TITLE: DETECTING AND PREVENTING SELFISH NODE ATTACKS IN MANETS: A LIGHTWEIGHT APPROACH

Teklu Wolde

HAWASSA UNIVERSITY, HAWASSA ETHIOPIA

JULY, 2025

DETECTING AND PREVENTING SELFISH NODE ATTACKS IN MANETS: A LIGHTWEIGHT APPROACH

Teklu Wolde

MAJOR ADVISOR: BERHANE W/GABRIEL (Asst. Professor)

A THESIS SUBMITTED TO DEPARTMENT OF COMPUTER SCINCE

HAWASSA UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE

OF MASTER OF SCIENCE IN COMPUTER SCIENCE

HAWASSA, ETHIOPIA

MAY, 2025

APPROVAL SHEET-I

This is to certify that the thesis entitled "***DETECTING AND PREVENTING SELFISH NODE ATTACKS IN MANETS: A LIGHTWEIGHT APPROACH***" submitted in partial fulfillment of the requirements for the degree of Master's with specialization in Computer Science, the Graduate Program of the Department/School of Informatics, and has been carried out by **Teklu Wolde**. Therefore, we recommend that the student has fulfilled the requirements and hence hereby can submit the thesis to the department.

Berhane W/Gabriel (Ass.Profesor)		19/5/2025
Name of major advisor	Signature	Date
Name of co-advisor	Signature	Date

ii

APPROVAL SHEET-II

We, the undersigned, members of the Board of Examiners of the final open defense by **Teklu Wolde** have read and evaluated his/her thesis entitled "***DETECTING AND PREVENTING SELFISH NODE ATTACKS IN MANETS: A LIGHTWEIGHT APPROACH***", and examined the candidate. This is, therefore, to certify that the thesis has been accepted in partial fulfillment of the requirements for the degree.

Berhane W/Gabriel		19/5/2025
Name of Major Advisor	Signature	Date
_____	_____	_____
Name of Internal Examiner-I	Signature	Date
_____	_____	_____
Name of Internal Examiner-II	Signature	Date
_____	_____	_____
Name of External examiner	Signature	Date
_____	_____	_____
SGS Approval	Signature	Date

Final approval and acceptance of the thesis is contingent upon the submission of the final copy of the thesis to the School of Graduate Studies (SGS) through the Department/School Graduate Committee (DGC/SGC) of the candidate's department.

Stamp of SGS Date: _____

Remark

- Use this form to submit the thesis with ***MINOR CORRECTION*** suggested by the examining board
- 3 copies

iii

ACKNOWLEDGEMENT

First of all, I would like to thank my Almighty God for giving me the strength, knowledge, ability, and opportunity to conduct this research study. Without His blessings, this achievement and accomplishment would not have been possible.

Secondly, I would like to express my sincere gratitude and thanks to my advisor, Berhane W/Gabriel, from whom I received amazing guidance and constructive ideas, offered with a great care.

Thirdly, I am also thankful to all my instructors for their valuable effort in my study.

I am Greatly thankful to the Dean, Program Manager and Program Chair, Faculty of Informatics Computer Science Department.

Finally, my heartfelt gratitude goes to my families and friends for helping to keep Moving in the right direction and strengthen me to pass the challenging times.

STATEMENT OF THE AUTHOUR

I hereby declare that this MSc thesis is my original work and has not been presented for a degree in any other university, and all sources of material used for this thesis have been duly acknowledged.

Name: ……...……. Signature: ……..……

Place: Institute of Technology, Hawassa University, Hawassa

Date of Submission: …….

Abstract

Mobile Ad hoc Networks (MANETs) are decentralized networks of self-organizing, mobile nodes communicating wirelessly without fixed infrastructure. Their dynamic nature, open medium, and lack of central control make them susceptible to routing attacks. A significant threat is selfish node behavior, where nodes participate in route discovery but refuse to forward data packets to conserve resources, as seen in the Ad-hoc On-demand Distance Vector (AODV) protocol.

This thesis introduces a MA (Mobile Agent) based lightweight system to detect and prevent Selfish Node Type 1 (SNT1) attacks in MANETs. The proposed system employs multiple MAs that utilize a watchdog and trust value technique to monitor neighboring nodes during data transmission. By overhearing network traffic, these MAs analyze node behavior to identify selfish activity, distinguishing it from other network issues. Upon detection, the system blocks the identified selfish nodes from participating in future route discovery processes.

Simulations conducted using NS-3 (Network Simulator 3) demonstrate that this proposed detection and prevention system significantly enhances network performance compared to standard AODV and existing detection mechanisms. Specifically, with 25 nodes and 6 selfish nodes, the lightweight system achieves substantially higher throughput and packet delivery ratios. While the prevention mechanism might introduce a slight increase in end-to-end delay due to the necessity of discovering new routes after isolating selfish nodes, the overall improvement in Quality of Service (QoS) by effectively mitigating data packet drops caused by selfish behaviour underscores the efficacy of this novel approach. The comparison with existing literature further validates the advancements offered by this lightweight system in addressing selfish node attacks in MANETs.

Keywords: MANET, AODV, MA, Watchdog, Selfish Node Type 1, QoS,

Lightweight system, trust value.

Table of Contents

ABBREVIATIONS

MANET	Mobile Ad-hoc Network
AODV	Ad-hoc On-demand Distance Vector
RREQ	Route Request
RREP	Route Reply
RERR	Route Error
MAC	Media Access Control
MA	Mobile Agent
NS-3	Network Simulator 3
QoS	Quality of Service
DSR	Dynamic Source Routing
DSDV	Destination Sequence Distance Vector
OLSR	Optimized Link State Routing
ZRP	Zone Routing Protocol
MPR	Multipoint Relay
TTL	Time to Live
HCR	Hybrid Cluster Routing
CLREQ	Cluster List Request
CLREP	Cluster List Reply
SNT1	Selfish Node Type 1
SNT2	Selfish Node Type 2
SNT3	Selfish Node Type 3
TORA	Temporarily Ordered Routing Algorithm
PDR	Packet Delivery Ratio
NRL	Normal Routing Load

LIST OF TABLES

LIST OF FIGURES

CHAPTER ONE
INTRODUCTION

1.1 Background of the Study

Mobile Ad hoc Network (MANET) is a collection of autonomous, self-organizing and freely moving mobile nodes connected through wireless links, without the support of any central infrastructure. MANET is vulnerable to various kinds of routing attacks due to its dynamic topology, mobility, open medium, and lack of central monitoring. So, security in MANET routing protocol is a vital issue and needs a mechanism to protect communication between nodes.

In recent times, wireless communications are growing tremendously and become one of the leading forces to improve computing industry. A wireless network is capable of joining two or more devices through wireless links. MANET is a group of autonomous, collaborative, multi-hoping and freely moving mobile nodes connected to exchange information without the support of any communications and centralized infrastructure [1]. The nodes in MANET are free to move, sometimes at the risk of security threat and highly vulnerable to different routing attacks in all layer of the network [11, 16]. The source node may not be in the same transmission range as the destination node in the network. Thus, the packet should pass multiple neighboring nodes (multiple hops) to reach the destination. From those intermediate nodes, one or more nodes have a chance to become selfish nodes and affect the normal operation of the routing protocol by dropping the data packets.

In this work, a Mobile agent (MA) based lightweight system is proposed to detect and prevent SNT1 (Selfish Node Type 1) attack in MANET. The system implements more than one MA nodes to overhear the network, analyze the selfish behavior of neighboring node and detect selfish nodes. Once a selfish node is detected, MA uses the watchdog and trust value technique to detect selfish node in its neighboring nodes during data communication.

The system implements more than one MA nodes to overhear the network, analyze the Selfish behavior of neighboring node and detect selfish nodes. Once a selfish node is detected,

MA identifies the data packet dropped due to the selfish node from other network related problems and blocks the participation of the selfish nodes in the routing discovery process. The system detects and prevent selfish node and avoid data packet drop due to selfish node and improve QoS (Quality of Service). Simulation is performed on NS-3 (Network Simulator 3) and performance of detection and prevention system with normal routing protocol and updated routing protocol is analyzed.

1.2 Application area of MANET

a) Military and defense

Mobile ad hoc network enables the military to keep data communication between soldieries, vehicles, and military data central command. MANETs are used as an important solution for military operations as it avoids a single point of failure.

b) Sensor network

Wireless sensor networks use sensors to provide a wireless communication infrastructure. Sensor nodes are small devices used for sensing physical environments, processing data, and communicating over the networks to the monitoring station. The application areas are military, health care, home security and environmental monitoring. Sensors are placed in harsh environment that makes the sensor nodes battery charging more difficult.

c) Emergency Services

In emergency situation such as earthquakes, flood and fire the wired networks could be destroyed. MANET is very useful in emergency and rescue applications. Immediate deployment of MANET in these scenarios could be built very quickly to restore communication compared to long time and costly efforts required for constructing wired networks.

d) Automotive Applications

Automobiles ought to communicate with street, traffic lights, and to other cars in the road. The network gives information to the drivers about road congestion information, accident-ahead warnings and helping to optimize traffic flow.

1.3 Routing in MANET

Routing in network is the process of selecting the shortest and most optimal path Between nodes in the networks. To enable communication among nodes in MANET, a routing protocol is required to establish routes and forward a data packet between sending and receiving nodes. Due to the limited transmission range, multiple hops [17] are needed to forward a data from source to destination nodes. Since MANET is an infrastructure less network, each mobile node operates not only as a host but also as a router by forwarding packets for other mobile nodes in the network. As bandwidth becomes limited, the routing protocols used for data packet transmission should be bandwidth efficient [16].

1.4 Statement of The Problem

Selfishness within nodes has imminent disastrous effect on MANET as it reduces the performance of overall network. Therefore, detecting and eliminating these selfish nodes is a vital step in ensuring a working system. [8] presented a method of detecting selfish nodes and tested the proposed method on an AODV (Ad hoc On-Demand Distance Vector) routing protocol. Although the method looks promising and efficient, it suffers from several limitations. After implementing the protocol and running the method on MATLAB, it has been found that their proposed selfish nodes detection mechanism has two prominent drawbacks.

Firstly, the method failed to notice the problem pertaining to the first type of selfish node; dropping RREQ (Route Request) packets. Due to the dynamic structure of MANET, a node may receive RREQ more than once and from different source nodes. This method suggests that when a node receives RREQ from a node with the same ID it has previously received, it will drop the packet, hence considered as potentially malicious node but it does not necessarily mean it is selfish.

Secondly, the method could not address an issue related to the second type of selfish node; dropping data packet. When a node receives a data packet, it forwards the packet to the neighboring node following the established route until the packet reaches the destination node. Problem arises when neighboring node may be out of coverage zone of the forwarding node. After sending a data packet, sending node does not know if the receiving node has

successfully received the packet, and it might identify the receiving node falsely as malicious in case no forwarding action has been performed by the receiving node. In both cases, the paper failed to see these problems and therefore could see the next node of a particular forwarding node as selfish in nature. This false decision will lead to lower performance as normal nodes could be terminated from the network while in fact, the nodes can participate in forwarding packets.

This paper proposes a method to minimize false detection of selfish nodes.In MANET, routing and guaranteeing quality of service (QoS) is much more challenging than wired networks, mainly due to node mobility, multi-hop communications, unpredictable topology change, and lack of central coordination.

When a sender needs to transfer data in MANET using AODV protocol, it initiates a route discovery process by broadcasting a RREQ message. The selfish node receives the RREQ packet, cooperates in route establishment process and becomes part of the active route. Once the source node receives a RREP (Route Reply) message, it starts sending data packets through the selfish nodes and it starts dropping the data packet. This loss of information degrades the network, reduce QoS and affect the overall performance of the network [10].

The number of packets received, throughput, routing overhead and packet delivery ratio of the AODV routing protocol with the existence of selfish node attack is highly affected. Most of the previous research work focused on only detection of selfish node attacks. Selfish nodes affect the transmission process by dropping the incoming data packets. Here, MA based system is implemented for selfish node detection and prevention during data transmission. Many authors have proposed to improve the quality of service of MANET routing protocol by providing different selfish node detection and prevention mechanisms.

There are several data dropping factors like network transmission errors, routing attack and selfish behavior of nodes. The network related errors cannot be completely avoided.
To improve the performance of a network a detection and prevention mechanism for routing attacks should be designed. The proposed lightweight selfish node detection and prevention system does not give solution for Selfish Node Type 2 (SNT2) and Selfish Node Type 3 (SNT3). Therefore, the main focus of this thesis is on presenting an improved lightweight

System for detection and prevention of SNT1 in MANET. The implementation of this system in AODV routing protocol, prevent the data packet drop due to selfish behavior of a node and improve the QoS and performance of a network.

1.5 Research question

This study attempts to answer the following research questions.

1. How the proposed detection and prevention mechanisms work to detect and prevent selfish node attack in MANET?
2. How much data packet are dropped specifically by the selfish nodes attack and other reasons?
3. What is the performance difference in AODV routing at the time of Selfish Node detection and prevention system?
4. How does the scalability of lightweight detection perform as the number of nodes and the Type 1 selfish attacks increase?

1.6 Objectives

General Objective

The main objective of this study is to develop a lightweight system for the detection and prevention of selfish node attack in MANET for AODV routing protocol.

Specific Objective

Investigation of various security challenges in MANET through literature review for identifying and exploring routing attacks, especially selfish node attack and

- Develop MA based detection system for the existence of selfish node attack in MANET.
- Identify data packet drop due to a selfish node attack and develop a prevention system for selfish node attack to avoid data packet drop in MANET.
- Make comparative analysis of data transmission in selfish node detection and proposed prevention system in AODV routing protocol.

1.7 Scope

The scope of this work is limited to the detection and prevention of Selfish Node Type 1 (SNT1). It does not cover other Selfish Node attack types. The system only detects and prevents a packet drop due to selfish node in the network. In the proposed method, MA node monitors the neighboring node for misbehaving activity. MA uses the watchdog mechanism and trust value to detect the selfish node in its neighbors. After the implementation of the system there may also exist packet drop due to other reasons.

1.8 Significance of the study

The ordinary AODV routing algorithm is highly vulnerable to different routing attacks and need modification, using different techniques.

This work proposes an improved detection. and prevention mechanism for selfish node attack in AODV routing using a lightweight system. This MA based detection and prevention is achieved by modifying the normal AODV routing algorithm by incorporating a trust and watchdog mechanisms. The system avoids data packet drop due to selfish node attack and improve the overall performance of the network. Therefore, this work will enhance the usefulness of MANETs by providing better QoS support routing technique.

CHAPTER TWO

LITERATURE REVIEW

This chapter aims to provide basic theoretical concepts related to MANET routing protocol and security during data packet routing. Section 2.1 covers the common routing protocols used in MANET. Section 2.2 discusses security threats in MANET and the different kinds of routing attacks. Section 2.3 and 2.4 deal with specific attack with in MANETs. 2.5 explores mobile agents and finally section 2.6 Review Related works.

2.1 MANET Routing Protocols

Routing in network is the process of selecting the shortest and most optimal path between nodes in the networks. To enable communication among nodes in MANET, a routing protocol is required to establish routes and forward a data packet between sending and receiving nodes. Due to the limited transmission range, multiple hops [17] are needed to forward a data from source to destination nodes.

A route can be affected by the mobility nature of the mobile nodes. Since mobile nodes share the same frequency channel, the capacity of the network bandwidth becomes limited. The routing protocols used for data packet transmission should be bandwidth efficient [16].

Each source node in the network discovers and maintains routes to destination in the network. The nodes that participate in the network are responsible for dynamically discovering other nodes to communicate with each other. When a given node wants to communicate with a node outside its transmission range, a multi-hop routing strategy is used and the packet must pass several intermediate nodes [13]. The basic phenomenon is that, whenever a new node enters into MANET, it must announce its arrival and presence by sending a HELLO packet to the neighboring nodes. The new comer node also listens to similar HELLO packet broadcasts by other mobile nodes. The primary challenge in MANET is to establish efficient route between a pair of nodes and to ensure secured and timely delivery of data packets. As shown in Figure 2.0, the protocols in MANET [4] can be classified based on routing strategy as reactive, proactive and hybrid protocols. In this section, each routing protocols are discussed in detail along with examples.

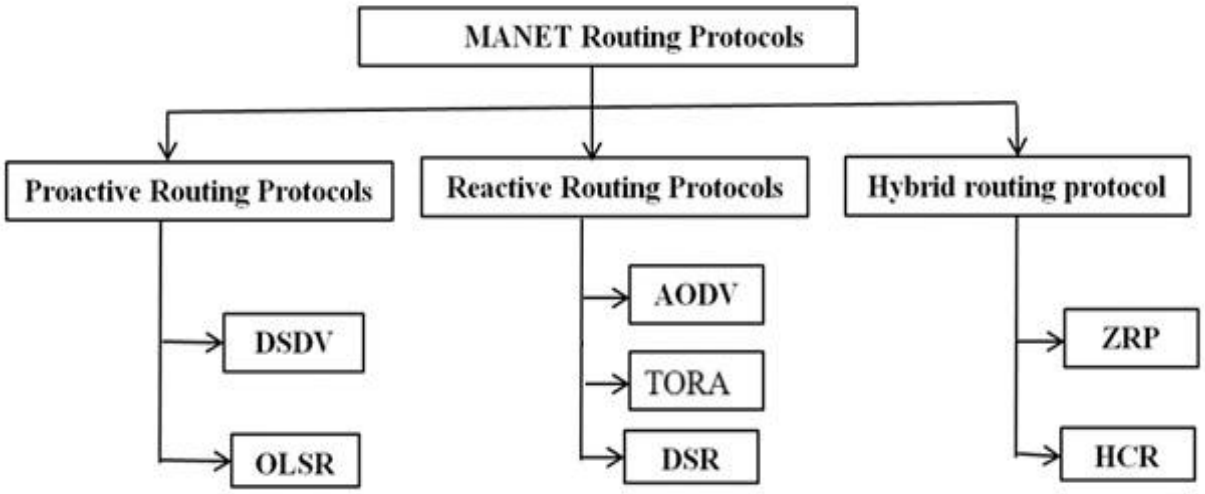


Figure 2.0: Classification of MANET routing protocols

2.1.1 Proactive routing protocol

Proactive routing protocol is also known as a table-driven routing protocols, each node of the network determine route to various node in advance and create a single or multiple routing tables that is regularly updated. Each node sends a broadcasting message to all the other nodes in the network in order to detect the changes in the network topology.

The information related to topology will be exchanged between the nodes updated regularly and it will create a high overhead. The amount of bandwidth consumed by this routing protocol will be very high due to sending messages constantly to keep the nodes routing information updated [19]. This protocol will be much efficient in the networks with small number of nodes. For a large network, it is difficult for each node to maintain large size routing information.

DSDV (Destination-Sequenced Distance Vector) routing protocol. It's a proactive, table-driven routing protocol used in mobile ad-hoc networks (MANETs). It incorporates sequence numbers to prevent routing loops, a common problem in distance vector protocols. Nodes

periodically broadcast their routing tables to neighbors, and updates are triggered by network topology changes.

OLSR (Optimized Link State Routing) Protocol. It is also a proactive routing protocol designed for MANETs. However, unlike DSDV which is distance-vector based, OLSR is a link-state routing protocol. It employs a technique called "multipoint relays" (MPRs) to optimize the flooding of control traffic. In OLSR, only a selected set of nodes (MPRs) forward broadcast messages, which significantly reduces overhead compared to traditional link-state protocols where every node retransmits. Nodes in OLSR periodically broadcast "hello" messages to discover their neighbors and "topology control" (TC) messages to disseminate information about the selected MPRs, allowing each node to build a partial view of the network topology and compute routes.

2.1.2 Reactive routing protocol

In reactive routing protocols, there is no need for the nodes to keep routing information [21]. It is on demand routing protocols, when the source mobile node has the packets for the destination node, it initiates the route discovery process and the route is present in the routing tables. Whenever a source wants to forward a data packet, route discovery will be done for that particular destination.

These protocols do not start route discovery by themselves, until it has been made a route request to find the appropriate route to destination. The process of route discovery occurs by flooding the route request packets throughout the mobile network [15]. Flooding is a reliable method of route discovery, but it uses high bandwidth and creates network overhead [28]. The active routes can be broken with the frequent changes in the topology of the network. These routes will be stored only for a certain time period in cache.

This section discusses the key differences and considerations of the reactive routing protocols ODV, TORA (Temporally Ordered Routing Algorithm), and DSR (Dynamic Source Routing).

Route Information:

- ✓ AODV uses traditional routing tables where each node knows the next hop to a destination.

- ✓ DSR uses source routing, meaning the source node knows the entire path (sequence of intermediate nodes) to the destination, and this path is included in the packet header.
- ✓ TORA uses a "height" metric and builds a DAG (Directed Acyclic Graph), aiming for localized updates rather than global knowledge of paths.

Overhead vs. Reactivity:

- ➢ AODV and DSR . They only perform route discovery when a route is needed.
- ➢ DSR avoids periodic updates entirely, making it efficient in terms of control overhead when the network is stable. However, the packet header size can be a concern for long routes.
- ➢ TORA is designed for highly dynamic environments, aiming to minimize the impact of topological changes by localizing control messages. Its "link reversal" mechanism is central to its adaptive nature.

Difficulty

- ▪ AODV is generally simpler to implement than TORA.
- ▪ DSR is conceptually straightforward but maintaining route caches effectively and dealing with stale routes can add complexity.
- ▪ TORA is considered more complex due to its "height" metric, link reversal algorithm, and multi-functional control packets.

AODV (Ad-hoc On-Demand Distance Vector) uses a route discovery process involving Route Request (RREQ) and Route Reply (RREP) messages and maintains routes as long as they are in use.

DSR (Dynamic Source Routing) protocol. In DSR, the source node determines the entire route to the destination and includes this route in the packet header. It relies on route discovery and route maintenance mechanisms but, unlike AODV, does not maintain routing tables at intermediate nodes

Advantages of reactive routing protocol

- ✓ Reduced overhead, routes are created only when needed, minimizing control traffic
- ✓ Efficient Resource Use, resources are not consumed maintaining unused routes.
- ✓ Scalability: Performs better in large, sparse networks with less frequent

communication [9].

- ✓ Simplicity,Nodes don't need to store or continuously update routing tables.

Disadvantage of reactive routing protocol

- ✓ Initial delay, Route discovery happens only when needed, causing a delay .
- ✓ Scalability issues, route discovery load can become significant in large networks.
- ✓ Fequent route rediscovery when topology changes.it requires new route.

2.1.3 Ad hoc On Demand Distance Vector (AODV) routing protocol

AODV is on demand and source driven reactive routing protocol that works on hop-by-hop routing mechanism [25]. The main mechanism behind this protocol is that, the source requests a route when it needs to send a packet to destination. Routes in routing table are deleted even if the route is valid to use the bandwidth efficiently. The route discovery process is frequent, but the performance is better in high mobility scenario. In AODV routing, a sequence numbers are used to determine up-to-date path to a destination. Every entry of the route information in the routing table has a sequence number.The sequence number acts as a route time stamp, ensuring freshness of the route. The nodes which are not present on the active route, will neither actively transmit the update packets nor try to maintain their routing tables up-to-date in AODV [23, 30, 38].

a) Route Discovery mechanism in AODV

The neighboring node will return a RREP (Request Reply) message if they have a route to the desired destination node. If they do not have the route to the destination, they forward the RREQ packet to other intermediate nodes except the original sender node.The neighboring node will return a RREP (Request Reply) message if they have a route to the desired destination node. If they do not have the route to the destination, they forward the RREQ packet to other intermediate nodes except the original sender node.
This process will continue until the destination node is discovered and route is established. The route request packet contains fields like source address, source sequence number,

broadcast ID, destination address, destination sequence number and hop count [5]. While transmitting the RREQ packet, if the source node is unaware of the destination, the destination sequence number will be empty. Hop count refers to the distance from source tothe destination node. In AODV protocol, sequence number and broadcast ID are the two counters that are maintained by every node in the network [22].

Upon receiving the RREQ, the destination node generates RREP packet and send back to the source nodes in the reverse path. Destination sequence number will be included by the destination node while sending the RREP packet. The RREP packet contains fields like source address, destination address, destination sequence number, hop count and lifetime. If source node receives more than one RREP packet, a route reply packet with higher destination sequence number is selected. If destination sequence numbers from different route are the same then, it will select RREP packet with a minimum hop count to the destination sequence number, to ensure the route is fresh and loop free. If the intermediate node destination sequence number is greater than the RREQ's sequence number, the intermediate node sends a RREP message to the source. The duplicate packets resulted due to route discovery process will be discarded by the intermediate nodes. If this intermediate node has a higher sequence number than the RREQ, then a route reply will be sent back to the source node. If the sequence number of the intermediate node is less than the incoming packet, then this RREQ will be broadcasted further [5, 7, 22, 30].

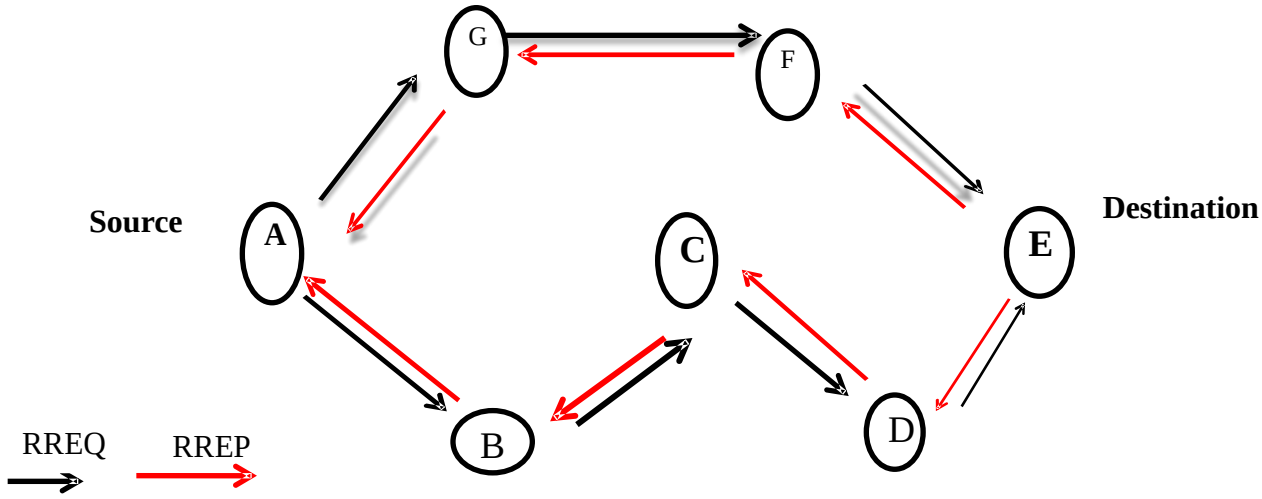


Figure 2.1 : Route discovery mechanism in AODV

In the Figure 2.1, the source node “A” initiates a route discovery and sends RREQ packet to its neighboring node “B” and “G” in its transmission range. If node “B” and “G” have a route to destination node “E”, they will send a RREP packet to the source. In this case, intermediate node “B”, “C”, “D”, “G” and “F” have no fresh route information to destination andthe RREQ packet is sent to the final destination node “E”. In receiving the RREQ, the destination node generates a RREP packet and sends it in the reverse direction from node “E” to node “A”. The source node receives two RREP packets from intermediate node “B” and “G”. The route reply with the highest destination sequence number and minimum hop count is received and the remaining RREP packets are discarded. When node “A” wants to establish a path to node “E”, the routing table at each intermediate node is updated and maintained for some time. Now node “A” sends a data packet to node “E” through the path A→G→F→E.

The broadcast ID, destination address, source address, destination sequence number and hop count are the major entries of the RREQ packet. Figure 2.2 [27], shows the format of RREQ message (packet) for AODV routing protocol.

Type	Flag	Reserved	Hop count
Broadcast id			
Destination IP address			
Destination Sequence number			
Source IP address			
Source sequence number			

Figure 2.2 *RREQ message format of AODV*

After successful forwarding of the RREQ packet to the final destination node or intermediate nodes with a fresh route to destination, a RREP packet is generated by the

destination node and sent back to the source node in the reverse direction. The content of the RREP packet is similar with the RREQ packet format except it includes the source sequence number and lifetime of the RREP packet. Figure 2.3 shows the format of RREP message for AODV routing protocol [27]**.**

Type	Flag	Reserved	Hop count
Broadcast id Destination IP address			
Destination Sequence number			
Source IP address			
Source sequence number			
Lifetime			

Figure 2.3: *RREP message format of AODV*

b) Route Maintenance in AODV

All nodes monitor their neighborhood by sending a HELLO packet. If HELLO messages do not reach the receiving node on some time interval, it indicates that there is a link failure between nodes. When a node in an active route gets lost, a Route Error (RERR) message is generated to notify the other nodes on both sides for link failure. Every node in the network checks whether the nearby nodes are actively participating in the routing process. A link failure is detected by sending the RERR control packet to a source node. When a node is unable to forward a packet to next hop or destination node, it generates RERR by tagging it with a higher destination sequence number. So, when a node receives RERR packet, it marks its own routing table entries as invalid and delete the particular route entry. When a source node receives a RERR message it initiates a new route discovery for the destination node [5, 7, 30, 35].

In the Figure 2.1, source node sends a RREQ message to its neighboring node B and G to reach the destination node E. But, while node D tries to forward the RREQ packet to node E, link failure is detected. Thus, node D will generate and send the RERR message to the source node "A" in the reverse direction. The source node initiates a new route discovery operation to get route to the destination node.

In AODV routing protocol, four types of control message namely RREQ, RREP, RERR and HELLO messages were used. RREQ and RREP message are used to discover a route to destination nodes. The RERR and HELLO control message are used for route maintenance and repair [5, 35].

Creating routing table in AODV

The AODV routing protocol discovers a route to destination and maintain a routing table. The routing table contains several fields related to wireless node communication in MANET. When a source node sends a RREQ packet to discover a path toward destination, it reach to the destination node, which further reply a RREP packet toward source node. On successful discovery of a route the node maintain routing information about the path in the routing table.As shown in figure 2.4 [27], routing table contains the information necessary to forward a packet to destination node.

Destination Ip address
Destination sequence number
Hop count
Next hop
First hop
Valid bit
Count

Figure 2.4: *Fields of AODV routing table*

Consider Figure 2.5. A simple MANET with mobile nodes S, A, B and D to explain the route discovery process and how to create a routing tables during route discovery.

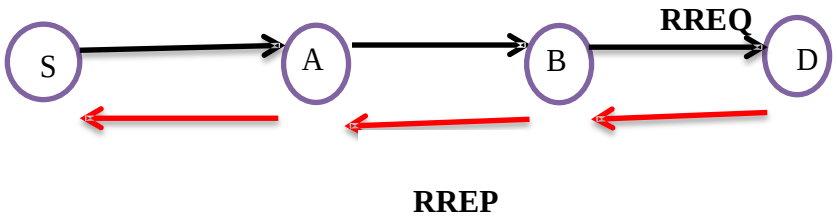


Figure 2.5: *A Mobile Ad-hoc network with 4 nodes*

Mobile node S is the source node, node D is a destination node and node A and B are intermediate nodes. When a node S wants to send data packet to the destination node D, S starts a route discovery process by flooding a RREQ packet.

The RREQ packet format of node S becomes <S, 1, 1, D, 0>. Here "S" is the source node address, "1" is the source sequence number, "1" is a broadcast id, "D" is a destination node address and "0" is a hop count respectively. At this step the destination sequence number is unknown and left empty in the RREQ packet. Since node S has only one neighboring node, it will send the RREQ packet to node A. Node A is also unaware about destination node D, it will broadcast the RREQ packet to node B after it updates the hop count in the routing table. Node A will increment the hop count to 1 and broadcast the RREQ packet to node B. The RREQ packet at this stage becomes <S, 1, 1, D, 1>. Then, the routing table of node A after receiving RREQ packet from node S for destination address, next hop, hop count and sequence number entry becomes <S, S. 1, 1>.

The hop count in the routing table at node B is 2 which indicate the distance from node B to reach node S through node A is 2 hops. Then, node B broadcast the updated RREQ packet to node D, and the format of this updated RREQ packet becomes <S, 1, 1, D, 2>. The routing table of node B after receiving RREQ packet from node A becomes <S, A, 2, 1>. The destination node (Node D) will create an entry on its routing table after receiving a RREQ

packet from node B. The routing table entry of node D after receiving RREQ packet from B becomes <S, B, 3, 1>.

The destination address on the RREQ packet matches with the address of node D (receiving node), node D initiate a RREP packet with the source node address, destination node address, destination sequence number and hop count values like <D, A, 120, 0 >.

The destination sequence number 120 is a random number generated by the destination node, which is greater than the sequence number in the RREQ packet.The route reply packet will be sent to the neighboring node of the destination node i.e. node B in the reverse direction. Thus, node B will create a new entry in its routing table. Table 2.1 contains the updated routing table of node B after receiving RREP packet from D

Destination	**Next Hop**	**Hop count**	**Sequence Number**
S	A	2	1
D	D	1	120

Table 2.1: *Routing table of node B in a given network*

Now, node B will broadcast the RREP packet to node A by incrementing the hop count by one. The format of the broadcasted RREP packet from node B to node A looks like < D, S, 120, 1 >. Table 2.2 shows the updated routing table of node A after receiving RREP packet from node B.

Destination	Next Hop	Hop count	Sequence
S	***S***	***1***	***1***
D	***B***	***2***	***120***

Table 2.2: *Updated routing table of node A in a given network*

Finally, node A will forward the incoming RREP packet to the sender node S. The format of the RREP packet looks like < S, S, 120, 2 >. Thus, the routing table entry of node S after receiving RREP packet from A becomes <D, A, 3, 120> for "destination address", "next hop", "hop count" and "destination sequence number" respectively.

In the above route discovery process the routing table from source node to destination node is updated. If another node in the network wants to send a data packet to another node they check the existence of route in their routing table and make decision. In the process of a route discovery, if an intermediate node receives more than one RREQ message from the same source, they will accept the first packet and discard the redundant RREQ packets by checking the broadcast id.

Maintaining Sequence Numbers in AODV

The sequence numbers are the most important feature of AODV for removing the old information from the network [26]. The use of a destination sequence number for each route entry is one of the distinguished features of AODV. The destination sequence number is created by the destination node, added to the RREP message and sends to source nodes. When a source node broadcast a RREQ packet to discover the route, the destination sequence number field is empty and incorporated by the destination node in RREP message. It acts as a times tamps and prevents the AODV protocol from loop problem and problem counting to infinity by removing the old route from the routing table. The destination sequence number in the routing table is updated whenever a node receives new information about the sequence number from RREQ, RREP, or RERR messages that may be received related to that destination [11, 35].

Destination node increments its own sequence number in two conditions [26].

- ✓ Immediately before a node initiate a route discovery process, it must increment its own sequence number. This prevents conflicts with previously established reverse routes towards the source of RREQ.
- ✓ Immediately before a destination node originates a RREP in response to a RREQ, it must update its own sequence number to the maximum of its current sequence number and the destination sequence number in the RREQ packet.

In order to ensure the incoming route information is up to date, the node compares its current sequence number value with the sequence number in the incoming RREP message. If the difference between currently stored sequence number and the incoming RREP sequence number is less than zero, then the information related to that destination in the AODV message must be discarded. The other circumstance in which a node may change the destination sequence number is when a link failure to the next hop is occurred. A node may change the sequence number in the routing table entry of a destination in one of the following conditions [26].

- ✓ It receives an AODV message with new information about the Sequence number for a destination node.
- ✓ The path towards the destination node expires or breaks.

2.1.4 Hybrid routing protocol

In MANET, a hybrid routing protocol combines the advantage of both proactive and reactive routing protocols to balance the delay and routing overhead [20]. In hybrid routing protocols, the nodes that have high level topological information maintains more routing information, which leads to more memory and power consumption. In hybrid routing protocol the whole network is divided into zones. Nodes in a zone are classified as interior or peripheral nodes. A node is said to be interior node if its distance from the source node is less than the radius of the zone and it is said to be peripheral if its distance from the central

node is same as the radius of the zone. The delay in delivery of data packet is avoided if the source and the destination node are in the same zone. If the source and the destination are present in two different zones, there will be a delay in delivering the packet as reactive routing protocol is employed and the routes are searched only on demand

2.2 Security in MANET

In MANET, vulnerability may occur due to entry and exit of nodes into/from the network. During the process of route discovery, the attacker node receive the RREQ packet and sends a false reply information to the source by generating a fake RREP control message before the original RREP arrives from any other nodes in the network. Some malicious node may enter into the network without informing other existing nodes and monitoring the process of communication between the nodes. Other malicious node present in the network successfully receives RREQ and replies with invalid path information by generating RREP control message and cause packet drop or create unnecessary packet delay [23].

The selfish node in the network, participates in the route discovery operation and drop the data packet of others node, to consume its hardware resources for its own communication only. Understanding attack's unique behavior helps to develop a mechanism to detect and prevent the impacts in the network.

In MANET, packets drop due to malicious behavior of nodes and other network related reasons. There are various reasons behind packet dropping in the network. Some of the causes for packet loss are [18]: Link failure, network congestiony, Lack of energy resourcesy, Overflow of the transmission queuey, Misbehavior of a malicious nodey, Too much bandwidth consumed by the attacker nodey, Selfish behavior of nodes.

2.3 MANET Protocol Stack and Associated **Attacks**

In MANET, protocol stacks [14] are used to make effective communication among wireless nodes. The data packet should pass application, presentationy, session,transport, network, data link and physical layers.

1. **Physical layer attacks**

In each layers of the MANET protocol stack, there are different kinds of attacks [14]. Eavesdropping and jamming are the major types of physical layer attacks [31]. In the eavesdropping attack, the attacker node makes itself as part of the communication path and becomes part of the route and starts tuning the receiver's frequency. Jamming attack is a type of denial of service attack which starts its damage by initially determining the communication frequency between the source and destination nodes.

2. **Data link layer attacks**

In the data link layer, nodes having malicious behavior, selfish behavior and nodes having a traffic analysis nature affects the data transmission activity. Attacks at this layer intentionally drop the data packet to conserve the node battery power and other computational resources. Some of the attacks analyze the flow of the traffic in the network to get important information about the network topology and routing mechanisms.

3. **Network layer attacks**

The network layer attack [31] affect the route discovery process and affect the data exchange between source and destination nodes. The black hole and wormhole attacks are common network layer attacks. The black hole attacker node advertises itself as having shortest route to destination node, attracts the data packet and finally drops it. The worm hole attack creates a tunnel between malicious nodes and results loss in legitimate route between source and destination nodes.

4. **Transport layer attacks**

The transport layer is responsible to create a connection between source and destination node and control the flow of data packets. In this layer, session hijacking and SYN flooding affect the communication by stopping the network services. The session hijacking is a kind of denial of service attack that affects the network by spoofing the victim node IP address. The SYN flooding [31] is also a kind of denial of service attack that opens a TCP connection with victim nodes.

5. Presentation Layer Attack

These attacks exploit vulnerabilities in how data is formatted, encrypted, compressed, or translated. Common attacks involve encryption flaws (e.g., SSL stripping to downgrade secure connections), data manipulation (altering compressed or encoded data to deceive applications), or malicious file formats designed to exploit processing vulnerabilities, leading to code execution.

6. Session Layer Attack

These attacks target the establishment, management, and termination of communication sessions. Examples include session hijacking, where an attacker takes over an authenticated user's session, or session fixation, where they force a user to use a known session ID. Denial-of-Service (DoS) attacks can also overwhelm session resources, preventing legitimate connections

7. Application layer attacks

In this layer, there are attacks that affect the application programs and functions of the operating system. The malicious code attack affects the applications and operating system. Another application layer attack is repudiation attack, that denies participation in parts of communication and brings a communication failure. In Table 2.4 [14], MANET layers, the attack type at each layer, modes of attacking and the result of attack were discussed.

LAYERS	ATTACK TYPE	MODE OF ATTACK	RESULT OF ATTACK
Physical	Eaves dropping	By receiver tunning to proper frequency	Reading messages by unintended receiver
	Jamming	By malicious node with known communication frequency	Prevents reception of legitimate packets
	Active interference	Blocks the communication channel	Change order messages
	Selfish misbehavior of nodes	Selfish nodes	Drops the packet

Data Link	Malicious behavior of nodes	Disrupts operation of routing protocol	Misdirects traffic
	Traffic Analysis	Topology information	Information to Unintended receiver
Network	Black hole attack	Fake optimum route message	Loss of confidential information on packet
	Wormhole attack	Tunnel between malicious nodes	Loss of safe route
	Rushing attack	Subvert route discovery process	Loss of safe route
Transport, Session and presentation	Session hijacking	Spoofs victim node IP address	Dos attacks
	SYN flooding attack	Open TCP connection with victim node	DoS attacks
Application	Malicious code attack	Viruses, worms	Attack to OS
	Repudiation attack	Denial of participation in parts of communication	Communication failure

Table 2.4: MANET routing attack at different layers

2.4 Attacks in MANET

In MANET, authentication is open and any malicious node may get routing taffic in order to disrupt the communication. Node misbehavior [7] refers to the deviation of node activity from normal routing and data packet forwarding. The routing protocol in MANETs is based on the active involvement of nodes to establish route between the nodes. The assumption is that all nodes are considered to be genuine nodes and cooperative to forward the incoming data packet. If one or more nodes act as malicious, it will disturb the routing operations and affect the performance of the network. Figure 2..4.1 [24], shows the major classification of MANET routing attack. In the following sections the MANET routing attacks are discussed in detail.

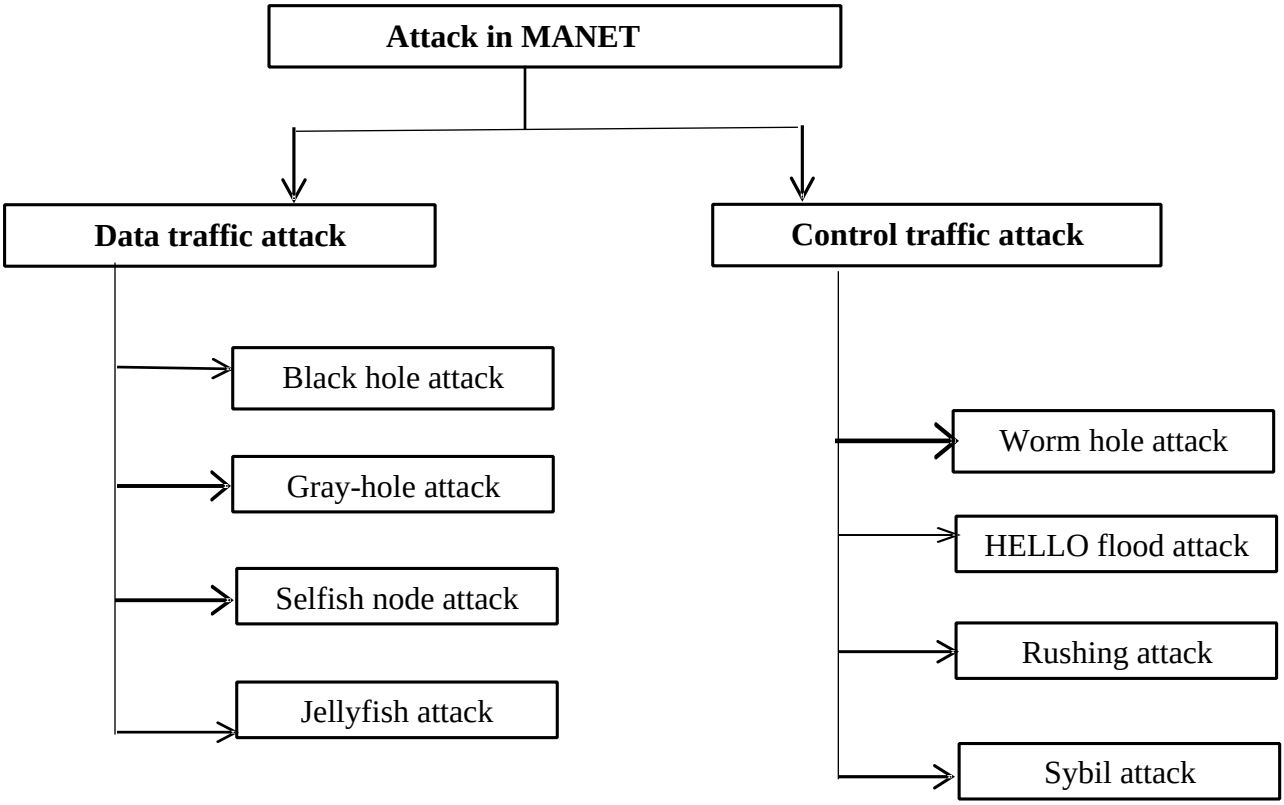


Figure 2.4.1: *Classification of routing attacks in MANET*

2.4.1. Data traffic attacks

Data traffic attack is an active attack that manipulates the operation of the network by changing and interfering with data. Data traffic attack drop or delay a data packet of other nodes which are forwarded through it to other neighboring nodes or destination node. The attacker node drops the entire or some of the received packet. This packet drop or delay will affect the quality of service and increases the end to end delay.

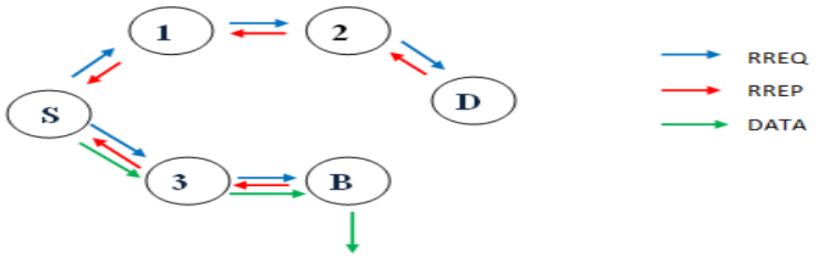
It totally changes the content of the packet transmitted by the source node. The attack may be internal or external attack. Internal attacks are carried out by nodes, which are actually the part of the network and external attacks are accomplished by nodes that do not belong to the domain of the network [31]. Some of the major data traffic attacks are discussed in the following sections.

a) Black hole attack

A black hole attack is kind of denial of service attack that occurs when a malicious node attacks the data by intentionally capturing and dropping it [7]. In MANET, any intermediate node having a fresh route to destination node can reply to Route Request (RREQ) sent by source node. Thus, a black hole node receive the RREQ packet and sends a RREP packet to source node claiming that it has a fresh route to destination node.

In protocol that use flooding for rout discovery like AODV, if any malicious node is present in the network it can simply send RREP packet with fresh value of destination sequence number. The destination sequence number generated by the black hole node is higher than the destination sequence number in RREQ. Thus, the source node assumes that the routing information is fresh and starts sending packets through malicious node to intended destination node [15, 37]. In a network, there are collaborative black hole attacks [6], in which a network having two or more number of black hole attacks.

Figure 2.4.2 [6], shows how a black hole node becomes part of the route and drops a data packet. Here node "S" is the source node, node "D" is the destination node and node "B" is the black hole node. The remaining nodes 1, 2, and 3 are normal nodes in the network. Node "S" broadcast a RREQ packet and node "B" claims that it has fresh route to the specified destination node as soon as it receives RREQ packets from node "S". Node "B" sends RREP to node "S" before any other normal nodes. In this way, node "S" assume that this is the active route to the destination node, ignore all other RREP message and send data packets through node "B". Finally, node "B" drops or delays all the data packets rather than forwarding it to the intended destination node "D".



F***igure 2.4.2:*** *Black hole attack in MANE*

b) Gray hole attack

In AODV, a gray-hole participate in the route discovery process and advertise itself as having a valid route to destination node, with the intention of intercepting packets. A gray-hole refers to a malicious node which initially responds to route requests normally, but when packets are sent to it for transmission, it drops those packets selectively. A gray-hole attack is a denial of service attacks that prevents the normal operation of MANET [9, 44]. In a given wireless network, a gray-hole may forward all the TCP packets and drop the UDP packets. A gray-hole attacker node may behave maliciously for some time duration by dropping packets and changed to normal behavior after a while [14]. In gray-hole detection mechanism, all nodes maintain their neighbor's data forwarding information. After a time interval each node checks if any neighbor were not communicated and then it starts the detection procedure by comparing the received and forwarded data packet [14].

c) Selfish node attack

Based on the nodes behavior, nodes in MANET are classified into cooperative node, failed node, malicious node and selfish nodes. Cooperative/normal nodes are genuine nodes in a network that actively participate in route discovery, maintenance and data forwarding but not in attacks. Failed nodes are nodes that do not participate in route discovery process. The node is changed into failed state when the node is out of the transmission range, low or no battery power, software and hardware problem. The malicious node participates in route discovery and launching of routing attacks. Finally, a selfish node participates in route discovery operation and route maintenance, but not in data packet forwarding.

Selfish node is a kind of denial of service attack, which drops data packet by refusing the data packet fully or partially. The selfish node is not cooperative to share its resources with the neighboring nodes to make effective communication. Receiving and forwarding other's node data packets consumes network bandwidth, local CPU time, memory, and energy of the selfish node [16]. Therefore, there is a strong motivation for a selfish node to deny packet forwarding to others, and interested to only packets addressed to it. In a given network, more than one selfish node may participate to drop the data packet. Once, the

selfish node participates in the network it will drop the data intentionally and affect the network performance [16]. In Figure 2.4.3 [16], node S is a selfish node, A is a source node, E is a destination node and the remaining node B, C and D, are normal node that cooperate in packet forwarding. The source node A broadcast a RREQ message to neighboring node

B, since node B have no fresh route to destination node, it will forward the RREQ packet to its neighboring node S and C. In this way the RREQ message reaches the destination node E and node E will initiate RREP message and send back to the source node. The selfish nodes behave as a normal node during the route discovery process and establish the path to destination. The source node select the route through node S having smaller hop count to destination. The route table of the intermediate node updated with a route information and a route to destination is established. Finally, the source node sends the data packet through S, and selfish node S drops all the incoming data packets from sender A.

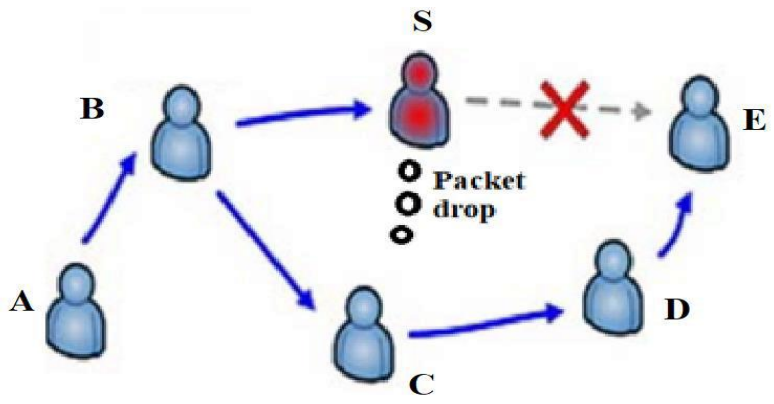


Figure 2.4.3*: Selfish node attack and data packet dropping in MANET*

- **Behavior of selfish nodes**

The selfish node attack may have the following characteristics depending on the type of selfish behavior of the node [10, 19].

- ✓ Drop the incoming data packets.

- ✓ Does not participate in route discovery and maintenance process.
- ✓ Do not reply or send to HELLO packet for route maintenance operation.
- ✓ Purposely postpone the RREQ and delay the route discovery operation.

- **Classification of Selfish nodes**

Based on the above characteristics, selfish node attack is classified into Selfish Node Type 1 (SNT1), Selfish Node Type 2 (SNT2) and Selfish Node Type 3 (SNT3). SNT1 participates in the route discovery and route maintenance operation, but do not participate in the data packet forwarding. SNT2 does not participate in the route discovery, route maintenance and data forwarding operation to save its resources. This type of node uses their energy only for the transmission of their own packets. SNT3 misbehaves based on the energy level of the node. The node behaves normally, if the node's energy lies between full energy E and a threshold T1. The node starts to behave as SNT1 when the energy level is between T1 and another threshold T2. If the energy levels lower than T2, the node behaves as SNT2 node. The value of T2 is less than T1 and the value of T1 is less than E. The selfish node attack mainly focuses on packet dropping, which leads to congestion in network [10, 21, 43].

- **Selfish node detection approach**

In this thesis, a detection and prevention mechanism is developed for SNT1. SNT1 participate in route discovery and route maintenance operation. The selfish node participates in route discovery operation by receiving and forwarding RREQ and RREP messages. The selfish node also participates in route maintenance operation sending and receiving HELLO packets periodically. Generally, the isolation and motivation approach are used to reduce the impact of selfish node attack in the network.

- **Selfish node detection and isolation approach**

The misbehavior of any node in the network is checked by its neighboring node. The existence of selfish node in the network is detected using a collaborative watchdog mechanism at each node. The watchdog at each node checks the misbehaving and data

dropping activity of the neighboring nodes. Watchdog buffer the recently sent packet and compare with the forwarded packet. If the data packet waits longer than the threshold timeout, the watchdogs increment a data forwarding failure. If the data forwarding failure exceeds a threshold, the node is considered as misbehaving node and announce to the source node about the behavior of the node. The working principle of watchdog is that "Number of incoming message equal to Number of outgoing messages". [47, 52].

In the proposed system, the trust value of each node is calculated by its neighboring node and gives information for the watchdog to identify the selfish node from the normal node. A trust value is calculated as the ratio of the number of packets forwarded by the neighbor to the number of packets received by the neighbor. If a node drops the entire incoming data packet (i.e. trust value=0), then MA detect and declare the respective node as a selfish nodes.

In Figure 2.4.3, node A is the source node and E is the destination node. The route path created by the routing discovery operation is A→B→S→E. The source node A start sending a data packet to node B. The watchdog at node A overhears the neighboring node B. Since, node B is a cooperative node it forward the data packet to node S. The watchdog at node B overhears node S, whether it forward the data packet or not. Since, node S is a selfish node in the network, it drops the data packet rather than forwarding to the final destination node E. The watchdog at node B detected node S is a selfish node and it forward the selfish information to its neighboring node. The selfish node is isolated or removed from the network; discard the current route and source node initiate a new route discovery process.
In [35], proposed a selfish node detection algorithm (SNDA) for identifying and isolation of selfish node from the network. In this selfish node isolation process, willingness of the node was asked before cooperating in the routing activities. In [36], an intrusion detection system monitoring (IDSM) approach was proposed for selfish node detection to monitor the entire network using the single cluster head node.

- **Selfish node detection and motivation approach**

This approach focuses on how to detect selfish nodes and motivate them to cooperate in data packet forwarding operation. The selfish node gets incentive when it forward others node

data packet. The selfish behavior of a node is detected using the watchdog protocol. Once the selfish node is detected by the watchdog, rather than isolating the node from the network, it is motivated and changes the node to cooperative node. In [34], an incentive mechanism and source credit checking is used in revival of selfish nodes in MANET**.** The proposed system involves in bringing back the selfish nodes to normal nodes.

d) Jellyfish attack

The jellyfish node participates in the route discovery process and becomes part of the active route. Once the malicious node is part of the active path, it starts dropping or delaying the data packets. The jellyfish attack can be classified as jellyfish recorder attack, periodic dropping attack and delay variance attacks [12].

In jellyfish recorders attack, packets arrive to the destination node in different order and send duplicate acknowledgement to the sender. The actual packet has reached at destination side; sender still believes that packet is lost. So sender retransmits the packet several times and that creates congestion in the network. In jellyfish periodic dropping attack, the attacker node drops the entire data or some portion of the data in some time interval. In jellyfish delay variance attack, the malicious node creates unwanted packet delay, packet loss and creates collusion. If acknowledgement of forwarding data packet is not received in threshold time interval, the network is congested with data packet and results data drop [14, 39, 40].

2.5 Mobile Agents (MA)

Mobile agents are software entities [8] that can overhear remotely or physically travel across a network, and perform tasks on machines that provide agent hosting capability. A mobile agent is a specific form of mobile code, within the field of code mobility [13]. MA acts on behalf of their sender and move independently between nodes in the network from source to destination. A mobile agent executes on a machine having route to destination. If a node does not contain the required services, the mobile agent can transfer itself to a new node. In a Multi-agent system, a set of intelligent agents interact with each other within an

environment [9]. A given mobile agent can cooperate with other mobile agent in the network environment and the end users have no control over the mobile agents [10].
Agents are the autonomous programs which sense the environment and acts upon the environment using its knowledge to achieve their goals. Agents are trained with previous intrusion, malicious activity by continuously monitoring the network [29]. An agent consists of three components: the program, the execution state of the program and the data [38].

2.5.1 Roles of Mobile Agent

MA are applied in a wide range of domains such as network management, E-ommerche, traffic control and robotics applications [17]. MA is also used in computing environment like grid computing, parallel computing, distributed computing and mobile computing [13]. In a distributed computing, MA creates a simple communication between clients and servers [17]. Ordinary routing protocols focused on finding shortest path between source and destination node rather than focusing on the quality of service path. Link failure due to high mobility of nodes and lack of sufficient energy of wireless nodes greatly affect the communication. To avoid link failure and to reduce communications overhead, stable routing is needed. The complexity of routing protocol due to its dynamic nature can be reduced with use of mobile agent. The mobile agents move to any node and update the routing table to transfer the processing task to the data source to reduce the network load.MA collects information about the network and computes the packet delivery ratio of the node. Then, it compares the packet delivery ratio with the predefined one and then gives responses to the source node accordingly [20, 43]. The Mobile Agent maintains entries <source node ID, destination node ID, HOP count, packet drop> on its data structure to perform the computation and comparison with threshold value. [30].

Characteristics of a mobile agent

a) Mobilithy: The mobile agent is a small unit of packets that migrate from the creator node to other nodes in a network to perform a certain task. This feature distributes the process and balance the network load.

b) Network awareness: Mobile Agents are capable of learning and searching for knowledge about their network topology and packet routing mechanisms. Agents transport

their state from one environment to another, without disturbing the previous data or state.

c) Communication intelligence: MA can communicate effectively with other agents, users and systems. In a given network there may be more than one mobile agent created by one or more mobile nodes to facilitate the communication process.

d) Autonomous: The mobile agents are autonomous, the agents are not only motivated by the outside actions initiated by the users but also they have internal events that decided their performance and behavior.

2.5.2 Security in mobile agents

Security is a very important concept in the growth and development of the mobile agent technology. Since, a mobile agent is moving from node to node, interact with other agents in the network, it is highly susceptible to attacks. A mobile agent authenticates itself to the node in the communication process. The agent encrypts and enforces a strong access control mechanism to protect its data from unauthorized access. To execute a mobile agent, the node must have access to its code, data and state. Thus, an agent that runs is exposed to security threats such as attack against denial of service, unauthorized access, confidentiality, availability and integrity.

In the propose system, nodes are clustered in the network and the MA node acts as a cluster head for overhearing and detection of selfish behavior of a node. The watchdog detects the selfish node using its trust value. The MA uses the watchdog mechanism to detect and prevent a selfish node. More than one MA nodes are designed to co-operate each other in the detection and prevention of a selfish node. When watchdog mechanism detects a selfish node based on a trust value, the MA node collects information about the entire network.
A given node may be watched by one or more MA nodes during data communication due to the mobility nature of the node. The MA nodes only detect data packet drop due to selfish behavior of a node. The MA cannot detect data packet due to other network related errors and other routing attacks. Here, the mobile agent will not move from node to node to detect the selfish behavior, rather it overhear the activity of the node and identify the node with a selfish behavior.

2.6 RELATED WORKS

Table 2.6 Related Works

Author(s) & Reference	Product/ Scheme Name	Attack(s) Detected/Prevented	Methodology/Key Features	Advantages	Disadvantages
K. Rama Abirami and M. G Sumithra [8]	Neighbor Credit Value Method	Selfish node behavior	Each node maintains and updates a neighbor credit table. Credit increases when a neighbor sends or forwards packets. Generates dummy data traffic from idle genuine nodes.	Successfully detects selfish neighbor nodes and avoids sending packets through them. Improves overall network performance. Prevents genuine nodes from being marked as malicious by generating dummy traffic.	Dummy data traffic generated in long intervals, potential impact on real traffic not discussed.
A. Meeran et al. [33]	Enhanced Watchdog Mechanism with Revival System	Selfish nodes	Node dropping packets beyond a threshold is selfish. Revival system uses incentive and source credit checking. Trust value (ratio of forwarded to received packets) differentiates nodes.	Does not remove selfish nodes; motivates them to return to normal using credit and incentive. Improves packet delivery ratio and throughput. Reduces the number of selfish nodes.	Selfish nodes are not fully removed from the network
V. Keerthika and R. C. Suganthe [34]	Collaborative Watchdog Mechanism	Selfish node information propagation delay	Collaborative watchdog propagates selfish information with route request message from the detecting node, not the source.	Only improves the end-to-end delay of the network.	It increase route overhead and latency
K. Anitha et al. [35]	Selfish Node Detection Algorithm (SNDA)	Selfish node	Calculates a selfish threshold value based on number of packets dropped. Classifies nodes as genuine, partially selfish, or selfish based on threshold.	Detects and prevents selfish nodes from the network.	High False Positives it may Incorrectly identifying genuine nodes as selfish
N. Veeraiah and B. T. Krishna [36]	Cluster-Head based Intrusion Detection System	Selfish node	Selects cluster heads for each cluster and a master cluster head to monitor. Master cluster head collects information to detect and eliminate selfish nodes.	Increases efficiency and QoS by reducing energy consumption.	Does not specify how the master cluster head evaluates the presence of selfish nodes in each cluster.

In MANET, routing protocol like AODV has its own weakness to secure data transmission between nodes. The attacker node affects the network during route discovery or data packet forwarding period. Research works done in the area of a mobile agent based single or cooperative black hole, gray-hole, DOS, jellyfish and selfish node attack detection and prevention mechanism are discussed.

A selfish node is a denial of service attack that participates in data packet dropping. The Selfish Node Type 1 (SNT1) receives and sends a RREQ, RREP and HELLO packet in the route discovery and route maintenance operations. Many proposed scheme detects the selfish nodes by modifying or enhancing the underlying MANET routing protocol. The selfish node is detected by modifying the routing algorithm and by implementing different technique to improve the performance of AODV. A collaborative watchdog detection mechanism and incentive based prevention of a selfish node is proposed by authors. In addition, research conducted on the area of mobile agent based selfish node detection is also discussed. To improve the performance of a network and maintain the quality of service, several security measures are proposed.

In most of the research works, the attacker node detection and prevention task is implemented by modifying the existing routing algorithm and propose a new method. The proposed work is simulated and the result is analyzed in network simulator software.

In section, literatures related with malicious nodes detection and prevention, selfish node detection, selfish node detection and prevention and mobile agent based selfish node detection and prevention mechanisms are discussed accordingly

A. Taggu *et al.* [1] proposed a reverse route (wtracert) application layer scheme for detecting a black hole attack in MANET using a mobile agent and next hop information. A node that suspects abnormal activity in the network run the wtracert to trace the path from the last working node to the final destination node. The intrusion detection process started when a node suspect's extremely high data packet drops in the network. MA was initiated and sent from the last normal working node and if the scheme detected any black hole node or broken link, it returned to the pervious node. The reverse route detects the existence of single or collaborative black hole attack successfully and improves the

detection time over the standard traceroute. The reverse route (wtracert) can be used to detect other types of MANET attacks in varying node mobility rate and different number of black hole nodes in DSR (Dynamic Source Routing).

Ani Taggu and Amar Taggu [7] proposed a TraceGray application layer intrusion detection scheme for DSR protocol based MANET using a mobile agent. The approach assumed that the next hop information is available to the node and the detection process starts after a node suspected extremely high packet dropping. A node that suspected abnormal activity started the detection process by sending a mobile agent and sent across the route discovered. After a MA traversed two hops successfully from the home context, it attempted to return back to the home context. The gray-hole node dropped data packets, since neither the source address nor the destination address in the data packet match with its own address. The timer expired and this timeout indicated a detection of a gray-hole in the network.

The TraceGray scheme detected single as well as multiple gray-holes in the MANET network, the MA overhead is recorded negligible and the detection time was also too small. If the next hop information in the node was not recent and route to destination node were not available, the timer was expired and route broken error is resulted. The gray-hole detection was only done at the application layer and other lower layers normal operations were not influenced. The attack in other MANET routing protocols like AODV is detected by this proposed scheme. TraceGray used one hop ping or MA transfer in order to avoid the packet loss in the intrusion detection process in MANET.

R. Lakhwani *et al.* [9] proposed agent based AODV protocol to detect and eliminate a black hole attacks in MANET. In the process of improving the AODV routing protocol algorithm, the sendReply() function is modified to include the flag on RREP packet and the recvReply() function is modified using a mobile agent to detect the malicious node in the network. The black hole attack was detected and prevented by the modified algorithm. The PDR (Packet Delivery Ratio) gave a significant improvement, the end-to-end delay gave a marginal rise and number of dropped packet greatly reduced in the proposed agent based AODV protocol as compared to the ordinary AODV protocol in the existence of black hole nodes in the network. The proposed agent based AODV detected and eliminated a single

and collaborative black hole attacks by putting a flag in the RREP packet. The malicious node detection and prevention in agent based AODV was easily done by modifying the sent and received function in the protocol.

C. Turguner [10] proposed a secure data transmission in MANET based network using mobile agents. It provides secured and fault tolerant communication environment between source and destination nodes. Symmetric and asymmetric encryption methods were used to encrypt the communication mobile agents and message. The sender node created private and public keys and encrypts the mobile agent. Before encryption, the keys were shared between sender and receiver nodes by Control and Authentication Service (CAS). The mobile agent formed a message block. The message and headers were inserted in to a hash function and a hash code was generated. The fault tolerance mechanisms masked the failures occurring in the system and created a fault tolerant communication in MANET. The results were not simulated in any network simulator and fault tolerance capacity was not checked by any network performance matrices. The authors only predicted that the proposed fault tolerance mobile agent based system improved the performance of the network. It was difficult to be ensured the impact of proposed system on the network performance and security.

V. Gaikwad and L. Ragha [11] proposed agent based mechanism to detect and avoid a cooperative black hole attack in AODV routing based MANET. The cooperative cluster gents work with a slight modification on AODV protocol and by adding routing information table. This approach effectively detected malicious nodes and mitigated the negative impact caused by a single and cooperative black hole attacks. The throughput, packet delivery ratio and end-to-end delay with different mobility rate were improved as compared with the original AODV protocol.

S. A. Mostafa *et al.* [32] proposed a Mobile Agent based AODV (MAAODV) protocol to calculate and select an efficient route to destination node from the route list in RREQ packet. The multi agent system was combined with AODV to implement MAAODV routing protocol. The source node received more than one RREP from the AODV route discovery and collect information about each route delay rate, number of hop count and energy level. The MAAODV keeps all routes and evaluate the quality of the route through a parameter (i.e., hop count, delay and energy) and choose the optimal route to forward a data packet.

The route having a highest energy, fewer hops and delay was set as a quality route and selected for data packet transmission. This work only focused on the availability of route to maximize the packet delivered to the destination nodes, not about the delay and the number of hops a packet passes to reach the destination node. The MAAODV improved the performance AODV protocol by reducing a link failure and increased the lifespan of the network. The proposed system was good to get a stable routing path between source and destination, but the end-to-end delay of the network were increased and affect the quality of service.

V. Keerthika and R. C. Suganthe [34] presented a watchdog mechanism to reduce the delay time for spreading the selfish information over the network. In the normal watchdog selfish node detection mechanism, once the selfish nodes were detected by the neighboring nodes, the node informs the source node and it spreads the selfish information to all the nodes. Once watchdog detected the selfish node, the collaborative watchdog propagates the selfish information with route request message to a network starting from the node that detected the selfish node, not from the source node. Only the end to end delay of the network is improved by the proposed system.

D. P. Sharma *et al.* [17] proposed a mechanism to discover the best possible path to improve the Quality of Service (QoS) in MANET using a mobile software agent by taking multiple QoS metrics like link stability, network congestion, delay & remaining energy. The proposed work was implemented in AODV routing protocol using mobile agent concepts. Path Selection Index (PSI) cost was added to the AODV RREQ packet format and sent to destination node using the available path in hop-by-hop manner to all one-hope neighbors. In the proposed system quality of service metrics, i.e., link stability, was calculated using RSS (Received Signal Strength); congestion was measured by the ratio of number of packets in the queue to buffer size and remaining energy was achieved by subtracting the consumed energy from the total energy. Queuing delay was considered as the waiting time in the queue for a packet before it was sent out onto a link. The PSI cost of each path was calculated while a source node discovers a route using the above combined QoS parameters. Based on the PSI value in the RREP message, a source node selected the optimal path to destination node. The proposed model improved the path cost, hop and path distance over the

conventional hop-by-hop AODV routing protocol. Adding PSI in RREQ and calculating its cost increased the route discovery overhead. The work indicated that the model improved more than one quality of service parameters at route discovery period.

A. K. Jain and A. Choorasiya [2] proposed a security enhancement for AODV routing protocol in MANET. The proposed AODV protocol detected and prevented the impact of black hole attack, gray-hole attack and DOS attacks. The sender node generated a dummy packet and sent it to the destination node. Using this packet, number of sent, received and forwarded packets were calculated for each node in the network. According to the packet delivery ratio, the threshold (variance) value was estimated and assigned to each node. Using the estimated variance, a trust value was evaluated for each node. By making several comparisons, the proposed algorithm assigned a weight for each node in the network. The selection of a next hope node was done by checking the maximum weight or trust value. If the weight of a given node was below the threshold limit for three times, the node added to a black list and was permanently blocked. The end-to-end delay of the proposed AODV protocol was lower than the original AODV. The PDR and throughput of a proposed AODV protocol is higher than the original AODV protocols. The threshold evaluation and assignment mechanism was not clearly specified in how much time a dummy packet was sent, received and forwarded. The proposed system was not specified whether collaborative attack is prevented or not. The enhanced system successfully detected and prevented the effect of black hole; gray-hole and DOS attacks in AODV based MANET.

S. R Deshmukh *et al.* [6] proposed AODV based secure routing against black hole attack in MANET by modifying the routing table and setting a validity bit in the RREP message. The system used the normal AODV route discovery process and the final destination nodes set a validity bit to the RREP packet of the routing table and send back to the source nodes. The intermediate nodes in the active path checked the validity bit set by the destination node, updated its routing table and forwarded to the source node. If the validity bit in the RREP message was equal to one, the message is forwarded back to source node and data packet transmission started. The proposed system prevented the black hole node from participation in the network and legitimacy of route is confirmed. The RREP message

without a validity bit were discarded, but the black hole node that generated a fake route was not blacklisted in the network, it sent RREP message continuously to the source node and creates a network overhead. The proposed method prevented the participation and effect of malicious node with minor modification to the routing table and with negligible overhead resulted due to validity bit.

P. Pooja B *et al.* [12] presented jellyfish attack detection and prevention mechanism in AODV based MANET. The proposed mechanism detected and prevented the impact of jellyfish delay variance attack by considering sent and received time of each packet, threshold time of packet and load of the network. The approach calculated the difference of sent and received time and compared with the threshold time value. If the difference was greater than the threshold value, delay is said to have occurred due to congestion or availability of jellyfish nodes.

The system checked the delay resulted due to the network congestion by comparing the load with a predefined load threshold. The approach compared normal AODV, AODV with jellyfish attack and proposed AODV. The PDR, throughput and delay of a proposed AODV is greatly improved over the AODV with existence of jellyfish attacks. Thus, the approach derived the packet delay occurred due to the jellyfish attack or network congestions. If delay occurred due to network congestion then, disabled retransmission of same packet and enabled selective acknowledgement. If the delay occurred due to jellyfish node, the system discarded the jellyfish node and sends a reply. A genuine Multipoint Relay (MPR) node was used for forwarding data packets. The approaches were not detected and prevented the impact of reorder and periodic dropping jellyfish attack in MANET.

CHAPTER THREE

PROPOSED WORK and METHODOLOGY

This chapter establishes the foundation for performance evaluation and subsequently delves into the specifics of a simulation environment designed to assess the efficacy of our proposed lightweight selfish node detection and prevention system within an AODV-based MANET.

This research examines how well a network performs. In MANETs, it's important to use common network performance metrics to ensure good quality of service. This paper proposes a mechanism to find and stop selfish nodes that are trying to avoid sharing resources (MA (mobile agent) based selfish node attack detection and prevention). To see how well this approach works, we'll evaluate it using various network performance measurements.

3.1 Network Performance Metrics

Measurements for network performance include packet delivery ratio (PDR), the time it takes for data to get from start to finish (end-to-end delay), throughput (data transfer rate), routing overhead (extra network traffic caused by routing), packet drops, and the number of packets received. We'll first discuss how these network performance metrics relate to security in MANETs, and then we'll simulate the results using the NS-3 network simulator.

I. Packet Delivery Ratio/Fraction

PDR is one of the network performance parameters used to measure how much of the data packet sent from the sender node reaches to the destination node. In other words, it defines the percentage of data packets successfully received by the destination node divided by the number of data packets generated and sent by the source node [18, 24].

$$\text{Packet Delivery Ration (PDR)} = \left[\frac{\text{No of Packet received}}{\text{No of Packet Sent}} \right] * 100 \qquad (1)$$

II. Average End to End delay

The end-to-end delay refers to how long it takes for a piece of information (a packet) to travel from the program that sent it (source application) to the program that receives it (destination application). It's basically the average amount of time between when the information is sent and when it's successfully received. [24, 27]. In other words, network delay refers to the time the first bit of the packet is put on the medium at the source node to the time the last bit of the packet is received at the receiver side. It includes all delay resulted due to queuing, network congestion, propagation delay, link failure, route discovery, etc.

$$\text{Avarage end to end delay} = \frac{\sum(\text{Time received} - \text{Time sent})}{\text{Total No of sent packets}} \quad (2)$$

The formula we presented, ∑(Time received - Time sent) / Total No. of sent packets, represents the average end-to-end delay for a single packet. It calculates the sum of the difference between "Time received" and "Time sent" for each packet, and then divides it by the total number of packets sent.

Total No. of sent packets: This is used for averaging. By dividing the total delay by the number of packets sent, we get the average delay per packet, which is essentially the average end-to-end delay.

III. Throughput

Throughput is the average rate of successful message delivery from source to destination node over a communication channel per unit time [2]. It refers to the packet delivered in total propagation time in mbps or data packet per second. It can be influenced by the type of transmission medium, network congestion, network delay, packet loss, link failure, protocol operation and routing attacks. In the proposed system, the throughput is improved by avoiding the effect of selfish node attack.

$$\text{Throughput} = \frac{(\text{No of packets received} * \text{Packet size})}{\text{Total Transmission time}} \quad (3)$$

IV. Normalized Routing Load (NRL)

The efficiency of a routing protocol can be measured by its routing load. This is calculated as the total number of routing control packets sent by all devices in the network divided by the number of data packets that successfully reach their destinations. These control packets are necessary for tasks like keeping routes up-to-date, finding new routes, and ensuring network security. However, they add extra information to data packets, increasing the overall network traffic and potentially slowing down communication [19].

$$\text{NRL} = \frac{\text{Total No of routing control packets}}{\text{Total No of received data packets}} \qquad (4)$$

V. Ratio of Received Packets

The ratio of received data is a key network performance parameter used to measure the success rate of data transmission. It represents the percentage of data packets successfully received by the destination node, divided by the total number of packets sent by the source node. High Ratio (close to 1 or 100%): This indicates efficient transmission. Most packets are reaching their destination.

$$\text{rate} = \frac{\text{Total number of packet received}}{\text{Total number of sent packet}} * 100$$

VI. Packet drops

In a given network, a packet is dropped of several reasons before it is received by the destination node. Packet drop is calculated by subtracting the number of data packets received at destination node from the number of data packets sent by source node [19]. Packet drop in a network can occur due to a variety of reasons that impact the efficiency and reliability of data transmission. Some of the reasons are:

- One primary cause is network congestion, where excessive traffic exceeds the capacity of network devices, leading to dropped packets as routers and switches become overwhelmed.
- Faulty hardware, such as malfunctioning routers, switches, or cables, can also introduce errors that result in packet loss.
- Additionally, software issues, including misconfigured network settings or outdated firmware, can disrupt proper data flow.
- Network interference, particularly in wireless networks, caused by physical obstacles or electromagnetic interference, can degrade signal quality and cause packet drop.
- Lastly, security measures, such as firewalls and intrusion prevention systems, may intentionally drop packets perceived as threats, impacting legitimate traffic if not properly configured.

In Table 3.1, the common network related reasons for data packet drop are presented.

Reason of the event	Description
END	Packet drop due to end of the simulation time
DUP	The data drop due to MAC (Media access control)
ERR	Drop due to MAC packet errors
RET	Drop due to MAC retry count exceed
STA	Drop by MAC invalid state
BSY	Drop due to MAC busy
NRTE	Drop RTR, No route to destination
TTL	Drop due to TTL (Time To Live) reached zero
TOUT	Packet is dropped due to timeout
IFQ	Packet drop due to no space in IFQ (Interface Queue)
ARP	Drop due to ARP full

Table 3. 1: *Network related data packet drop reasons in MANET*

3.2 Highlighting the Simulation Setup

The simulation of the proposed lightweight system will be done on Network Simulator (NS-3) software to achieve the research objectives and answer the basic research questions**.** NS-3 is a discrete event driven simulator that helps to analyze the performance of routing Protocols over routing attacks. It is an open source, flexible and modular nature that combines C++ and Object-oriented Tool Command Language (OTcl) programing language. NS-3 is used to implement a mobile agent based system to detect and prevent a selfish node attack in AODV routing. It enables to compare the performance of AODV routing protocol with the presence of selfish node attack and mobile agent based selfish node prevention system using different network performance metrics. Ubuntu 20.4 LTS 64-bit open source Linux operating system is installed on Intel core i7, 8^{th} generation, 1.8 GHZ processor, 8GB RAM and 1TB hard disk (190 GB for Ubuntu 20.4) laptop physical machine. NS-3 and GNU plot (GNU plot is a powerful, command-line driven, and open-source utility used for visualizing simulation results) are installed and configured on Ubuntu 20.4 operating system. A mobile agent based selfish node detection and prevention mechanism is simulated with different numbers of nodes (10, 25 and 50) by applying different number of selfish node attack (3, 6 and 6) respectively.

In the simulation environment we use MA nodes, normal nodes and selfish nodes to simulate the proposed system. The MA nodes are the master nodes that monitor the activity of its neighboring nodes for selfish behavior. The MA nodes communicate with each other like a cluster head to get the total data packet dropped by each selfish node, but they cannot send and receive any kinds of data packets. The normal nodes are individual cooperative nodes that participate in route discovery, maintenance and data packet forwarding. The selfish nodes (SNT1) are nodes with a selfish behavior that participate in the route discovery process but not in data forwarding.

3.3 Comparison of different network simulation toolkits

Each simulator has its own advantages and disadvantages based on the studies conducted. So Selecting the right simulation toolkit is crucial.We compared NS-3, OPNET, OMNeT++, QualNet, and GloMoSIM, and finally explained our choice of NS-3.

1. OPNET (Optimized Network Engineering Tool)

OPNET a Discrete-event network simulator is in designed for commercial.It not open source.This simulator uses C and C++ among its strengths it has a comprehensive GUI (graphical user interface) it offers a powerful graphical user interface for network design, configuration and analysis.

it Provides a vast library of pre-built models for various protocols and devices, making it quick to set up common scenarios. And widely used in industry for network planning, analysis, and optimization. And it has Strong capabilities for data visualization and performance analysis.

Among the weakness of this simulator, it has high cost and often prohibitive for academic research budgets.It Limits transparency and makes it difficult to deeply modify or understand the underlying implementation of models. It is less flexible for Novel Protocol Development: While extensible, developing entirely new protocols or making fundamental changes can be more challenging compared to open-source alternatives.

2. OMNeT++ (Objective Modular Network Testbed in C++)

OMNeT++ is a modular, component-based, discrete-event simulation framework. It has a free license for non-commercial use and exclusively utilizes C++. Among its strengths, it emphasizes component-based design, which promotes code reusability. It also comes with an Eclipse-based IDE (integrated Development Environment) and a rich graphical runtime environment, making it easier to trace and debug simulations. INET, a popular framework built on OMNeT++, is specifically designed for internet-related simulations and provides a wide range of network models.

Regarding its weaknesses, OMNeT++ offers fewer pre-built network models compared to OPNET. While INET provides many, setting up complex network scenarios from scratch might require more effort than with OPNET. Furthermore, it requires C++ knowledge and an understanding of its component-based architecture.

3.QualNet (Qualitative Network)

QualNet is a discrete-event network simulator, recognized for its high-fidelity models and accuracy in predicting network performance, particularly in wireless and mixed-platform networks. As a

commercial product derived from GLoMoSIM, it is not open source and primarily uses C++ for its underlying architecture.

Among QualNet's key strengths are its robust design for simulating large and heterogeneous networks, its professional support and training, and its ability to merge with other simulation tools and real hardware. However, its commercial nature presents significant weaknesses, including a high cost that can be prohibitive for academic use, and limitations in transparency and deep customization compared to open-source alternatives.

4. GloMoSIM (GLObal MObile Information System Simulator)

GloMoSIM is a library-based simulator capable of both sequential and parallel execution. It is an open-source project, though its maintenance is currently less active. Developed using the C programming language, GloMoSIM's primary strength lies in its specialized design for wireless and mobile communication systems, including its inherent support for parallel simulation. However, its weaknesses include a reduced active development status, with QualNet serving as its commercial successor. Furthermore, its scope is somewhat limited, as it primarily focuses on wireless networks and may not be as comprehensive for diverse network scenarios compared to other simulators. Lastly, being an older technology, GloMoSIM may lack support for newer protocols and technologies.

5. NS-3 (Network Simulator 3)

NS-3 is a Discrete-event network simulator it is Open-source . It primarily uses C++, with Python scripting interfaces.

Strengths:

-High Realism**:** Designed to be highly realistic, including frameworks for using real application and kernel code. Supports integration with virtual machine environments and testbeds.

-*Open Source and Community Driven***:** Fosters a large and active community of developers and users, leading to continuous development, bug fixes, and a wide range of available models.

-*Detailed Packet-Level Simulation***:** Provides fine-grained control over network events at the packet level.

-*Extensibility***:** Highly extensible due to its C++ core, allowing researchers to develop new protocols, models, and functionalities.

-*Tracing and Analysis***:** Offers a robust tracing subsystem for analyzing simulation data

-*Scalability:* Capable of simulating large-scale networks with millions of nodes.

-*Modular Architecture***:** Components are designed modularly, promoting reusability and easier development.

Weaknesses:

- Requires strong C++ programming skills.

-*Limited GUI Support:* it's primarily command-line driven,

Why we prefer NS-3 is in our Study

Given network simulation toolkits, NS-3 is often the preferred choice for academic research due to several key advantages:

Open Source and Free: is open-source software , NS-3 is freely available for use, modification, and distribution. This is a significant advantage for academic institutions and individual researchers who often have limited budgets. Commercial simulators like OPNET and QualNet are prohibitively expensive.

High Realism NS-3 is designed for network research, aiming to create very realistic models of networks and devices. It can run actual program and system code, so its simulations are very similar to real-world networks. This means our research findings will be more trustworthy and useful.

Flexibility and Extensibility: NS-3 is written mainly in C++ and also works with Python, making it super flexible.

- **Scalability for Diverse Research Needs:** NS-3 can handle simulations ranging from small, simple topologies to very large and complex networks, making it suitable for a wide array of research topics, including: Wireless networks (Wi-Fi, LTE, 5G, IoT, WSNs) ,
  -Wired networks (TCP/IP performance, routing protocols)
  -Mobile ad-hoc networks (MANETs) and vehicular ad-hoc networks (VANETs)
  -Software-Defined Networking (SDN) and Network Function Virtualization (NFV)

- **Academic Acceptance:** NS-3 is widely recognized and accepted within the academic community as a robust and reliable research tool. Research papers based on NS-3 simulations are commonly published in reputable conferences and journals.

3.3 Simulation environment

The lightweight mobile agent based selfish node detection and prevention will be implemented and simulated on NS-3. More than one node sends and receives a data packet in the simulation period in mobility environment. A given node can be monitored by one or more MA node at a time due to the mobility nature of the nodes. The same simulation parameters and values will be used for the detection and prevention module.

The data packet drop due to the selfish node attack is identified and recorded. The value of the parameters for the detection and prevention module is similar for 10, 25 and 50 nodes in the simulation environment. Table 3.2 presents the common simulation parameters that will be used for detection and proposed prevention methods for all scenarios.

Parameters	**Value**
Simulator	NS-3
Routing protocols	AODV
Simulation Time	100s
Simulation Area(m)	1000 X 1000
Number of Nodes	10, 25, 50
Propagation model MAC type	TwoRayGround Mac/802_11
Channel Type	WirelessChannel
Transport Layers	TCP, UDP
Traffic Type	CBR, FTP
Packet Size	512 bytes
Data Rate	5 mbps
Mobility rate Type of Attacks	Random Selfish node attack

Table 3.2: *Common simulation parameters for detection and prevention methods*

The simulation parameter values were determined based on a combination of factors, including:

- ✓ **Standard Practices and Realism:** Many parameters were chosen to align with common practices in MANET simulation studies and to reflect realistic network conditions. For instance, NS-3 is a widely accepted and robust simulator for network research. AODV is a well-established reactive routing protocol, and 802.11 is the prevalent MAC standard for wireless ad-hoc networks.
- ✓ **Scalability and Performance Evaluation:**
  - **Number of Nodes (10, 25, 50):** These values were specifically selected to evaluate the scalability of the proposed lightweight approach across small, medium, and larger network densities. This allows for observing how throughput and packet delivery ratios change with increasing network size.
  - **Simulation Time (100s):** This duration was deemed sufficient to allow the network to stabilize, for routing paths to be established, and for the effects of selfish node attacks to manifest and be observed over a reasonable period.
  - **Simulation Area (1000 x 1000 m):** This provides a sufficiently large area for nodes to move and for diverse network topologies to form, while remaining computationally manageable.
- ✓ **Comprehensive Traffic and Mobility Scenarios:**
  - **Transport Layers (TCP, UDP) and Traffic Types (CBR, FTP):** Including both TCP (reliable) and UDP (unreliable) transport layers, along with CBR (constant bit rate) and FTP (bursty) traffic types, allows for a comprehensive evaluation of the lightweight approach's performance under various application requirements and traffic patterns.
  - **Packet Size (512 bytes) and Data Rate (5 mbps):** These are typical values for wireless network traffic, helping to simulate realistic data transmission conditions.
  - **Mobility Rate (Random):** Random mobility is a standard model for MANETs, simulating the dynamic and unpredictable nature of node movement in ad-hoc environments, which is crucial for testing the robustness of the routing and security mechanisms.

- ✓ **Focus on Attack Analysis:**
  - **Type of Attacks (Selfish Node Attack):** This parameter is directly tied to the thesis objective, focusing the simulation on the specific threat model being addressed by the lightweight approach. The number of selfish nodes (as discussed previously) was varied to test the system's resilience against different attack intensities.
- ✓ **Propagation Model (TwoRayGround) and Channel Type (WirelessChannel):** These models provide a more realistic representation of radio signal propagation in a typical outdoor environment, accounting for direct path and ground reflections, which is important for accurate network performance analysis.

In summary, the parameter values were chosen to create a controlled yet representative simulation environment that enables a thorough and meaningful evaluation of the proposed lightweight detection and prevention mechanism for selfish node attacks in MANETs.

In Table 3.3, the simulation parameters and its values that vary with the number of nodes (10, 25, 50) and the mobile agent code and data size for the prevention system is given. Here, the mobile agent code size and data size indicates the size of MA data and code in the proposed prevention system.

To evaluate the performance of the proposed detection and prevention system, it is simulated with varying number of nodes and varying number of selfish nodes in the network. We select 10, 25 and 50 node scenario to analyze the behavior of the network in linear approach. In the proposed detection and prevention system we use 3 selfish nodes for 10 nodes scenario and 6 selfish nodes for 25 and 50 numbers of nodes scenario. In the detection and prevention simulation environment, the selected selfish nodes are enough to give coverage in the simulation environments and analyze the impact of selfish node in MANET.

Parameters	Number of nodes		
	10	25	50
Mobile agent code size	8.5 KB	13.5 KB	20.3 KB
Mobile agent data size	5.8 MB	19.3 MB	73.6 MB
Number of selfish node attack	3	6	6
MA nodes	4,5	2,3,6,14,15	1,4,17,22,24

Table 3.3: *Specific simulation parameters which varies with the number of nodes*

The criteria for selecting the simulation parameters were primarily driven by the need to thoroughly evaluate the proposed lightweight approach's performance, scalability, and robustness under varying network conditions and attack intensities. Specifically:

- **Number of Nodes (10, 25, 50):** These values were chosen to represent small, medium, and relatively larger network densities. This allows for the assessment of the protocol's scalability and its performance characteristics (throughput, packet delivery ratio) as the network size increases, demonstrating its applicability across different MANET deployments.
- **Mobile Agent Size:** This typically refers to the data payload or complexity of a single mobile agent. While not explicitly provided, it would generally be chosen to reflect realistic mobile agent overheads and their impact on network resources.
- **Number of Data Size:** This parameter would typically be selected to simulate varying traffic loads and data volumes. This allows for an evaluation of the system's efficiency in handling different amounts of legitimate data alongside the detection and prevention mechanisms.

- **Number of Selfish Node Attacks (3, 6, 6 for 10, 25, 50 nodes respectively):** These numbers were selected to simulate different *proportions* or *densities* of selfish nodes within the network. This allows for a comprehensive evaluation of the detection and prevention mechanism's effectiveness and resilience against varying levels of attack intensity, from a moderate presence to a more significant proportion of malicious actors.
- **Number of Mobile Agents (MA) (6, 14, 15, 17, 22):** These specific values were chosen to investigate the impact of mobile agent density on the overall system performance. By varying the number of MAs, the aim was to analyze the trade-offs between the overhead introduced by the agents and their effectiveness in terms of detection speed, accuracy, and the overall improvement in network metrics like throughput and packet delivery ratio.

3.5 Selfish node detection and proposed prevention system

3.5.1 Selfish node detection system

The normal AODV routing protocol initiates a route discovery process by flooding a RREQ packet and receives a RREP packets to create a route. The source node sends its data packet to destination node through the intermediate nodes. Nodes which are configured as selfish nodes participate in the route discovery process and become part of the active route. The selfish nodes drop all the routing data packets of other nodes' data to save their resources. Detection and identification of nodes with a selfish behavior is done by analyzing the trace file using AWK script after data transmission takes place and routing information is stored (An AWK script is a program written in the AWK programming language).The detection process identifies the selfish nodes and packet dropped by the respective selfish nodes. The detection mechanism is simulated with varying number of nodes and selfish nodes.

3.5.2 Proposed selfish node detection and prevention system

In MANET, once the route to destination node is discovered and established by the ordinary AODV routing protocol, a data transmission is started. Due to several data packet dropping factors, the route cannot give guaranty for successful delivery of the data to destination node. During data transmission, the trust module in the intermediate node calculates the trust value of the neighboring nodes, identify the misbehaving nodes and sends the information to watchdog method. Trust value is calculated as the number of packet forwarded to the neighbor divided by number of packet received from the neighbor node. Since, the SNT1drops all the incoming data packet from the neighboring node, the trust value of the selfish node is zero.

The watchdog mechanism uses the misbehaving information from trust mechanism and watches the selfish behavior of the node using different parameters. The MA uses the watchdog mechanism to watch the mobile nodes for selfish behavior/data packet dropping reasons. MA uses collaborative watchdog mechanism and trust value to detect the selfish node. Due to the mobility nature of a node at the time of data transmission, data may be forwarded from source to destination at different routes. The MA node starts watching the data drop at the neighboring node, identify the reasons for data packet drop and prevent the selfish node permanently from participating in the network.

In figure 3.1, the lightweight AODV routing algorithm uses a mobile agent to detect data packet drop using the watchdog mechanism. The route discovery process will be done using the ordinary AODV routing protocol and a route to destination is established.

A source node sends the data packet to destination and MA starts overhearing the packet drop at each node in the active path using the watchdog methods. A trust value will be used to identify the node as normal node or selfish node. The proposed systems overhears data drop in the neighboring nodes using a watchdog method and trust value, declare the node as a selfish node, inform the selfish behavior of the node to its neighboring nodes and prevent the node permanently from participating in the network.

The proposed prevention system can detect and prevent multiple selfish nodes in a network.

Finally, the MA discards the current route information and the source node initiates a new route discovery operation.

Selfish node detection and prevention procedures:-

When a source node is ready to send data to destination, the ordinary AODV routing protocol initiates a route discovery process by forwarding a RREQ packet. On receiving an RREQ, the destination node or intermediate node returns an RREPmessage. The routing table of nodes in the routing path updates their routing table information accordingly.

1. The source node forwards the packet to its first intermediate node in the path, the intermediate node receives the packet and forwards to the next intermediate node and so on until it is received by the destination node.
2. The watchdog method identifies the behavior of the neighboring node using the trust value of each node.
3. The MA overhears a group of mobile nodes in the network for the existence of selfish behavior in the group in a cooperative manner. The MA uses the watchdog echanism to detect selfish nodes in the network.
4. When a selfish node is detected and prevented, the data packet is received by the destination node, and it sends an acknowledgemetn to the source node and waits for the next packet. The data packet sent by the source node can be dropped by other network errors and other kinds of routing attacks
5. If MA detects a selfish node in the active path, the node is declared as a selfish node and MA removes it from the network. The nodes' selfish behavior information is sent to all neighboring nodes.
6. The route information related to the selfish node is removed from the routing table of source nodes in the active path and source initiates a new route discovery process to find a legitimate route to the destination node.

7. If source nodes have only one route to the destination node i.e. through a selfish node, after the removal of a selfish node, the sender waits until neighboring nodes exist in its radio frequency range. MANET topology is dynamic due to mobility nature of the nodes and source wait for new selfish node free route to forward the data packet to the destination node.

Flow chart for selfish node detection and prevention system

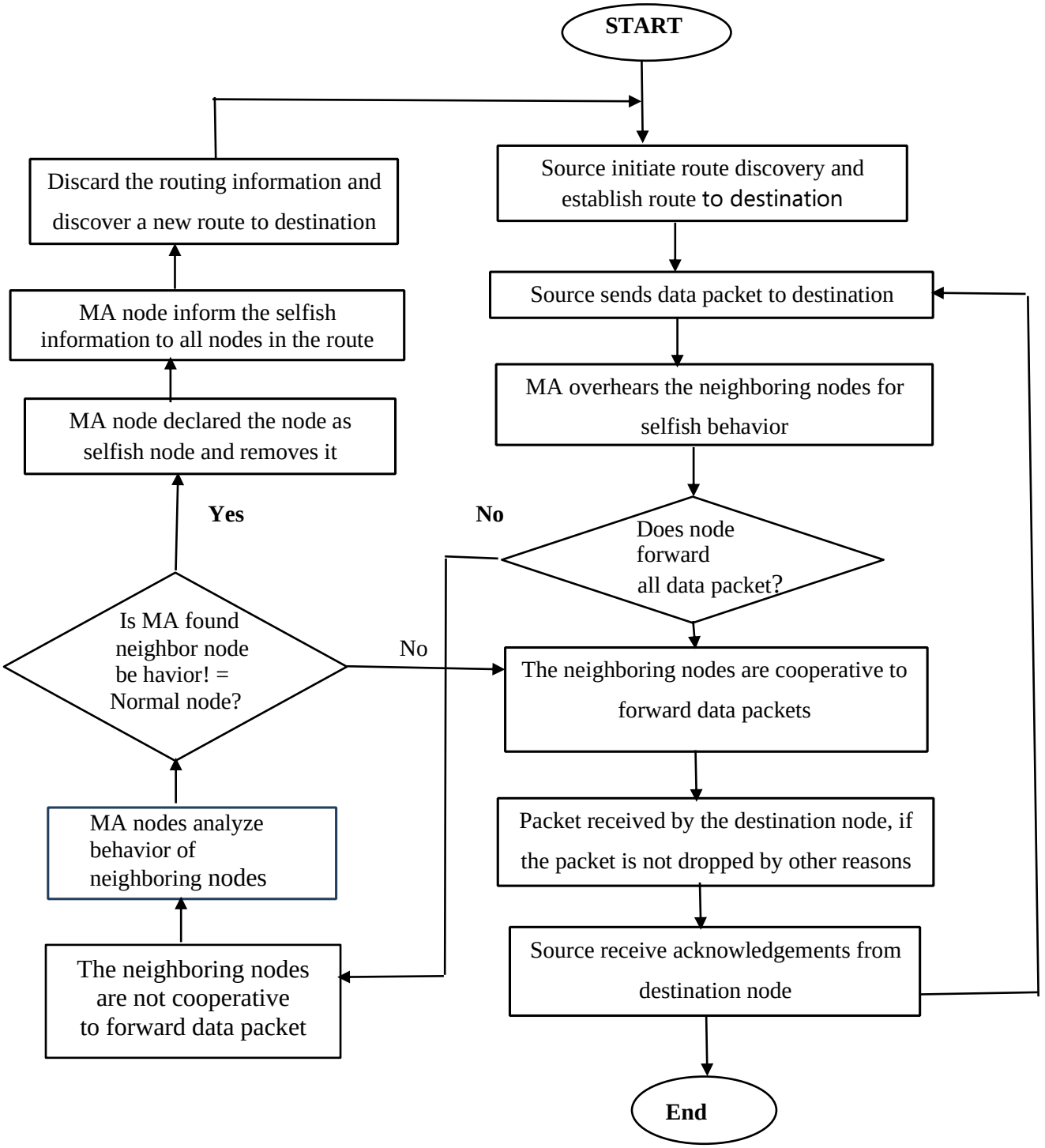


Figure 3. 1: *Flow chart for selfish node detection and prevention system*

Summary

To detect and prevent a selfish node from MANET based AODV routing protocol, the routing algorithm of AODV routing protocol is modified using MA based trust and watchdog mechanism. To check the result and the performance of the proposed system, experimentation based methods will be used and the detection and prevention system is simulated on NS-3 network simulator.

In the simulation environment, the selfish node detection and the selfish node detection & prevention scenarios are simulated with 10, 25, 50 number of nodes and 3, 6, 6 number of selfish nodes respectively. The detection systems detect the selfish node at the end of the simulation by analyzing the behavior of the node in the active route using the information in a trace file. To analyze the behavior of each node in the communication process and identify the misbehaving node, AWK script is will be used to perform the analysis.

During the proposed lightweight selfish node detection and prevention system, the detection and prevention of selfish node will be done at real time or during data transmission using MA.The MA uses the selfish information about each node in the active path using watchdog and trust method to detect, declare and block the selfish node from the network. Once the selfish node is removed from the network, route information related to the selfish node is removed and a new route discovery operation is initiated.

CHAPTER FOUR

RESULTS AND DISCUSSION

In this chapter, performance of the proposed detection and prevention system is evaluated. The result of selfish node detection and prevention system is simulated for 10, 25 and 50 numbers of mobile nodes. In the following section, the simulation result for the performance parameters are presented in line or bar graph and the results obtained are discussed.

4.1 Performance Evaluation of the proposed detection and prevention system

a) Number of Packets Received

The total number of successfully received data packet is influenced by selfish behavior of the node, network related problems and other MANET routing attacks. The number of successfully received packets by the destination node is presented in Figure 5.1. The amount of data received in the network after the implementation of the proposed prevention system is better than the data packet received during presence of selfish node. The data received is compared with the total number of sent data packet from source nodes to destination nodes in the simulation period in all scenarios. Table 5.1 shows the total number of packets sent by the source node, the number of packets received by the destination node and the packet drop in the proposed selfish node attack. The ratio of number of packets received to the number of packets sent is better in the selfish node detection and prevention method is better than in the normal AODV routing during selfish node detection.

Number of Nodes	Selfish node detection system				Proposed selfish node detection and prevention system			
	Sent packet	Received packet	Packet drop	Packet drops by SNT1	Sent packet	Received packet	Packet drop	Packet drop by SNT1
10	1,043	370	673	485	2,295	1,651	644	0
25	2,101	454	1,647	548	4,645	3,193	1,452	0
50	3,353	882	2,471	1604	5,736	4,424	1,312	7

Table 5.1: *Data packet sent, received and drop with different number of nodes*

As shown in above table, when number nodes are 50 and selfish node is presented in the system, out of 3,353 packets, only 882 packets (26.30) are successfully received at destination and after implementation of the proposed prevention method from 5,736 data packet send by the source node, 4,424 packets (77.13%) are received by the destination. In general, the proposed lightweight mobile agent based selfish node detection and prevention mechanism improves the total number of packets received by the destination node.

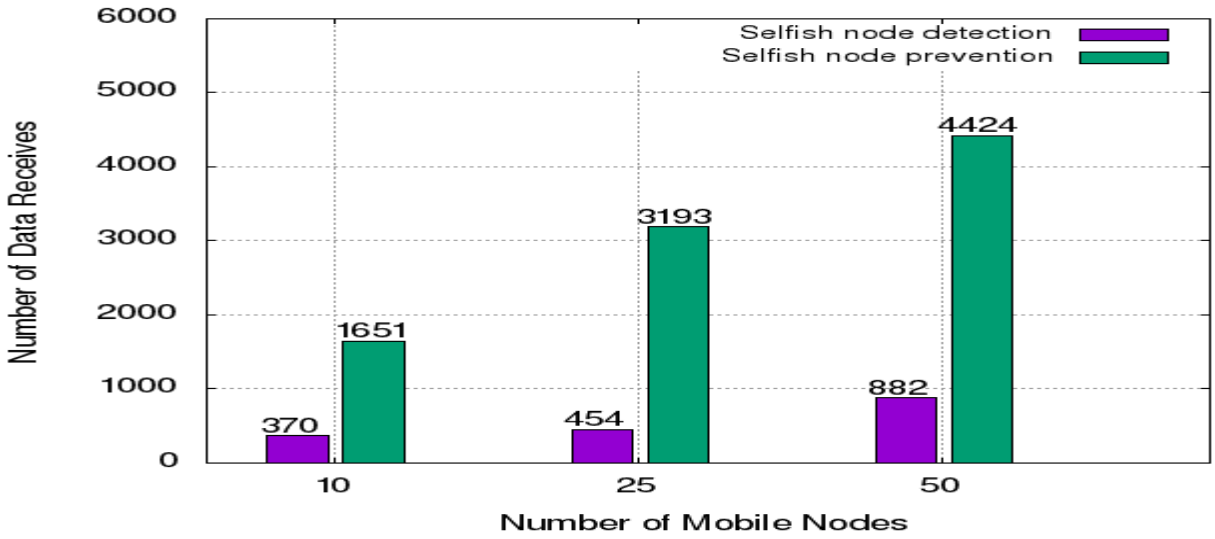


Figure 5.1: *Packet received by the destination nodes*

b) Number of data packet drop

Besides the selfish behavior of a node, a packet drop may result by other routing attack and network communication errors like link failure, network congestion, etc. Figure 5.2 indicates the total number of data packet drop by the selfish node, other routing attack and network related errors while data transmission takes place. The number of data packet lost after implementing the proposed selfish node attack prevention mechanism is highly improved. Here, it shows the total data packet drop in the network with varying number of nodes during detection and implementation of the proposed system. In Table 5.1, the data packet drop due to selfish node attack and total number of packet drop due to several network and security related problems are clearly presented that is shown in following figure:

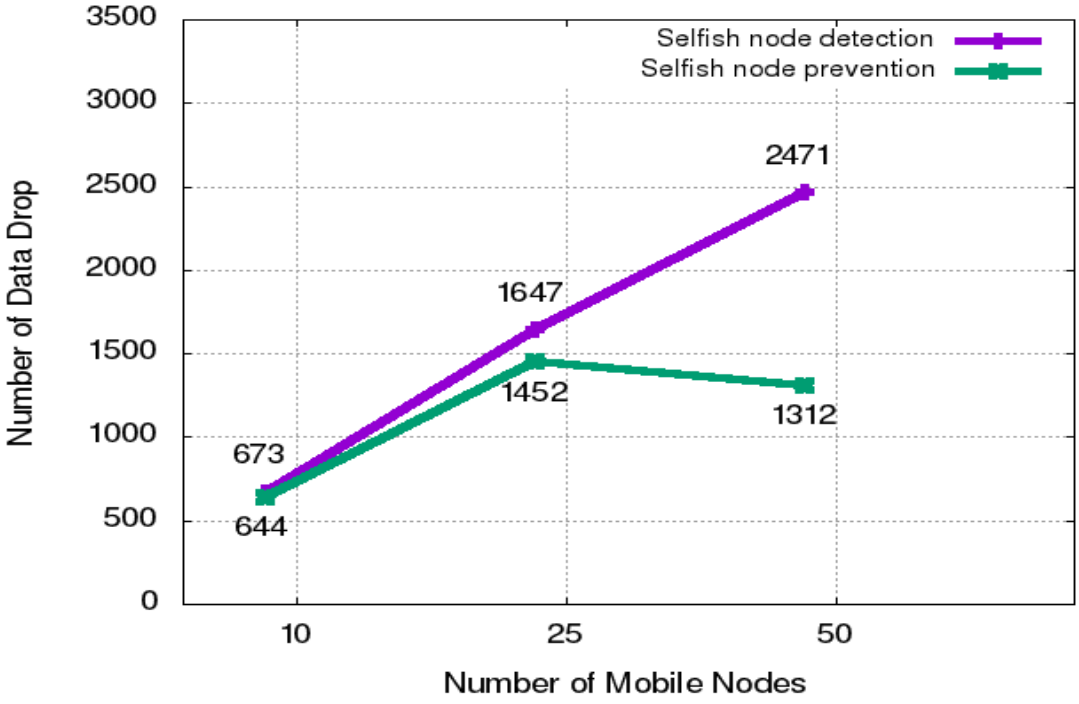


Figure 5.2: *Packet drop in the network during communication*

c) Packet captured and drop by selfish node

In MANET, a data packet is lost due to several reasons including selfish behavior of a node, network related issues and other routing attacks. Figure 5.3 presents the data packet captured and dropped by selfish behavior of a node in the network. After the implementation of the proposed lightweight system, the number of packets dropped by the selfish node is too small.As shown in Table 5.1 and Figure 5.3, when the number of node in the network is 10 and number of selfish node is 3, the proposed system minimize the number of dropped packet to 0 as compared with dropped packet in the presence of selfish node without prention were 485. When the number of node in the network is 25 and number of selfish node is 6, the proposed system minimize the number of packet drop as compared to selfish node without prevention from 548 to 0. When the number of node in the network is 50 and number of nodes with selfish behavior is 6, the proposed mobile agent based selfish node prevention system minimizes the number of packet drop as compared to selfish node without prevention from 1607 to 7 only.

The detection and prevention of selfish node attack start after a source node sends a data

packet to destination nodes. In prevention process, before the MA node detect and prevents the selfish node from the network, selfish nodes which are very close to the source nodes in the active route, drops some packet (e.g., 7 packets in scenario 3). Once the proposed system detects and prevents all the selfish node in the network, there is no packet drop due to selfish node for the next communication in all scenarios.

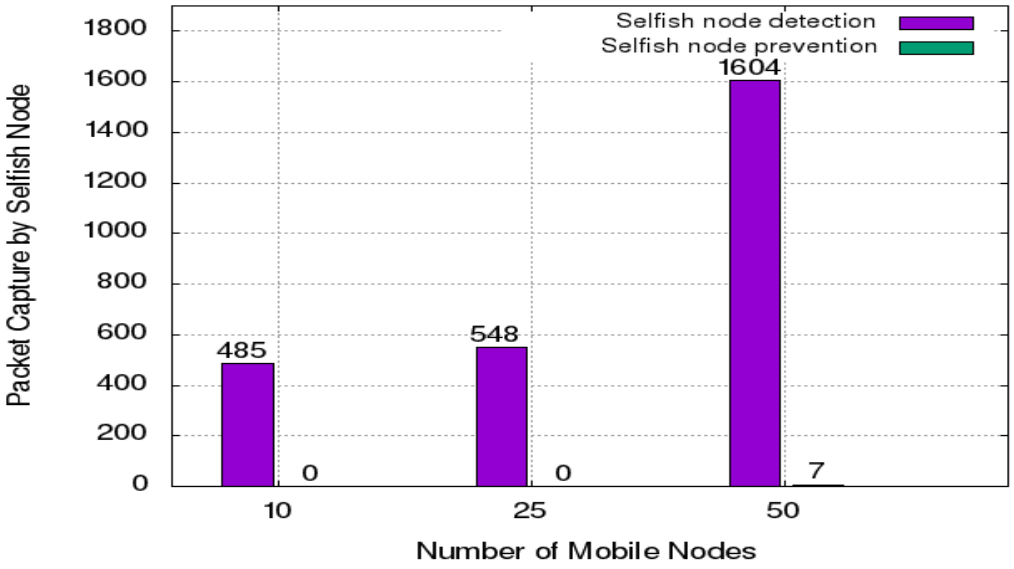


Figure 5.3: *Packet captured and dropped by selfish nodes*

d) Packet Delivery Ratio (PDR)

From the simulation result, the total number of packet sent by all source nodes, data packet successfully received by all the destination nodes, packet drop by selfish node and total packet drop in the network are presented in Table 5.1.

In the Figure 5.4, the packet delivery ratio of the proposed MA based system and presence of selfish node is shown and it is concluded that proposed prevention system is better than the selfish node attack detection in all three scenarios. When the proposed system is implemented in scenario one (number of node 10 and number of selfish node 3), the PDR of the network is improved from 35.47% to 71.94%. When the proposed system is implemented in scenario two (number of node 25 and number of selfish node 6), the PDR of the network is improved from 21.61% to 68.74%. Similarly, when the proposed system is

implemented in scenario three (number of node 50 and number of selfish node 6), the PDR of the network is improved from 26.30% to 77.13%.

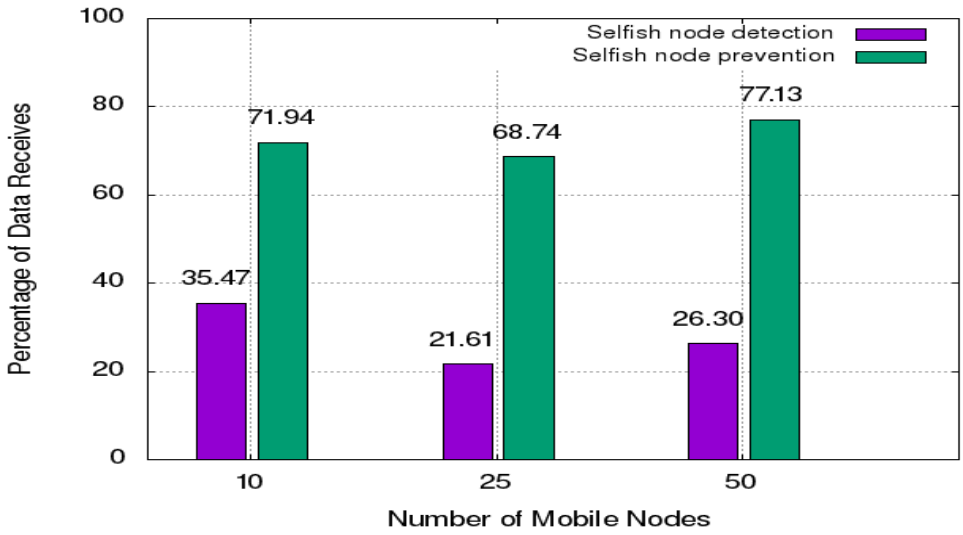


Figure 5.4: *Comparison of Packet delivery ratio with or without selfish node*

e) Normalized Routing Load (NRL)

The selfish node attack in the network increases the routing load. If the routing load of the network is smaller, the network is more robust and performs well. As shown in Figure 5.5, the proposed mobile agent based selfish node prevention mechanism reduces the routing load. After prevention of the selfish node from the network, the network routing load is significantly reduced.

When the number of node in the network increases, the routing overhead also increases. This is because if the number of nodes that participate in the data transmission process increase, mobility of node also increases and the nodes create large amount of routing control packets. Figure 5.5 shows the comparison of routing overhead with presence of selfish node and after removal of selfish node. In the proposed prevention system for 10, 25 and 50 nodes, the routing overhead is recorded only 0.12, 0.35 and 1.48 respectively.

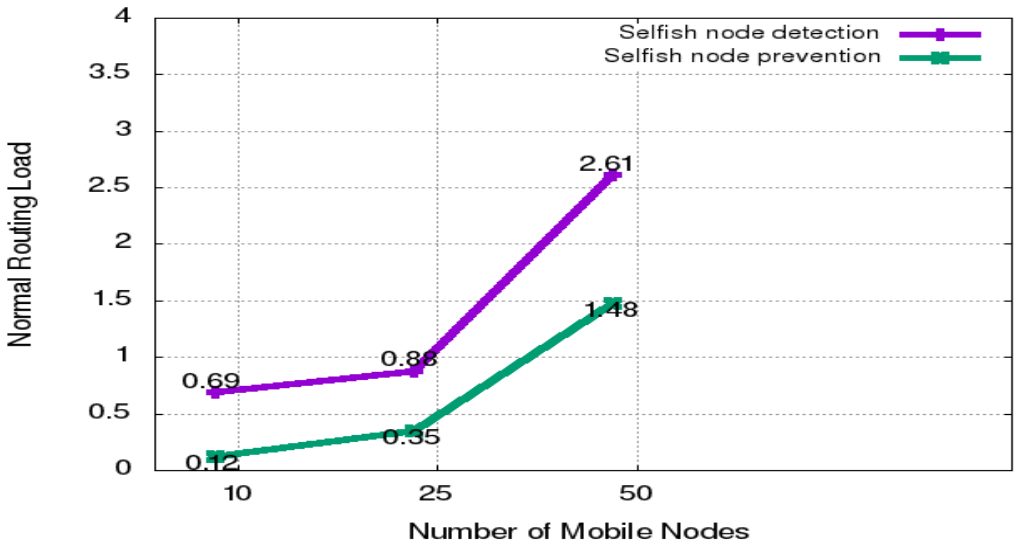


Figure 5.5: *Normalized routing load of detection and prevention system*

f) Average end-to-end delay

In the proposed system, MA performs detection of selfish behavior, removal of selfish node from the network, discarding the existing RREP message and initiating a new RREQ to forward the dropped packet increases the end to end delay of the network. Figure 5.6 clearly shows that the delay during the implementation of the proposed selfish node prevention system slightly increases over the selfish node detection system.

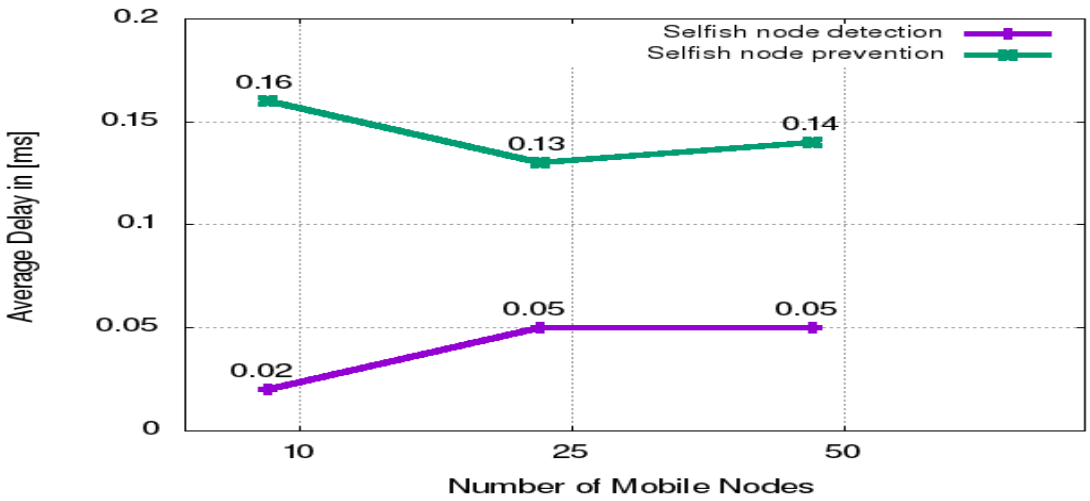


Figure 5.6: *Average end to end delay of a detection and prevention system*

g) Throughput of a network

As presented in the Figure 5.7, the proposed selfish node prevention method greatly improves the throughput of the network. The throughput of the network (in Mbps) with proposed selfish node prevention method over the detection system with varying number of node (10, 25, 50) is also improved. In the detection and prevention system, when the number of nodes in the system is 10, the throughput is improved from 0.18 to 0.83. When the number of nodes in the network is 25, the throughput is improved from 0.23 to 1.60. Similarly, when the number of nodes is 50, throughput increases from 0.44Mbps to 2.21Mbps.

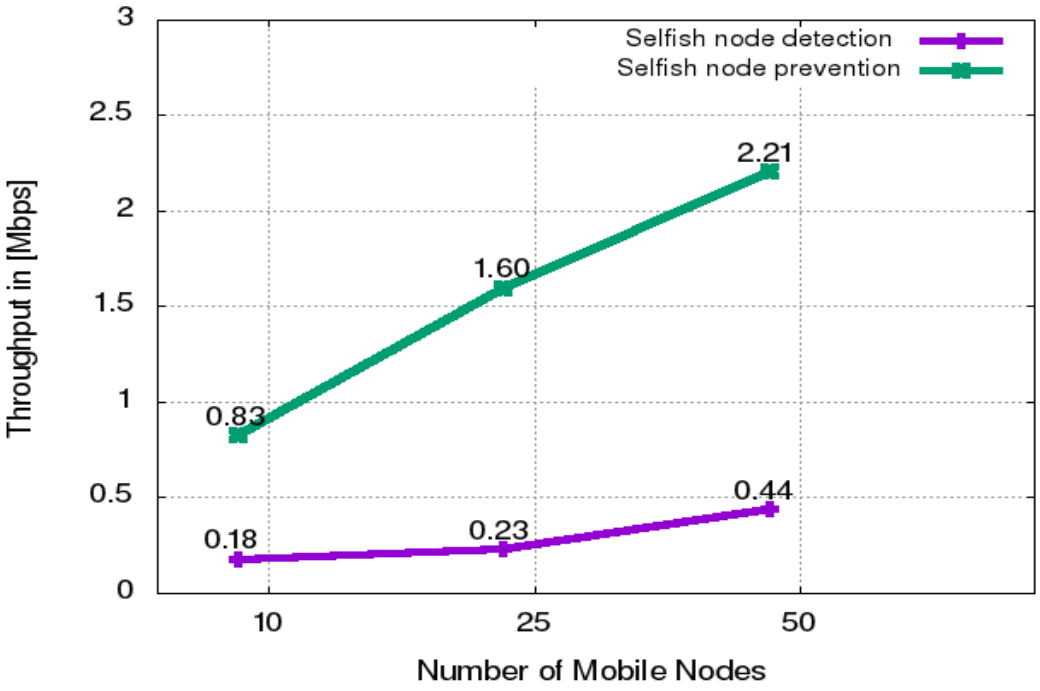


Figure 5.7: *Throughput of the network*

As the number of node in the network increase, the throughput of the network also increases. Improvement in the throughput of a network indicates that, after the removal of the selfish node attack, the data packet transmission rate and bandwidth utilization between source and destination is increased.

4.2 Result comparison with existing literature

In this thesis work, a lightweight selfish node detection and prevention system is proposed to improve the performance of the network and QoS. MA uses the functionality of watchdog and trust method to detect and prevent selfish node attacks.

Odedra *et al.* [39] proposed a selfish node detection and prevention mechanism using watchdog technique. It identified the selfish node by comparing the number of packet drop with the predefined packet drop threshold value. The proposed system put the packet dropping node in a suspected list. When the suspect list beyond the limit it blacklist the selfish node and remove it from the network. The network is simulated with varying number of nodes and varying number of selfish nodes to get and analyze the throughput and PDR of the network. Table 5.2 demonstrates the comparison between the proposed work and sample existing literature result with varying number of nodes and selfish nodes using throughput and PDR metrics. The throughput and PDR of our proposed detection and prevention system with 25 nodes and 6 selfish nodes is greatly improved. To compare the result of the proposed lightweight system with the state of the art, we took an existing literature [39] and the result is compared interms of throughput and PDR.

performance metrics	No of nodes	No of selfish nodes	Result comparison			
			[39]		This thesis work	
			Detection system	Proposed work	Detection system	Proposed work
Throughput (mbps)	25	6	0.00767	0.01886	0.23	1.60
PDR (%)	25	6	8.45	15.3	21.61	68.74

Table 5. 2*: Sample result comparison between proposed system and existing literature*

The proposed system is designed to work well even as the network grows bigger (scalability). We tested it with small networks (10 nodes), medium networks (25 nodes), and larger ones (50 nodes) to see how good it is at sending data (throughput) and how many packets actually reach their destination (packet delivery ratio) as more devices join. Because the system is lightweight it handle more devices without problems.

When it comes to telling the difference between a selfish attack and just normal network issues (like a weak signal or too much traffic), the system looks for specific bad behaviors. Normal network problems happen by accident, but selfish nodes *choose* to act badly. They might Drop some data on purpose. They consistently throw away actual data packets but still pass along control messages or only drop packets meant for certain places to save their own battery or resources. They might Refuse to send packets: Even if they have a clear path and are able to send packets, they just don't. They might also Lie about routes: They might mess with the network's rules (like AODV) to make routes look better or worse than they are, just so they don't have to do much work.

The lightweight system watches for these specific bad actions. It can tell that a node is selfish if it keeps doing these things, even when everything else in the network is fine. For example, if a node always fails to send packets when it should, it's flagged as selfish. But if it only drops packets because the network is really busy or a connection genuinely breaks, then it's not considered selfish. The "lightweight" part means it does this checking without using up a lot of the network's power or communication.

Summary

This proposed work is considered "lightweight" primarily because the overhead introduced by its detection and prevention mechanisms is minimal. This means it doesn't significantly consume network resources (like bandwidth, processing power, or battery life of the nodes) while operating. Lightweight nature contributing to its ability to scale without significantly degrading overall network performance, implying that the monitoring and analysis are done with minimal computational and communication overhead. This is vital in MANETs where nodes often have limited resources.

The core unique contribution of this proposed system, lies in its specific and efficient method for differentiating selfish node attacks from non-selfish network issues.

The unique additions are:

- ✓ Behavioral Pattern Analysis: Instead of complex cryptographic methods,The system uniquely focuses on identifying "specific behavioral patterns indicative of malicious intent." This implies a targeted and efficient way of observing node actions.
- ✓ Deliberate Deviation Detection: the proposed method specifically highlights the ability to distinguish "deliberate deviation from standard protocol behavior" (selfishness) from "transient network conditions or legitimate protocol operations" (non-selfish issues like congestion or link failures). This precise differentiation, based on criteria like "Selective Packet Dropping," "Packet Non-Forwarding," and "Abnormal Routing Behavior," forms the unique core of the lightweight detection mechanism.

In essence, the uniqueness lies in achieving effective selfish node detection and prevention with very low resource consumption, by intelligently analyzing specific malicious behaviors rather than employing resource-intensive general monitoring or security protocols.

In this chapter, we have provided an evaluation study on detection and proposed Lightweight detection & prevention system to detect and prevent selfish node attack in AODV routing protocols. The performance of the routing protocols is measured in terms of the most widely used QoS performance metrics including number of successful packet received, number of packet drop, normalized routing overhead, packet delivery ratio, throughput and average end-to-end delay. Performance analysis has been conducted by considering detection and proposed lightweight detection & prevention scenarios with varying number of nodes and selfish nodes.As a result, the number of packet received by the destination node increases and number of packet dropped due to selfish node are highly reduced. The throughput, PDR and normalized routing load of the proposed detection and prevention lightweight system is better than the detection system. Generally, the proposed lightweight detection and prevention system avoids data packet drop due to the selfish behavior of a node and improve the network quality of services.

CHAPTER FIVE

CONCLUSIONS AND FUTURE WORKS

5.1 Conclusions

The MA based selfish node detection and prevention system integrates trust value and watchdog mechanism. In a proposed system, each node calculates a trust value of its neighboring node. This trust value informs the watchdog about potential selfish behavior. During data transmission, MA nodes overhear the activity of their neighboring nodes for the selfish behavior using the watchdog method and block the selfish nodes from the network. Since the selfish node drops the entire data packet forwarded through it, the trust value of the node is zero.

In the detection system, the selfish node is detected and identified by analyzing the trace file using AWK script. The packet drop due to the selfish behavior of the node is easily identified by its state (d) and event (event=”LOOP”). The result of the detection system is simulated in NS-3 Network Simulator and the result is compared with the proposed lightweight detection and prevention system. Finally, the performance of selfish node detection and proposed prevention system is evaluated using different network performance metrics.

From the analysis it was observed that:-

- The number of data packet received and the packet delivery ratio of the proposed lightweight detection and prevention system increase with varying number of nodes and selfish nodes.
- The number of data packet drop in the network and the number of data packet captured and dropped by a selfish node in the proposed lightweight detection and prevention system are smaller than the detection system.
- In the proposed lightweight detection and prevention system the end-to-end

  delay of the network increases. This delay results because of the lightweight system

detect, identify the selfish node, discard the current route related to the selfish node, blocks the node, discover a new route to forward the data packet consume some time and increase the average end to end delay of the network.

- The proposed selfish node detection and prevention system improves the throughput and normalized routing load of the network.

The proposed lightweight system is designed only to detect and prevent the data drop by selfish nodes attack during data transmission in AODV routing and improve performance of the network. The proposed mobile agent based detection and prevention system designed to remove the selfish node and avoid the data drop by a selfish nodes. As indicated in the simulation result, there are packet drop due to other reasons like link failure, network congestion, MAC error, no space in IFQ (Interface Queue), no route to destination etc. This lightweight detection and prevention system cannot give solution to all kinds of data dropping reasons.In conclusion, the proposed lightweight detection and prevention system successfully detect and prevent the selfish node attack in the network and improves most of the network performance parameters and QoS. The proposed lightweight mechanism performs better in terms of PDR, throughput and normalized routing load with reasonable slight end-to-end delay increment compared to the detection system in AODV based routing.

5.2 Future work

We know that new technologies like *Artificial Intelligence (AI)*, *Machine Learning (ML),* and *Deep Learning (DL)* can greatly improve how we find and stop cyberattacks, especially against "selfish nodes" in mobile networks called MANETSs.

Our main goal in this study was to create a simple, quick solution for MANETs because these networks have limited resources. However, we agree that AI, ML and DL could make our solution much better. These technologies can help our system adapt, recognize complex patterns, and even predict future attacks.

So, for our future work, we highly suggest looking into these ideas:

- *Combine Systems***:** Use our simple attack-finding methods first. Then, use AI,ML to dig deeper

into strange behavior or find more hidden selfish attacks.

- *Simple Machine Learning***:** Explore using small, efficient ML programs that don't need a lot of computing power on MANET devices. These could include methods like one-class SVMs, k-means clustering (to find unusual things), or basic neural networks.
- *Predicting Attacks***:** Use ML to guess when a node might act selfishly by looking at past network information. This would help us prevent attacks before they even happen.
- *Learning Defenses:* See how "reinforcement learning" could allow devices to automatically learn and change their defenses as selfish attacks change.

Adding these smart technologies would definitely make our defense systems stronger and smarter, making MANETs more secure even with their resource limits.

The proposed lightweight detection and prevention technique successfully detect and prevent SNT1 in AODV based MANET. SNT2 do not participate in route discovery, maintenance and ignored by AODV routing protocol during route discovery operation. SNT3 misbehaves based on the available energy of the nodes.

To detect and prevent SNT3 the available energy of each node is considered and studied in the detection process.

The following are summarized suggested areas for future work:

- ❖ A mobile agent based mechanism for SNT2 and SNT3 detection and prevention Will be studied.
- ❖ Study the other routing layer attacks using MA based detection and prevention mechanism to reduce data packet dropping due to data packet dropping attacks.

REFERENCES

[1] A. Taggu, A. Mungoli and A. Taggu, *"ReverseRoute: An Application-layer Scheme for Detecting Blackholes in MANET using Mobile Agents"*, The Technology Innovation Management and Engineering Science International Conference, 2018.

[2] A. K. Jain and A. Choorasiya, *"Security Enhancement of AODV Routing Protocol in Mobile Ad Hoc Network",* Proceedings of the 2nd International Conference on Communication and Electronics Systems, 2017, pp. 958-959.

[3] Zalte S.S. and Ghorpade V.R, *"Intrusion Detection System for MANET"*, International Conference for Convergence in Technology, April 2018, pp.1-2.

[4] B. Kumari and D. Vydeki, "*Performance Analysis of MANET In The Presence of Malicious Nodes"*, IEEE, 2017, pp. 79-81.

[5] K. Tamizarasu and M. Rajaram, *"The Effective and Efficient of AODV Routing protocol for Minimized End-to-End Delay in MANET*", International Journal of Advanced Research in Computer and Communication Engineering, Vol.1, June 2012, pp. 250-252.

[6] Deepthi V S and Vagdevi S, *"Behaviour Analysis and Detection of Black hole Attacker Node under Reactive Routing Protocol in MANETs"*, IEEE, 2018.

[7] Ani Taggu and Amar Taggu, *"TraceGray: An application-layer scheme for Intrusion detection in MANET using mobile agents"*, Third International Conference on Communication Systems and Networks", 2011.

[8] R. Abirami and G. Sumithra, *"Preventing the impact of selfish behavior under MANET using Neighbor Credit Value based AODV routing algorithm"*, Indian Academy of Sciences, 2018, pp. 3-7.

[9] R. L akhwani, S. Suhane and A. Motwani, *"Agent based AODV Protocol to Detect and Remove Black Hole Attacks*", International Journal of Computer Applications, Vol.59, No.8, December 2012, pp. 35-37.

[10] C. Turguner, *"Secure Data Dissemination in MANETs by means of Mobile*

Agents", IEEE, 2014.

[11] V. Gaikwad and L. Ragha, *"Security Agents for Detecting and Avoiding Cooperative Blackhole Attacks in MANET"*, International Conference on Applied and Theoretical Computing and Communication Technology, 2015, pp. 306-311.

[12] P. Pooja B., P. Manish M. and P. Megha B. "*Jellyfish Attack Detection and Prevention in MANET*", IEEE 3rd International Conference on Sensing, Signal Processing and Security, 2017, pp. 54-60.

[13] A. Aranganathan and C. D. Suriyakala, *"Mobile Agent based Security in MANETS against Sybil Attack",* International Conference on Control, Instrumentation, Communication and Computational Technologies, 2014.

[14] A. V. Panicker and Jisha G, "*Network Layer Attacks and Protection in MANETA Survey",* International Journal of Computer Science and Information Technologies, Vol. 5, 2014.

[15] E. Elmahdi, S. Yoo and K. Sharshembiev, "*Securing Data Forwarding against Black hole Attacks in Mobile Ad Hoc Networks",* IEEE, 2018, pp. 463-466.

[16] Karthik Pai B. H., Nagesh H. R and N. Chiplunkar, *"Mitigation and Performance evaluation Mechanism for Selfish Node Attack in MANETs",* IEEE, 2017.

[17] V. R. Verma, D. P. Sharma and C. S. Lamba, "*QOS Improvement in MANET Routing by Route Optimization through Convergence of Mobile Agent",* IEEE Uttar Pradesh Section International Conference on Electrical, Electronics and Computer Engineering, 2018.

[18] N. Soliyal and H. S. Bhadauria, "*Preventing Packet Dropping Attack on AODV Based Routing in Mobile Ad-Hoc MANET",* International Conference on Advances in Computing, Communications and Informatics, 2016.

[19] J. Rahman, MD. A. M. Hasan, and K. B. Islam *"Comparative Analysis the Performance of AODV, DSDV and DSR Routing Protocols in Wireless Sensor Network",* International Conference on Electrical and Computer Engineering, December, 2012, pp. 283-286.

[20] M. K. Singh and S. N. Thakur, "*Comparison of DSDV, DSR and ZRP Routing Protocols in MANETs"*, International Journal of Computer Applications, Vol.108, 2014.

[21] Y. Bai, Y. Mai and N. Wang, "*Performance Comparison and Evaluation of the Proactive and Reactive Routing Protocols for MANETs",* IEEE, 2017.

[22] P.R. Muchintala, "*Routing protocols for MANET",* Unpublished Masters project, Department of Telecommunications, State University of New York, 2016.

[23] M. K Dholey and M. K Saha, "*A Security Mechanism In DSR Routing For MANET"*, Proceedings of the second International Conference on Trends in Electronics and Informatics, 2018.

[24] A. Bhattacharyya, A. Banerjee, H. N. Saha, D. Bhattacharjee and D. Bose,

"*Different types of attacks in Mobile ADHOC Network: Prevention and mitigation techniques"*, Unpublished, Calcutta University, Department of Computer Science & Engineering, 2012.

[25] Alamsyah, E. Setijadi, K. E. Purnama and M. H. Purnomo, "*Performance Comparative of AODV, AOMDV and DSDV Routing Protocol in MANET Using NS2*",International seminar on Application for Technology of Information and Communication, 2018, pp. 286-287.

[26] P.K. Maurya, G. Sharma,V. Sahu and V. Roberts, "*An Overview of AODV Routing Protocol"*,Int. Journal of Modern Engineering Research, Vol.2, June 2012, pp. 728-731.

[27] N. Arya, U. Singh and S. Singh, "*Detecting and Avoiding of Worm Hole Attack and CollaborativeBlackhole attack on MANET using Trusted AODV Routing Algorithm",* IEEE International Conference on Computer, Communication and Control, 2015.

[28] S. Saranya1 and R. M. Chezian, "*Comparison of Proactive, Reactive and Hybrid Routing Protocol in MANET",* International Journal of Advanced Research in Computer and Communication Engineering, Vol.5, 2016.

[29] T. S. John and A. Aranganathan, *"Performance analysis of proposed Mobile autonomous agent for detection of malicious node and protecting against attacks in MANET"*, International Conference on Communication and Signal Processing, 2014.

[30] D. B. Roy and R. Chaki, "*Detection of Denial of Service Attack Due to Selfish Node in MANET by Mobile Agent",* Communications in Computer and Information Science] Recent Trends in Wireless and Mobile Networks, Vol.162, 2011, pp. 15-18.

[31] N. Thakur, D. Bisen, N. Gupta, "*Proposed Agent Based Black hole Node Detection Algorithm for Ad-Hoc Wireless Network*", International Journal on Computational Science & Applications, Vol.5, 2015.

[32] S.A. M. Alicia, Y.C. Tang, M. H. Hassan and S. H. Khaleefah, *"A Multi-Agent Ad Hoc On-Demand Distance Vector for Improving the Quality of Service in MANETs*", International Symposium on Agent, Multi-Agent Systems and Robotics, 2018.

[33] A. Meeran, Praveen A N and Ratheesh T K, *"Enhanced System For Selfish Node Revival Based On Watchdog Mechanism*", International Conference on Trends in Electronics and Informatics, 2017, pp.332-334.

[34] V. Keerthika and R. C and Suganthe, "*WATCHDOG: Reduce time delay for Spreading selfish information in MANET*", International Conference on Information Communication and Embedded Systems, 2013.

[35] M. M. Musthafa, K. Vanitha, and K. Anitha, "*An Efficient Approach to Identify Selfish Node in MANET*", International Conference on Computer Communication and Informatics, 2020.

[36] Veeraiah and B. T. Krishna, "*Selfish Node Detection IDSM Based Approach Using Individual Master Cluster Node",* Proceedings of the Second International Conference on Inventive Systems and Control, 2018, pp.427-429.

[37] M. D. Serrat, J. Cano, C. T. Calafate, and P. Manzoni, *"Improving Selfish Node Detection in MANETs Using a Collaborative Watchdog",* IEEE, Vol.16, 2012.

[38] M. N Anjum, C. Chowdhury and S. Neogy, "*Implementing Mobile Agent based Secure Load Balancing Scheme for MANET",* IEEE, 2019.

[39] L. Odedra, A. Revar and M. H. Lunagaria, "*Detection and prevention of selfish Attack in MANET using dynamic learning*", IOSR Journal of Computer Engineering, Vol.18, 2016, pp.54-61.

APPENDICES

APPENDIX A: Sample C++ code for selfish node and MA implementation

```
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/mobility-module.h"
#include "ns3/wifi-module.h"
#include "ns3/internet-module.h"
#include "ns3/applications-module.h"
#include "ns3/aodv-helper.h"
#include "ns3/energy-module.h"
#include "ns3/point-to-point-module.h"
#include "ns3/netanim-module.h"
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
using namespace ns3;
NS_LOG_COMPONENT_DEFINE("AODVWirelessEnergySimulation");
void
RecordUdpStats(Ptr<PacketSink> sink, Ptr<PacketSink> sink2, std::ofstream& outputFile)
{
   double now = Simulator::Now().GetSeconds();
   uint64_t packetsReceived = sink->GetTotalReceivedBytes();
   uint64_t packetsReceived2 = sink2->GetTotalReceivedBytes();
   NS_LOG_INFO("Time: " << now << "s, UDP Sink 1 Received: " << packetsReceived << "
bytes");
   NS_LOG_INFO("Time: " << now << "s, UDP Sink 2 Received: " << packetsReceived2 << "
bytes");
   outputFile << now << "\t" << packetsReceived << "\t" << packetsReceived2 << std::endl;
   Simulator::Schedule(Seconds(0.05), &RecordUdpStats, sink, sink2, std::ref(outputFile));
```

```
}
int
main(int argc, char* argv[])
{
   // 1. Simulation Parameters
   double simulationTime = 100.0;
   uint32_t numNodes = 10;
   double maxX = 1000.0;
   double maxY = 1000.0;
   uint32_t ifqLen = 10;
     double antennaX = 0.0;
   double antennaY = 0.0;
   double antennaZ = 1.5;
   double antennaGt = 1.0;
   double antennaGr = 1.0;
   double cpThresh = 20.0;
   double csThresh = 1.559e-11;
   double rxThresh = 3.652e-10;
   double rb = 2e6;
   double pt = 0.2828;
   double freq = 914e6;
   double l = 0.2;
   double ptConsume = 0.660;
   double prConsume = 0.395;
   double pIdle = 0.035;
   std::string wifiDataMode = "OfdmRate6Mbps"; // A common basic rate in NS3
   double initialEnergyNode[10] = {92, 89, 65, 88, 77, 90, 78, 65, 86, 70};
   double txPower = 1.0;
   double rxPower = 0.8;
```

```
double idlePower = 0.0; // Tcl script had 0.0 for idlePower, but P_idle_ 0.035 for Phy
double sensePower = 0.5;
double sleepPower = 0.5;
uint32_t cbrPacketSize = 1000;
double cbrInterval = 0.15;
Time tcpStartTime = Seconds(1.0);
Time cbrStartTime = Seconds(1.0);
CommandLine cmd(__FILE__);
cmd.AddValue("simulationTime", "Total simulation time in seconds", simulationTime);
cmd.AddValue("numNodes", "Number of nodes in the simulation", numNodes);
cmd.AddValue("maxX", "X-dimension of the simulation area", maxX);
cmd.AddValue("maxY", "Y-dimension of the simulation area", maxY);
cmd.Parse(argc, argv);
  if (numNodes > 10)
{
   NS_FATAL_ERROR("Number of nodes exceeds predefined initial energy array size.");
}
 // 2. Configure Tracing
AsciiTraceHelper ascii;
 AnimationInterface anim("aodv10.xml");
anim.SetMaxByX(maxX);
anim.SetMaxByY(maxY);
// 3. Create Nodes
NodeContainer nodes;
nodes.Create(numNodes);
anim.NodeContainerPhyAssociations(nodes, YansWifiPhyHelper::GetTypeId());
anim.NodeColor(nodes.Get(0), 0, 0, 255); // Blue
anim.NodeColor(nodes.Get(1), 255, 0, 0); // Red
anim.NodeColor(nodes.Get(2), 210, 105, 30); // Chocolate
```

```
    anim.NodeColor(nodes.Get(3), 165, 42, 42); // Brown
    anim.NodeColor(nodes.Get(4), 210, 180, 140); // Tan
    anim.NodeColor(nodes.Get(5), 255, 215, 0); // Gold
    anim.NodeColor(nodes.Get(6), 0, 0, 0); // Black
    // No specific colors for nodes 7, 8, 9, default will be used or can be set.
    // 4. Configure Wireless Channel and PHY
    YansWifiChannelHelper channel;
    channel.SetPropagationLoss("ns3::TwoRayGroundPropagationLossModel"); // val(prop)
    channel.SetPropagationDelay("ns3::ConstantSpeedPropagationDelayModel");
   Config::SetDefault("ns3::WifiRemoteStationManager::NonUnicastMode",
StringValue(wifiDataMode));
    Config::SetDefault("ns3::WifiRemoteStationManager::RtsCtsThreshold", UintegerValue(0)); //
Disable RTS/CTS by default
    // Configure the WifiPhy
    YansWifiPhyHelper phy;
    phy.SetChannel(channel.Create());
    phy.Set("TxPowerStart", DoubleValue(pt)); // Transmitter power
    phy.Set("TxPowerEnd", DoubleValue(pt));
    phy.Set("CcaMode1Threshold", DoubleValue(cpThresh)); // Clear Channel Assessment
threshold
    phy.Set("RxSensitivity", DoubleValue(rxThresh));     // Receiver sensitivity (RXThresh_)
     Config::SetDefault("ns3::WifiRadioEnergyModel::TxCurrentA", DoubleValue(ptConsume));
    Config::SetDefault("ns3::WifiRadioEnergyModel::RxCurrentA", DoubleValue(prConsume));
    Config::SetDefault("ns3::WifiRadioEnergyModel::IdleCurrentA", DoubleValue(pIdle));
    Config::SetDefault("ns3::WifiRadioEnergyModel::SleepCurrentA", DoubleValue(0.001)); // A
small sleep current
    // 5. Configure MAC and install devices
    WifiMacHelper mac;
    mac.SetType("ns3::AdhocWifiMac"); // Ad-hoc mode
    mac.SetQueue("ns3::DropTailQueue", "MaxPackets", UintegerValue(ifqLen));
```

```
    WifiHelper wifi;
    wifi.SetRemoteStationManager("ns3::ConstantRateWifiManager",
                    "DataMode", StringValue(wifiDataMode));
    NetDeviceContainer devices;
    devices = wifi.Install(phy, mac, nodes);
  phy.EnablePcap("aodv10", devices);
    // 6. Install Internet Stack and AODV (equivalent to -adhocRouting AODV)
    AodvHelper aodv;
    InternetStackHelper internet;
    internet.SetRoutingHelper(aodv);
    internet.Install(nodes);
    Ipv4AddressHelper address;
    address.SetBase("10.0.0.0", "255.0.0.0");
    Ipv4InterfaceContainer interfaces = address.Assign(devices);
    // 7. Configure Mobility (equivalent to Antenna/OmniAntenna, set X_, Y_, Z_ and $ns at
"$node(X) setdest")
    MobilityHelper mobility;
    mobility.SetMobilityModel("ns3::WaypointMobilityModel"); // Use WaypointMobilityModel
for setdest commands
    // Create and set initial positions for nodes
    Ptr<ListPositionAllocator> positionAlloc = CreateObject<ListPositionAllocator>();
    std::vector<Vector> initialPositions(numNodes);
    initialPositions[0] = Vector(700.0, 240.0, antennaZ);
    initialPositions[1] = Vector(500.0, 600.0, antennaZ);
    initialPositions[2] = Vector(76.0, 533.0, antennaZ);
    initialPositions[3] = Vector(55.0, 10.0, antennaZ);
    initialPositions[4] = Vector(350.0, 210.0, antennaZ);
    initialPositions[5] = Vector(700.0, 800.0, antennaZ);
    initialPositions[6] = Vector(171.0, 718.0, antennaZ);
    initialPositions[7] = Vector(105.0, 314.0, antennaZ);
```

```
initialPositions[8] = Vector(500.0, 20.0, antennaZ);
initialPositions[9] = Vector(832.0, 551.0, antennaZ);
for (uint32_t i = 0; i < numNodes; ++i)
{
   positionAlloc->Add(initialPositions[i]);
}
mobility.SetPositionAllocator(positionAlloc);
mobility.Install(nodes);
  for (uint32_t i = 0; i < numNodes; ++i)
{
   Ptr<WaypointMobilityModel> wpm = nodes.Get(i)->GetObject<WaypointMobilityModel>();
   if (wpm == 0)
   {
      NS_FATAL_ERROR("WaypointMobilityModel not found for node " << i);
   }
       if (i == 0)    {
      wpm->AddWaypoint(Waypoint(Seconds(5.0), Vector(450.0, 50.0, antennaZ)));
      wpm->AddWaypoint(Waypoint(Seconds(simulationTime - 0.01), Vector(450.0, 50.0,
antennaZ))); // Hold position till end
   }
   else if (i == 1)   {
      wpm->AddWaypoint(Waypoint(Seconds(10.0), Vector(650.0, 740.0, antennaZ)));
      wpm->AddWaypoint(Waypoint(Seconds(simulationTime - 0.01), Vector(650.0, 740.0,
antennaZ)));
   }
   else if (i == 2)   {
      wpm->AddWaypoint(Waypoint(Seconds(5.0), Vector(450.0, 300.0, antennaZ)));
      wpm->AddWaypoint(Waypoint(Seconds(simulationTime - 0.01), Vector(450.0, 300.0,
antennaZ)));
   }
```

```
else if (i == 3)     {
    wpm->AddWaypoint(Waypoint(Seconds(6.0), Vector(10.0, 300.0, antennaZ)));
    wpm->AddWaypoint(Waypoint(Seconds(simulationTime - 0.01), Vector(10.0, 300.0,
antennaZ)));
}
else if (i == 4)   {
    wpm->AddWaypoint(Waypoint(Seconds(3.0), Vector(17.0, 102.0, antennaZ)));
    wpm->AddWaypoint(Waypoint(Seconds(simulationTime - 0.01), Vector(17.0, 102.0,
antennaZ)));
}
else if (i == 5)   {
    wpm->AddWaypoint(Waypoint(Seconds(14.0), Vector(810.0, 634.0, antennaZ)));
    wpm->AddWaypoint(Waypoint(Seconds(simulationTime - 0.01), Vector(810.0, 634.0,
antennaZ)));
}
else if (i == 6)     {
    wpm->AddWaypoint(Waypoint(Seconds(13.0), Vector(925.0, 700.0, antennaZ)));
    wpm->AddWaypoint(Waypoint(Seconds(simulationTime - 0.01), Vector(925.0, 700.0,
antennaZ)));
}
else if (i == 7)     {
    wpm->AddWaypoint(Waypoint(Seconds(20.0), Vector(50.0, 250.0, antennaZ)));
    wpm->AddWaypoint(Waypoint(Seconds(simulationTime - 0.01), Vector(50.0, 250.0,
antennaZ)));
}
else if (i == 8)   {
    wpm->AddWaypoint(Waypoint(Seconds(21.0), Vector(60.0, 200.0, antennaZ)));
    wpm->AddWaypoint(Waypoint(Seconds(simulationTime - 0.01), Vector(60.0, 200.0,
antennaZ)));
}
else if (i == 9)   {
```

```
        wpm->AddWaypoint(Waypoint(Seconds(24.0), Vector(750.0, 610.0, antennaZ)));
         wpm->AddWaypoint(Waypoint(Seconds(simulationTime - 0.01), Vector(750.0, 610.0,
antennaZ)));
      }    }
   // 8. Energy Model Setup
    for (uint32_t i = 0; i < numNodes; ++i)
   {
      Ptr<Node> node = nodes.Get(i);
      Ptr<BasicEnergySource> energySource = CreateObject<BasicEnergySource>();
      energySource->SetInitialEnergy(initialEnergyNode[i]);
      Ptr<DeviceEnergyModel> radioEnergyModel = CreateObject<WifiRadioEnergyModel>();
      node->AddDeviceEnergyModel(radioEnergyModel);
      node->AggregateObject(energySource);
         energySource->TraceConnectWithoutContext("RemainingEnergy",
MakeCallback([i](double oldValue, double remainingEnergy) {
         NS_LOG_INFO("Time: " << Simulator::Now().GetSeconds() << "s, Node " << i << "
Remaining Energy: " << remainingEnergy << "J");
      }));
      radioEnergyModel->TraceConnectWithoutContext("TotalEnergyConsumption",
MakeCallback([i](double oldValue, double totalEnergy) {
         NS_LOG_INFO("Time: " << Simulator::Now().GetSeconds() << "s, Node " << i << " Total
Radio Energy Consumed: " << totalEnergy << "J");
      }));
   }
     anim.SetNodeDescription(nodes.Get(5), "MA1"); // node(5)
   anim.SetNodeDescription(nodes.Get(4), "MA2"); // node(4)
   // 9. Applications
     Ptr<Node> tcpSourceNode1 = nodes.Get(3);
   Ptr<Node> tcpSinkNode1 = nodes.Get(2);
    TcpNewRenoHelper tcpReno;
   tcpReno.SetAttribute("SegmentSize", UintegerValue(1000)); // Default segment size is often
```

```
536, adjust if needed
      BulkSendHelper ftp1("ns3::TcpSocketFactory",
                  InetSocketAddress(interfaces.GetAddress(tcpSinkNode1->GetId()), 9)); // Port 9
for TCP Sink
      ApplicationContainer clientApps1 = ftp1.Install(tcpSourceNode1);
   clientApps1.Start(tcpStartTime);
   clientApps1.Stop(Seconds(simulationTime - 0.01)); // Stop just before simulation end
   PacketSinkHelper sink1("ns3::TcpSocketFactory",
                   InetSocketAddress(Ipv4Address::GetAny(), 9));
   ApplicationContainer serverApps1 = sink1.Install(tcpSinkNode1);
   serverApps1.Start(tcpStartTime);
   serverApps1.Stop(Seconds(simulationTime));
     Ptr<Node> tcpSourceNode2 = nodes.Get(1);
   Ptr<Node> tcpSinkNode2 = nodes.Get(6);
   BulkSendHelper ftp2("ns3::TcpSocketFactory",
                  InetSocketAddress(interfaces.GetAddress(tcpSinkNode2->GetId()), 10)); //
Different port
   ApplicationContainer clientApps2 = ftp2.Install(tcpSourceNode2);
   clientApps2.Start(tcpStartTime);
   clientApps2.Stop(Seconds(simulationTime - 0.01));
   PacketSinkHelper sink2("ns3::TcpSocketFactory",
                   InetSocketAddress(Ipv4Address::GetAny(), 10));
   ApplicationContainer serverApps2 = sink2.Install(tcpSinkNode2);
   serverApps2.Start(tcpStartTime);
   serverApps2.Stop(Seconds(simulationTime));
    Ptr<Node> udpSourceNode = nodes.Get(7);
   Ptr<Node> udpSinkNode = nodes.Get(9);
   OnOffHelper cbrSource("ns3::UdpSocketFactory",
                    InetSocketAddress(interfaces.GetAddress(udpSinkNode->GetId()), 11)); // Port 11
for UDP Sink
```

```
    cbrSource.SetAttribute("OnTime",
StringValue("ns3::ConstantRandomVariable[Constant=1.0]"));

    cbrSource.SetAttribute("OffTime",
StringValue("ns3::ConstantRandomVariable[Constant=0.0]")); // Always On

    cbrSource.SetAttribute("PacketSize", UintegerValue(cbrPacketSize));

    cbrSource.SetAttribute("DataRate", StringValue(std::to_string(cbrPacketSize * 8 / cbrInterval /
1000000) + "Mbps")); // Calculate data rate

    ApplicationContainer cbrApps = cbrSource.Install(udpSourceNode);

    cbrApps.Start(cbrStartTime);

    cbrApps.Stop(Seconds(simulationTime - 0.01));

    Ptr<PacketSink> udpSink =
DynamicCast<PacketSink>(PacketSinkHelper("ns3::UdpSocketFactory",
                                             InetSocketAddress(Ipv4Address::GetAny(), 11))
                                        .Install(udpSinkNode)
                                        .Get(0));

    udpSink->Start(cbrStartTime);

    udpSink->Stop(Seconds(simulationTime));

    std::ofstream udpOutputFile("udp_stats.txt");

    udpOutputFile << "Time\tSink4_ReceivedBytes\tSink_Placeholder_ReceivedBytes" <<
std::endl; // Placeholder for second sink

    Simulator::Schedule(Seconds(0.05), &RecordUdpStats, udpSink, udpSink,
std::ref(udpOutputFile)); // Passing sink twice for simplicity

    // 10. Simulation Execution

      NS_LOG_INFO("Starting simulation for " << simulationTime << " seconds...");

    Simulator::Stop(Seconds(simulationTime));

    Simulator::Run();

    Simulator::Destroy();

    udpOutputFile.close();

    NS_LOG_INFO("Simulation finished.");

    return 0;

}
```

APPENDIX B: C++ Script for evaluating network performance metrics

```
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/mobility-module.h"
#include "ns3/wifi-module.h"
#include "ns3/internet-module.h"
#include "ns3/applications-module.h"
#include "ns3/aodv-helper.h"
#include "ns3/energy-module.h"
#include "ns3/point-to-point-module.h"
#include "ns3/netanim-module.h"
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <map> // For start_time, end_time
using namespace ns3;
NS_LOG_COMPONENT_DEFINE("AODVWirelessEnergySimulationWithMetrics");
// --- Global Variables for Statistics
uint32_t g_sends = 0;
uint32_t g_recvs = 0;
uint32_t g_pkt_forwarded = 0;
uint32_t g_routing_packets = 0;
uint32_t g_total_hop_count = 0;
uint32_t g_atk = 0; // For "Selfish Attack Drop Data"
uint32_t g_rtp = 0; // For "Routing Protocol Packets" (message type 404)
// For End-to-End Delay
std::map<uint32_t, double> g_start_time; // packetId -> time
std::map<uint32_t, double> g_end_time;   // packetId -> time
```

```
// --- Trace Callbacks
void AppTxTrace(Ptr<const Packet> packet, const ApplicationContainer& apps)
{
   g_sends++;
   uint32_t packetId = packet->GetUid(); // NS3's unique packet ID
   g_start_time[packetId] = Simulator::Now().GetSeconds();
   NS_LOG_INFO("AppTxTrace: Packet " << packetId << " sent at " <<
Simulator::Now().GetSeconds() << "s");
}
void AppRxTrace(Ptr<const Packet> packet, const ApplicationContainer& apps)
{
   g_recvs++;
   uint32_t packetId = packet->GetUid(); // NS3's unique packet ID
   g_end_time[packetId] = Simulator::Now().GetSeconds();
   NS_LOG_INFO("AppRxTrace: Packet " << packetId << " received at " <<
Simulator::Now().GetSeconds() << "s");
}
void RouterForwardTrace(Ptr<const Packet> packet, Ptr<Ipv4RoutingProtocol> routing)
{
    if (packet->GetSize() > 0) // Simple check to filter control/empty packets
   {
      Ptr<Packet> copy = packet->Copy();
      // Check for TCP/UDP headers
      if (copy->RemoveHeader<TcpHeader>() || copy->RemoveHeader<UdpHeader>())
      {
         g_pkt_forwarded++;
      }
   }
   NS_LOG_INFO("RouterForwardTrace: Packet forwarded at " << Simulator::Now().GetSeconds()
<< "s");
```

```
}
void AodvRoutingPacketTrace(Ptr<const Packet> packet)
{
   // This trace source is usually specific to the routing protocol.
   // AODV packets (RREQ, RREP, RERR, RREG) would typically have an AODvHeader.
   // We would connect to a trace source that specifically fires for AODV control packets.
   // For demonstration, let's assume any packet with AodvHeader is a routing packet.
   Ptr<Packet> copy = packet->Copy();
   if (copy->RemoveHeader<AodvHeader>()) // Check if it has an AODV header
   {
      g_routing_packets++;
      NS_LOG_INFO("AodvRoutingPacketTrace: AODV packet counted at " <<
Simulator::Now().GetSeconds() << "s");
   }
}
void PacketDropTrace(Ptr<const Packet> packet, const NetDevice::DropReason& reason)
{
  }
// --- Main Simulation Function ---
int
main(int argc, char* argv[])
{
   // 1. Simulation Parameters
   double simulationTime = 100.0;
   uint32_t numNodes = 10;
   double maxX = 1000.0;
   double maxY = 1000.0;
   uint32_t ifqLen = 10;
   double antennaZ = 1.5;
```

```
double pt = 0.2828;
double cpThresh = 20.0;
double rxThresh = 3.652e-10;
double ptConsume = 0.660;
double prConsume = 0.395;
double pIdle = 0.035;
std::string wifiDataMode = "OfdmRate6Mbps";
double initialEnergyNode[10] = {92, 89, 65, 88, 77, 90, 78, 65, 86, 70};
uint32_t cbrPacketSize = 1000;
double cbrInterval = 0.15;
Time tcpStartTime = Seconds(1.0);
Time cbrStartTime = Seconds(1.0);
CommandLine cmd(__FILE__);
cmd.AddValue("simulationTime", "Total simulation time in seconds", simulationTime);
cmd.AddValue("numNodes", "Number of nodes in the simulation", numNodes);
cmd.AddValue("maxX", "X-dimension of the simulation area", maxX);
cmd.AddValue("maxY", "Y-dimension of the simulation area", maxY);
cmd.Parse(argc, argv);
if (numNodes > 10)
{
   NS_FATAL_ERROR("Number of nodes exceeds predefined initial energy array size.");
}
// 2. Configure Tracing (NetAnim and PCAP)
AnimationInterface anim("aodv_metrics.xml");
anim.SetMaxByX(maxX);
anim.SetMaxByY(maxY);
// 3. Create Nodes
NodeContainer nodes;
nodes.Create(numNodes);
```

```
anim.NodeContainerPhyAssociations(nodes, YansWifiPhyHelper::GetTypeId());
anim.NodeColor(nodes.Get(0), 0, 0, 255); // Blue
anim.NodeColor(nodes.Get(1), 255, 0, 0); // Red
anim.NodeColor(nodes.Get(2), 210, 105, 30); // Chocolate
anim.NodeColor(nodes.Get(3), 165, 42, 42); // Brown
anim.NodeColor(nodes.Get(4), 210, 180, 140); // Tan
anim.NodeColor(nodes.Get(5), 255, 215, 0); // Gold
anim.NodeColor(nodes.Get(6), 0, 0, 0); // Black
// 4. Configure Wireless Channel and PHY
YansWifiChannelHelper channel;
channel.SetPropagationLoss("ns3::TwoRayGroundPropagationLossModel");
channel.SetPropagationDelay("ns3::ConstantSpeedPropagationDelayModel");
Config::SetDefault("ns3::WifiRemoteStationManager::NonUnicastMode",
StringValue(wifiDataMode));
Config::SetDefault("ns3::WifiRemoteStationManager::RtsCtsThreshold", UintegerValue(0));
YansWifiPhyHelper phy;
phy.SetChannel(channel.Create());
phy.Set("TxPowerStart", DoubleValue(pt));
phy.Set("TxPowerEnd", DoubleValue(pt));
phy.Set("CcaMode1Threshold", DoubleValue(cpThresh));
phy.Set("RxSensitivity", DoubleValue(rxThresh));
Config::SetDefault("ns3::WifiRadioEnergyModel::TxCurrentA", DoubleValue(ptConsume));
Config::SetDefault("ns3::WifiRadioEnergyModel::RxCurrentA", DoubleValue(prConsume));
Config::SetDefault("ns3::WifiRadioEnergyModel::IdleCurrentA", DoubleValue(pIdle));
Config::SetDefault("ns3::WifiRadioEnergyModel::SleepCurrentA", DoubleValue(0.001));
// 5. Configure MAC and install devices
WifiMacHelper mac;
mac.SetType("ns3::AdhocWifiMac");
mac.SetQueue("ns3::DropTailQueue", "MaxPackets", UintegerValue(ifqLen));
```

```
WifiHelper wifi;
wifi.SetRemoteStationManager("ns3::ConstantRateWifiManager",
                             "DataMode", StringValue(wifiDataMode));
NetDeviceContainer devices;
devices = wifi.Install(phy, mac, nodes);
phy.EnablePcap("aodv_metrics", devices);
// 6. Install Internet Stack and AODV
AodvHelper aodv;
InternetStackHelper internet;
internet.SetRoutingHelper(aodv);
internet.Install(nodes);
Ipv4AddressHelper address;
address.SetBase("10.0.0.0", "255.0.0.0");
Ipv4InterfaceContainer interfaces = address.Assign(devices);
// 7. Configure Mobility
MobilityHelper mobility;
mobility.SetMobilityModel("ns3::WaypointMobilityModel");
Ptr<ListPositionAllocator> positionAlloc = CreateObject<ListPositionAllocator>();
std::vector<Vector> initialPositions(numNodes);
initialPositions[0] = Vector(700.0, 240.0, antennaZ);
initialPositions[1] = Vector(500.0, 600.0, antennaZ);
initialPositions[2] = Vector(76.0, 533.0, antennaZ);
initialPositions[3] = Vector(55.0, 10.0, antennaZ);
initialPositions[4] = Vector(350.0, 210.0, antennaZ);
initialPositions[5] = Vector(700.0, 800.0, antennaZ);
initialPositions[6] = Vector(171.0, 718.0, antennaZ);
initialPositions[7] = Vector(105.0, 314.0, antennaZ);
initialPositions[8] = Vector(500.0, 20.0, antennaZ);
initialPositions[9] = Vector(832.0, 551.0, antennaZ);
```

```
  for (uint32_t i = 0; i < numNodes; ++i)
  {
    positionAlloc->Add(initialPositions[i]);
  }
  mobility.SetPositionAllocator(positionAlloc);
  mobility.Install(nodes);
  for (uint32_t i = 0; i < numNodes; ++i)
  {
    Ptr<WaypointMobilityModel> wpm = nodes.Get(i)->GetObject<WaypointMobilityModel>();
    if (wpm == 0)
    {
      NS_FATAL_ERROR("WaypointMobilityModel not found for node " << i);
    }
    if (i == 0)
    {
      wpm->AddWaypoint(Waypoint(Seconds(5.0), Vector(450.0, 50.0, antennaZ)));
      wpm->AddWaypoint(Waypoint(Seconds(simulationTime - 0.01), Vector(450.0, 50.0,
antennaZ)));
    }
    else if (i == 1)    {
      wpm->AddWaypoint(Waypoint(Seconds(10.0), Vector(650.0, 740.0, antennaZ)));
      wpm->AddWaypoint(Waypoint(Seconds(simulationTime - 0.01), Vector(650.0, 740.0,
antennaZ)));
    }
    else if (i == 2)    {
      wpm->AddWaypoint(Waypoint(Seconds(5.0), Vector(450.0, 300.0, antennaZ)));
      wpm->AddWaypoint(Waypoint(Seconds(simulationTime - 0.01), Vector(450.0, 300.0,
antennaZ)));
    }
    else if (i == 3)  {
```

```
        wpm->AddWaypoint(Waypoint(Seconds(6.0), Vector(10.0, 300.0, antennaZ)));
        wpm->AddWaypoint(Waypoint(Seconds(simulationTime - 0.01), Vector(10.0, 300.0,
antennaZ)));
    }
    else if (i == 4)    {
        wpm->AddWaypoint(Waypoint(Seconds(3.0), Vector(17.0, 102.0, antennaZ)));
        wpm->AddWaypoint(Waypoint(Seconds(simulationTime - 0.01), Vector(17.0, 102.0,
antennaZ)));
    }
    else if (i == 5)   {
        wpm->AddWaypoint(Waypoint(Seconds(14.0), Vector(810.0, 634.0, antennaZ)));
        wpm->AddWaypoint(Waypoint(Seconds(simulationTime - 0.01), Vector(810.0, 634.0,
antennaZ)));
    }
    else if (i == 6)    {
        wpm->AddWaypoint(Waypoint(Seconds(13.0), Vector(925.0, 700.0, antennaZ)));
        wpm->AddWaypoint(Waypoint(Seconds(simulationTime - 0.01), Vector(925.0, 700.0,
antennaZ)));
    }
    else if (i == 7)    {
        wpm->AddWaypoint(Waypoint(Seconds(20.0), Vector(50.0, 250.0, antennaZ)));
        wpm->AddWaypoint(Waypoint(Seconds(simulationTime - 0.01), Vector(50.0, 250.0,
antennaZ)));
    }
    else if (i == 8)    {
        wpm->AddWaypoint(Waypoint(Seconds(21.0), Vector(60.0, 200.0, antennaZ)));
        wpm->AddWaypoint(Waypoint(Seconds(simulationTime - 0.01), Vector(60.0, 200.0,
antennaZ)));
    }
    else if (i == 9)  {
        wpm->AddWaypoint(Waypoint(Seconds(24.0), Vector(750.0, 610.0, antennaZ)));
```

```
        wpm->AddWaypoint(Waypoint(Seconds(simulationTime - 0.01), Vector(750.0, 610.0,
antennaZ)));
      }
   }
   // 8. Energy Model Setup (Corrected Aggregation)
   for (uint32_t i = 0; i < numNodes; ++i)
   {
      Ptr<Node> node = nodes.Get(i);
      Ptr<BasicEnergySource> energySource = CreateObject<BasicEnergySource>();
      energySource->SetInitialEnergy(initialEnergyNode[i]);
      Ptr<WifiRadioEnergyModel> radioEnergyModel = CreateObject<WifiRadioEnergyModel>();
      // Corrected: Aggregate WifiRadioEnergyModel with the WifiNetDevice
      Ptr<WifiNetDevice> wifiDevice = DynamicCast<WifiNetDevice>(devices.Get(i));
      wifiDevice->AggregateObject(radioEnergyModel); // Attach to the device
      node->AggregateObject(energySource); // Attach energy source to the node
      energySource->TraceConnectWithoutContext("RemainingEnergy", MakeCallback([i](double
oldValue, double remainingEnergy) {
         NS_LOG_INFO("Time: " << Simulator::Now().GetSeconds() << "s, Node " << i << "
Remaining Energy: " << remainingEnergy << "J");
      }));
      radioEnergyModel->TraceConnectWithoutContext("TotalEnergyConsumption",
MakeCallback([i](double oldValue, double totalEnergy) {
         NS_LOG_INFO("Time: " << Simulator::Now().GetSeconds() << "s, Node " << i << " Total
Radio Energy Consumed: " << totalEnergy << "J");
      }));
   }
   anim.SetNodeDescription(nodes.Get(5), "MA1"); // node(5)
   anim.SetNodeDescription(nodes.Get(4), "MA2"); // node(4)
   // 9. Applications
   Ptr<Node> tcpSourceNode1 = nodes.Get(3);
   Ptr<Node> tcpSinkNode1 = nodes.Get(2);
```

```
TcpNewRenoHelper tcpReno;
tcpReno.SetAttribute("SegmentSize", UintegerValue(1000));
BulkSendHelper ftp1("ns3::TcpSocketFactory",
  InetSocketAddress(interfaces.GetAddress(2), 9)); // index for node 2
ApplicationContainer clientApps1 = ftp1.Install(tcpSourceNode1);
clientApps1.Start(tcpStartTime);
clientApps1.Stop(Seconds(simulationTime - 0.01));
PacketSinkHelper sink1("ns3::TcpSocketFactory",
            InetSocketAddress(Ipv4Address::GetAny(), 9));
ApplicationContainer serverApps1 = sink1.Install(tcpSinkNode1);
serverApps1.Start(tcpStartTime);
serverApps1.Stop(Seconds(simulationTime));
Ptr<Node> tcpSourceNode2 = nodes.Get(1);
Ptr<Node> tcpSinkNode2 = nodes.Get(6);
BulkSendHelper ftp2("ns3::TcpSocketFactory",
      InetSocketAddress(interfaces.GetAddress(6), 10)); //  index for node 6
ApplicationContainer clientApps2 = ftp2.Install(tcpSourceNode2);
clientApps2.Start(tcpStartTime);
clientApps2.Stop(Seconds(simulationTime - 0.01));
PacketSinkHelper sink2("ns3::TcpSocketFactory",
                InetSocketAddress(Ipv4Address::GetAny(), 10));
ApplicationContainer serverApps2 = sink2.Install(tcpSinkNode2);
serverApps2.Start(tcpStartTime);
serverApps2.Stop(Seconds(simulationTime));
Ptr<Node> udpSourceNode = nodes.Get(7);
Ptr<Node> udpSinkNode = nodes.Get(9);
OnOffHelper cbrSource("ns3::UdpSocketFactory",
                InetSocketAddress(interfaces.GetAddress(9), 11)); //  index for node 9
cbrSource.SetAttribute("OnTime",
```

```
StringValue("ns3::ConstantRandomVariable[Constant=1.0]"));

   cbrSource.SetAttribute("OffTime",
StringValue("ns3::ConstantRandomVariable[Constant=0.0]"));

   cbrSource.SetAttribute("PacketSize", UintegerValue(cbrPacketSize));

   cbrSource.SetAttribute("DataRate", StringValue(std::to_string(cbrPacketSize * 8 / cbrInterval /
1000000) + "Mbps"));

   ApplicationContainer cbrApps = cbrSource.Install(udpSourceNode);

   cbrApps.Start(cbrStartTime);

   cbrApps.Stop(Seconds(simulationTime - 0.01));


   Ptr<PacketSink> udpSink =
DynamicCast<PacketSink>(PacketSinkHelper("ns3::UdpSocketFactory",
                                            InetSocketAddress(Ipv4Address::GetAny(), 11))
                                    .Install(udpSinkNode)
                                    .Get(0));

   udpSink->Start(cbrStartTime);

   udpSink->Stop(Seconds(simulationTime));

   // --- Connecting Trace Sources for Metrics ---

   // 1. Packet Delivery Fraction (Sends/Receives)

   // The "Tx" trace source fires when the application sends a packet.

   clientApps1.Get(0)->TraceConnectWithoutContext("Tx", MakeBoundCallback(&AppTxTrace,
clientApps1));

   clientApps2.Get(0)->TraceConnectWithoutContext("Tx", MakeBoundCallback(&AppTxTrace,
clientApps2));

   cbrApps.Get(0)->TraceConnectWithoutContext("Tx", MakeBoundCallback(&AppTxTrace,
cbrApps));

   // For PacketSink (TCP/UDP receive):

   // The "Rx" trace source fires when the application receives a packet.

   serverApps1.Get(0)->TraceConnectWithoutContext("Rx", MakeBoundCallback(&AppRxTrace,
serverApps1));

   serverApps2.Get(0)->TraceConnectWithoutContext("Rx", MakeBoundCallback(&AppRxTrace,
serverApps2));
```

```
udpSink->TraceConnectWithoutContext("Rx", MakeBoundCallback(&AppRxTrace, udpSink));


// 2. Total Packets Forwarded (Data packets) and Total Hop Counts
   for (uint32_t i = 0; i < numNodes; ++i)
{
   Ptr<Ipv4L3Protocol> ipv4 = nodes.Get(i)->GetObject<Ipv4>()->GetRoutingProtocol();
   ipv4->TraceConnectWithoutContext("Tx", MakeCallback(&RouterForwardTrace, ipv4));
 }
// 3. AODV Routing Overhead (Routing Packets)
  for (uint32_t i = 0; i < numNodes; ++i)
{
   Ptr<Queue> queue = DynamicCast<Queue>(mac.GetQueue(i)); // to trace drops from the MAC queue
   if (queue)
   {
      queue->TraceConnectWithoutContext("Drop", MakeCallback(&PacketDropTrace));
   }   }
// 10. Simulation Execution
NS_LOG_INFO("Starting simulation for " << simulationTime << " seconds...");
Simulator::Stop(Seconds(simulationTime));
Simulator::Run();
Simulator::Destroy();
// End of Simulation: Calculate and Print Metrics
uint32_t final_drop = g_sends - g_recvs; // Data packets dropped
double total_sum_delay = 0;
uint32_t successful_recvs_for_delay = 0;
for (auto const& [packetId, endTime] : g_end_time)
{
   if (g_start_time.count(packetId) && endTime >= 0) // Check if start time exists and packet was truly received
```

```
    {
        double startTime = g_start_time[packetId];
        double packet_duration = endTime - startTime;
        if (packet_duration > 0) // Ensure valid positive duration
        {
            total_sum_delay += packet_duration;
            successful_recvs_for_delay++;
        }   }    }
    double avg_delay = 0;
    if (successful_recvs_for_delay > 0)
    {
        avg_delay = total_sum_delay / successful_recvs_for_delay;
    }
    double normalized_routing_load = 0;
    if (g_recvs > 0)
    {
        normalized_routing_load = (double)g_routing_packets / g_recvs;
    }
    double packet_delivery_ratio = 0;
    if (g_sends > 0)
    {
        packet_delivery_ratio = ((double)g_recvs / g_sends) * 100.0;
    }
     double throughput_mbps = 0;
    Ptr<PacketSink> sink1_ptr = DynamicCast<PacketSink>(serverApps1.Get(0));
    Ptr<PacketSink> sink2_ptr = DynamicCast<PacketSink>(serverApps2.Get(0));
    Ptr<PacketSink> udpSink_ptr = DynamicCast<PacketSink>(udpSink);
    uint64_t totalRxBytes = sink1_ptr->GetTotalReceivedBytes() + sink2_ptr-
>GetTotalReceivedBytes() + udpSink_ptr->GetTotalReceivedBytes();
```

```
   throughput_mbps = (totalRxBytes * 8.0) / (simulationTime * 1000000.0); // Convert bytes to bits, divide by time in seconds, then to Mbps

   // Average Hop Count: This would require a custom header in

      double avg_hop_count = 0;

   if (g_recvs > 0)

   {    }

  std::cout << "\n--- NS-3 Simulation Metrics ---" << std::endl;

   std::cout << "\t Data Send \t " << g_sends << std::endl;

   std::cout << "\t Data Receive \t " << g_recvs << std::endl;

   std::cout << "\t Total Packets Forward \t " << g_pkt_forwarded << std::endl;

   std::cout << "\t ROUTINGPKTS \t " << g_routing_packets << std::endl;

   std::cout << "\t PDR \t " << std::fixed << std::setprecision(2) << packet_delivery_ratio << "%" << std::endl;

   std::cout << "\t Normal Routing Load \t " << std::fixed << std::setprecision(2) << normalized_routing_load << std::endl;

   std::cout << "\t No. of dropped data \t " << final_drop << std::endl;

   std::cout << "\t Throughput of the network (Mbps)\t " << std::fixed << std::setprecision(2) << throughput_mbps << std::endl;

   std::cout << "\t Average End to End Delay \t " << std::fixed << std::setprecision(5) << avg_delay << "s" << std::endl;

   std::cout << "\t Average HopCount \t " << std::fixed << std::setprecision(2) << avg_hop_count << " (Requires custom hop counter)" << std::endl;

   std::cout << "\t Selfish Attack Drop Data\t " << g_atk << " (Requires custom attack logic)" << std::endl;

   NS_LOG_INFO("Simulation finished.");

   return 0;

}

template<typename T>

Callback<void, Ptr<const Packet>, T&> MakeBoundCallback(void (*f)(Ptr<const Packet>, T&), T& boundArg)

{

   return MakeCallback(f, boundArg);
```

```
}

Callback<void, Ptr<const Packet>> MakeBoundCallback(void (*f)(Ptr<const Packet>, const
ApplicationContainer&), const ApplicationContainer& apps)

{

   return MakeCallback(f, apps);   }

Callback<void, Ptr<const Packet>> MakeBoundCallback(void (*f)(Ptr<const Packet>,
Ptr<Ipv4RoutingProtocol>), Ptr<Ipv4RoutingProtocol> routing)

{

   return MakeCallback(f, routing);

  }
```

APPENDIX C: Selfish node prevention technique sample codes

```
#include "aodv-trust.h"
#include "ns3/log.h"
#include "ns3/node.h"
#include "ns3/simulator.h"
#include "ns3/ipv4-address.h"
#include "ns3/ipv4-header.h"
#include "ns3/ipv4-l3-protocol.h"
#include "ns3/udp-header.h"
#include "ns3/tcp-header.h"
#include "ns3/aodv-header.h" // For AODV control packets
#include "ns3/net-device.h"
#include "ns3/wifi-net-device.h"
#include "ns3/packet-sink.h"
#include "ns3/applications-module.h" // For application trace
#include "ns3/assert.h"
#include "ns3/pointer.h"
#include "ns3/object.h"
#include "ns3/ipv4-interface.h"
NS_LOG_COMPONENT_DEFINE("AodvTrust");
namespace ns3 {
/* TrustPacketTag Implementation */
NS_OBJECT_ENSURE_REGISTERED(TrustPacketTag);
TypeId TrustPacketTag::GetTypeId(void)
{
  static TypeId tid = TypeId("ns3::TrustPacketTag")
    .SetParent<Tag>()
```

```
    .AddConstructor<TrustPacketTag>();
  return tid;
}

TypeId TrustPacketTag::GetInstanceTypeId(void) const
{
  return GetTypeId();
}
uint32_t TrustPacketTag::GetSerializedSize(void) const
{
  return 4 + 4; // Ipv4Address (4 bytes) + uint32_t (4 bytes)
}
void TrustPacketTag::Serialize(TagBuffer i) const
{
  i.WriteU32(m_originalSource.Get());
  i.WriteU32(m_packetSequenceNumber);
}
void TrustPacketTag::Deserialize(TagBuffer i)
{
  m_originalSource = Ipv4Address(i.ReadU32());
  m_packetSequenceNumber = i.ReadU32();
}
void TrustPacketTag::Print(std::ostream &os) const
{
  os << "TrustPacketTag: Src=" << m_originalSource << ", Seq=" <<
m_packetSequenceNumber;
}
```

```
void TrustPacketTag::SetOriginalSource(Ipv4Address src)
{
  m_originalSource = src;
}
Ipv4Address TrustPacketTag::GetOriginalSource(void) const
{
  return m_originalSource;
}
void TrustPacketTag::SetPacketSequenceNumber(uint32_t seq)
{
  m_packetSequenceNumber = seq;
}
uint32_t TrustPacketTag::GetPacketSequenceNumber(void) const
{
  return m_packetSequenceNumber;
}
/* TrustEntry Implementation */
/* AodvTrustManager Implementation*/
NS_OBJECT_ENSURE_REGISTERED(AodvTrustManager);
TypeId AodvTrustManager::GetTypeId(void)
{
  static TypeId tid = TypeId("ns3::AodvTrustManager")
    .SetParent<Object>()
    .AddConstructor<AodvTrustManager>()
    .AddAttribute("TrustLogFile",
                  "Name of the file to log trust values.",
```

```
                StringValue(""),
                MakeStringAccessor(&AodvTrustManager::m_trustLogFilename),
                MakeStringChecker());
  return tid;
}
AodvTrustManager::AodvTrustManager()
  : m_trustLogFileInitialized(false)
{
}
AodvTrustManager::~AodvTrustManager()
{
  if (m_trustLogFile.is_open())
  {
    m_trustLogFile.close();
  }
}
void AodvTrustManager::Initialize(Ptr<Node> node)
{
  m_node = node;
  if (!m_trustLogFileInitialized)
  {
     std::string actualFilename = m_trustLogFilename;
     if (actualFilename.empty())
     {
        actualFilename = "trust_node_" + std::to_string(m_node->GetId()) + ".log";
     }
```

```
        LogTrustToFile(actualFilename);
        m_trustLogFileInitialized = true;
      }
      NS_LOG_INFO("TrustManager for Node " << m_node->GetId() << " initialized.");
    }
    Ptr<TrustEntry> AodvTrustManager::LookupTrust(Ipv4Address nodeAddress)
    {
      auto it = m_trustTable.find(nodeAddress);
      if (it != m_trustTable.end())
      {
        return it->second;
      }
      return nullptr;
    }
    void AodvTrustManager::UpdateTrust(Ipv4Address nodeAddress, double trustChange,
Ipv4Address prevNode, Ipv4Address nextNode)
    {
      Ptr<TrustEntry> entry = LookupTrust(nodeAddress);
      if (entry)
      {
        entry->trustValue += trustChange;
        entry->lastUpdated = Simulator::Now();
        NS_LOG_DEBUG("Node " << m_node->GetId() << ": Trust updated for " << nodeAddress
<< " to " << entry->trustValue);
      }
      else
      {
```

```
        entry = Create<TrustEntry>(nodeAddress, 0.0 + trustChange, Simulator::Now());
        m_trustTable[nodeAddress] = entry;
        NS_LOG_DEBUG("Node " << m_node->GetId() << ": New trust entry for " << nodeAddress
<< " with " << entry->trustValue);
      }

      if (m_trustLogFile.is_open())
      {
        m_trustLogFile << "Time: " << Simulator::Now().GetSeconds() << "s, "
                       << "NodeID: " << m_node->GetId() << ", "
                       << "TargetIP: " << nodeAddress << ", "
                       << "TrustValue: " << entry->trustValue << ", "
                       << "PrevHopIP: " << prevNode << ", "
                       << "NextHopIP: " << nextNode << std::endl;
      }    }
    void AodvTrustManager::LogTrustToFile(std::string filename)
    {
      if (m_trustLogFile.is_open())
      {
        m_trustLogFile.close();
      }
      m_trustLogFile.open(filename.c_str(), std::ios_base::app); // Open in append mode
      if (!m_trustLogFile.is_open())
      {
        NS_LOG_ERROR("Failed to open trust log file: " << filename);
      }
      m_trustLogFileInitialized = true;
```

```
}
/* AodvTrustWatchdog Implementation */

NS_OBJECT_ENSURE_REGISTERED(AodvTrustWatchdog);
TypeId AodvTrustWatchdog::GetTypeId(void)
{
  static TypeId tid = TypeId("ns3::AodvTrustWatchdog")
    .SetParent<Object>()
    .AddConstructor<AodvTrustWatchdog>()
    .AddAttribute("WatchdogTimeout",
                  "Timeout for detecting dropped packets.",
                  TimeValue(Seconds(0.5)),
                  MakeTimeAccessor(&AodvTrustWatchdog::m_watchdogTimeout),
                  MakeTimeChecker());
  return tid;
}
AodvTrustWatchdog::AodvTrustWatchdog()
{
}
AodvTrustWatchdog::~AodvTrustWatchdog()
{
  m_watchdogEvent.Cancel();
}
void AodvTrustWatchdog::Initialize(Ptr<Node> node, Ptr<AodvTrustManager> trustManager,
Time timeout)
{
  m_node = node;
```

```
    m_trustManager = trustManager;

    m_watchdogTimeout = timeout;

    m_watchdogEvent = Simulator::Schedule(m_watchdogTimeout,
&AodvTrustWatchdog::CheckWatchedPackets, this);

    NS_LOG_INFO("Watchdog for Node " << m_node->GetId() << " initialized with timeout " <<
m_watchdogTimeout.GetSeconds() << "s.");

    Ptr<Ipv4> ipv4 = m_node->GetObject<Ipv4>();

    if (ipv4)

    {

     ipv4->TraceConnectWithoutContext("Tx", MakeCallback(&AodvTrustWatchdog::OnIpv4Tx,
this));

    }

    else

    {

     NS_LOG_ERROR("Node " << m_node->GetId() << " has no Ipv4 Protocol installed for
Watchdog.");

    }

     for (uint32_t i = 0; i < m_node->GetNDevices(); ++i)

    {

     Ptr<NetDevice> device = m_node->GetDevice(i);

     if (device) // Check all devices

     {

       device->TraceConnectWithoutContext("PromiscRx",
MakeCallback(&AodvTrustWatchdog::OnNetDevicePromiscRx, this));

     }    }    }

  Ipv4Address AodvTrustWatchdog::GetIpv4AddressFromNetDeviceAddress(Ptr<NetDevice>
device, const Address& netDeviceAddress)

  {

    Ptr<Ipv4> ipv4 = m_node->GetObject<Ipv4>();
```

```
    if (ipv4)
    {
      for (uint32_t i = 0; i < ipv4->GetNInterfaces(); ++i)
      {
        if (ipv4->GetNetDevice(i) == device)
        {
          if (device->GetAddress() == netDeviceAddress)
          {
            return ipv4->GetAddress(i, 0).GetLocal();
          }  }    }  }
      return Ipv4Address::GetZero();
  }
  void AodvTrustWatchdog::OnIpv4Tx(Ptr<const Packet> packet, Ptr<Ipv4> ipv4,
Ptr<Ipv4Route> route, Ptr<const Ipv4Interface> oif)
  {
   if (!route)
    {
      NS_LOG_DEBUG("OnIpv4Tx: Packet with no route, not watching. Node " << m_node-
>GetId());
      return; // Not a forwarded packet (or route already expired)
    }
    Ipv4Header ipHeader;
    packet->PeekHeader(ipHeader);
    Ptr<Packet> tempPacket = packet->Copy();
    if (tempPacket->RemoveHeader<aodv::AodvHeader>())
    {
      NS_LOG_DEBUG("OnIpv4Tx: AODV control packet, not watching. Node " << m_node-
>GetId());
```

```
        return;
    }
    tempPacket = packet->Copy();
    bool isDataPacket = false;
    if (tempPacket->RemoveHeader<UdpHeader>() || tempPacket->RemoveHeader<TcpHeader>())
    {
        isDataPacket = true;
    }
     if (!isDataPacket)
    {
        NS_LOG_DEBUG("OnIpv4Tx: Not a data packet, not watching. Node " << m_node->GetId());
        return; // Only watchdog data packets
    }
    Ipv4Address nextHopIp = route->GetNextHop();
    NS_ASSERT (!nextHopIp.IsAny()); // Should have a concrete next hop
    uint32_t packetUid = packet->GetUid();
    TrustPacketTag tag;
    bool tagFound = packet->PeekPacketTag(tag);
    uint32_t packetSeqNum = tagFound ? tag.GetPacketSequenceNumber() : packetUid;
    Ipv4Address originalSource = tagFound ? tag.GetOriginalSource() : ipHeader.GetSource();
    NS_LOG_INFO("Node " << m_node->GetId() << " (IP: " << ipv4->GetAddress(0,0).GetLocal() << ") "
                        << "is watching packet UID " << packetUid
                        << " (App Seq: " << packetSeqNum << ") "
                        << "destined for " << ipHeader.GetDestination()
                        << " via next hop " << nextHopIp
```

```
                << " at time " << Simulator::Now().GetSeconds() << "s.");
      m_watchedPackets[std::make_pair(packetSeqNum, nextHopIp)] = Simulator::Now();
    }
    void AodvTrustWatchdog::OnNetDevicePromiscRx(Ptr<NetDevice> device, Ptr<const Packet>
packet, uint16_t protocol, const Address &sender)
    {
      if (protocol != Ipv4L3Protocol::PROT_NUMBER)
      {
        return;
      }
       Ipv4Header ipHeader;
      packet->PeekHeader(ipHeader);
      Ptr<Packet> tempPacket = packet->Copy();
      if (tempPacket->RemoveHeader<aodv::AodvHeader>())
      {
        return;
      }
      tempPacket = packet->Copy();
      bool isDataPacket = false;
      if (tempPacket->RemoveHeader<UdpHeader>() || tempPacket->RemoveHeader<TcpHeader>())
      {
        isDataPacket = true;
      }
      if (!isDataPacket)
      {
        return; // Not a data packet, not relevant for this watchdog
      }
```

```
  Ipv4Address observedSenderIp = GetIpv4AddressFromNetDeviceAddress(device, sender);
    if (observedSenderIp.IsZero())
    {
      NS_LOG_DEBUG("OnNetDevicePromiscRx: Could not determine IP for sender MAC " << sender << ". Ignoring packet for watchdog.");
      return;
    }
    uint32_t packetUid = packet->GetUid();
    TrustPacketTag tag;
    bool tagFound = packet->PeekPacketTag(tag);
    uint32_t packetSeqNum = tagFound ? tag.GetPacketSequenceNumber() : packetUid; // Use UID if no custom tag
    std::pair<uint32_t, Ipv4Address> key = std::make_pair(packetSeqNum, observedSenderIp);
     auto it = m_watchedPackets.find(key);
    if (it != m_watchedPackets.end())
    {
     NS_LOG_INFO("Node " << m_node->GetId() << ": Watchdog CONFIRMED packet UID " << packetUid
                 << " (App Seq: " << packetSeqNum << ") "
                 << "was retransmitted by next hop " << observedSenderIp
                 << " at " << Simulator::Now().GetSeconds() << "s. Trust increased.");
      m_trustManager->UpdateTrust(observedSenderIp, 0.05); // Increase trust by 0.05
      m_watchedPackets.erase(it);
    }
    else
    {
      NS_LOG_DEBUG("Node " << m_node->GetId() << ": Promiscuously received packet UID " << packetUid
```

```
                    << " from " << observedSenderIp << " but not currently watching it.");
    }   }


   void AodvTrustWatchdog::OnApplicationRx(Ptr<const Packet> packet, Ptr<const Application>
app)
   {
     Ipv4Header ipHeader;
     packet->PeekHeader(ipHeader);
     Ipv4Address src = ipHeader.GetSource();
     NS_LOG_INFO("Node " << m_node->GetId() << ": Application received packet UID " <<
packet->GetUid()
      << " from " << src << " at " << Simulator::Now().GetSeconds() << "s.");
      m_trustManager->UpdateTrust(src, 0.05); // This means the source delivered successfully to
me.
   }
   void AodvTrustWatchdog::CheckWatchedPackets()
   {
     Time currentTime = Simulator::Now();
     std::vector<std::pair<uint32_t, Ipv4Address>> timedOutPackets;
     for (auto const& [key, sentTime] : m_watchedPackets)
     {
       if ((currentTime - sentTime) > m_watchdogTimeout)
       {
         timedOutPackets.push_back(key);
       }    }
     for (auto const& key : timedOutPackets)
     {
       Ipv4Address nextHopIp = key.second;
```

```
      NS_LOG_WARN("Node " << m_node->GetId() << ": Watchdog TIMEOUT for packet (App
Seq: " << key.first << ")"
                  << " expected from next hop " << nextHopIp
                  << " at " << currentTime.GetSeconds() << "s. Trust DECREASED.");
      m_trustManager->UpdateTrust(nextHopIp, -0.1);
      m_watchedPackets.erase(key);
    }
    if (!m_watchedPackets.empty())
    {
       m_watchdogEvent = Simulator::Schedule(m_watchdogTimeout,
&AodvTrustWatchdog::CheckWatchedPackets, this);
    }   }
  /*TrustAodvRoutingProtocol Implementation*/
  NS_OBJECT_ENSURE_REGISTERED(TrustAodvRoutingProtocol);
  TypeId TrustAodvRoutingProtocol::GetTypeId(void)
  {
    static TypeId tid = TypeId("ns3::TrustAodvRoutingProtocol")
      .SetParent<aodv::RoutingProtocol>()
      .AddConstructor<TrustAodvRoutingProtocol>();
    return tid;
  }
  TrustAodvRoutingProtocol::TrustAodvRoutingProtocol()
    : aodv::RoutingProtocol() // Call base class constructor
  {
    m_trustManager = CreateObject<AodvTrustManager>();
    m_watchdog = CreateObject<AodvTrustWatchdog>();
  }
```

```
TrustAodvRoutingProtocol::~TrustAodvRoutingProtocol()
{    }
 void TrustAodvRoutingProtocol::DoInitialize()
{
  aodv::RoutingProtocol::DoInitialize();
   m_trustManager->Initialize(GetNode());
  m_watchdog->Initialize(GetNode(), m_trustManager);
  NS_LOG_INFO("TrustAodvRoutingProtocol initialized for Node " << GetNode()->GetId());
  this));
}
void TrustAodvRoutingProtocol::DoDispose()
{
  m_trustManager = nullptr;
  m_watchdog = nullptr;
  aodv::RoutingProtocol::DoDispose();
}
Ptr<Ipv4Route> TrustAodvRoutingProtocol::RouteInput (Ptr<const Packet> p, const Ipv4Header
&header, Ptr<const Ipv4Interface> incomingInterface)
{
    return aodv::RoutingProtocol::RouteInput(p, header, incomingInterface);
}
Ipv4RouteList TrustAodvRoutingProtocol::RouteOutput (Ptr<Packet> p, const Ipv4Header
&header)
{
   Ipv4RouteList conventionalRoutes = aodv::RoutingProtocol::RouteOutput(p, header);
  if (conventionalRoutes.empty())
  {
```

```
        NS_LOG_DEBUG("TrustAodv: No conventional route found for " <<
header.GetDestination());

        return conventionalRoutes; // No route, nothing to do

    }

     Ptr<Ipv4Route> bestTrustedRoute = nullptr;

    double highestTrust = -1.0; // Trust values are generally positive, start low

    Ptr<Ipv4Route> primaryRoute = conventionalRoutes.front();

    Ipv4Address nextHop = primaryRoute->GetNextHop();

    Ptr<TrustEntry> nextHopTrust = m_trustManager->LookupTrust(nextHop);

    double currentNextHopTrust = (nextHopTrust) ? nextHopTrust->trustValue : 0.0;

    NS_LOG_INFO("TrustAodv: Node " << GetNode()->GetId() << " evaluating route for " <<
header.GetDestination()

                            << " via next hop " << nextHop << " with trust " << currentNextHopTrust);

     if (currentNextHopTrust < 0.1) // Example threshold

    {

        NS_LOG_WARN("TrustAodv: Node " << GetNode()->GetId() << " dropping/avoiding
packet to " << header.GetDestination()

                                << " due to low trust in next hop " << nextHop << " (" <<
currentNextHopTrust << ")");

            return Ipv4RouteList();

     }

    return conventionalRoutes;

   }

  void TrustAodvRoutingProtocol::RecvReply (Ptr<Packet> p, Ptr<aodv::AodvHeader> hdr,
Ptr<Ipv4Interface> incomingInterface)

   {

      aodv::RoutingProtocol::RecvReply(p, hdr, incomingInterface);

      Ipv4Address rrepSender = incomingInterface->GetAnyAddress().GetLocal(); // This node's
address on that interface
```

```
        if (hdr->GetPacketType() == aodv::AodvHeader::AODV_REPLY)

        {

             NS_LOG_DEBUG("TrustAodv: Node " << GetNode()->GetId() << " received AODV
Reply from "

                                  << hdr->GetSourceAddress() << " for destination " << hdr-
>GetDestinationAddress());

        } }

    void TrustAodvRoutingProtocol::RecvRequest (Ptr<Packet> p, Ptr<aodv::AodvHeader> hdr,
Ptr<Ipv4Interface> incomingInterface)

    {

         aodv::RoutingProtocol::RecvRequest(p, hdr, incomingInterface);

         Ipv4Address rreqSender = incomingInterface->GetAnyAddress().GetLocal();

        if (hdr->GetPacketType() == aodv::AodvHeader::AODV_REQUEST)

        {

         NS_LOG_DEBUG("TrustAodv: Node " << GetNode()->GetId() << " received AODV
Request from "

                                  << hdr->GetSourceAddress() << " for destination " << hdr-
>GetDestinationAddress());

        } }}
```