    {
      NS_ASSERT_MSG (false, "Received a packet from an unknown socket"
   );
    }
  NS_LOG_INFO ("AODV node " << this << " received a AODV packet from "
     << sender << " to " << receiver);

  if(EnableWrmAttack) //CNLAB
    {
    // cout <<endl<<"Received AODV Packet at Wormhole Node"<<endl;
    // cout<<"Sender IP Address-"<<sender<<endl;
    // cout<<"First End of Wormhole Tunnel"<<FirstEndWifiWormTunnel<<
   endl;
    // cout<<"Receiver IP Address-"<<receiver<<endl;
    // cout<<"Second End of Wifi Tunnel"<<SecondEndWifiWormTunnel;

     if(sender==FirstEndOfWormTunnel && receiver==
   SecondEndWifiWormTunnel)
       {
       // cout<<"Received by Second Wifi Wrm Tunnel"<<endl;
          receiver=SecondEndOfWormTunnel;
       }
     if(sender==SecondEndOfWormTunnel && receiver==
   FirstEndWifiWormTunnel)
       {
```

```
        // cout<<"Received by First Wifi Wm Tunnel"<<endl;
          receiver=FirstEndOfWormTunnel;
        }
      }

  UpdateRouteToNeighbor (sender, receiver);
  TypeHeader tHeader (AODVTYPE_RREQ);
  packet->RemoveHeader (tHeader);
  if (!tHeader.IsValid ())
    {
      NS_LOG_DEBUG ("AODV message " << packet->GetUid () << " with
   unknown type received: " << tHeader.Get () << ". Drop");
      return; // drop
    }
  switch (tHeader.Get ())
    {
    case AODVTYPE_RREQ:
      {
        RecvRequest (packet, receiver, sender);
        break;
      }
    case AODVTYPE_RREP:
      {
        RecvReply (packet, receiver, sender);
        break;
      }
    case AODVTYPE_RERR:
      {
        RecvError (packet, sender);
        break;
      }
    case AODVTYPE_RREP_ACK:
      {
        RecvReplyAck (sender);
        break;
      }
    }
}

bool
RoutingProtocol::UpdateRouteLifeTime (Ipv4Address addr, Time lifetime)
{
  NS_LOG_FUNCTION (this << addr << lifetime);
  RoutingTableEntry rt;
  if (m_routingTable.LookupRoute (addr, rt))
    {
      if (rt.GetFlag () == VALID)
        {
          NS_LOG_DEBUG ("Updating VALID route");
          rt.SetRreqCnt (0);
          rt.SetLifeTime (std::max (lifetime, rt.GetLifeTime ()));
          m_routingTable.Update (rt);
          return true;
        }
    }
  return false;
```

```
}

void
RoutingProtocol::UpdateRouteToNeighbor (Ipv4Address sender, Ipv4Address receiver)
{
  NS_LOG_FUNCTION (this << "sender " << sender << " receiver " << receiver);
  RoutingTableEntry toNeighbor;
  if (!m_routingTable.LookupRoute (sender, toNeighbor))
    {
      Ptr<NetDevice> dev = m_ipv4->GetNetDevice (m_ipv4->GetInterfaceForAddress (receiver));
      RoutingTableEntry newEntry (/*device=*/ dev, /*dst=*/ sender, /*know seqno=*/ false, /*seqno=*/ 0,
                                              /*iface=*/ m_ipv4->GetAddress (m_ipv4->GetInterfaceForAddress (receiver), 0),
                                              /*hops=*/ 1, /*next hop=*/ sender, /*lifetime=*/ ActiveRouteTimeout);
      m_routingTable.AddRoute (newEntry);
    }
  else
    {
      Ptr<NetDevice> dev = m_ipv4->GetNetDevice (m_ipv4->GetInterfaceForAddress (receiver));
      if (toNeighbor.GetValidSeqNo () && (toNeighbor.GetHop () == 1) && (toNeighbor.GetOutputDevice () == dev))
        {
          toNeighbor.SetLifeTime (std::max (ActiveRouteTimeout, toNeighbor.GetLifeTime ()));
        }
      else
        {
          RoutingTableEntry newEntry (/*device=*/ dev, /*dst=*/ sender, /*know seqno=*/ false, /*seqno=*/ 0,
                                                  /*iface=*/ m_ipv4->GetAddress (m_ipv4->GetInterfaceForAddress (receiver), 0),
                                                  /*hops=*/ 1, /*next hop=*/ sender, /*lifetime=*/ std::max (ActiveRouteTimeout, toNeighbor.GetLifeTime ()));
          m_routingTable.Update (newEntry);
        }
    }

}

void
RoutingProtocol::RecvRequest (Ptr<Packet> p, Ipv4Address receiver, Ipv4Address src)
{
  NS_LOG_FUNCTION (this);
  RreqHeader rreqHeader;
  p->RemoveHeader (rreqHeader);

  // A node ignores all RREQs received from any node in its blacklist
  RoutingTableEntry toPrev;
```

```
if (m_routingTable.LookupRoute (src, toPrev))
  {
    if (toPrev.IsUnidirectional ())
      {
        NS_LOG_DEBUG ("Ignoring RREQ from node in blacklist");
        return;
      }
  }

uint32_t id = rreqHeader.GetId ();
Ipv4Address origin = rreqHeader.GetOrigin ();

/*
 *  Node checks to determine whether it has received a RREQ with the same Originator IP Address and RREQ ID.
 *  If such a RREQ has been received, the node silently discards the newly received RREQ.
 */
if (m_rreqIdCache.IsDuplicate (origin, id))
  {
    NS_LOG_DEBUG ("Ignoring RREQ due to duplicate");
    return;
  }

// Increment RREQ hop count
uint8_t hop = rreqHeader.GetHopCount () + 1;
rreqHeader.SetHopCount (hop);

/*
 *  When the reverse route is created or updated, the following actions on the route are also carried out:
 *  1. the Originator Sequence Number from the RREQ is compared to the corresponding destination sequence number
 *     in the route table entry and copied if greater than the existing value there
 *  2. the valid sequence number field is set to true;
 *  3. the next hop in the routing table becomes the node from which the  RREQ was received
 *  4. the hop count is copied from the Hop Count in the RREQ message;
 *  5. the Lifetime is set to be the maximum of (ExistingLifetime, MinimalLifetime), where
 *     MinimalLifetime = current time + 2*NetTraversalTime - 2*HopCount*NodeTraversalTime
 */
RoutingTableEntry toOrigin;
if (!m_routingTable.LookupRoute (origin, toOrigin))
  {
    Ptr<NetDevice> dev;

    //CNLAB
    if(EnableWrmAttack && (src==FirstEndOfWormTunnel))
        {
      // cout<<"ENTER IN THE ATTACK WRM2";
        dev = m_ipv4->GetNetDevice (m_ipv4->GetInterfaceForAddress (SecondEndOfWormTunnel));
```

```
          receiver=SecondEndOfWormTunnel;
          }
      else if(EnableWrmAttack && (src==SecondEndOfWormTunnel))
          {
        // cout<<"ENTER IN THE ATTACK WRM1";
          dev = m_ipv4->GetNetDevice (m_ipv4->GetInterfaceForAddress (FirstEndOfWormTunnel));
          receiver=FirstEndOfWormTunnel;
          }
          else
          dev = m_ipv4->GetNetDevice (m_ipv4->GetInterfaceForAddress (receiver));

      RoutingTableEntry newEntry (/*device=*/ dev, /*dst=*/ origin, /*validSeno=*/ true, /*seqNo=*/ rreqHeader.GetOriginSeqno (),
                                              /*iface=*/ m_ipv4->GetAddress (m_ipv4->GetInterfaceForAddress (receiver), 0), /*hops=*/ hop,
                                              /*nextHop*/ src, /*timeLife=*/ Time ((2 * NetTraversalTime - 2 * hop * NodeTraversalTime)));
      m_routingTable.AddRoute (newEntry);
    }
  else
    {
      if (toOrigin.GetValidSeqNo ())
        {
          if (int32_t (rreqHeader.GetOriginSeqno ()) - int32_t (toOrigin.GetSeqNo ()) > 0)
            toOrigin.SetSeqNo (rreqHeader.GetOriginSeqno ());
        }
      else
        toOrigin.SetSeqNo (rreqHeader.GetOriginSeqno ());
      toOrigin.SetValidSeqNo (true);
      toOrigin.SetNextHop (src);
      toOrigin.SetOutputDevice (m_ipv4->GetNetDevice (m_ipv4->GetInterfaceForAddress (receiver)));
      toOrigin.SetInterface (m_ipv4->GetAddress (m_ipv4->GetInterfaceForAddress (receiver), 0));
      toOrigin.SetHop (hop);
      toOrigin.SetLifeTime (std::max (Time (2 * NetTraversalTime - 2 * hop * NodeTraversalTime),
                                      toOrigin.GetLifeTime ()));
      m_routingTable.Update (toOrigin);
      //m_nb.Update (src, Time (AllowedHelloLoss * HelloInterval));
    }


  RoutingTableEntry toNeighbor;
  if (!m_routingTable.LookupRoute (src, toNeighbor))
    {
      NS_LOG_DEBUG ("Neighbor:" << src << " not found in routing table. Creating an entry");
      Ptr<NetDevice> dev = m_ipv4->GetNetDevice (m_ipv4->GetInterfaceForAddress (receiver));
      RoutingTableEntry newEntry (dev, src, false, rreqHeader.
```

```
GetOriginSeqno (),
                                                m_ipv4->GetAddress (
m_ipv4->GetInterfaceForAddress (receiver), 0),
                                                1, src,
ActiveRouteTimeout);
      m_routingTable.AddRoute (newEntry);
    }
  else
    {
      //sethops added. CNLAB
      toNeighbor.SetLifeTime (ActiveRouteTimeout);
      toNeighbor.SetValidSeqNo (false);
      toNeighbor.SetSeqNo (rreqHeader.GetOriginSeqno ());
      toNeighbor.SetFlag (VALID);
      toNeighbor.SetOutputDevice (m_ipv4->GetNetDevice (m_ipv4->
GetInterfaceForAddress (receiver)));
      toNeighbor.SetInterface (m_ipv4->GetAddress (m_ipv4->
GetInterfaceForAddress (receiver), 0));
      toNeighbor.SetHop (1);
      toNeighbor.SetNextHop (src);
      m_routingTable.Update (toNeighbor);
    }
  m_nb.Update (src, Time (AllowedHelloLoss * HelloInterval));

  NS_LOG_LOGIC (receiver << " receive RREQ with hop count " <<
static_cast<uint32_t>(rreqHeader.GetHopCount ())
                          << " ID " << rreqHeader.GetId ()
                          << " to destination " << rreqHeader.GetDst ()
);

  //  A node generates a RREP if either:
  //  (i)  it is itself the destination,
  if (IsMyOwnAddress (rreqHeader.GetDst ()))
    {
      m_routingTable.LookupRoute (origin, toOrigin);
      NS_LOG_DEBUG ("Send reply since I am the destination");
      SendReply (rreqHeader, toOrigin);
      return;
    }

  /*
   * (ii) or it has an active route to the destination, the
destination sequence number in the node's existing route table
entry for the destination
   *      is valid and greater than or equal to the Destination
Sequence Number of the RREQ, and the "destination only" flag is
NOT set.
   */
  RoutingTableEntry toDst;
  Ipv4Address dst = rreqHeader.GetDst ();

  if (IsMalicious || m_routingTable.LookupRoute (dst, toDst))
    {
      /*
       * Drop RREQ, This node RREP wil make a loop.
       */
```

```
    if (toDst.GetNextHop () == src)
      {
        NS_LOG_DEBUG ("Drop RREQ from " << src << ", dest next hop " << toDst.GetNextHop ());
        return;
      }
    /*
     * The Destination Sequence number for the requested destination is set to the maximum of the corresponding value
     * received in the RREQ message, and the destination sequence value currently maintained by the node for the requested destination.
     * However, the forwarding node MUST NOT modify its maintained value for the destination sequence number, even if the value
     * received in the incoming RREQ is larger than the value currently maintained by the forwarding node.
     */
    if (IsMalicious || ((rreqHeader.GetUnknownSeqno () || (int32_t (toDst.GetSeqNo ()) - int32_t (rreqHeader.GetDstSeqno ()) >= 0))
        && toDst.GetValidSeqNo ()) )
      {
        if (IsMalicious || (!rreqHeader.GetDestinationOnly () && toDst.GetFlag () == VALID))
          {
            m_routingTable.LookupRoute (origin, toOrigin);
            /* Code added by Shalini Satre, Wireless Information Networking Group (WiNG), NITK Surathkal for simulating Blackhole Attack
             * If node is malicious, it creates false routing table entry having sequence number much higher than
             * that in RREQ message and hop count as 1.
             * Malicious node itself sends the RREP message,
             * so that the route will be established through malicious node.
             */
            if(IsMalicious)
            {
              Ptr<NetDevice> dev = m_ipv4->GetNetDevice (m_ipv4->GetInterfaceForAddress (receiver));
              RoutingTableEntry falseToDst(dev,dst,true,rreqHeader.GetDstSeqno()+100,m_ipv4->GetAddress (m_ipv4->GetInterfaceForAddress (receiver),0),1,dst,ActiveRouteTimeout);

              SendReplyByIntermediateNode (falseToDst, toOrigin, rreqHeader.GetGratiousRrep ());
              return;
            }
            /* Code for Blackhole Attack Simulation ends here */
            SendReplyByIntermediateNode (toDst, toOrigin, rreqHeader.GetGratiousRrep ());
            return;
          }
        rreqHeader.SetDstSeqno (toDst.GetSeqNo ());
        rreqHeader.SetUnknownSeqno (false);
      }
```

```
    }

  for (std::map<Ptr<Socket>, Ipv4InterfaceAddress>::const_iterator j =
         m_socketAddresses.begin (); j != m_socketAddresses.end (); ++
     j)
    {
      Ptr<Socket> socket = j->first;
      Ipv4InterfaceAddress iface = j->second;
      Ptr<Packet> packet = Create<Packet> ();
      packet->AddHeader (rreqHeader);
      TypeHeader tHeader (AODVTYPE_RREQ);
      packet->AddHeader (tHeader);
      // Send to all-hosts broadcast if on /32 addr, subnet-directed
     otherwise
      Ipv4Address destination;
      if (iface.GetMask () == Ipv4Mask::GetOnes ())
        {
          destination = Ipv4Address ("255.255.255.255");
        }
      else
        {
          destination = iface.GetBroadcast ();
        }
      m_lastBcastTime = Simulator::Now ();
      Simulator::Schedule (Time (MilliSeconds (m_uniformRandomVariable
     ->GetInteger (0, 10))), &RoutingProtocol::SendTo, this, socket,
     packet, destination);

    }
}

void
RoutingProtocol::SendReply (RreqHeader const & rreqHeader,
     RoutingTableEntry const & toOrigin)
{
  NS_LOG_FUNCTION (this << toOrigin.GetDestination ());
  /*
   * Destination node MUST increment its own sequence number by one if
     the sequence number in the RREQ packet is equal to that
   * incremented value. Otherwise, the destination does not change its
     sequence number before generating the RREP message.
   */
  if (!rreqHeader.GetUnknownSeqno () && (rreqHeader.GetDstSeqno () ==
     m_seqNo + 1))
    m_seqNo++;
  RrepHeader rrepHeader ( /*prefixSize=*/ 0, /*hops=*/ 0, /*dst=*/
     rreqHeader.GetDst (),
                                              /*dstSeqNo=*/ m_seqNo, /*
     origin=*/ toOrigin.GetDestination (), /*lifeTime=*/ MyRouteTimeout
     );
  Ptr<Packet> packet = Create<Packet> ();
  packet->AddHeader (rrepHeader);
  TypeHeader tHeader (AODVTYPE_RREP);
  packet->AddHeader (tHeader);
  Ptr<Socket> socket = FindSocketWithInterfaceAddress (toOrigin.
     GetInterface ());
```

```
  NS_ASSERT (socket);
  socket->SendTo (packet, 0, InetSocketAddress (toOrigin.GetNextHop (), AODV_PORT));
}

void
RoutingProtocol::SendReplyByIntermediateNode (RoutingTableEntry & toDst, RoutingTableEntry & toOrigin, bool gratRep)
{
  NS_LOG_FUNCTION (this);
  RrepHeader rrepHeader (/*prefix size=*/ 0, /*hops=*/ toDst.GetHop (), /*dst=*/ toDst.GetDestination (), /*dst seqno=*/ toDst.GetSeqNo (),
                                              /*origin=*/ toOrigin.GetDestination (), /*lifetime=*/ toDst.GetLifeTime ());
  /* If the node we received a RREQ for is a neighbor we are
   * probably facing a unidirectional link... Better request a RREP-ack
   */

  ///Attract node to set up path through malicious node

  if(IsMalicious)                          //Shalini Satre
  {
     rrepHeader.SetHopCount(1);
  }

  if (toDst.GetHop () == 1 )
    {
      rrepHeader.SetAckRequired (true);
      RoutingTableEntry toNextHop;
      m_routingTable.LookupRoute (toOrigin.GetNextHop (), toNextHop);
      toNextHop.m_ackTimer.SetFunction (&RoutingProtocol::AckTimerExpire, this);
      toNextHop.m_ackTimer.SetArguments (toNextHop.GetDestination (), BlackListTimeout);
      toNextHop.m_ackTimer.SetDelay (NextHopWait);
    }
  toDst.InsertPrecursor (toOrigin.GetNextHop ());
  toOrigin.InsertPrecursor (toDst.GetNextHop ());
  m_routingTable.Update (toDst);
  m_routingTable.Update (toOrigin);

  Ptr<Packet> packet = Create<Packet> ();
  packet->AddHeader (rrepHeader);
  TypeHeader tHeader (AODVTYPE_RREP);
  packet->AddHeader (tHeader);
  Ptr<Socket> socket = FindSocketWithInterfaceAddress (toOrigin.GetInterface ());
  NS_ASSERT (socket);
  socket->SendTo (packet, 0, InetSocketAddress (toOrigin.GetNextHop (), AODV_PORT));

  // Generating gratuitous RREPs
  if (gratRep)
    {
```

```
        RrepHeader gratRepHeader (/*prefix size=*/ 0, /*hops=*/ toOrigin
     .GetHop (), /*dst=*/ toOrigin.GetDestination (),
                                                        /*dst seqno=*/
     toOrigin.GetSeqNo (), /*origin=*/ toDst.GetDestination (),
                                                        /*lifetime=*/
     toOrigin.GetLifeTime ());
        Ptr<Packet> packetToDst = Create<Packet> ();
        packetToDst->AddHeader (gratRepHeader);
        TypeHeader type (AODVTYPE_RREP);
        packetToDst->AddHeader (type);
        Ptr<Socket> socket = FindSocketWithInterfaceAddress (toDst.
     GetInterface ());
        NS_ASSERT (socket);
        NS_LOG_LOGIC ("Send gratuitous RREP " << packet->GetUid ());
        socket->SendTo (packetToDst, 0, InetSocketAddress (toDst.
     GetNextHop (), AODV_PORT));
      }
}

void
RoutingProtocol::SendReplyAck (Ipv4Address neighbor)
{
  NS_LOG_FUNCTION (this << " to " << neighbor);
  RrepAckHeader h;
  TypeHeader typeHeader (AODVTYPE_RREP_ACK);
  Ptr<Packet> packet = Create<Packet> ();
  packet->AddHeader (h);
  packet->AddHeader (typeHeader);
  RoutingTableEntry toNeighbor;
  m_routingTable.LookupRoute (neighbor, toNeighbor);
  Ptr<Socket> socket = FindSocketWithInterfaceAddress (toNeighbor.
    GetInterface ());
  NS_ASSERT (socket);
  socket->SendTo (packet, 0, InetSocketAddress (neighbor, AODV_PORT));
}

void
RoutingProtocol::RecvReply (Ptr<Packet> p, Ipv4Address receiver,
    Ipv4Address sender)
{
  NS_LOG_FUNCTION (this << " src " << sender);
  RrepHeader rrepHeader;
  p->RemoveHeader (rrepHeader);
  Ipv4Address dst = rrepHeader.GetDst ();
  NS_LOG_LOGIC ("RREP destination " << dst << " RREP origin " <<
    rrepHeader.GetOrigin ());

  uint8_t hop = rrepHeader.GetHopCount () + 1;
  rrepHeader.SetHopCount (hop);

  // If RREP is Hello message
  if (dst == rrepHeader.GetOrigin ())
    {
      ProcessHello (rrepHeader, receiver);
      return;
    }
```

```
  /*
   * If the route table entry to the destination is created or updated, then the following actions occur:
   * -  the route is marked as active,
   * -  the destination sequence number is marked as valid,
   * -  the next hop in the route entry is assigned to be the node from which the RREP is received,
   *    which is indicated by the source IP address field in the IP header,
   * -  the hop count is set to the value of the hop count from RREP message + 1
   * -  the expiry time is set to the current time plus the value of the Lifetime in the RREP message,
   * -  and the destination sequence number is the Destination Sequence Number in the RREP message.
   */
  Ptr<NetDevice> dev = m_ipv4->GetNetDevice (m_ipv4->GetInterfaceForAddress (receiver));
  RoutingTableEntry newEntry (/*device=*/ dev, /*dst=*/ dst, /*validSeqNo=*/ true, /*seqno=*/ rrepHeader.GetDstSeqno (),
                                          /*iface=*/ m_ipv4->GetAddress (m_ipv4->GetInterfaceForAddress (receiver), 0),/*hop=*/ hop,
                                          /*nextHop=*/ sender, /*lifeTime=*/ rrepHeader.GetLifeTime ());
  RoutingTableEntry toDst;
  if (m_routingTable.LookupRoute (dst, toDst))
    {
      /*
       * The existing entry is updated only in the following circumstances:
       * (i) the sequence number in the routing table is marked as invalid in route table entry.
       */
      if (!toDst.GetValidSeqNo ())
        {
          m_routingTable.Update (newEntry);
        }
      // (ii)the Destination Sequence Number in the RREP is greater than the node's copy of the destination sequence number and the known value is valid,
      else if ((int32_t (rrepHeader.GetDstSeqno ()) - int32_t (toDst.GetSeqNo ())) > 0)
        {
          m_routingTable.Update (newEntry);
        }
      else
        {
          // (iii) the sequence numbers are the same, but the route is marked as inactive.
          if ((rrepHeader.GetDstSeqno () == toDst.GetSeqNo ()) && (toDst.GetFlag () != VALID))
            {
              m_routingTable.Update (newEntry);
            }
```

```
          // (iv)  the sequence numbers are the same, and the New Hop
  Count is smaller than the hop count in route table entry.
          else if ((rrepHeader.GetDstSeqno () == toDst.GetSeqNo ()) &&
   (hop < toDst.GetHop ()))
            {
              m_routingTable.Update (newEntry);
            }
        }
    }
  else
    {
      // The forward route for this destination is created if it does
  not already exist.
      NS_LOG_LOGIC ("add new route");
      m_routingTable.AddRoute (newEntry);
    }
  // Acknowledge receipt of the RREP by sending a RREP-ACK message
  back
  if (rrepHeader.GetAckRequired ())
    {
      SendReplyAck (sender);
      rrepHeader.SetAckRequired (false);
    }
  NS_LOG_LOGIC ("receiver " << receiver << " origin " << rrepHeader.
  GetOrigin ());
  if (IsMyOwnAddress (rrepHeader.GetOrigin ()))
    {
      if (toDst.GetFlag () == IN_SEARCH)
        {
          m_routingTable.Update (newEntry);
          m_addressReqTimer[dst].Remove ();
          m_addressReqTimer.erase (dst);
        }
      m_routingTable.LookupRoute (dst, toDst);
      SendPacketFromQueue (dst, toDst.GetRoute ());
      return;
    }

  RoutingTableEntry toOrigin;
  if (!m_routingTable.LookupRoute (rrepHeader.GetOrigin (), toOrigin)
   || toOrigin.GetFlag () == IN_SEARCH)
    {
      return; // Impossible! drop.
    }
  toOrigin.SetLifeTime (std::max (ActiveRouteTimeout, toOrigin.
  GetLifeTime ()));
  m_routingTable.Update (toOrigin);

  // Update information about precursors
  if (m_routingTable.LookupValidRoute (rrepHeader.GetDst (), toDst))
    {
      toDst.InsertPrecursor (toOrigin.GetNextHop ());
      m_routingTable.Update (toDst);

      RoutingTableEntry toNextHopToDst;
      m_routingTable.LookupRoute (toDst.GetNextHop (), toNextHopToDst)
```

```
;
      toNextHopToDst.InsertPrecursor (toOrigin.GetNextHop ());
      m_routingTable.Update (toNextHopToDst);

      toOrigin.InsertPrecursor (toDst.GetNextHop ());
      m_routingTable.Update (toOrigin);

      RoutingTableEntry toNextHopToOrigin;
      m_routingTable.LookupRoute (toOrigin.GetNextHop (),
    toNextHopToOrigin);
      toNextHopToOrigin.InsertPrecursor (toDst.GetNextHop ());
      m_routingTable.Update (toNextHopToOrigin);
    }

  Ptr<Packet> packet = Create<Packet> ();
  packet->AddHeader (rrepHeader);
  TypeHeader tHeader (AODVTYPE_RREP);
  packet->AddHeader (tHeader);
  Ptr<Socket> socket = FindSocketWithInterfaceAddress (toOrigin.
    GetInterface ());
  NS_ASSERT (socket);
  socket->SendTo (packet, 0, InetSocketAddress (toOrigin.GetNextHop ()
    , AODV_PORT));
}

void
RoutingProtocol::RecvReplyAck (Ipv4Address neighbor)
{
  NS_LOG_FUNCTION (this);
  RoutingTableEntry rt;
  if(m_routingTable.LookupRoute (neighbor, rt))
    {
      rt.m_ackTimer.Cancel ();
      rt.SetFlag (VALID);
      m_routingTable.Update (rt);
    }
}

void
RoutingProtocol::ProcessHello (RrepHeader const & rrepHeader,
    Ipv4Address receiver )
{
  NS_LOG_FUNCTION (this << "from " << rrepHeader.GetDst ());
  /*
   *  Whenever a node receives a Hello message from a neighbor, the
    node
   * SHOULD make sure that it has an active route to the neighbor, and
   * create one if necessary.
   */
  RoutingTableEntry toNeighbor;
  if (!m_routingTable.LookupRoute (rrepHeader.GetDst (), toNeighbor))
    {
      Ptr<NetDevice> dev = m_ipv4->GetNetDevice (m_ipv4->
    GetInterfaceForAddress (receiver));
      RoutingTableEntry newEntry (/*device=*/ dev, /*dst=*/ rrepHeader
    .GetDst (), /*validSeqNo=*/ true, /*seqno=*/ rrepHeader.
```

```
GetDstSeqno (),
                                               /*iface=*/ m_ipv4->
GetAddress (m_ipv4->GetInterfaceForAddress (receiver), 0),
                                               /*hop=*/ 1, /*nextHop=*/
 rrepHeader.GetDst (), /*lifeTime=*/ rrepHeader.GetLifeTime ());
      m_routingTable.AddRoute (newEntry);
    }
  else
    {
      toNeighbor.SetLifeTime (std::max (Time (AllowedHelloLoss *
HelloInterval), toNeighbor.GetLifeTime ()));
      toNeighbor.SetSeqNo (rrepHeader.GetDstSeqno ());
      toNeighbor.SetValidSeqNo (true);
      toNeighbor.SetFlag (VALID);

      //CNLAB

      Ipv4Address wrmDst=rrepHeader.GetDst();
      if(EnableWrmAttack && wrmDst==FirstEndOfWormTunnel)
      {
        // cout<<"RREP Helper Contains First End P2P interface Address
";
        toNeighbor.SetOutputDevice (m_ipv4->GetNetDevice (m_ipv4->
GetInterfaceForAddress (SecondEndOfWormTunnel)));
        toNeighbor.SetInterface (m_ipv4->GetAddress (m_ipv4->
GetInterfaceForAddress (SecondEndOfWormTunnel), 0));
          toNeighbor.SetHop (1);
          toNeighbor.SetNextHop (rrepHeader.GetDst ());
      }
      else if(EnableWrmAttack && wrmDst==SecondEndOfWormTunnel)
      {
        // cout<<"RREP Helper Contains Second End P2P interface
Address";
        toNeighbor.SetOutputDevice (m_ipv4->GetNetDevice (m_ipv4->
GetInterfaceForAddress (FirstEndOfWormTunnel)));
        toNeighbor.SetInterface (m_ipv4->GetAddress (m_ipv4->
GetInterfaceForAddress (FirstEndOfWormTunnel), 0));
          toNeighbor.SetHop (1);
          toNeighbor.SetNextHop (rrepHeader.GetDst ());

      }

      else
      {
          toNeighbor.SetOutputDevice (m_ipv4->GetNetDevice (m_ipv4->
GetInterfaceForAddress (receiver)));
          toNeighbor.SetInterface (m_ipv4->GetAddress (m_ipv4->
GetInterfaceForAddress (receiver), 0));
          toNeighbor.SetHop (1);
          toNeighbor.SetNextHop (rrepHeader.GetDst ());
      }

      m_routingTable.Update (toNeighbor);
    }
  if (EnableHello)
    {
```

```
      m_nb.Update (rrepHeader.GetDst (), Time (AllowedHelloLoss *
   HelloInterval));
    }
}

void
RoutingProtocol::RecvError (Ptr<Packet> p, Ipv4Address src )
{
  NS_LOG_FUNCTION (this << " from " << src);
  RerrHeader rerrHeader;
  p->RemoveHeader (rerrHeader);
  std::map<Ipv4Address, uint32_t> dstWithNextHopSrc;
  std::map<Ipv4Address, uint32_t> unreachable;
  m_routingTable.GetListOfDestinationWithNextHop (src,
   dstWithNextHopSrc);
  std::pair<Ipv4Address, uint32_t> un;
  while (rerrHeader.RemoveUnDestination (un))
    {
      for (std::map<Ipv4Address, uint32_t >::const_iterator i =
           dstWithNextHopSrc.begin (); i != dstWithNextHopSrc.end ();
   ++i)
      {
        if (i->first == un.first)
          {
            unreachable.insert (un);
          }
      }
    }

  std::vector<Ipv4Address> precursors;
  for (std::map<Ipv4Address, uint32_t >::const_iterator i = unreachable
   .begin ();
       i != unreachable.end ();)
    {
      if (!rerrHeader.AddUnDestination (i->first, i->second))
        {
          TypeHeader typeHeader (AODVTYPE_RERR);
          Ptr<Packet> packet = Create<Packet> ();
          packet->AddHeader (rerrHeader);
          packet->AddHeader (typeHeader);
          SendRerrMessage (packet, precursors);
          rerrHeader.Clear ();
        }
      else
        {
          RoutingTableEntry toDst;
          m_routingTable.LookupRoute (i->first, toDst);
          toDst.GetPrecursors (precursors);
          ++i;
        }
    }
  if (rerrHeader.GetDestCount () != 0)
    {
      TypeHeader typeHeader (AODVTYPE_RERR);
      Ptr<Packet> packet = Create<Packet> ();
      packet->AddHeader (rerrHeader);
```

```
      packet->AddHeader (typeHeader);
      SendRerrMessage (packet, precursors);
    }
  m_routingTable.InvalidateRoutesWithDst (unreachable);
}

void
RoutingProtocol::RouteRequestTimerExpire (Ipv4Address dst)
{
  NS_LOG_LOGIC (this);
  RoutingTableEntry toDst;
  if (m_routingTable.LookupValidRoute (dst, toDst))
    {
      SendPacketFromQueue (dst, toDst.GetRoute ());
      NS_LOG_LOGIC ("route to " << dst << " found");
      return;
    }
  /*
   *  If a route discovery has been attempted RreqRetries times at the maximum TTL without
   *  receiving any RREP, all data packets destined for the corresponding destination SHOULD be
   *  dropped from the buffer and a Destination Unreachable message SHOULD be delivered to the application.
   */
  if (toDst.GetRreqCnt () == RreqRetries)
    {
      NS_LOG_LOGIC ("route discovery to " << dst << " has been attempted RreqRetries (" << RreqRetries << ") times");
      m_addressReqTimer.erase (dst);
      m_routingTable.DeleteRoute (dst);
      NS_LOG_DEBUG ("Route not found. Drop all packets with dst " << dst);
      m_queue.DropPacketWithDst (dst);
      return;
    }

  if (toDst.GetFlag () == IN_SEARCH)
    {
      NS_LOG_LOGIC ("Resend RREQ to " << dst << " ttl " << NetDiameter);
      SendRequest (dst);
    }
  else
    {
      NS_LOG_DEBUG ("Route down. Stop search. Drop packet with destination " << dst);
      m_addressReqTimer.erase (dst);
      m_routingTable.DeleteRoute (dst);
      m_queue.DropPacketWithDst (dst);
    }
}

void
RoutingProtocol::HelloTimerExpire ()
{
```

```
  NS_LOG_FUNCTION (this);
  Time offset = Time (Seconds (0));
  if (m_lastBcastTime > Time (Seconds (0)))
    {
      offset = Simulator::Now () - m_lastBcastTime;
      NS_LOG_DEBUG ("Hello deferred due to last bcast at:" << m_lastBcastTime);
    }
  else
    {
      SendHello ();
    }
  m_htimer.Cancel ();
  Time diff = HelloInterval - offset;
  m_htimer.Schedule (std::max (Time (Seconds (0)), diff));
  m_lastBcastTime = Time (Seconds (0));
}

void
RoutingProtocol::RreqRateLimitTimerExpire ()
{
  NS_LOG_FUNCTION (this);
  m_rreqCount = 0;
  m_rreqRateLimitTimer.Schedule (Seconds (1));
}

void
RoutingProtocol::RerrRateLimitTimerExpire ()
{
  NS_LOG_FUNCTION (this);
  m_rerrCount = 0;
  m_rerrRateLimitTimer.Schedule (Seconds (1));
}

void
RoutingProtocol::AckTimerExpire (Ipv4Address neighbor, Time blacklistTimeout)
{
  NS_LOG_FUNCTION (this);
  m_routingTable.MarkLinkAsUnidirectional (neighbor, blacklistTimeout);
}

void
RoutingProtocol::SendHello ()
{
  NS_LOG_FUNCTION (this);
  /* Broadcast a RREP with TTL = 1 with the RREP message fields set as follows:
   *   Destination IP Address         The node's IP address.
   *   Destination Sequence Number    The node's latest sequence number.
   *   Hop Count                      0
   *   Lifetime                       AllowedHelloLoss * HelloInterval
   */
  for (std::map<Ptr<Socket>, Ipv4InterfaceAddress>::const_iterator j =
```

```
     m_socketAddresses.begin (); j != m_socketAddresses.end (); ++j)
    {
      Ptr<Socket> socket = j->first;
      Ipv4InterfaceAddress iface = j->second;
      RrepHeader helloHeader (/*prefix size=*/ 0, /*hops=*/ 0, /*dst=*/ iface.GetLocal (), /*dst seqno=*/ m_seqNo,
                                               /*origin=*/ iface.GetLocal (),/*lifetime=*/ Time (AllowedHelloLoss * HelloInterval));
      Ptr<Packet> packet = Create<Packet> ();
      packet->AddHeader (helloHeader);
      TypeHeader tHeader (AODVTYPE_RREP);
      packet->AddHeader (tHeader);
      // Send to all-hosts broadcast if on /32 addr, subnet-directed otherwise
      Ipv4Address destination;
      if (iface.GetMask () == Ipv4Mask::GetOnes ())
        {
          destination = Ipv4Address ("255.255.255.255");
        }
      else
        {
          destination = iface.GetBroadcast ();
        }
      Time jitter = Time (MilliSeconds (m_uniformRandomVariable->GetInteger (0, 10)));
      Simulator::Schedule (jitter, &RoutingProtocol::SendTo, this , socket, packet, destination);
    }
}

void
RoutingProtocol::SendPacketFromQueue (Ipv4Address dst, Ptr<Ipv4Route> route)
{
  NS_LOG_FUNCTION (this);
  QueueEntry queueEntry;
  while (m_queue.Dequeue (dst, queueEntry))
    {
      DeferredRouteOutputTag tag;
      Ptr<Packet> p = ConstCast<Packet> (queueEntry.GetPacket ());
      if (p->RemovePacketTag (tag) &&
          tag.GetInterface() != -1 &&
          tag.GetInterface() != m_ipv4->GetInterfaceForDevice (route->GetOutputDevice ()))
        {
          NS_LOG_DEBUG ("Output device doesn't match. Dropped.");
          return;
        }
      UnicastForwardCallback ucb = queueEntry.GetUnicastForwardCallback ();
      Ipv4Header header = queueEntry.GetIpv4Header ();
      header.SetSource (route->GetSource ());
      header.SetTtl (header.GetTtl () + 1); // compensate extra TTL decrement by fake loopback routing
      ucb (route, p, header);
```

```
    }
}

void
RoutingProtocol::SendRerrWhenBreaksLinkToNextHop (Ipv4Address nextHop)
{
  NS_LOG_FUNCTION (this << nextHop);
  RerrHeader rerrHeader;
  std::vector<Ipv4Address> precursors;
  std::map<Ipv4Address, uint32_t> unreachable;

  RoutingTableEntry toNextHop;
  if (!m_routingTable.LookupRoute (nextHop, toNextHop))
    return;
  toNextHop.GetPrecursors (precursors);
  rerrHeader.AddUnDestination (nextHop, toNextHop.GetSeqNo ());
  m_routingTable.GetListOfDestinationWithNextHop (nextHop, unreachable
    );
  for (std::map<Ipv4Address, uint32_t >::const_iterator i = unreachable
    .begin (); i
       != unreachable.end ();)
    {
      if (!rerrHeader.AddUnDestination (i->first, i->second))
        {
          NS_LOG_LOGIC ("Send RERR message with maximum size.");
          TypeHeader typeHeader (AODVTYPE_RERR);
          Ptr<Packet> packet = Create<Packet> ();
          packet->AddHeader (rerrHeader);
          packet->AddHeader (typeHeader);
          SendRerrMessage (packet, precursors);
          rerrHeader.Clear ();
        }
      else
        {
          RoutingTableEntry toDst;
          m_routingTable.LookupRoute (i->first, toDst);
          toDst.GetPrecursors (precursors);
          ++i;
        }
    }
  if (rerrHeader.GetDestCount () != 0)
    {
      TypeHeader typeHeader (AODVTYPE_RERR);
      Ptr<Packet> packet = Create<Packet> ();
      packet->AddHeader (rerrHeader);
      packet->AddHeader (typeHeader);
      SendRerrMessage (packet, precursors);
    }
  unreachable.insert (std::make_pair (nextHop, toNextHop.GetSeqNo ()))
    ;
  m_routingTable.InvalidateRoutesWithDst (unreachable);
}

void
RoutingProtocol::SendRerrWhenNoRouteToForward (Ipv4Address dst,
                                               uint32_t dstSeqNo,
```

```
Ipv4Address origin)
{
  NS_LOG_FUNCTION (this);
  // A node SHOULD NOT originate more than RERR_RATELIMIT RERR messages per second.
  if (m_rerrCount == RerrRateLimit)
    {
      // Just make sure that the RerrRateLimit timer is running and will expire
      NS_ASSERT (m_rerrRateLimitTimer.IsRunning ());
      // discard the packet and return
      NS_LOG_LOGIC ("RerrRateLimit reached at " << Simulator::Now ().GetSeconds () << " with timer delay left "
                                                << m_rerrRateLimitTimer.GetDelayLeft ().GetSeconds ()
                                                << "; suppressing RERR");
      return;
    }
  RerrHeader rerrHeader;
  rerrHeader.AddUnDestination (dst, dstSeqNo);
  RoutingTableEntry toOrigin;
  Ptr<Packet> packet = Create<Packet> ();
  packet->AddHeader (rerrHeader);
  packet->AddHeader (TypeHeader (AODVTYPE_RERR));
  if (m_routingTable.LookupValidRoute (origin, toOrigin))
    {
      Ptr<Socket> socket = FindSocketWithInterfaceAddress (
          toOrigin.GetInterface ());
      NS_ASSERT (socket);
      NS_LOG_LOGIC ("Unicast RERR to the source of the data transmission");
      socket->SendTo (packet, 0, InetSocketAddress (toOrigin.GetNextHop (), AODV_PORT));
    }
  else
    {
      for (std::map<Ptr<Socket>, Ipv4InterfaceAddress>::const_iterator i =
             m_socketAddresses.begin (); i != m_socketAddresses.end (); ++i)
        {
          Ptr<Socket> socket = i->first;
          Ipv4InterfaceAddress iface = i->second;
          NS_ASSERT (socket);
          NS_LOG_LOGIC ("Broadcast RERR message from interface " << iface.GetLocal ());
          // Send to all-hosts broadcast if on /32 addr, subnet-directed otherwise
          Ipv4Address destination;
          if (iface.GetMask () == Ipv4Mask::GetOnes ())
            {
              destination = Ipv4Address ("255.255.255.255");
            }
          else
            {
```

```
                destination = iface.GetBroadcast ();
              }
            socket->SendTo (packet->Copy (), 0, InetSocketAddress (destination, AODV_PORT));
          }
      }
}

void
RoutingProtocol::SendRerrMessage (Ptr<Packet> packet, std::vector<Ipv4Address> precursors)
{
  NS_LOG_FUNCTION (this);

  if (precursors.empty ())
    {
      NS_LOG_LOGIC ("No precursors");
      return;
    }
  // A node SHOULD NOT originate more than RERR_RATELIMIT RERR messages per second.
  if (m_rerrCount == RerrRateLimit)
    {
      // Just make sure that the RerrRateLimit timer is running and will expire
      NS_ASSERT (m_rerrRateLimitTimer.IsRunning ());
      // discard the packet and return
      NS_LOG_LOGIC ("RerrRateLimit reached at " << Simulator::Now ().GetSeconds () << " with timer delay left "
                                                 << m_rerrRateLimitTimer.GetDelayLeft ().GetSeconds ()
                                                 << "; suppressing RERR");
      return;
    }
  // If there is only one precursor, RERR SHOULD be unicast toward that precursor
  if (precursors.size () == 1)
    {
      RoutingTableEntry toPrecursor;
      if (m_routingTable.LookupValidRoute (precursors.front (), toPrecursor))
        {
          Ptr<Socket> socket = FindSocketWithInterfaceAddress (toPrecursor.GetInterface ());
          NS_ASSERT (socket);
          NS_LOG_LOGIC ("one precursor => unicast RERR to " << toPrecursor.GetDestination () << " from " << toPrecursor.GetInterface ().GetLocal ());
          Simulator::Schedule (Time (MilliSeconds (m_uniformRandomVariable->GetInteger (0, 10))), &RoutingProtocol::SendTo, this, socket, packet, precursors.front ());
          m_rerrCount++;
        }
      return;
    }
```

```
  // Should only transmit RERR on those interfaces which have
    precursor nodes for the broken route
  std::vector<Ipv4InterfaceAddress> ifaces;
  RoutingTableEntry toPrecursor;
  for (std::vector<Ipv4Address>::const_iterator i = precursors.begin
    (); i != precursors.end (); ++i)
    {
      if (m_routingTable.LookupValidRoute (*i, toPrecursor) &&
          std::find (ifaces.begin (), ifaces.end (), toPrecursor.
    GetInterface ()) == ifaces.end ())
        {
          ifaces.push_back (toPrecursor.GetInterface ());
        }
    }

  for (std::vector<Ipv4InterfaceAddress>::const_iterator i = ifaces.
    begin (); i != ifaces.end (); ++i)
    {
      Ptr<Socket> socket = FindSocketWithInterfaceAddress (*i);
      NS_ASSERT (socket);
      NS_LOG_LOGIC ("Broadcast RERR message from interface " << i->
    GetLocal ());
      // std::cout << "Broadcast RERR message from interface " << i->
    GetLocal () << std::endl; //added. CNLAB
      // Send to all-hosts broadcast if on /32 addr, subnet-directed
    otherwise
      Ptr<Packet> p = packet->Copy (); //added. CNLAB
      Ipv4Address destination;
      if (i->GetMask () == Ipv4Mask::GetOnes ())
        {
          destination = Ipv4Address ("255.255.255.255");
        }
      else
        {
          destination = i->GetBroadcast ();
        }
      Simulator::Schedule (Time (MilliSeconds (m_uniformRandomVariable
    ->GetInteger (0, 10))), &RoutingProtocol::SendTo, this, socket, p,
     destination);
    }
}

Ptr<Socket>
RoutingProtocol::FindSocketWithInterfaceAddress (Ipv4InterfaceAddress
    addr ) const
{
  NS_LOG_FUNCTION (this << addr);
  for (std::map<Ptr<Socket>, Ipv4InterfaceAddress>::const_iterator j =
         m_socketAddresses.begin (); j != m_socketAddresses.end (); ++
    j)
    {
      Ptr<Socket> socket = j->first;
      Ipv4InterfaceAddress iface = j->second;
      if (iface == addr)
        return socket;
```

```
    }
  Ptr<Socket> socket;
  return socket;
}

//added: CNLAB
Ptr<Socket>
RoutingProtocol::FindSubnetBroadcastSocketWithInterfaceAddress (
    Ipv4InterfaceAddress addr ) const
{
  NS_LOG_FUNCTION (this << addr);
  for (std::map<Ptr<Socket>, Ipv4InterfaceAddress>::const_iterator j =
         m_socketSubnetBroadcastAddresses.begin (); j !=
    m_socketSubnetBroadcastAddresses.end (); ++j)
    {
      Ptr<Socket> socket = j->first;
      Ipv4InterfaceAddress iface = j->second;
      if (iface == addr)
        return socket;
    }
  Ptr<Socket> socket;
  return socket;
}

//added: CNLAB
void
RoutingProtocol::DoInitialize (void)
{
  NS_LOG_FUNCTION (this);
  uint32_t startTime;
  if (EnableHello)
    {
      m_htimer.SetFunction (&RoutingProtocol::HelloTimerExpire, this);
      startTime = m_uniformRandomVariable->GetInteger (0, 100);
      NS_LOG_DEBUG ("Starting at time " << startTime << "ms");
      m_htimer.Schedule (MilliSeconds (startTime));
    }
  Ipv4RoutingProtocol::DoInitialize ();
}

} //namespace aodv
} //namespace ns3
```

Appendix B

NS-3 802.11s modules

In this appendix, the modules implemented in C++ and how they interconnect with each other is presented.

B.1 SensorHelper

void SetSpreadInterfaceChannels

Parameters: (ChannelPolicy)
Set the channel policy which can be SPREAD CHANNELS or ZERO CHANNEL: Spread or not spread frequency channels of MP interfaces. If set to true different non-overlapping 20MHz frequency channels will be assigned to different mesh point interfaces.

void SetStackInstaller

Parameters: (std::string type, std::string n0 = ””, const AttributeValue & v0 = Empty- AttributeValue (),...)
You need to tell which Sensor stack do you want, in this case Wormhole11sStack, so you can use the characteristics of the 802.11s. n0 and v0 are the name and the value of the attribute to set, respectively. For example you can set the root node in the sensor network.

void SetNumberOfInterfaces

Parameters: (uint32 t nInterfaces)
Set a number of interfaces in a sensor network.

NetDeviceContainer Install

Parameters: (const WifiPhyHelper & phyHelper, NodeContainer c)
Install 802.11s sensor device and protocols on given node list. The phyHelper is the Wifi PHY helper and c is the list of nodes to install. This function returns the list of created sensor point devices.

void SetMacType

Parameters: (std::string n0 = "", const AttributeValue & v0 = EmptyAttributeValue (),...)
Uses the class SensorWifiInterfaceMac and n0 and v0 are the name and the value of the attribute to set, respectively. The values that can be set are the next ones:

- BeaconInterval : Beacon Interval. Initial value: 0.5 seconds
- RandomStart: Window when beacon generating starts (uniform random) in seconds. Initial value: 0.5 seconds
- BeaconGeneration: Enable/Disable Beaconing. Initial value: enabled
- TxOkHeader : The header of successfully transmitted packet.
- TxErrHeader : The header of unsuccessfully transmitted packet.

This class uses the RegularWifiMac class where you can set the QoS support which enable 802.11e/WMM-style (By default is disable). And at the same time this class has as parent class WifiMac which we can use to modify values like CTS timeout, ACK timeout, SIFTS, EIFS-DIFS, duration of a slot, PIFS or the Ssid.

void SetRemoteStationManager

Parameters: (std::string type, std::string n0 = " ", const AttributeValue & v0 = Empty- AttributeValue (),...)
With this function, using the variable type we define which station manager do we want. A part from a constant bit rate value, the following rate control algorithms implemented in NS-3: AARF, AARF-CD, AMRR, ARF, CARA, Ideal, Minstrel, ONOE and RRAA. The one selected by default is the ARF. They all use has as parent class WifiRemoteStationManager, and using the n0 and v0 you can set the maximum number of retransmission attempts for an RTS and data packets as well as the threshold to decide when to use a RTS/CTS handshake before sending a data packet and the one to decide when to fragment them. As described in IEEE Std. 802.11-2007, Section 9.2.6. and 9.4. This value will not have any effect on some rate control algorithms. Here we can also set a wifi mode for non-unicast transmissions.

void SetNumberOfInterfaces

Parameters: (uint32 t nInterfaces)
Set a number of interfaces in a Sensor network.

void SetStandard

Parameters: (enum WifiPhyStandard standard)
Allows you to select the following standards: 802.11 with 5 or 10 Mhz, 802.11a and 802.11b. The one set by default is the 802.11a.

void AddInterface

Parameters: (Ptr <NetDevice>port)
Attach new interface to the station. Interface must support 48-bit MAC address and only SensorPointDevice can have IP address, but not individual interfaces.

bool SetMtu

Parameters: (const uint16 t mtu)
Set the MAC-level Maximum Transmission Unit in bytes and returns whether the MTU value was within legal bounds. Override for default MTU defined on a per-type basis.

void SetRoutingProtocol

Parameters: (Ptr <SensorL2RoutingProtocol>protocol)
Register the Sensor routing protocol to be used by this snesor point. Protocol must be already installed on this snesor point.