University of Nebraska - Lincoln

DigitalCommons@University of Nebraska - Lincoln

Theses, Dissertations, and Student Research from Electrical & Computer Engineering

Electrical & Computer Engineering, Department of

Fall 12-1-2022

# A Stacking-Based Misbehavior Detection System in Vehicular Communication Networks

Troy Green
*University of Nebraska-Lincoln*, tmgreen@unomaha.edu

Follow this and additional works at: https://digitalcommons.unl.edu/elecengtheses

Part of the Computer Engineering Commons, and the Other Electrical and Computer Engineering Commons

Green, Troy, "A Stacking-Based Misbehavior Detection System in Vehicular Communication Networks" (2022). *Theses, Dissertations, and Student Research from Electrical & Computer Engineering*. 137.
https://digitalcommons.unl.edu/elecengtheses/137

This Article is brought to you for free and open access by the Electrical & Computer Engineering, Department of at DigitalCommons@University of Nebraska - Lincoln. It has been accepted for inclusion in Theses, Dissertations, and Student Research from Electrical & Computer Engineering by an authorized administrator of DigitalCommons@University of Nebraska - Lincoln.

# A STACKING-BASED MISBEHAVIOR DETECTION SYSTEM IN VEHICULAR COMMUNICATION NETWORKS

By

Troy M Green

A THESIS

Presented to the faculty of

The Graduate College at the University of Nebraska

In partial fulfillment of the requirements

For the Degree of Master of Science

Major: Telecommunications Engineering

Under the supervision of Professor Yi Qian

Lincoln, Nebraska

August, 2022

A STACKING-BASED MISBEHAVIOR DETECTION SYSTEM IN VEHICULAR COMMUNICATION NETWORKS

Troy M Green, MS

University of Nebraska 2022

Advisor: Yi Qian

Over the past few decades communication systems for vehicles have continued to advance. Communications between these vehicles can be classified into safety related and non safety related messages. An example of a safety related message would be one vehicle warning others of an icy road it encountered, where a non safety related communication would be a passenger streaming a movie. In either case it's important to secure the communications so that the system continues to behave as expected. In this thesis we propose a Misbehavior Detection System (MDS), which is a system that monitors messages sent between vehicles, and detects misbehaviors for possible attacks. In our approach we use a stacking based machine learning algorithm to determine if vehicles are misbehaving. We then compare this approach to other MDS to determine if our approach makes a measurable difference. In the analysis and comparison section of this thesis, we evaluate the simulation and performance data, showing that our protocol has an accuracy of 91.8%.

# Content

**1 Introduction** **1**

**2 Vehicular Communication Network Model and Security Architecture** **5**

**3 Background and Literature Reviews** **9**

3.1 Machine Learning Algorithms 9

3.2 Literatures in Misbehavior Detection System and Intrusion Detection Systems 13

**4 The Misbehavior Detection System and Evaluations** **16**

4.1 Proposed Misbehavior Detection System 16

4.2 Random Forest and Decision Tree 21

4.3 Artificial Neural Network 23

4.4 Simulation Tools and Evaluation Set Up 25

4.5 Performance Evaluation Metrics 27

4.6 Simulation Scenarios 28

4.7 Analysis and Comparison 43

**5 Future work** **46**

**6 Conclusion** **49**

**Bibliography** **52**

**Appendix A Modification Attack Source Code** **56**

**Appendix B Sybil Attack Source Code** **60**

**Appendix C DoS Attack Source Code** **65**

**Appendix D Blackhole/Whitehole Source Code** **69**

**Appendix E Location tracking Source Code** **73**

**Appendix F GreenMDS Source Code** **77**

# Figures


**2.1 Network model** 5

**2.2 Security Architecture** 6

**3.1 Example Decision Tree Model** 9

**3.2 Example Random Forest Model** 10

**3.3 Example ANN model** 11

**3.4 Example stacking model** 12

**4.1 Proposed MDS Architecture** 16

**4.2 Training Proposed MDS** 16

**4.3 Normalized Data** 18

**4.4 Proposed Stacking Architecture** 19

**4.5 Training Stacked based MDS** 20

**4.6 Tree Split** 21

**4.7 Navigating the tree** 21

**4.8: Random Forest** 22

**4.9 ANN model** 23

**4.10 Gradient descent** 25

**4.11 Message Modification Senario** 30

**4.12 Attackers velocity for message modification scenario** 31

**4.13 Attacked vehicles velocity for message modification scenario** 31

**4.14 Sybil attack scenario** 32

**4.15 Attackers velocity for sybil attack scenario** 33

**4.16 Attacked vehicles velocity for sybil attack scenario** 34

**4.17 DoS attack scenario** 35

**4.18 Attacked vehicles velocity for DoS attack** 36

**4.19 Blackhole attack scenario** 36

**4.20 Normal Behavior** 37

**4.21 Blackhole attacked vehicle** 37

**4.22 Grayhole attack scenario** 38

**4.23 Normal vehicle** **39**

**4.24 White hole attacked vehicle** **39**

**4.25 Location tracking attack scenario** **40**

**4.26 Ground of truth** **43**

**4.27 JSONlog.json** **43**

# Equations

**4.1 Mean Square Error** **24**

**4.2 Gradient Descent** **25**

**4.3 Accuracy** **28**

**4.4 True Positive Rate** **28**

**4.5 False Positive Rate** **28**

# Chapter 1

# Introduction

Vehicular communication networks have the potential to revolutionize the automotive industry. Communications between vehicles can provide life saving safety messages, provide passengers with entertainment, and help automatic driving. For example a vehicle driving over an icy road could send a safety message to other vehicles, letting drivers know the route is not safe and to take another route. Another example of a safety message could be a vehicle noticing someone suddenly stopped and informing other vehicles in the area they will need to stop. This message could prevent a wreck or allow them to pick another safer route. Using these safety messages will be crucial for self-driving cars, as they will use the information to help them make decisions about the route.

Though vehicular networks have lots of potential, a lot of challenges still remain. In [1] the authors present many of the challenges that remain. They classify them into the following: physical layer, synchronization, multimedia broadcast multicast service (MBMS), resource allocation, and security. Issues still remain in the physical layer, as vehicles struggle to communicate when moving at high speeds and don't support high carrier frequency. There are also issues keeping vehicles in sync with base stations, as high speeds cause lots of changes in the topology of the network. MBMS struggles with

large overlapping areas, and there are issues keeping resources separated between vehicles and cellular users. Then there are issues with keeping the system secure, as safety related messages are broadcast to all vehicles and remain unencrypted. As this thesis is focused on creating a MDS, we will focus on the security challenges in vehicular networks.

Vehicular networks are susceptible to a variety of attacks layed out in [2]. These would include the following:

- *Bogus message* : An attacker can send a false message to vehicles, in an attempt to modify other vehicle's behavior.
- *Message modification* : An attacker captures a legitimate message, modifies the contents, and sends it out in an attempt to also modify other vehicle's behavior
- *Sybil attack* : Sybil attack occurs when an attacker attempts to send a false message from multiple vehicles
- *DoS* : When an attacker attempts to spam a system with messages to stop a service, it's known as denial of service or DoS attacks
- *Eavesdropping* : A passive attack, where an attacker captures messages in order to steal information from the system
- *Impersonate attack* : An attacker sends false messages while pretending to be a legitimate user.
- *Replay attack* : Replay attack occurs when an attacker captures a message and resends it at a later time.
- *Black hole attack* : Black hole attack occurs when an attacker attempts to drop all of the messages like a black hole instead of forwarding them

- *Gray hole attack* : A gray hole attack occurs when a vehicle drops some packets selectively
- *Location tracking* : Where an attacker tracks a legitimate user's location, through the vehicular network.

In order to prevent some of these attacks we design and study a Misbehavior Detection System (MDS), which will analyze messages to ensure they act normally. As discussed in [3], an MDS can be classified into the following categories: signature-based detection, specification-based detection, and anomaly detection. Signature-based detection is where patterns for normal and malicious behaviors are stored and compared to the incoming messages. Specification-based detection is where a set of conditions is defined and the message has to follow these conditions. Anomaly detection involves the creation of a model for normal and malicious behaviors. The proposed system will use anomaly detection, so we will create a model to detect malicious messages.

For the proposed MDS we will need to create a model using a machine learning algorithm. There are a variety of machine learning algorithms we can choose from, including the following: :

- *K-nearest neighbor (KNN)* : Operates under the idea that similar data clumps together.
- *Support vector machine (SVM)* : Uses a hyperplane to divide data into two sets.
- *Decision tree (DT)* : Uses a hierarchical structure similar to a flow chart.
- *Naive bayes (NB)* : A probabilistic algorithm that gets the probability of all the features.
- *Random forest (RF)* : RF uses multiple trees.

- *Artificial neural network (ANN)* : Based on how the human brain operates and makes predictions.
- *Logistic regression (LR)* : Uses logistic function to classify data as either true or false.

To improve upon the machine learning algorithms, we can introduce an ensemble technique. This is where we combine multiple machine learning algorithms to improve the overall accuracy of the model. The main techniques are as follows: bagging, boosting, and stacking. Where bagging averages the results together, boosting is where we keep applying the machine learning algorithm until it meets an acceptable accuracy, and stacking is a two layer approach where the outputs of different machine learning algorithms feed the input of another machine learning algorithm.

We introduce a stacking based MDS, that uses a combination of DT, RF and ANN. We compare our algorithm to the one in [16] and [19], looking to see if stacking can improve the overall accuracy of the model. To do this we set up three different attack scenarios that we attempt to detect using our MDS. These include the following: message modification attack, Sybil attack, and DOS attack. We also run our algorithm against known malicious data, to see if it can detect it from normal data. We get this test data from the VeReMi dataset.

# Chapter 2

# Vehicular Communication Network Model and Security Architecture

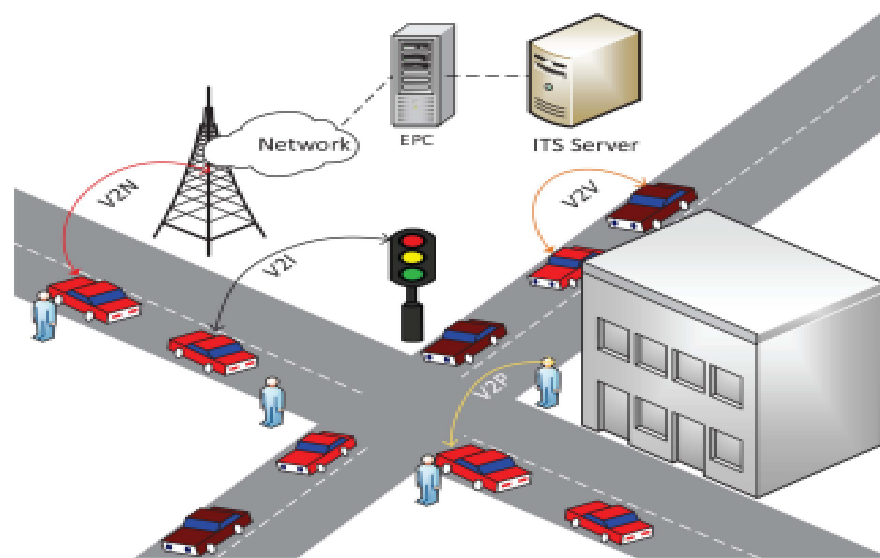


Figure 2.1 Network model

We show the vehicle communication network model and security architecture in this chapter that will be used in the rest of this thesis. We assume that in a vehicle there are two main components: the application unit, and the on board unit (OBU). The application unit is responsible for the sensors and processing of data, while the OBU is responsible for the communications between the vehicles. Data being sent to the vehicle

passes through the OBU, and is processed by the application unit. Data being sent moves in the opposite direction, going from the application unit to the OBU.

Once data leaves the OBU, it is either sent to a vehicle within range or sent to a road side unit (RSU). The RSU can either be a base station or another receiver that connects one cluster of cars to another cluster. This allows any vehicle to communicate with any other vehicle on the network. Figure 2.1 shows the network model for a vehicular network. This figure also shows a variety of communication types, including: Vehicle to network (V2N), Vehicle to infrastructure (V2I), vehicle to pedestrian (V2P), and vehicle to vehicle (V2V). The more generic term V2X means any of the mentioned communication types.

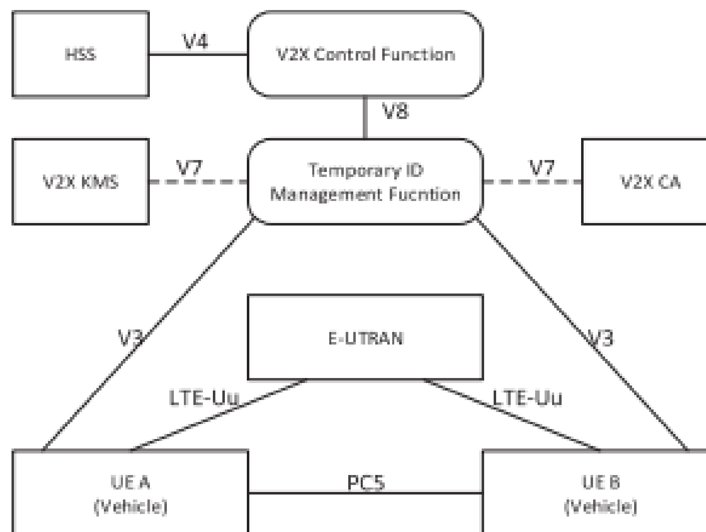


Figure 2.2 Security Architecture

Secure vehicular communications must satisfy a variety of requirements. These requirements would include the following:

- *Authentication* : Messages must be authenticated before further action is taken.
- *Integrity* : A message has not been altered.

- *Non-repudiation* : The sender/receiver cannot deny a message was sent/received.
- *Availability* : The system is up and working whenever needed.
- *Anonymity* : Users' real identities are protected.
- *Unlinkability* : A link cannot be established between a real id and temporary id.
- *Conditional traceability* : A legitimate user identity remains protected but a malicious user identity can be traced.
- *Efficient revocation* : Vehicle revocation is efficient and scalable.
- *Location privacy preservation* : A user's location cannot be traced.

All these security requirements ensure a vehicular network is secure and ready to be deployed, even though deploying a security architecture can still be a challenge. In [1] and [2] a general security architecture is discussed as shown in Figure 2.2. The security architecture shown in Figure 2.2, adds the following components:

- *V2X Control function (VCF)* : Responsible for authorization and authentication.
- *Temporary ID Management Function (TIMF)* : Responsible for issuing temporary ids and credentials.
- *Key Management Service (KMS)* : Responsible for creating and distributing keys.
- *Certificate authority (CA)* : Responsible for creating and distributing certificates.

The system will deploy a KMS and CA depending on the requirements.

In [2] the authors introduce an additional component to the security architecture, known as a trust authority (TA). The TA is responsible for keeping track of real IDs, location of APs, and reputation scores. The reputation is a number that keeps track of how trustworthy a vehicle is. If this value drops below a certain value it is considered a

malicious user and added to a revocation list of users. This will keep legitimate users from talking to known malicious vehicles.

The downside of using a reputation score is ultimately the revocation list, as this solution does not grow well. It is possible for the revocation list to expand forever, and as a result the performance of the system will degrade as time goes on. The revocation list can be updated, but then it's possible for the malicious users to attack the system.

Despite the disadvantage, in [3] the authors successfully implemented a trust based MBS. This system combines trust with machine learning in a way to detect malicious users. Machine learning and trust made a powerful combination, as even if one fails the other technique can catch the malicious user. For example, if the CRL gets cleared the machine learning technique will still be able to detect the malicious user.

# CHAPTER 3

# Background and Literature Reviews

## 3.1 Machine Learning Algorithms

As mentioned in chapter 1, there are a variety of machine learning algorithms we can use. In this thesis we will use three different algorithms, and will be focusing on them. These would include: decision tree (DT), random forest (RF), and artificial neural network (ANN). Where DT uses a tree to make decisions, RF is an extension of decision tree, instead of just using one tree they have many. ANN on the other hand is modeled off the human brain, and looks to replicate its process of thinking and predicting.

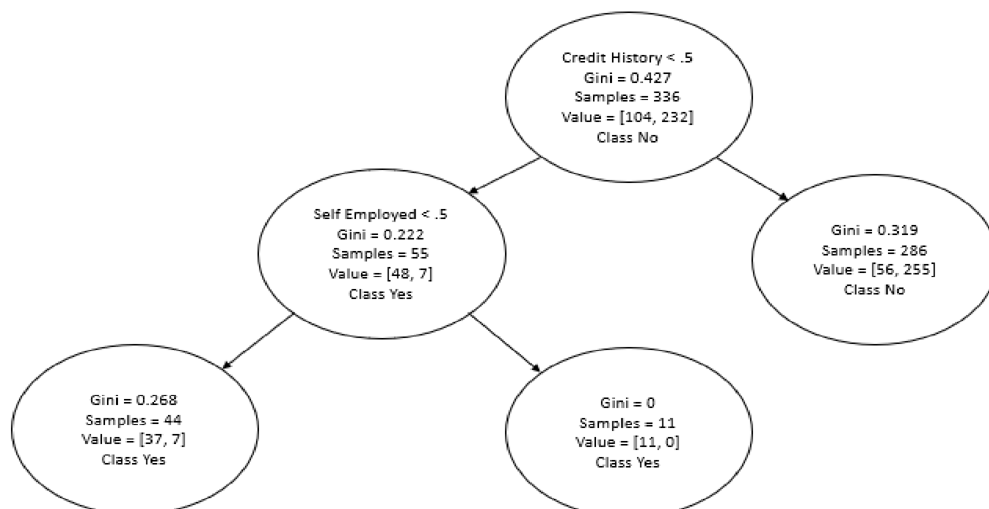


Figure 3.1 Example Decision Tree Model

DT is a type of ML that partitions training data into groups, based on a set of rules. These rules are represented as nodes, and these nodes are organized into a tree. Following nodes down the tree, results in a prediction.

Figure 3.1 shows an example of a DT. In this model they attempt to create a model to determine if an individual should receive a loan, so they look at credit history and if they are self employed. Each node has a few different values, these would include:

- Condition: If true go to the right node, otherwise the left node
- Gini: A variable that determines distribution among classes
- Value: The number datasets that belong to one class or the other
- Class: Either yes they get the loan or no they dont.

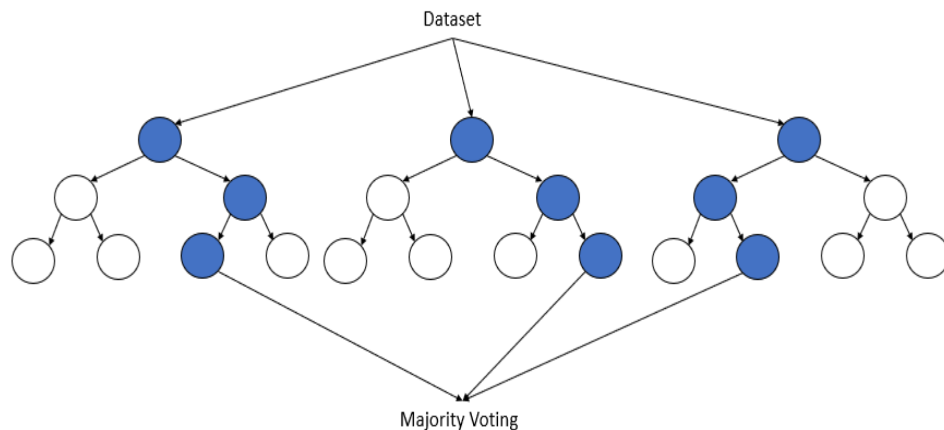


Figure 3.2 Example Random Forest Model

As mentioned before a RF is a collection of decision trees, making it an ensemble technique. In [6] the authors compare RF to decision trees, and give a good overview of the two algorithms. In order to combine the results of these trees together they use a majority vote, so whichever answer has the most votes will become the prediction. Prior to combining the results from each tree though, it must first generate all of the trees, thus

creating the forest. To do this it first collects training data, then each tree selects data at random, finally constructing a decision tree. Once the decision tree has been constructed it is free to conduct voting and get the results. Figure 3.2 shows an example of a RF model.

It concludes in [6] that the RF has a better accuracy when dealing with a large amount of data and the decision tree is better suited for smaller data sets. It should also be noted that both algorithms do better when a larger pool of data is available. They use a variety of data to make these decisions and find that the RF algorithm has an accuracy of 96.13% when the data set has 699 instances of data, but dropped to 69.23% when it has 286 instances of data. On the other hand, the decision tree has an accuracy of 75.52% when the data set has 286 instances of data, and increased to 94.6 when the data set increased to 699 instances.

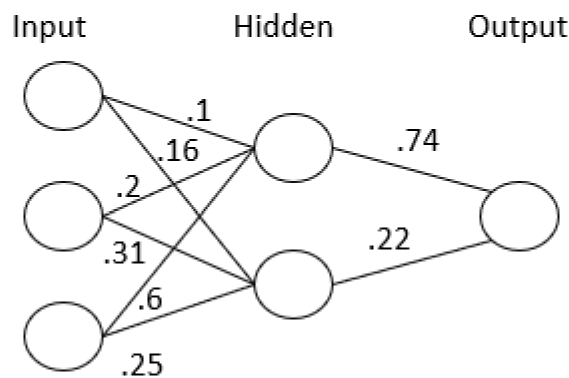


Figure 3.3 Example ANN model

ANN on the other hand is based on the human brain, making decisions and predictions in a similar manner to humans. As shown in Figure 3.1, the architecture of this algorithm is divided into three layers: input, hidden, and output. The input layer is simply the input to the algorithm, the hidden layer is where all the calculations are done,

and the output layer contains the predictions. The hidden layer can be composed into multiple layers as well, giving the algorithm more depth. Input through the hidden layer goes through a variety of transformations that give us the overall output. These transformations are determined by a weighted value on the links between nodes and activation function. The weighted values are determined using training data, and passed into the transformation function.

In [14] the authors give a detailed description of ANN, and show how to use it in an Intrusion Detection System (IDS). They conclude that their detection system has an accuracy of 98.31%, out performing all of the other compared algorithms including RF. This paper gives a practical example of how ANN can be applied to an intrusion detection system.

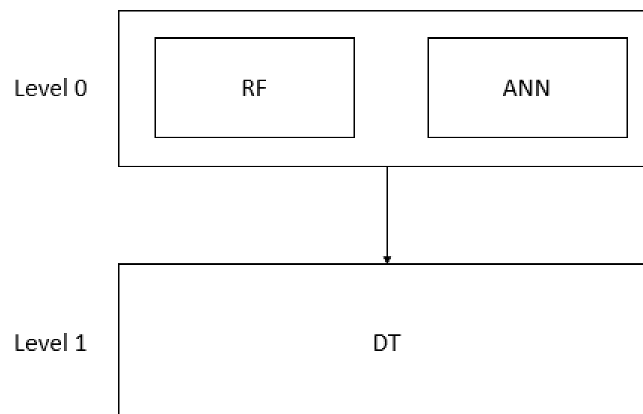


Figure 3.4 Example stacking model

As mentioned before, machine learning algorithms can be combined to make a more accurate algorithm. This combination is known as an ensemble technique, examples

include: RF, bagging, boosting, stacking, and more. In our proposed MDS we will use stacking in an attempt to make a more accurate algorithm.

In [8] the authors propose an algorithm to detect misbehaving vehicles that stacks random forest and xgboost, and shows that the algorithm has an accuracy of 98.44%. They also give an in-depth breakdown of how stacking works, showing that the algorithm has two levels: level 0, and level 1. Level 0 involves multiple machine learning algorithms making predictions, then level 1 takes the predictions and uses them as inputs to another machine learning algorithm. Figure 3.2 shows an example of this breakdown.

In [16] the authors propose an IDS that uses stacking and feature elimination. The main difference between this algorithm and [8] is the use of feature elimination, which is where they remove unneeded data. In this case they attempt to classify data into four categories, and if it doesn't match these scenarios, it is removed from the data set. This algorithm has the highest accuracy of 99%.

## 3.2 Literatures in Misbehavior Detection System and Intrusion Detection Systems

In [4] the authors propose a machine learning and reputation based MDS. In this system they use a variety of machine learning algorithms to determine if a vehicle is misbehaving. These algorithms would include the following: k-nearest neighbor, logistic regression, decision tree, random forest, and bagging. They also use reputation as a means of tracking if a vehicle has a history of misbehaving and revokes vehicles that have a low reputation.

In [5] the authors propose a statistical collaborative detection system that uses the Greenshield model to generate the needed statistics.  This model uses three parameters:

flow (q) , density (k), and velocity (v), where flow is vehicles per hour, density is vehicles per kilometer, and velocity is the speed of the vehicle. The IDS is also collaborative, meaning that data is shared among nodes in the system. This allows the nodes to collect data on each other and make an educated guess as to what other flow values should be.

In [12] the authors propose an Intrusion Detection System (IDS) using a combination of logistic regression and boosting. The IDS is divided into three parts: input, analysis, and output. The input section involves acquiring data that can be used to create a model. The analysis part is where it can apply machine learning and create the model. In this part it starts by doing feature ranking and selection. Where different features are ranked by occurrence, then passed to the machine learning algorithm. In this case the authors applied logistic regression and XGBoost to try to evaluate data as malicious or not. They conclude that XGBoost is more accurate than logistic regression.

In [16] the authors propose an Intrusion Detection System, using an ensemble technique known as stacking. The algorithm is divided into three stages: data preprocessing, feature elimination, and stacking. In the first stage training data is normalized into a common format, so that the algorithms can understand the data and be trained. In the feature elimination stage, the algorithm attempts to eliminate unneeded data. This means any data that can’t be classified into the four attacks are eliminated, along with duplicated data. The final phase is where stacking is applied, and where the final output is generated. They stack the following algorithms: random forest, adaboost, GBDT then combine the results using a decision tree. The proposed algorithm always performs better than other algorithms with an accuracy of 99%.

In [18] the authors propose an IDS that detects Distributed Denial of Service (DDOS) attacks, using a combination of machine learning and common feature extraction. This paper looks at three machine learning algorithms to determine if a data set is malicious. They compare decision tree, random forest, and gradient boost. They combine these algorithms with a technique for filtering out the average feature values, leaving the more discriminative features within the data set. They evaluate the technique with each of the machine learning algorithms mentioned before, and find which algorithm is more accurate in different situations. Decision tree and gradient boost perform worse as the percent error increases. Random forest performs well with only a little bit of training, so therefore random forest seems to be the more reliable algorithm.

In [19] the authors propose a misuse IDS based on a three layer Recurrent Neural Network (RNN). The proposed model uses an input layer, 2 hidden layers, and an output layer. The two hidden layers group the input layer data into different feature categories, the connection between the two hidden layers is considered partial. The output has five output neurons, representing the different attack scenarios and the normal condition.  The paper then evaluates and compares the algorithm to others. In this case they evaluate detection rate (DR), false alarm rate (FAR), and cost per example (CPE). The proposed algorithm has a detection rate of 94.1%, a false alert rate of .38%, and a cost of 0.16666.

# CHAPTER 4

# The Misbehavior Detection System and Evaluations

## 4.1 Proposed Misbehavior Detection System

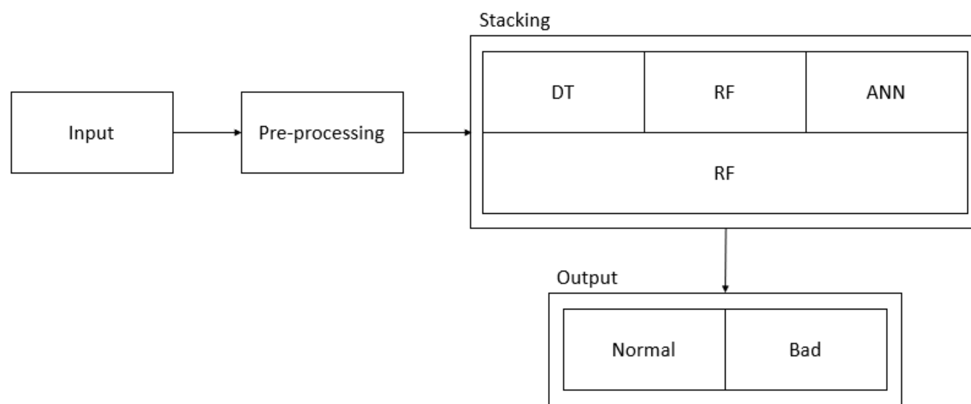


Figure 4.1 Proposed MDS Architecture

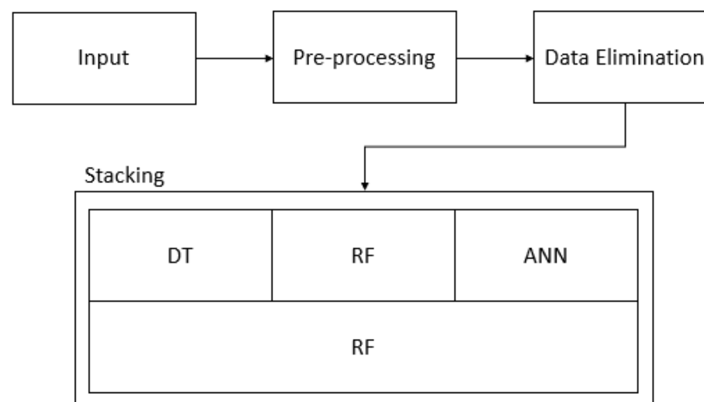


Figure 4.2 Training Proposed MDS

In Figure 4.1 the proposed architecture of the MDS is shown. This architecture includes the following parts:

- Input: Is where the user passes in data we want to classify
- Preprocessing: Is where the input data is normalized.
- Stacking: The proposed algorithm that classifies the data.
- Output: The classification, either good or bad.

The proposed system makes use of RF, DT, and ANN. Then stacks the results together in an attempt to improve the overall accuracy of the system. The general idea of each of these algorithms have been discussed in chapter 3.1 and will discuss our implementation further in the chapter.

Figure 4.2 shows how we train the proposed system. The training phase is divided into the following:

- Input: Where all of the training data is collected from the user
- Preprocessing: Input data is normalized
- Data elimination: Similar data is removed from the training data
- Stacking: Each of the used algorithms are trained with the remaining data

Figure 4.1 and 4.2 are very similar, but details two different processes. In the input phase we are working with a list of data as opposed to a single data point. We are also applying data elimination to eliminate data that is too similar to the rest of our data. Also the Stacking phase is also different, as we are training up each of our algorithms, instead of using them to classify a single piece of data. The preprocessing section on the other hand is the same regardless of which phase we are in.

Position

Position Noise

Speed

Speed Noise

Figure 4.3 Normalized data

In order to process data it is normalized into a known log class. This class pulls out the relevant features and converts it into a number that we can then use for comparison. As shown in Figure 4.3, we are interested in the following features: position, position noise, speed, and speed noise. Where all features are a vector in the form (x, y, z). At this point the data is still a string so we convert each feature into a double, then add all the features together. This gives us a value that we can then use in the data elimination phase. Going forward we will call this the normalized value.

The first part of the elimination phase is to organize the data by normalized value. Once the data is organized, we look to eliminate data that is similar to the average normalized value. So we calculate the average value and look to eliminate data that is similar to this average. To do this we run through all the data and compare the normalized value to the average and see if it's within a threshold, if it is, we remove it from our list. The next part is to eliminate redundant data, so again we look at the normalized value. As the list of data is already organized we can assume that similar redundant data will be next to each other and therefore removed if the previous value is the same.

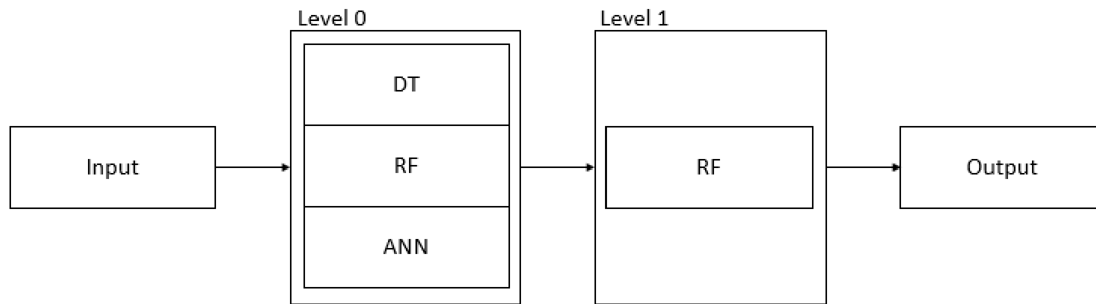


Figure 4.4 Proposed Stacking Architecture

Once the data has been entered into the system, we want to run it through the stacking phase of the proposed MDS. As mentioned in chapter 3.1, stacking is divided into two different levels. The first level runs data through a couple of different machine learning algorithms. In our case we are using DT, RF, and ANN. The second level takes data from the first level and combines it using another machine learning algorithm. Our proposed system uses random forest to combine the results of the first level. Figure 4.4 shows the proposed stacking architecture.

In order to train the model we split our training data into two different parts, the first is for level 0 and the second is for level 1. The process of splitting data into multiple sets is known as folding the data, where each split is a fold. We use the first fold to train RF, DT, and ANN. So, at this point we have three machine learning models that can be used to make predictions, but need to train the level 1 algorithm. To do this we run the level 0 algorithms against the second fold to make predictions, and train the level one model, another instance of RF with their outputs. At this point the level 0 models are discarded and we are left with only the level 1 model for making predictions. Figure 4.5 shows how level 0 models are trained and feed the level 1 model.

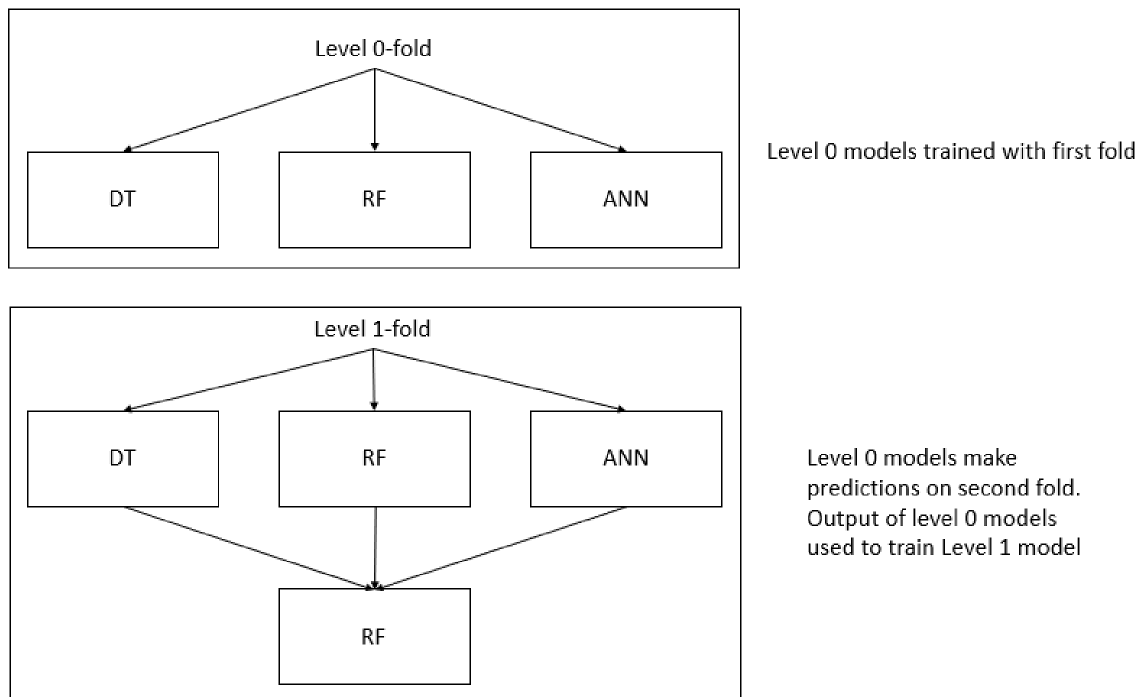


Figure 4.5: Training Stacked based MDS

Once the stacking phase has been completed, we should have a model that is capable of making predictions. The output phase classifies output from the stacking phase into two different types. These types are as follows: normal and bad. So if the output reports this is a normal feature, then it should be safe for a vehicle to use the corresponding data. If the output reports it as bad, then the data is considered malicious.

## 4.2 Random Forest and Decision Tree

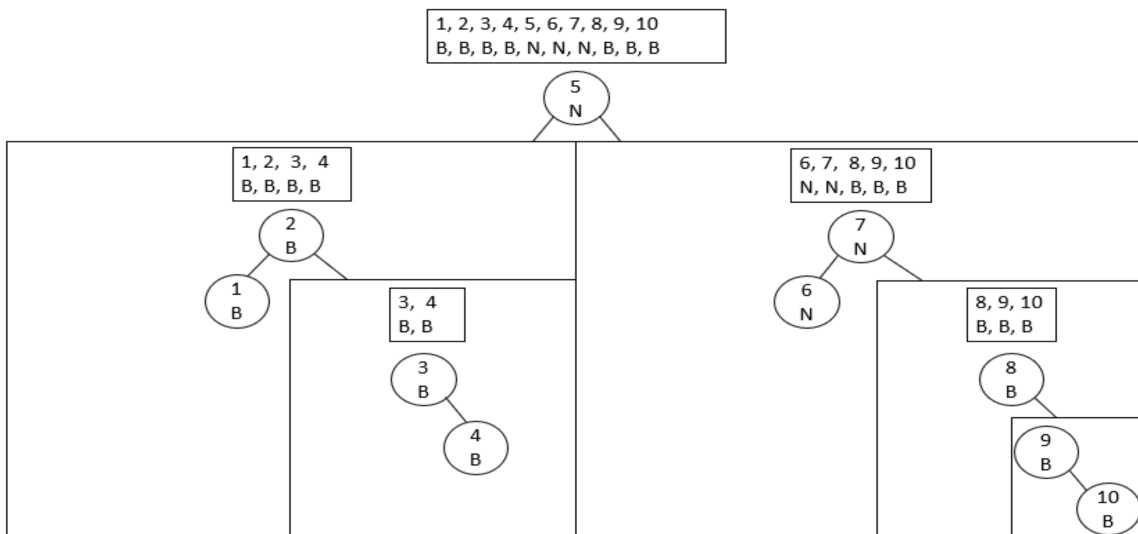


Figure 4.6 Tree Split

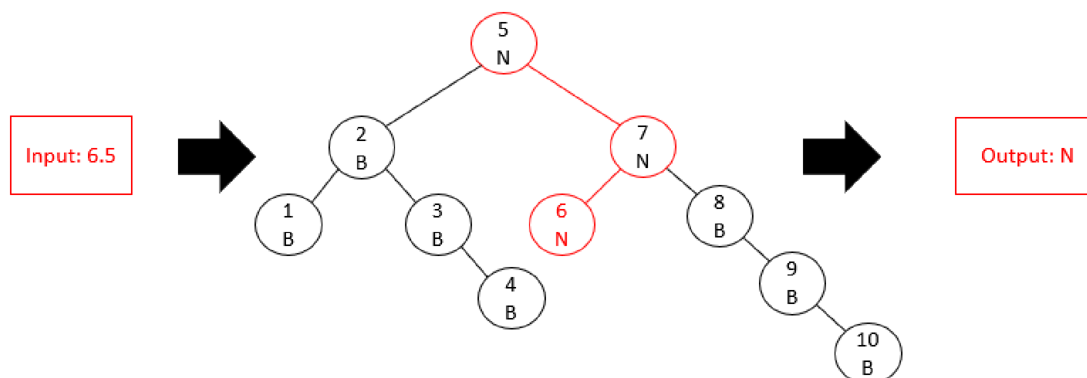


Figure 4.7 Navigating the tree

As part of the proposed system, we had to implement a version of Decision Tree and Random Forest. This is because we stacked Random Forest in an attempt to create a more accurate system. Our implementation of Decision Tree is split into two phases, the first is training. During this phase an organized set of data is passed into the algorithm

and is turned into a tree using a recursive algorithm. This tree represents a model that we can then use in the testing phase. Figure 4.6 shows an example of this process.

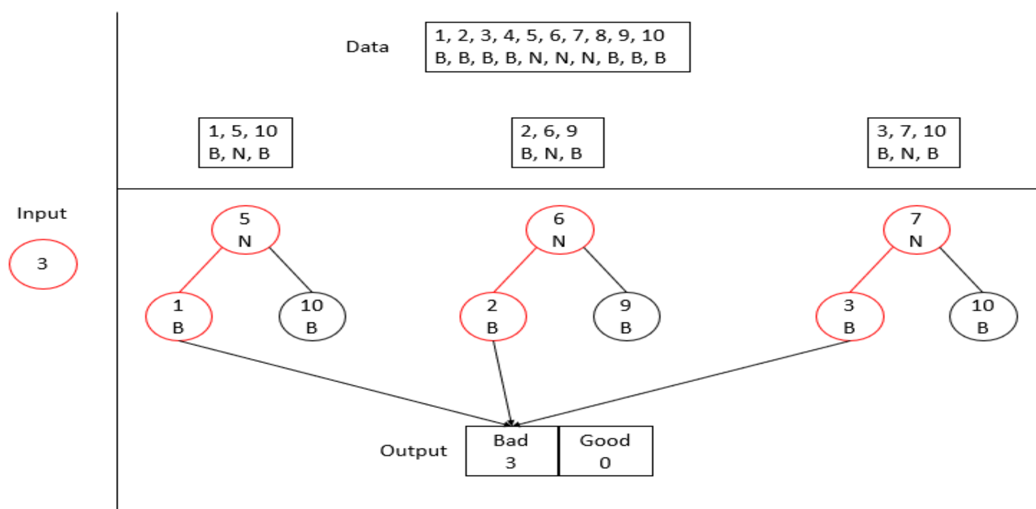


Figure 4.8: Random Forest

Once the training phase is complete and we have a working model, we can then move into the testing phase, where we can classify the testing data into either normal or into one of the attack scenarios. In this phase of the application, data is passed to the model and compared to the other nodes.

The data is first compared to the root of the tree, if the testing data is greater than the tree data it will continue down the right side of the tree. If the training data is smaller it will go down the left side of the tree. If the data is equal, we can say the training node and testing data are the same and can classify them the same. We will continue to move down the tree until either the data is equal, we are on a leaf node, or the next node is not defined. At this point we will classify the data the same as the node we are on. Figure 4.7 shows an example of this process.

For the next part of the proposed system we had to create a random forest. As mentioned before a random forest is a combination of many decision trees. For our

system we created 128 trees, each having 200 nodes. Each node is randomly selected from training data. Once the forest is created, we need it to make a prediction. For this we implement voting, where each tree votes if the testing data is good or bad. Figure 4.8 shows an example of a random forest.

## 4.3 Artificial Neural Network

As mentioned in section 3.1, ANN is a machine learning algorithm based on the human brain. In our proposed system we implement ANN with the following: input layer, hidden layer one, hidden layer two, and an output layer. For our system we implement 4 input nodes, 60 nodes within the first hidden layer, 30 for the second hidden layer, and one output node. For the input layer we pass in the features from section 4.1. These features will activate nodes in hidden layer one, which in turn will activate nodes in the hidden layer two, and in turn will activate the output node. We decide if a node is active or not using the Rectified Linear Activation function. If the output node activates, we predict the data is bad, otherwise the data is normal. Figure 4.9 shows an example of this model.

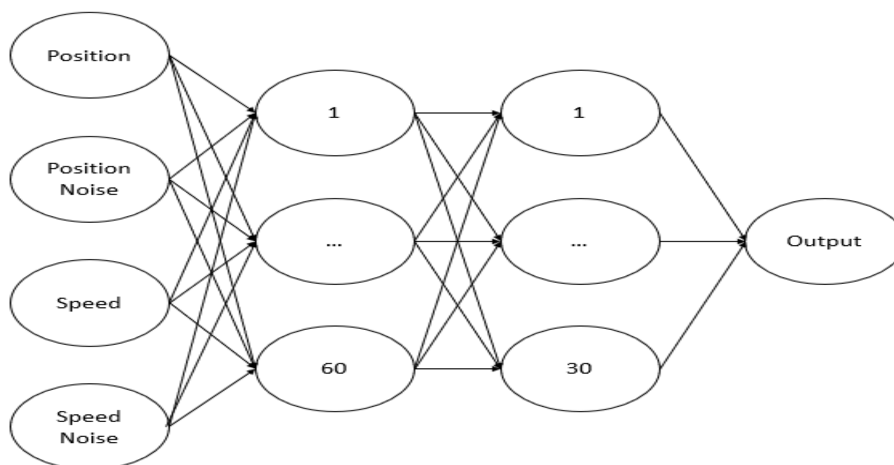


Figure 4.9 ANN model

Rectified Linear Activation is a simple function we can use to determine if a node is activated or not. This function simply returns the max of either 0 or the calculated value of the node. So if the value is zero or below, zero is returned and the node is considered inactive. On the flip side, if the calculated value is above zero, the value is returned and the node is considered active. This function has the advantage of being computationally simple and has a linear relationship.

To train the data we must adjust weights on the links between the nodes. To do this we first assign random values to each of the weights. First we go through the feedforward stage, where a prediction is made using the current values of the weights. Then we go through the feedback stage, where we use the prediction and expected result to calculate the error. Using this error value, we adjust the weights. From there we keep adjusting the weight until the algorithm makes the correct prediction.

To calculate the error of a prediction, we use the mean squared error (MSE) function. The MSE is the following:

$$\frac{1}{n}\sum_{i=1}^{n}(Yi + \hat{Y}i)^2 \tag{4.1}$$

- n: is the total number of data points
- $Y_i$: is the expected value
- $\hat{Y}_i$: is the predicted value
- i: is a single data point

Once the error is calculated we can pass it to a function that adjusts the weights.

Once we have the error, we can pass it to a function that adjusts the weight. This function uses gradient descent. The idea behind this is to find a local minimum of a function by slowly adjusting the weights. To do this we first assign a random value to the

weight, then subtract a small value from it. Figure 4.10 shows an example of how the weights are adjusted.

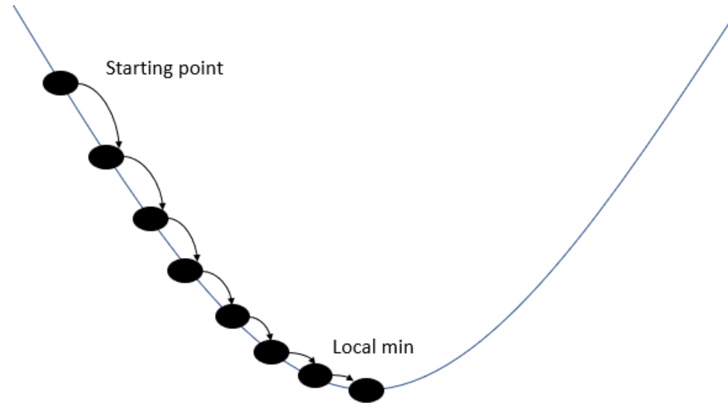


Figure 4.10: Gradient descent

Gradient descent can be represented as the following:

$$a_{n+1} = a_n - \gamma F(x) \tag{4.2}$$

- $a_{n+1}$: is the new weight we can apply
- $a$ : is the original weight
- γ: is the learning rate, a constant used to determine how fast to adjust the weights
- F: is a function, in our case we use tanh
- $x$: is the variable we pass into the equation. In our case its the error

## 4.4 Simulation Tools and Evaluation Set Up

To implement and analyze our proposed MDS model we use the following simulation tools: OMNET++ [5,6], SUMO [6], and Veins [5], where OMNET++ is a simulator to create and test network applications, SUMO generates vehicle data, and Veins is built on top of OMNET++ to simulate vehicular networks.

OMNET++ is an open source, modular application that developers can create plugins and tools for. The application is built on top of the popular IDE (Integrated Development Environment) known as eclipse, that many software engineers are familiar with. OMNET++ allows programmers to define .ned files, which define a network. Using this file, programmers can create network nodes and link the nodes together to form a network, then use C++ to define how these nodes will behave. From this a programmer can define a variety of networks and implement different protocols on that network.

SUMO is a full suite of traffic modeling utilities [6]. This is an open source tool that anyone can download and contribute to, which makes it popular for mobility simulation. Road networks can be generated using a tool called "netgen" or importing a digital road map. Using these road networks, a trip can be calculated using the application "dfrouter". One possible application for SUMO is V2X communication networks, as users can add network scenarios to the modeled vehicles. In [6], the authors introduce several recent research projects, including the following: iTetris, VABENE, cityMobil. They demonstrate the extensions to the application scenario, including: Emission and Noise Modeling, and Person based Intermodal Traffic Simulation. They finally show add-ons to SUMO, including the following: car-following and lane change API, model improvements, interoperability, network editor.

Veins is a library built using OMNET++, that defines all of the components of a vehicular network. Using this library users can interface with SUMO to get data, create vehicles that can communicate to each other, and through RSUs to ensure groups of vehicles can communicate to other groups that are out of range. This library acts as the glue in between OMNET++ and SUMO, creating different vehicular networks.

In [5] the authors introduce an alternative to Veins, called VACaMobil. This is a vehicular communication simulator built on top of OMNET++. They introduce some of the features that are added to OMNET++, including the following: set number of average vehicles, the upper bound, and the lower bound. To do this the simulator is given three tasks: (1). Manages when new vehicles get introduced, (2). Assigns a random route to the vehicle, (3). Determines the type of vehicle to be added.

## 4.5 Performance Evaluation Metrics

Most of the projects in the related work section used the same metrics to evaluate their algorithms. They use the following three metrics to evaluate their algorithms: accuracy, sensitivity, and true positive rate. Accuracy is a percent value to measure how often the algorithm successfully detected a malicious message. Sensitivity is how likely the algorithm is to identify the message is malicious, whether it's true or not. Finally the true positive rate is the actual number of malicious messages in the experiment.

In this thesis we will use the following metrice, to evaluate the proposed MDS:

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN} \quad (4.3)$$

$$True\ Positive\ Rate = \frac{TP}{TP+FN} \quad (4.4)$$

$$False\ Positive\ Rate = \frac{FP}{FP+TN} \quad (4.5)$$

- True Positive (TP) the number of malicious messages successfully detected
- False Positive (FP) the number of normal messages that were said to be positive
- True Negative (TN) the number of normal messages that were normal
- False Negative (FN) the number of malicious messages that were said to be normal

There are other metrics that could be used, for example, in [13] the authors use the following: Accuracy, Precision, Recal, F1 Score, and Log Loss. Precision is the percentage of correctly classified data, Recall is the same as sensitivity, F1 Score is a measure of the accuracy, and log loss is a function used to evaluate the performance of the algorithm.

In order to calculate the Accuracy, Sensitivity, and False Positive rate we will need to collect data from a series of experiments. While running these experiments we will need to keep track of the TP, FP, TN. To do this we use the results that OMNET++ can generate. We can compare our proposed scheme to several existing ones.

## 4.6 Simulation Evaluation Scenarios

We evaluate a variety of different attack scenarios including: message modification, Sybil, DoS, blackhole, grayhole, and location tracking. Message modification attack is where the attacker is altering the content of a message, Sybil attack sends false messages with a variety of senders, DoS attack attempts to spam a node with messages to disable it, blackhole drops all packets, grayhole drops some of the packets, and location tracking attempts to gain the location of a particular vehicle.

In [20] the authors give an in-depth analysis of the Sybil attack in a vehicular network. This paper defines a Sybil attack, which is where false messages are broadcast but with different identities. In this way an attacker can pretend to be multiple vehicles reporting the same issue, thus making the false message more convincing to other vehicles.

The authors of [21] explore different DoS attacks in a VANET environment. There are two types of application layer DoS attack in the paper, one where the attacker targets a single user and continuously sends safety messages to them. This prevents a single user from getting a potential safety related message in a timely manner. The other type is when the attacker focuses on the RSU, in this scenario an attacker sends bad messages to the RSU preventing it from getting messages from legitimate users in a timely manner.

The authors of [22] conduct an in depth analysis of man-in-the-middle (MITM) attacks in VANETs. They classify MITM attacks into passive attacks where an attacker captures data in an attempt to gain information about different users on the network. The example given in the paper is to track law enforcement or emergency services. The other classification is active, where an attacker purposely attempts to manipulate the behavior of vehicles by dropping, delaying, or modifying a message.

The authors of [2] cover black hole attacks in vehicular networks. This is a network layer attack that involves a vehicle dropping all incoming messages. This makes it so any important safety messages will be dropped. Going back to our initial example at the beginning of our paper, there could be a slippery road ahead that one vehicle attempts to notify others about. It could get forwarded to an attacker that has implemented a black hole, dropping the message and forcing other vehicles onto the slippery road.

The authors of [2] also cover white hole attackers in vehicular networks. This attack is very similar to black hole, but selectively drops packets. Just like the black hole attack this is also a type of network layer attack. Looking at our example at the beginning of our paper, this would functionally work just like the black hole. Where the safety

message is dropped. The difference is that all other types of messages continue to be processed as normal. This means the attacker could potentially blend in more with the other traffic and not blatantly look like a misbehaving vehicle.

The Location Tracking scenario is a passive type of attack, also covered by the authors in [2]. In this type of attack no modification to the message contents occurs, nothing is added, nothing is removed. Instead the attacker just listens to incoming messages in an attempt to find an id and GPS coordinates associated with the id. Then using the GPS coordinates, the attacker can track a particular vehicle.

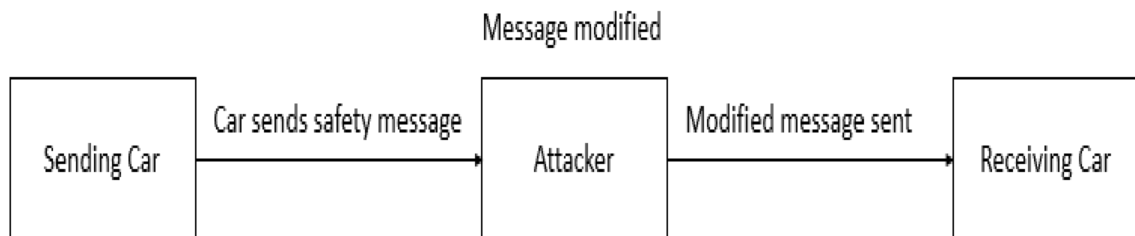


Figure 4.11 Message Modification Senario

We use the example project provided by Veins as a starting point for all of our scenarios. This project simulates 194 cars leaving the University of Erlangen-Nuremberg. When the wreck occurs, vehicles send out safety messages letting each other know that a wreck has occurred and to take another route. Looking at the reports generated from the scenario, we can see that most of the cars stop moving as the velocity of the vehicles is zero during this time.

Figure 4.11 shows a model of the message modification scenario. In this scenario the attacker waits for an accident to occur and captures the message, then resends the message at a later time. This way vehicles moving along the same route will divert,

thinking an accident has occurred. In reality no accidents have occurred and the attacker has the road to himself.

In order to simulate Figure 4.11 we add an attacker vehicle to the Veins example. This attacker vehicle mostly behaves the same as the other vehicles, but stores off any safety related message. As the only one in this example is an accident, we know what to expect. Then once the safety related message is stored, we use the position update function to constantly resend the same safety message. When other vehicles receive this message they will divert to a different route, clearing the way for the attacker.

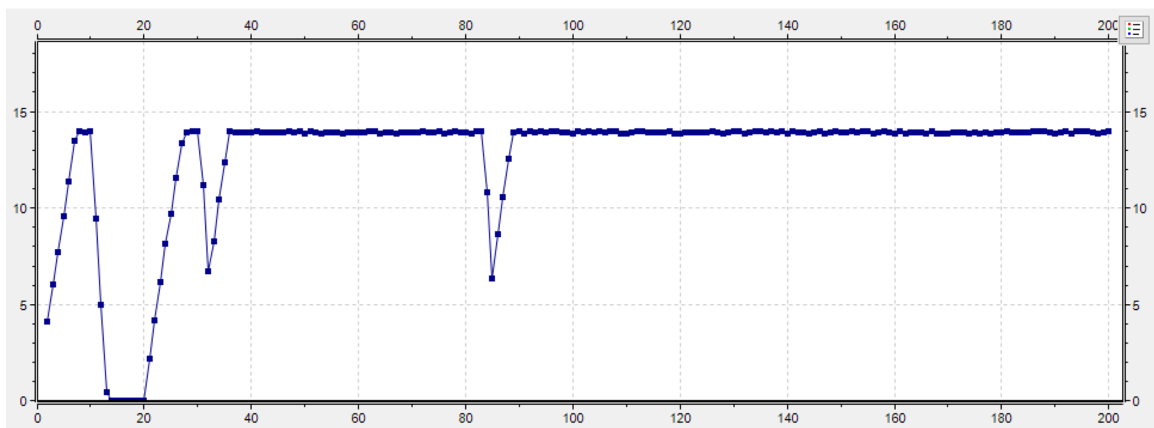


Figure 4.12 Attackers velocity for message modification scenario

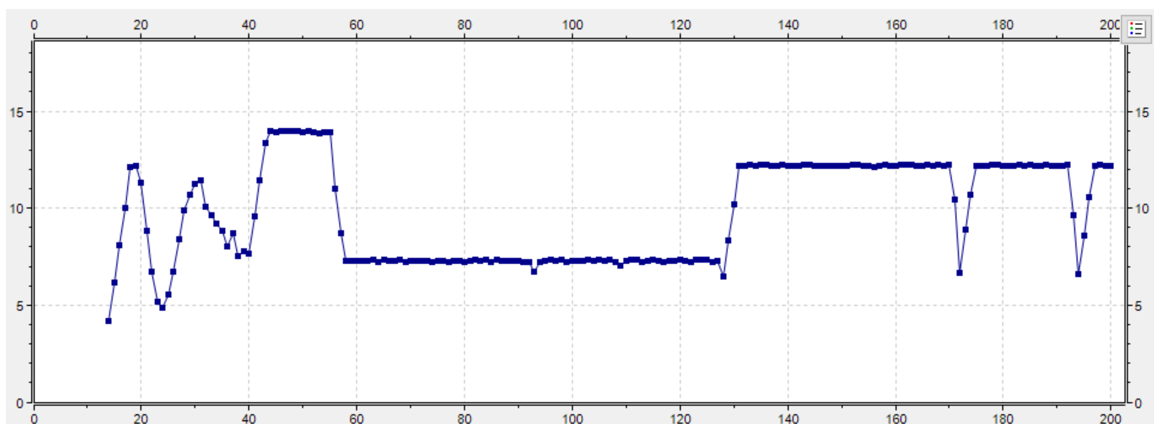


Figure 4.13 Attacked vehicles velocity for message modification scenario

Figure 4.12 shows the attackers velocity throughout the scenario. You can see at time unit 12 the velocity becomes zero, this is because an accident has occurred. At this time the attacker captures the message, so that he can send it at a later time. Once the attacker starts sending out the false message the vehicle maintains the max velocity (14 mph).

On the flip side, Figure 4.13 shows the velocity of an attacked vehicle. You can see that the vehicle reaches the max velocity, right before receiving the malicious message. Once the message is received the vehicle slows down to about 7 mph for a significant portion of time. The vehicle never reaches the max velocity again, and does not reach its destination before the simulation ends. See appendix A for the Message Modification attack scenario source code.

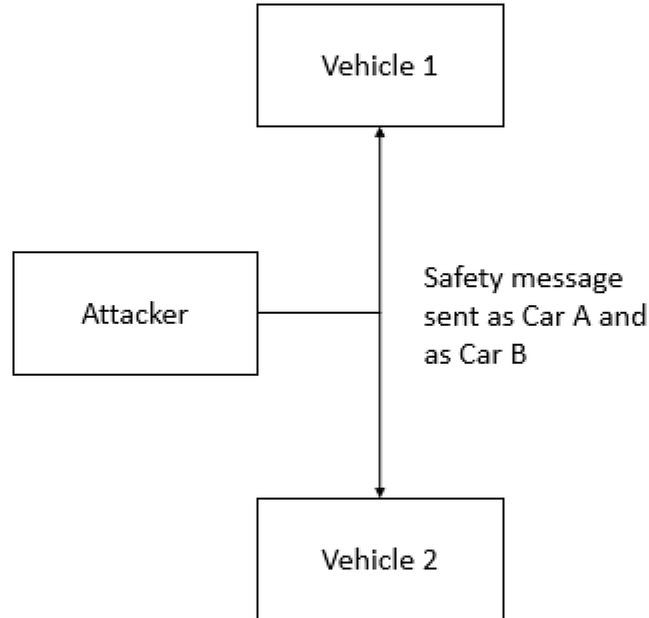


Figure 4.14 Sybil attack scenario

Figure 4.14 shows a model of the Sybil attack scenario. In this scenario an attacker will send out a safety message as both car A and car B. In this way the attacker attempts to fool other vehicles that a wreck has occurred and to take another route, thus once again clearing the way for the attacker. Much like the message modification scenario, we add an attacker to the example veins project. We also add a beacon message to the simulation, so that vehicles will announce themselves to the network. This allows the attacker to obtain the identities of two different vehicles. Once the attacker has these two different identities they send out accident messages as the two different vehicles. This way it will look like two different vehicles are reporting the accident, that are not the attacker, making the attack seem more believable.

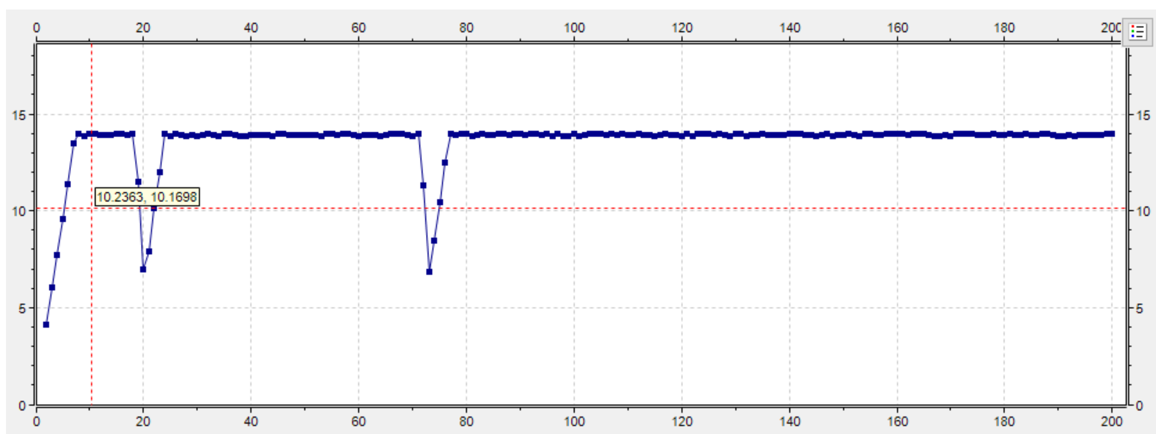


Figure 4.15 Attackers velocity for sybil attack scenario

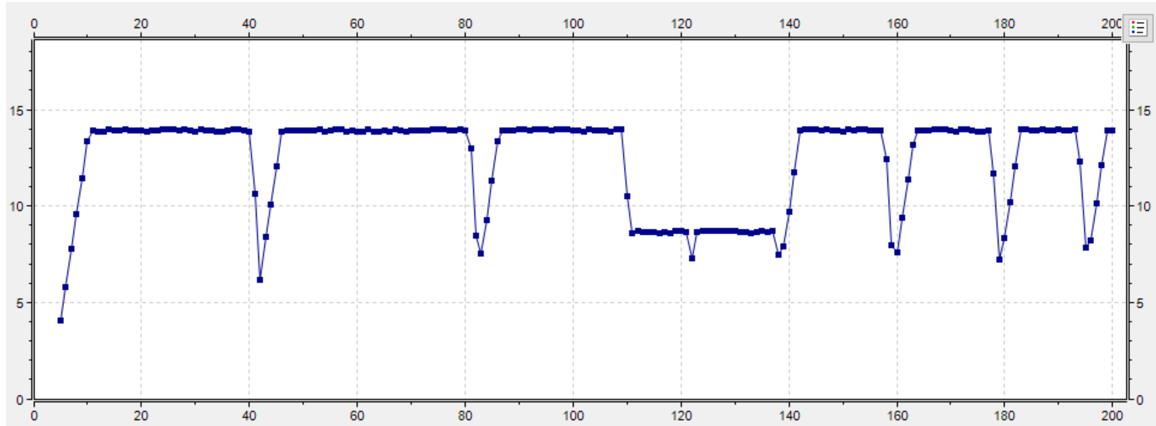

Figure 4.16 Attacked vehicles velocity for Sybil attack scenario

Figure 4.15 shows the velocity of the attackers' vehicle during the sybil attack scenario. Looking at this graph we can see that the attacker has small dips in velocity, indicating that they made a turn, however, for the most part they maintained the max velocity throughout the simulation.

On the other hand Figure 4.16 shows the speed of an attacked vehicle. This vehicle received a false safety message, telling it that a wreck was ahead and to divert along another route. Looking at this graph we can see that there are many small drops in the vehicle's velocity, indicating that the vehicle had to make many turns. Since the attacker only had a couple of dips in comparison, we can see that the attacked vehicle took a different, longer route. The attacked vehicle did not reach its destination before the simulation ended. See appendix B for the Sybil attack scenario source code.

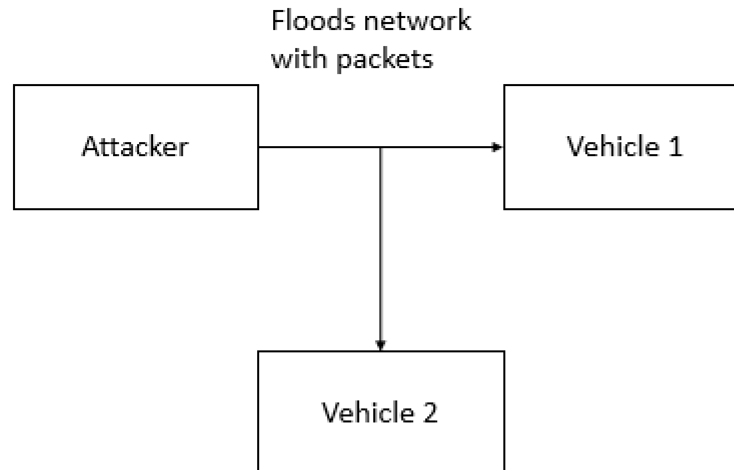


Figure 4.17 DoS attack scenario

Figure 4.17 shows a model of the DoS attack scenario. In this scenario, an attacker floods the network so that other vehicles cannot receive safety related messages. This scenario is different from the other scenarios, in that it prevents vehicles from receiving safety related messages at all. In this way when a wreck actually occurs, no one will redirect and a traffic jam will occur. Though the attacker will not benefit from this scenario, it is possible that extremists might want to take advantage of this vulnerability.

In order to simulate this scenario, we add an attacker vehicle to the example Veins project. This attacker vehicle will immediately start spamming the network with beacon messages, thus each legitimate vehicle will have to spend time processing these worthless messages.

When the wreck occurs, the majority of the vehicles fail to receive the safety messages in a timely manner. This means by the time they process the safety related messages they are already stuck in the traffic jam, and cannot do anything with this information.

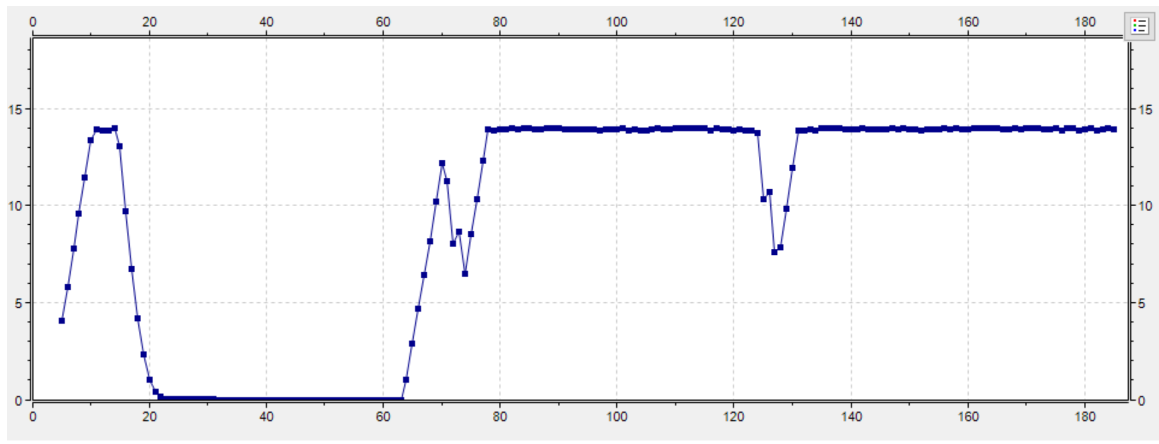


Figure 4.18 Attacked vehicles velocity for DoS attack

Figure 4.18 shows one of the attacked vehicles during the DoS scenario. We can see that at around time unit 20, the vehicle's velocity reached zero and was not moving. This is the same for all of the vehicles in the network, as the attacker is spamming the network with worthless beacon messages. As a result everyone gets stuck in the traffic jam and cannot divert. See appendix C for the DoS attack scenario source code.

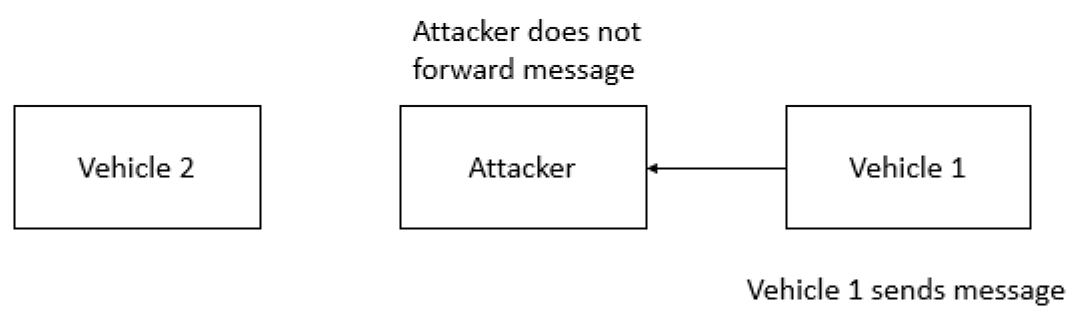


Figure 4.19 Blackhole attack scenario

Figure 4.19 shows a model of a blackhole attack. In this scenario we have a vehicle that attempts to send out a safety message, in our case trying to inform others of a car crash ahead. However when the attacker receives the message he drops it, not allowing others to know the accident has occurred. This means other drivers would drive

to where the accident occurred and caused a traffic jam, and the attacker can divert taking a shorter path.

In appendix E we have attached the source code to simulate a black hole attack. In this simulation we utilize the same example project as a starting point as the others. We then limit the simulation to 3 cars. One that forwards a safety message, a malicious user that drops all packets, and a victim that does not get the safety message. We limit this simulation as a means to limit the amount of messages in the system and help us create a point to point message system. Normally Veins broadcasts the messages to all cars connected to the RSU.

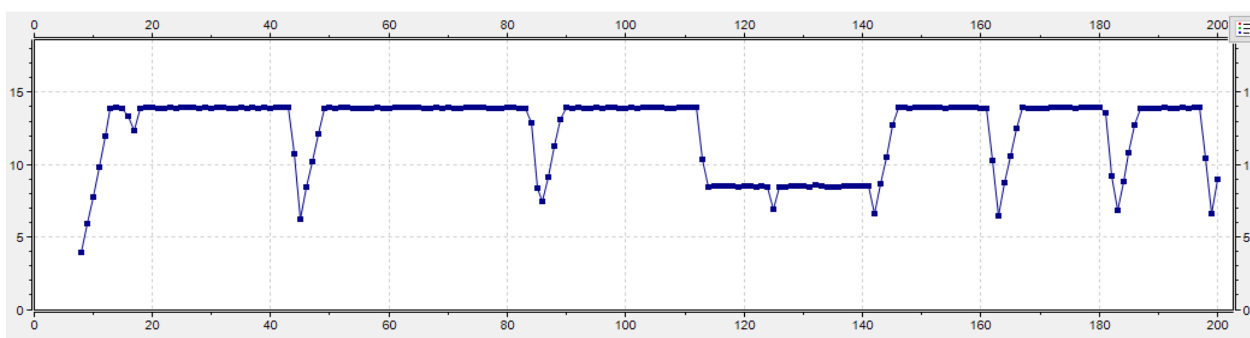


Figure 4.20 Normal Behavior

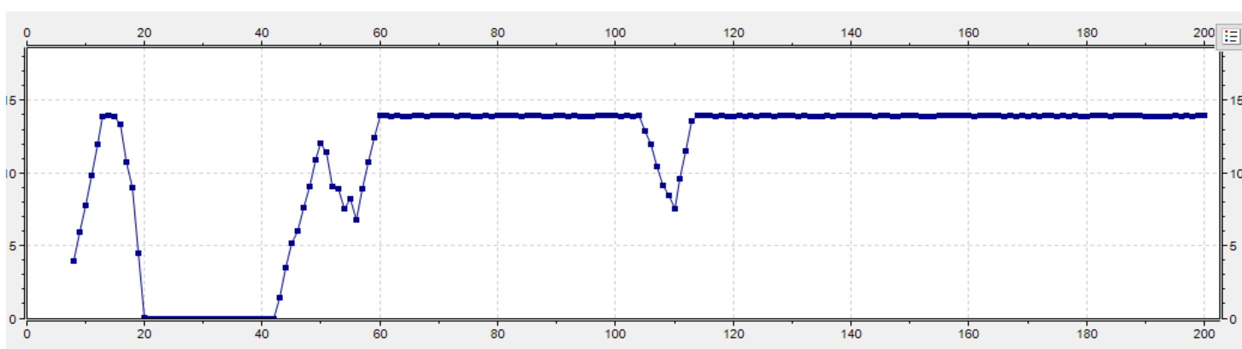


Figure 4.21 Blackhole attacked vehicle

In figure 4.20 you see the behavior of a vehicle diverting away from a wreck. At around time increment 20 the wreck occurs, which is where you see a small dip in its

velocity indicating a turn. From there its velocity pretty much remains at its max with the occasional dip, again indicating a turn.

On the other hand if you look at figure 4.21, where the same vehicle is attacked, you can see that its velocity drops to zero at time increment 20 and remains until around time increment 40. So looking at figure 4.21 you can see that the vehicle was unable to divert and got stuck in a traffic jam.

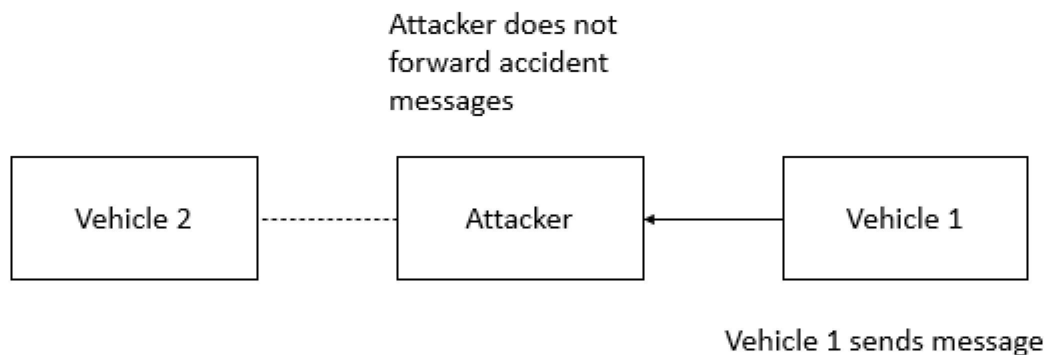


Figure 4.22 Grayhole attack scenario

Figure 4.22 shows a model for a Grayhole attack. In this scenario a vehicle will send a message out to the system. The attacker will receive this message and check what kind of message it is. If the message is a safety message it will drop the message. Otherwise it will continue to work as expected, forwarding the message to the other vehicles. For us this means it should forward the beacon messages, but drop the message about the accident.

For the Grayhole attack we start with the same source code as we used in the blackhole attack, but make it so that the beacon messages are not dropped. As the blackhole already does everything we need and more, we just removed part of the code that drops the beacon message, and then simulate the results again.

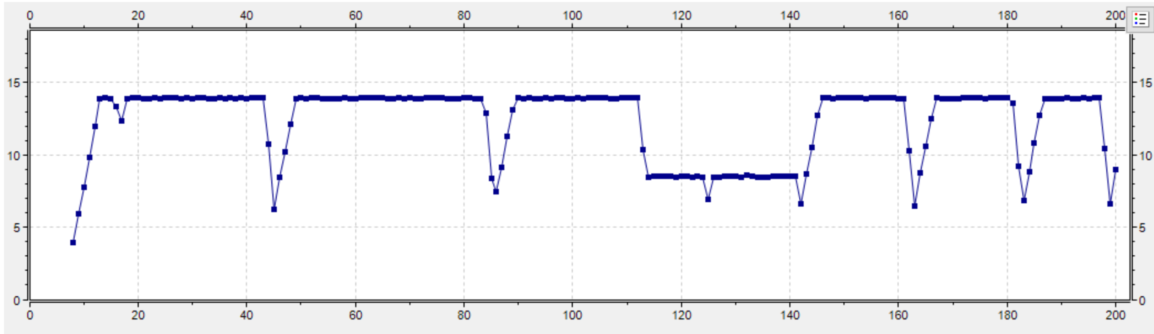


Figure 4.23 Normal vehicle

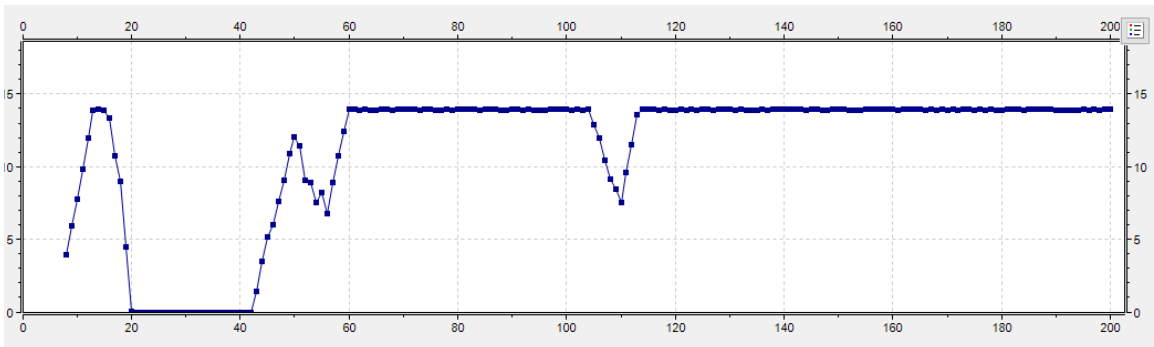


Figure 4.24 White hole attacked vehicle

Figure 4.23 and 4.24 show that the white hole scenarios performed the same as the black hole attack scenario. It made no difference in terms of the attack if we let other types of messages through or not. In figure 4.23 you see that the vehicle was able to divert. In figure 4.24 you see that the vehicle was still unable to divert and got stuck in a traffic jam.

This shows that it’s possible to let certain messages through, while still dropping others. An attacker could take advantage of this to fool systems into thinking it's a well intended user and not malicious.

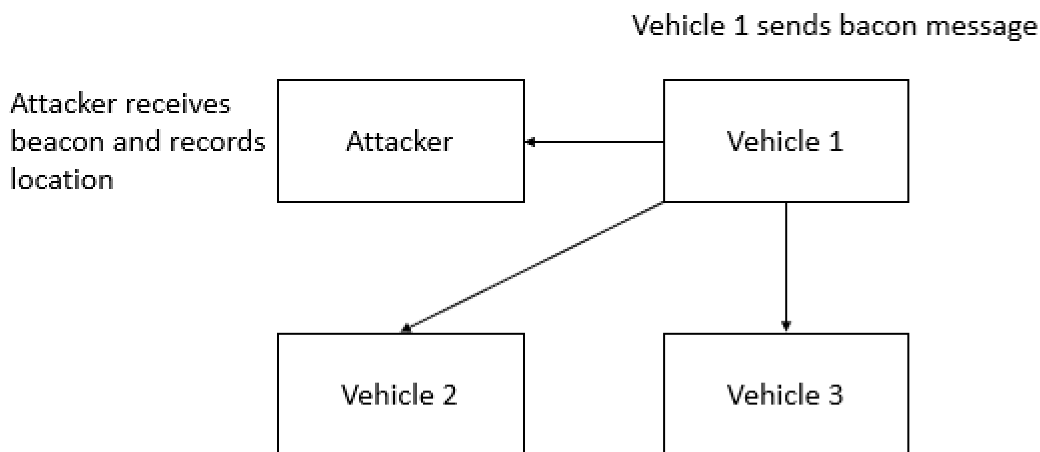


Figure 4.25 Location Tracking scenario

Figure 4.25 details how the location tracking scenario works. In this scenario a vehicle sends out a message, in our case a beacon message. All the vehicles in the system will receive this message and act normally, however the attacker will now know the location of the sender and be able to track the vehicle as a result.

In order to simulate the Location tracking, we again started with the same example project. We again added a beacon message, just like in many of the other simulations. In the beacon message we added the coordinates of the vehicle before sending out the message. Then when the attacker received a beacon message they would be able to print the message out on the screen, thus simulating location tracking. In addition to this we added an identifier to the beacon as well, so the attacker could filter out the beacons to the one they are interested in.

In order to simulate the location tracking scenario, we add an attacking vehicle that will capture messages from other vehicles and record its location. In this simulation each vehicle is sending out a beacon that contains its identification and its location. The

vehicle will then send out the message to anyone interested including the attacker. It's at this time the attacker captures the message.

In table 4.1 it shows that we captured a total of 19 beacon messages, with the vehicle's identity and its X and Y coordinate. These coordinates are for the screen location or in pixels and not the GPS coordinates, however it still shows that we are able to capture the location of vehicle 5 and track where it is located.

| **Packet** | **Identity** | **X** | **Y** |
|---|---|---|---|
| 1 | 5 | 2350.16 | 1639.28 |
| 2 | 5 | 2420.82 | 1700.74 |
| 3 | 5 | 2442.58 | 1817.28 |
| 4 | 5 | 2448.43 | 1956.23 |
| 5 | 5 | 2437.8 | 2095.19 |
| 6 | 5 | 2311.37 | 2117.66 |
| 7 | 5 | 2175.03 | 2096.36 |
| 8 | 5 | 2086.52 | 2019.26 |
| 9 | 5 | 1988.45 | 1920.25 |
| 10 | 5 | 1889.83 | 1821.86 |
| 11 | 5 | 1791.11 | 1723.57 |
| 12 | 5 | 1691.6 | 1626.05 |
| 13 | 5 | 1592.52 | 1528.22 |
| 14 | 5 | 1488.15 | 1435.96 |
| 15 | 5 | 1364.26 | 1373.99 |
| 16 | 5 | 1226.78 | 1352.5 |
| 17 | 5 | 1088.04 | 1338.96 |
| 18 | 5 | 949.24 | 1327.13 |

| 19 | 5 | 810.324 | 1316.37 |
|---|---|---|---|

Table 4.1 Captured Packets

In the message modification and sybil attack scenarios the proposed system is able to detect the bad message, and the vehicle is able to ignore it. The attacked vehicles velocity remains unchanged and looks very similar to the the attackers velocity, and thus successfully prevents the attack. This being said the data to train the model was derived from the simulation scenario, and as a result should not be used to make conclusions about the proposed MDS. For this we will use data readily available to the community.

For the DOS scenario this is a different story, as the attacker is spamming the network with worthless packets. As a result of this, the vehicle's message buffer gets filled before it can access a legitimate message. Once pulled out of the buffer the vehicle can detect the bad message, but at this point it is too late. For a DOS attack, another system is needed.

Attempting to detect the blackhole, whitehole, and location tracking scenarios with our MDS proved to not be possible. This is because these attacks don't send out messages of their own. The blackhole and whitehole make no modification, instead they just drop packets and in the location tracking attack just records data. As a result of this there is nothing for our MDS to detect, and shows that more is needed to prevent these types of attack.

## 4.7 Performance Analysis

| Type | time | sender | attackType | msgId | pos | Pos_noise | spd | Spd_noise |
|---|---|---|---|---|---|---|---|---|

Figure 4.26 Ground of truth

| Type | recTime | sendTime | sender | msgId | pos | Pos_noise | spd | Spd_noise |
|---|---|---|---|---|---|---|---|---|

Figure 4.27 JSONlog.json

In order to train our algorithm and compare it with other implementations, we used the popular VeReMi dataset. The dataset is readily available on github and can be pulled down and used by anybody. This dataset contains many different scenarios that we can use to test our system, in this case we used *attack type one* which is a location based attack. In this type of attack scenario the attacker modifies their GPS coordinates, pretending to be somewhere else.

There are two types of log files we are interested in from VeReMi. The GroundTruthlog.json, which tells us what messages are good and which are bad. Figure 4.26 shows the ground of truth data. In addition to this they provide us the JSONlog.json file, which has all of the vehicle logs. Figure 4.27 shows the features within the JSONlog.json file. Within these logs, there is a combination of good and bad data. These logs contain the features from section 4.6, we need as input for the proposed system. It is the job of our system to decide if a log is normal or bad.

In order to train our model we combined several JSONlog.json files, giving us a total of 100,290 data points. This gave us a very large data set to train our model up and get the best results possible. Once the model was trained we needed data to test it against, so we took a different JSONlog.json file with 4726 data points to test against.

While building up the proposed system we created a few different variants we could use to compare against. The first was an implementation utilizing DT. Testing our Decision Tree implementation with the VeReMi dataset, we found the following:

- Accuracy = 0.925
- TP rate = 1.00
- FP rate = 0.0836

This implementation has the highest accuracy and true positive rate of all our implementations. In fact it was able to detect all of the bad log messages, however it also had the highest false positive rate. So it was more likely than the other models to say a message is bad, even when it's not. The next implementation utilized random forest. The first of our ensemble methods. Implementing random forest improves the false positive rate, but at the cost of our accuray and true positive rate. We found the following:

- accuracy = 0.913
- true positive rate = 0.821
- FP rate = 0.0743

Testing our implementation of ANN we found the following:

- Accuracy = 0.909
- TP rate = 0.561
- FP rate = 0.0148

Comparing this to random forest and decision tree, we see a good improvement in the false positive rate, but at a high hit to the true positive rate.

Testing the proposed system we found the following:

- Accuracy = 0.918
- TP rate = .820
- FP rate = 0.0685

So our proposed system had the second highest accuracy of the different implementations, and the second lowest false positive rate. Stacking the different machine learning algorithms seems to have given us a balance between the extremes of DT and ANN.

# CHAPTER 5

# Future Work

Though we were able to simulate multiple attacks on vehicular networks and successfully detect the attacks with the proposed system, improvements could be made. [16] implemented a stacked based IDS that had an accuracy of .99. This IDS implemented a technique known as feature elimination, as well as using a variety of different ML algorithms.

Feature elimination is a technique where irrelevant features are removed from the system and not used to train the data. Doing this has two advantages: A higher accuracy, and faster processing times. Accuracy is improved as the model does not make corrections based on irrelevant data, and the processing time is improved as we no longer have to process all of the data.

Another useful technique known as feature selection chooses the most relevant or useful features from the data. In this situation the best features would be selected and passed as inputs to our machine learning algorithms, thus automatically selecting the inputs instead of the designer hard coding the inputs. This would also have the advantage of keeping all of the data for training. In this situation our implementation of DT would still need to eliminate data as it needs the data to be unique, but this could be handled as part of the DT training phase.

A useful improvement to the proposed system would be to replace our data elimination phase, with either the feature elimination or feature selection. Where the data

elimination phase did seem to help speed up the application, it also seemed to hurt the accuracy of some of our ML algorithms using ensemble techniques. This is a result of looking at the entire piece of data and eliminating data that is too similar, instead of looking at the individual features within the data and either eliminating or deciding which would be better to use.

Another way we could increase the performance would be to increase the number of folds when training the stack. For our implementation we used two folds: one to train the first layer, and another to train the second. Increasing the number of folds means our models would be more diverse, giving the level 0 and 1 models a chance to have different results. In [16] they utilized a total of 5 folds, but do not offer any insight into how many folds are optimal. More research is needed to determine this.

[16] also utilized different ML algorithms then we did. In layer 0 they used the following: RF, adaboost, and GBDT. For their layer 1 ML algorithm they used DT. It may be that this combination of ML is ideal, but more experimentation would be needed. In our case we utilized DT, RF, and ANN for layer 0 and RF for layer 1.

[3] looked into using DT and RF for an MDS. They found that DT had a TP rate of .96 and RF had a TP rate of .98. This would seem to indicate that these algorithms are well suited for an MDS. [19] looked into using a variety of different ML algorithms for an IDS, including Neural Networks. They found that ANN had a TP rate of .741 and an FP rate of .379. The FP rate is high in comparison to other algorithms, such as DT or RF that are consistently under .1. The system also had a lower TP rate in comparison to DT or RF, that were consistently achieving a TP rate above .9. This would seem to indicate that ANN is probably not the best choice to use for an IDS or MDS.

In addition to increasing the performance of our system, future work is needed to look into different types of systems that could complement our own. This is because our system is unable to detect attacks that do not send a message. When we ran simulations of the blackhole attack, whitehole, and location tracking our system was unable to detect the attack. In the blackhole and whitehole attack, the malicious vehicle just drops the message, meaning there was nothing for us to detect. In the location tracking attack, everything worked as normal, the attacker just took note of a vehicle's coordinates, so again nothing else to detect.

One system that might complement our system well is a reputation system. In this type of system, each vehicle is assigned a reputation. This reputation goes up whenever they send a good message, and it goes down when they either don't respond, or the message is determined to be bad. This would prevent an attacker from ignoring messages as they need a decent reputation to have messages sent to them, and they need to continue to send good messages to maintain that reputation. This in theory should be able to prevent a blackhole and white hole attack.

A reputation system would not help with the location tracking though as the vehicle continues to work as normal, but using a pudo id that is changing could potentially prevent this type of attack. This changing id could prevent location tracking because an attacker needs to know a vehicle's id to determine what vehicle they are tracking. If this id changes occasionally then an attacker might be able to get a vehicle's coordinates, but would not know which vehicle was there.

# CHAPTER 6

# Conclusion

Vehicular networks have the potential to revolutionize the automotive industry. They are the backbone of self-driving cars, provide life saving data, and can entertain the passengers. For example, say it's raining and the road is slippery. When one car goes over the slippery road, it can relay a message to the other cars letting them know the road is unsafe and to take another route. This service has the potential to prevent accidents, but without security it could be misused.

One possibility is that an attacker could want a clear road, so they could reach their destination before other drivers. So they send a false safety message saying an accident has occurred ahead. This would cause the other vehicles to divert and take another route, in an attempt to avoid the accident. However there was no accident and now the attacker has the road to themselves. This is one of many examples of how an attacker could manipulate the system to their advantage showing the importance of data security in a vehicular network.

In this thesis we proposed a stacking based MDS that utilized a combination of DT, RF, and ANN at level 0 and RF at level 0. We also implemented a variety of different simulations, showing how a malicious user could attack the system and how our MDS could detect these attacks. We also showed the limitations of a MDS, as it was unable to

detect malicious users that were not modifying messages. This means we were able to detect the message modification, DoS, and sybil attacks. However we were unable to detect the blackhole, whitehole, and location tracking attacks.

We also ran our algorithm against the VeReMi dataset, to determine the accuracy of our system. We ran it against four different versions of our system. We found that the proposed system had an accuracy of .918, the ANN had an accuracy of .909, RF had an accuracy of .913, and DT had an accuracy of .925.

We had six different scenarios: message modification, sybil, denial of service, blackhole, whitehole, and location tracking. We created a simulation for each of these using OMNET++ and the Veins library. These simulations showed how an attacker could potentially manipulate the system without some sort of network security. In both the message modification and Sybil simulations, the attacker was able to change the route of an attacked vehicle. In the DoS simulation, the attacker was able to prevent the users from changing routes when needed.

For both the message modification and Sybil attack simulation, the proposed system was able to detect and prevent the attack. It did this by detecting a bad message, and ignoring it if the message was bad. In these attacks the prevention is simple as we only need to ignore the message.

On the other hand our system was able to detect the DoS attack, but we could not simply ignore the message in this case. That is because in this kind of attack, the message buffer is filled with bad messages. So the attacked vehicle can't get to the message in a timely manner, even though it knows that the messages are bad. Another system is needed for this kind of attack.

For the blackhole, whitehole, and location tracking attack simulations, the proposed system was unable to detect the attack. This is because these attacks do not modify a message, as a result there is nothing for our MDS to detect. This means another system is needed to detect this type of attack.

# Bibliography

[1] S. Gyawali, S. Xu, Y. Qian, R. Q. Hu, “Challenges and Solutions for Cellular based V2X Communications”, IEEE Communications Surveys and Tutorials, Vol.23, No.1, pp.222-255, First Quarter 2021.

[2] J. Huang, D. Fang, Y. Qian, R. Q. Hu, “Recent Advances and Challenges in Security and Privacy for V2X Communications”, IEEE Open Journal of Vehicular Technology, Vol.1, No.1, pp.244-266, June 2020.

[3] S. Gyawali, Y. Qian, “Misbehavior Detection using Machine Learning in Vehicular Communication Networks”, Proceedings of IEEE ICC 2019, May 20-24, 2019.

[4] K. Zaidi; M. B. Milojevic; V. Rakocevic; A. Nallanathan, M. Rajarajan, “Host-Based Intrusion Detection for VANETs: A Statistical Approach to Rogue Node Detection”, IEEE Transactions on Vehicular Technology, Vol.65, No.8, pp.6703-6714, August 2016.

[5] M. Báguena, S. M. Tornell, Á. Torres, C. T. Calafate, J. Cano and P. Manzoni, "VACaMobil: VANET Car Mobility Manager for OMNeT++," 2013 IEEE International Conference on Communications Workshops (ICC), 2013, pp. 1057-1061.

[6] Behrisch, M., Bieker, L., Erdmann, J., Krajzewicz, D.: Sumo–simulation of urban mobility: an overview. In: Proceedings of SIMUL 2011, The Third International Conference on Advances in System Simulation. ThinkMind (2011).

[7] Odegua, “An empirical study of ensemble techniques (bagging, boost-ing and s tacking),” in Proc. Conf.: Deep Learn. IndabaXAt, 2019.

[8] Abhilash Sonker, Dr. R K Gupta, “A New Combination of Machine Learning Algorithms using Stacking Approach for Misbehavior Detection in VANETs”, Department of CSE&IT, MITS, Gwalior, India, 2020.

[9] Jehad Ali, Rehanullah Khan, Nasir Ahman, Imran Maqsood, “Random Forests and Decision Trees”, IJCSI International Journal of Computer Science Issues, Vol. 9, Issue 5, No 3, 2012.

[10] Nitze, I. Schulthess, U. Asche, H., “Comparison of machine learning algorithms random forest, artificial neural network and support vector machine to maximum likelihood for supervised crop type classification”, Riode Janeiro, Brazil, 2012, Volume 35.

[11] Maulud D, Abdulazeez, “A Review on Linear Regression Comprehensive in Machine Learning”, J. Appl. Sci. Technology, 2020, pp. 140–147.

[12] Siddique, K., Akhtar, Z., Lee, H., Kim, W., Kim, Y.: “Toward bulk synchronous parallel-based machine learning techniques for anomaly detection in high-speed big data networks”, Symmetry , 2017.

[13] Andrew Churcher, Remat Ullah, Jawad Ahmad, Sadaqat ur Rehman, Fawad Masood, Mandar Gogate, Fehaid Alqahtani, Boubakr Nour, and William J Buchanan, “An Experimental Analysis of Attack Classification Using Machine Learning in IoT Networks”, Napir University, 2021; 21(2):446.

[14] Johnson Singh, K., Thongam, K., & De, T. (2016). "Entropy-based application layer DDoS attack detection using artificial neural networks", Entropy, 18(10), 350.

[15] Albulayhi, Smadi, A.A., Sheldon, F.T., Abercrombie, R.K, "Intrusion Detection Taxonomy, Reference Architecture, and Analyses", Sensors, 2021, 21, 6432.

[16] W. Lian, G. Nie , B. Jia , D. Shi, Qi Fan, Y. Liang "An Intrusion Detection Method Based on Decision Tree-Recursive Feature Elimination in Ensemble Learning", Mathematical Problems in Engineering Volume 2020, pp.1-15. 2020.

[17] S. K. Biswas, "Intrusion detection using machine learning: A comparison study," International Journal of Pure and Applied Mathematics, vol. 118, no. 19, pp. 101–114, 2018.

[18] Maranhao, Joao Paulo & Javidi da Costa, Joao Paulo & Pignaton de Freitas, Edison & Javidi, Elnaz & de Sousa Junior, Rafael, "Error-Robust Distributed Denial of Service Attack Detection Based on an Average Common Feature Extraction Technique", Sensors. 20. 5845. 10.3390/s20205845.

[19] Sheikhan, M., Jadidi, Z. & Farrokhi, A. "Intrusion detection using reduced-size RNN based on feature grouping". Neural Comput & Applic 21, 1185–1190 (2012).

[20] Amit A Mane, "Sybil attack in VANET". IJCER, 2250 – 3005 (2016).

[21] Varsha Raghuwanshi, Simmi Jain, "Denial of Service Attack in VANET: A Survey", NRI Institute of Information Science & Technology, Bhopal, IJETT (2015).

[22] Farhan Ahmad, Asma Adnane, Virginia N. L. Franqueira, Fatih Kurugollu, Lu Liu, "Man-In-The-Middle Attacks in Vehicular Ad-Hoc Networks: Evaluating the Impact of Attackers' Strategies ", University of Derby, , Loughborough University, MDPI (2018).

# Appendix A

# Message Modification Attack Source

```
#pragma once
#include "veins/veins.h"
#include "veins/modules/application/ieee80211p/DemoBaseApplLayer.h"
#include "veins/modules/application/traci/TraCIDemo11pMessage_m.h"

using namespace omnetpp;

namespace veins {

static bool badGuy = true;
static int nodes = 0;

class VEINS_API MessageModificatoinSenario : public DemoBaseApplLayer {
public:
    void initialize(int stage) override;
    void finish() override;

protected:
    simtime_t lastDroveAt;
    bool sentMessage;
    int currentSubscribedServiceId;

    void onBSM(DemoSafetyMessage* bsm) override;
    void onWSM(BaseFrame1609_4* frame) override;
    void onWSA(DemoServiceAdvertisment* wsa) override;

    void handleSelfMsg(cMessage* msg) override;
    void handlePositionUpdate(cObject* obj) override;

private:
    bool bad = false;
    bool hasMessage = false;
    TraCIDemo11pMessage capturedMessage;
    int wait = 0;
};

} // namespace veins
```

```
#include "veins/modules/application/MessageModificationSenario/MessageModificationSenario.h"
#include "veins/modules/application/traci/TraCIDemo11pMessage_m.h"

using namespace veins;

Define_Module(veins::MessageModificatoinSenario);

void MessageModificatoinSenario::initialize(int stage)
{
  DemoBaseApplLayer::initialize(stage);
  if (stage == 0) {
    sentMessage = false;
    lastDroveAt = simTime();
    currentSubscribedServiceId = -1;
  }

  ++nodes;

  if(badGuy && nodes%2 == 0) {
    bad = true;
    badGuy = false;
  }
}

void MessageModificatoinSenario::finish()
{
  DemoBaseApplLayer::finish();
  // statistics recording goes here

  if(bad) {
    badGuy = true;
  }
}

void MessageModificatoinSenario::onBSM(DemoSafetyMessage* bsm)
{
  // Your application has received a beacon message from another car or RSU
  // code for handling the message goes here
}

void MessageModificatoinSenario::onWSM(BaseFrame1609_4* frame)
{
  TraCIDemo11pMessage* wsm = check_and_cast<TraCIDemo11pMessage*>(frame);
  if(bad) {
    hasMessage = true;
    capturedMessage = *wsm;
```

```
    }

  findHost()->getDisplayString().setTagArg("i", 1, "green");

  if (mobility->getRoadId()[0] != ':') traciVehicle->changeRoute(wsm->getDemoData(), 9999);
  if (!sentMessage) {
     sentMessage = true;
     // repeat the received traffic update once in 2 seconds plus some random delay
     wsm->setSenderAddress(myId);
     wsm->setSerial(3);
     scheduleAt(simTime() + 2 + uniform(0.01, 0.2), wsm->dup());
  }
}

void MessageModificatoinSenario::onWSA(DemoServiceAdvertisment* wsa)
{
   if (currentSubscribedServiceId == -1) {
      mac->changeServiceChannel(static_cast<Channel>(wsa->getTargetChannel()));
      currentSubscribedServiceId = wsa->getPsid();
      if (currentOfferedServiceId != wsa->getPsid()) {
         stopService();
         startService(static_cast<Channel>(wsa->getTargetChannel()), wsa->getPsid(), "Mirrored Traffic
            Service");
      }
   }
}

void MessageModificatoinSenario::handleSelfMsg(cMessage* msg)
{
   if (TraCIDemo11pMessage* wsm = dynamic_cast<TraCIDemo11pMessage*>(msg)) {
     // send this message on the service channel until the counter is 3 or higher.
     // this code only runs when channel switching is enabled
     sendDown(wsm->dup());
     wsm->setSerial(wsm->getSerial() + 1);
     if (wsm->getSerial() >= 3) {
        // stop service advertisements
        stopService();
        delete (wsm);
     }
     else {
        scheduleAt(simTime() + 1, wsm);
     }
  }
  else {
     DemoBaseApplLayer::handleSelfMsg(msg);
  }
```

```
}

void MessageModificatoinSenario::handlePositionUpdate(cObject* obj)
{
  if (mobility->getSpeed() < 1 && !bad) {
    if (simTime() - lastDroveAt >= 10 && sentMessage == false) {
      findHost()->getDisplayString().setTagArg("i", 1, "red");
      sentMessage = true;

      TraCIDemo11pMessage* wsm = new TraCIDemo11pMessage();
      populateWSM(wsm);
      wsm->setDemoData(mobility->getRoadId().c_str());

      // host is standing still due to crash
      if (dataOnSch) {
        startService(Channel::sch2, 42, "Traffic Information Service");
        // started service and server advertising, schedule message to self to send later
        scheduleAt(computeAsynchronousSendingTime(1, ChannelType::service), wsm);
      }
      else {
        // send right away on CCH, because channel switching is disabled
        sendDown(wsm);
      }
    }
  } else if(bad && hasMessage){
    if(wait < 20) {
      ++wait;
    } else {
      TraCIDemo11pMessage* wsm = new TraCIDemo11pMessage();
      populateWSM(wsm);
      wsm->setDemoData(mobility->getRoadId().c_str());

      wsm->setSentFrom(capturedMessage.getSenderModule(),
          capturedMessage.getSenderGate()->getBaseId(),
          capturedMessage.getSendingTime());
      sendDown(wsm);
      wait = 0;
    }
  }
}
```

# Appendix B

# Sybil Attack Scenario Source

```
#pragma once
#include "veins/veins.h"
#include "veins/modules/application/ieee80211p/DemoBaseApplLayer.h"

using namespace omnetpp;

namespace veins {

static bool badGuy = true;

class VEINS_API SybilSenario : public DemoBaseApplLayer {
public:
    void initialize(int stage) override;
    void finish() override;

protected:
    simtime_t lastDroveAt;
    bool sentMessage;
    int currentSubscribedServiceId;

    void onBSM(DemoSafetyMessage* bsm) override;
    void onWSM(BaseFrame1609_4* frame) override;
    void onWSA(DemoServiceAdvertisment* wsa) override;

    void handleSelfMsg(cMessage* msg) override;
    void handlePositionUpdate(cObject* obj) override;

private:
    bool bad = false;
    bool captured1Becon = false;
    DemoSafetyMessage capturedBecon1;
    bool captured2Becon = false;
    DemoSafetyMessage capturedBecon2;
};

} // namespace veins
```

```
#include "veins/modules/application/sybilSenario/SybilSenario.h"
#include "veins/modules/application/traci/TraCIDemo11pMessage_m.h"

using namespace veins;

Define_Module(veins::SybilSenario);

void SybilSenario::initialize(int stage)
{
  DemoBaseApplLayer::initialize(stage);
  if (stage == 0) {
    sentMessage = false;
    lastDroveAt = simTime();
    currentSubscribedServiceId = -1;
  }

  if(badGuy) {
    bad = true;
    badGuy = false;
  }
}

void SybilSenario::finish()
{
  DemoBaseApplLayer::finish();
  // statistics recording goes here

  if(bad) {
    badGuy = true;
  }
}

void SybilSenario::onBSM(DemoSafetyMessage* bsm)
{
  // Your application has received a beacon message from another car or RSU
  // code for handling the message goes here
  if(bad) {
    if(!captured1Becon) {
      captured1Becon = true;
      capturedBecon1 = *bsm;
    } else if (!captured2Becon) {
      captured2Becon = true;
      capturedBecon2 = *bsm;
    }
  }
}
```

```
void SybilSenario::onWSM(BaseFrame1609_4* frame)
{
    TraCIDemo11pMessage* wsm = check_and_cast<TraCIDemo11pMessage*>(frame);

   findHost()->getDisplayString().setTagArg("i", 1, "green");

   if (mobility->getRoadId()[0] != ':') traciVehicle->changeRoute(wsm->getDemoData(), 9999);
   if (!sentMessage) {
       sentMessage = true;
       // repeat the received traffic update once in 2 seconds plus some random delay
       wsm->setSenderAddress(myId);
       wsm->setSerial(3);
       scheduleAt(simTime() + 2 + uniform(0.01, 0.2), wsm->dup());
   }
}

void SybilSenario::onWSA(DemoServiceAdvertisment* wsa)
{
    if (currentSubscribedServiceId == -1) {
        mac->changeServiceChannel(static_cast<Channel>(wsa->getTargetChannel()));
        currentSubscribedServiceId = wsa->getPsid();
        if (currentOfferedServiceId != wsa->getPsid()) {
            stopService();
            startService(static_cast<Channel>(wsa->getTargetChannel()), wsa->getPsid(), "Mirrored Traffic
                Service");
        }
    }
}

void SybilSenario::handleSelfMsg(cMessage* msg)
{
    if (TraCIDemo11pMessage* wsm = dynamic_cast<TraCIDemo11pMessage*>(msg)) {
       // send this message on the service channel until the counter is 3 or higher.
       // this code only runs when channel switching is enabled
       sendDown(wsm->dup());
       wsm->setSerial(wsm->getSerial() + 1);
       if (wsm->getSerial() >= 3) {
           // stop service advertisements
           stopService();
           delete (wsm);
       }
       else {
           scheduleAt(simTime() + 1, wsm);
       }
   }
```

```
  else {
     DemoBaseApplLayer::handleSelfMsg(msg);
  }
}

void SybilSenario::handlePositionUpdate(cObject* obj)
{
   if (mobility->getSpeed() < 1) {
     if (simTime() - lastDroveAt >= 10 && sentMessage == false) {
        findHost()->getDisplayString().setTagArg("i", 1, "red");
        sentMessage = true;

        TraCIDemo11pMessage* wsm = new TraCIDemo11pMessage();
        populateWSM(wsm);
        wsm->setDemoData(mobility->getRoadId().c_str());

        // host is standing still due to crash
        if (dataOnSch) {
           startService(Channel::sch2, 42, "Traffic Information Service");
           // started service and server advertising, schedule message to self to send later
           scheduleAt(computeAsynchronousSendingTime(1, ChannelType::service), wsm);
        }
        else {
           // send right away on CCH, because channel switching is disabled
           sendDown(wsm);
        }
     }
  }
  else {
     lastDroveAt = simTime();
     DemoSafetyMessage* becon = new DemoSafetyMessage();
     populateWSM(becon);
     BaseApplLayer::sendDown(becon);

     if(bad && captured1Becon && captured2Becon) {
         findHost()->getDisplayString().setTagArg("i", 1, "red");
         sentMessage = true;

         TraCIDemo11pMessage* wsm = new TraCIDemo11pMessage();
         populateWSM(wsm);
         wsm->setDemoData(mobility->getRoadId().c_str());

         wsm->setSentFrom(capturedBecon1.getSenderModule(),
               capturedBecon1.getSenderGate()->getBaseId(),
               capturedBecon1.getSendingTime());
         sendDown(wsm);
```

```
        TraCIDemo11pMessage* wsm2 = new TraCIDemo11pMessage();
        populateWSM(wsm2);
        wsm2->setDemoData(mobility->getRoadId().c_str());

        wsm2->setSentFrom(capturedBecon2.getSenderModule(),
            capturedBecon2.getSenderGate()->getBaseId(),
            capturedBecon2.getSendingTime());
        sendDown(wsm2);
    }
  }
}
```

# Appendix C

# DOS Attack Scenario Source

```
#pragma once

#include "veins/veins.h"
#include "veins/modules/application/ieee80211p/DemoBaseApplLayer.h"
#include "veins/modules/application/traci/TraCIDemo11pMessage_m.h"

using namespace omnetpp;

namespace veins {

static bool badGuy = true;
static int nodes = 0;

class VEINS_API DosSenario : public DemoBaseApplLayer {
public:
   void initialize(int stage) override;
   void finish() override;

protected:
   simtime_t lastDroveAt;
   bool sentMessage;
   int currentSubscribedServiceId;

   void onBSM(DemoSafetyMessage* bsm) override;
   void onWSM(BaseFrame1609_4* frame) override;
   void onWSA(DemoServiceAdvertisment* wsa) override;

   void handleSelfMsg(cMessage* msg) override;
   void handlePositionUpdate(cObject* obj) override;

private:
   bool bad = false;
};

} // namespace veins
```

```
#include "veins/modules/application/DosSenario/DosSenario.h"
#include "veins/modules/application/traci/TraCIDemo11pMessage_m.h"

using namespace veins;

Define_Module(veins::DosSenario);

void DosSenario::initialize(int stage)
{
    DemoBaseApplLayer::initialize(stage);
    if (stage == 0) {
        sentMessage = false;
        lastDroveAt = simTime();
        currentSubscribedServiceId = -1;
    }

    ++nodes;

    if(badGuy && nodes%2 == 0) {
        bad = true;
        badGuy = false;
    }
}

void DosSenario::finish()
{
    DemoBaseApplLayer::finish();
    // statistics recording goes here

    if(bad) {
        badGuy = true;
    }
}

void DosSenario::onBSM(DemoSafetyMessage* bsm)
{
    // Your application has received a beacon message from another car or RSU
    // code for handling the message goes here
}

void DosSenario::onWSM(BaseFrame1609_4* frame)
{
   TraCIDemo11pMessage* wsm = check_and_cast<TraCIDemo11pMessage*>(frame);
   findHost()->getDisplayString().setTagArg("i", 1, "green");

   if (mobility->getRoadId()[0] != ':') traciVehicle->changeRoute(wsm->getDemoData(), 9999);
```

```
  if (!sentMessage) {
    sentMessage = true;
    // repeat the received traffic update once in 2 seconds plus some random delay
    wsm->setSenderAddress(myId);
    wsm->setSerial(3);
    scheduleAt(simTime() + 2 + uniform(0.01, 0.2), wsm->dup());
  }
}

void DosSenario::onWSA(DemoServiceAdvertisment* wsa)
{
   if (currentSubscribedServiceId == -1) {
     mac->changeServiceChannel(static_cast<Channel>(wsa->getTargetChannel()));
     currentSubscribedServiceId = wsa->getPsid();
     if (currentOfferedServiceId != wsa->getPsid()) {
       stopService();
       startService(static_cast<Channel>(wsa->getTargetChannel()), wsa->getPsid(), "Mirrored Traffic
         Service");
     }
   }
}

void DosSenario::handleSelfMsg(cMessage* msg)
{
   if (TraCIDemo11pMessage* wsm = dynamic_cast<TraCIDemo11pMessage*>(msg)) {
    // send this message on the service channel until the counter is 3 or higher.
    // this code only runs when channel switching is enabled
    sendDown(wsm->dup());
    wsm->setSerial(wsm->getSerial() + 1);
    if (wsm->getSerial() >= 3) {
      // stop service advertisements
      stopService();
      delete (wsm);
    }
    else {
      scheduleAt(simTime() + 1, wsm);
    }
  }
  else {
    DemoBaseApplLayer::handleSelfMsg(msg);
  }
}

void DosSenario::handlePositionUpdate(cObject* obj)
{
   if (mobility->getSpeed() < 1 && !bad) {
```

```
        if (simTime() - lastDroveAt >= 10 && sentMessage == false) {
            findHost()->getDisplayString().setTagArg("i", 1, "red");
            sentMessage = true;

            TraCIDemo11pMessage* wsm = new TraCIDemo11pMessage();
            populateWSM(wsm);
            wsm->setDemoData(mobility->getRoadId().c_str());

            // host is standing still due to crash
            if (dataOnSch) {
                startService(Channel::sch2, 42, "Traffic Information Service");
                // started service and server advertising, schedule message to self to send later
                scheduleAt(computeAsynchronousSendingTime(1, ChannelType::service), wsm);
            }
            else {
                // send right away on CCH, because channel switching is disabled
                sendDown(wsm);
            }
        }
    } else if(bad){
        for (int i = 0; i<1000; i++) {
            DemoSafetyMessage* becon = new DemoSafetyMessage();
            populateWSM(becon);
            BaseApplLayer::sendDown(becon);
        }
    }
}
```

# Appendix D

# Blackhole/Whitehole Attack Scenario Source

```
#pragma once
#include "veins/veins.h"
#include "veins/modules/application/ieee80211p/DemoBaseApplLayer.h"
#include "greenMds/api_include/api/FeatureFactory.h"
#using namespace omnetpp;
namespace veins {
static bool badGuy = true;
static int cars = 0;
//static api::FeatureFactory sybilFactory;
class VEINS_API WhiteHoleSenario : public DemoBaseApplLayer {
public:
   void initialize(int stage) override;
   void finish() override;
protected:
   simtime_t lastDroveAt;
   bool sentMessage;
   bool beconSent;
   int currentSubscribedServiceId;
   void onBSM(DemoSafetyMessage* bsm) override;
   void onWSM(BaseFrame1609_4* frame) override;
   void onWSA(DemoServiceAdvertisment* wsa) override;
   void handleSelfMsg(cMessage* msg) override;
   void handlePositionUpdate(cObject* obj) override;
private:
   bool bad = false;
   int id=0;
   int steps=0;
};
} // namespace veins

#include "veins/modules/application/WhiteHoleSenario/WhiteHoleSenario.h"
#include "veins/modules/application/traci/TraCIDemo11pMessage_m.h"
using namespace veins;
Define_Module(veins::WhiteHoleSenario);
void WhiteHoleSenario::initialize(int stage)
{
   DemoBaseApplLayer::initialize(stage);
   if (stage == 0) {
      sentMessage = false;
      lastDroveAt = simTime();
      currentSubscribedServiceId = -1;
   }
   beconSent = false;
   id = cars;
```

```
    ++cars;
    if(id == 3) {
        std::cout << "Bad Guy Set! " << std::endl;
        bad = true;
    }
}
void WhiteHoleSenario::finish()
{
    DemoBaseApplLayer::finish();
    // statistics recording goes here
}
void WhiteHoleSenario::onBSM(DemoSafetyMessage* bsm)
{
    // Your application has received a beacon message from another car or RSU
    // code for handling the message goes here
    if(id == 3 && beconSent == false) { // Add && !bad to make blackhole attack
        std::cout << "Received becon at " << id << std::endl;
        //forward becon message
        DemoSafetyMessage* dup = new DemoSafetyMessage(*bsm);
        dup->setPsid(5);
        scheduleAt(simTime(), dup);
        beconSent = true;
    }
}
void WhiteHoleSenario::onWSM(BaseFrame1609_4* frame)
{
    TraCIDemo11pMessage* wsm = check_and_cast<TraCIDemo11pMessage*>(frame);
    std::cout << "Safety message: " << wsm->getPsid() << "My Id: " << id << std::endl;
   if(wsm->getPsid() == id) {
      findHost()->getDisplayString().setTagArg("i", 1, "green");

      if (mobility->getRoadId()[0] != ':') {
         traciVehicle->changeRoute(wsm->getDemoData(), 9999);
      }
      if (!sentMessage && !bad) {
         sentMessage = true;
         // repeat the received traffic update once in 2 seconds plus some random delay
         wsm->setSenderAddress(myId);
         wsm->setPsid(5);
         wsm->setSerial(3);
         scheduleAt(simTime(), wsm->dup());
      }
   }
}
void WhiteHoleSenario::onWSA(DemoServiceAdvertisment* wsa)
{
```

```
    if (currentSubscribedServiceId == -1) {
        mac->changeServiceChannel(static_cast<Channel>(wsa->getTargetChannel()));
        currentSubscribedServiceId = wsa->getPsid();
        if (currentOfferedServiceId != wsa->getPsid()) {
            stopService();
            startService(static_cast<Channel>(wsa->getTargetChannel()), wsa->getPsid(), "Mirrored Traffic
                Service");
        }
    }
}

void WhiteHoleSenario::handleSelfMsg(cMessage* msg)
{
    if (TraCIDemo11pMessage* wsm = dynamic_cast<TraCIDemo11pMessage*>(msg)) {
       // send this message on the service channel until the counter is 3 or higher.
       // this code only runs when channel switching is enabled
       sendDown(wsm->dup());
       wsm->setSerial(wsm->getSerial() + 1);
       if (wsm->getSerial() >= 3) {
          // stop service advertisements
          stopService();
          delete (wsm);
       }
       else {
          scheduleAt(simTime() + 1, wsm);
       }
   }
   else {
       DemoBaseApplLayer::handleSelfMsg(msg);
   }
}
void WhiteHoleSenario::handlePositionUpdate(cObject* obj)
{
    if (mobility->getSpeed() < 1) {
       if (simTime() - lastDroveAt >= 2 && sentMessage == false) {
          findHost()->getDisplayString().setTagArg("i", 1, "red");
          sentMessage = true;
          TraCIDemo11pMessage* wsm = new TraCIDemo11pMessage();
          populateWSM(wsm);
          wsm->setDemoData(mobility->getRoadId().c_str());
          wsm->setPsid(3);
          std::cout << "Sending Safety message: " << id << std::endl;
          // host is standing still due to crash
          if (dataOnSch) {
             startService(Channel::sch2, 42, "Traffic Information Service");
             // started service and server advertising, schedule message to self to send later
```

```
            scheduleAt(computeAsynchronousSendingTime(1, ChannelType::service), wsm);
        }
        else {
            // send right away on CCH, because channel switching is disabled
            sendDown(wsm);
        }
    }
  }
  ++steps;
  if(steps == 30 && beconSent == false) {
    std::cout << "Sending Becon message to 3" << std::endl;
    lastDroveAt = simTime();
    DemoSafetyMessage* becon = new DemoSafetyMessage();
    populateWSM(becon);
    ChannelMobilityPtrType const mobility = check_and_cast<ChannelMobilityPtrType>(obj);
    Coord pos = mobility->getPositionAt(simTime());
    becon->setSenderPos(pos);
    becon->setPsid(3);
    BaseApplLayer::sendDown(becon);
    steps = 0;
    beconSent = true;
  }
}
```

# Appendix E

# Location Tracking Attack Source

```
#pragma once
#include "veins/veins.h"
#include "veins/modules/application/ieee80211p/DemoBaseApplLayer.h"
#include "greenMds/api_include/api/FeatureFactory.h"
using namespace omnetpp;
namespace veins {
static bool badGuy = true;
static int cars = 0;
//static api::FeatureFactory sybilFactory;
class VEINS_API LocationTrackingSenario : public DemoBaseApplLayer {
public:
   void initialize(int stage) override;
   void finish() override;
protected:
   simtime_t lastDroveAt;
   bool sentMessage;
   int currentSubscribedServiceId;
   void onBSM(DemoSafetyMessage* bsm) override;
   void onWSM(BaseFrame1609_4* frame) override;
   void onWSA(DemoServiceAdvertisment* wsa) override;
   void handleSelfMsg(cMessage* msg) override;
   void handlePositionUpdate(cObject* obj) override;
private:
   bool bad = false;
   int id=0;
   int steps=0;
};
} // namespace veins
#include "veins/modules/application/LocationTrackingSenario/LocationTrackingSenario.h"
#include "veins/modules/application/traci/TraCIDemo11pMessage_m.h"

using namespace veins;

Define_Module(veins::LocationTrackingSenario);

void LocationTrackingSenario::initialize(int stage)
{
   DemoBaseApplLayer::initialize(stage);
   if (stage == 0) {
      sentMessage = false;
```

```
        lastDroveAt = simTime();
        currentSubscribedServiceId = -1;
    }

    id = cars;
    ++cars;

    if(badGuy) {
        std::cout << "Bad Guy Set! " << std::endl;
        bad = true;
        badGuy = false;
    }
}

void LocationTrackingSenario::finish()
{
    DemoBaseApplLayer::finish();
    // statistics recording goes here

    if(bad) {
        badGuy = true;
    }
}

void LocationTrackingSenario::onBSM(DemoSafetyMessage* bsm)
{
    // Your application has received a beacon message from another car or RSU
    // code for handling the message goes here
    if(bad && bsm->getPsid() == 5) {
        std::cout << "psid: " << bsm->getPsid() << std::endl;
        std::cout << "x: " << bsm->getSenderPos().x << std::endl;
        std::cout << "y: " << bsm->getSenderPos().y << std::endl;
        std::cout << "z: " << bsm->getSenderPos().z << std::endl;
    }
}

void LocationTrackingSenario::onWSM(BaseFrame1609_4* frame)
{
    TraCIDemo11pMessage* wsm = check_and_cast<TraCIDemo11pMessage*>(frame);

   findHost()->getDisplayString().setTagArg("i", 1, "green");

   if (mobility->getRoadId()[0] != ':') {
      traciVehicle->changeRoute(wsm->getDemoData(), 9999);
   }
```

```
  if (!sentMessage) {
     sentMessage = true;
     // repeat the received traffic update once in 2 seconds plus some random delay
     wsm->setSenderAddress(myId);
     wsm->setSerial(3);
     scheduleAt(simTime() + 2 + uniform(0.01, 0.2), wsm->dup());
  }
}

void LocationTrackingSenario::onWSA(DemoServiceAdvertisment* wsa)
{
   if (currentSubscribedServiceId == -1) {
      mac->changeServiceChannel(static_cast<Channel>(wsa->getTargetChannel()));
      currentSubscribedServiceId = wsa->getPsid();
      if (currentOfferedServiceId != wsa->getPsid()) {
         stopService();
         startService(static_cast<Channel>(wsa->getTargetChannel()), wsa->getPsid(), "Mirrored Traffic
            Service");
      }
   }
}

void LocationTrackingSenario::handleSelfMsg(cMessage* msg)
{
   if (TraCIDemo11pMessage* wsm = dynamic_cast<TraCIDemo11pMessage*>(msg)) {
     // send this message on the service channel until the counter is 3 or higher.
     // this code only runs when channel switching is enabled
     sendDown(wsm->dup());
     wsm->setSerial(wsm->getSerial() + 1);
     if (wsm->getSerial() >= 3) {
        // stop service advertisements
        stopService();
        delete (wsm);
     }
     else {
        scheduleAt(simTime() + 1, wsm);
     }
  }
  else {
     DemoBaseApplLayer::handleSelfMsg(msg);
  }
}

void LocationTrackingSenario::handlePositionUpdate(cObject* obj)
{
  ++steps;
```

```
  if(steps == 10) {
    std::cout << "sending becon" << std::endl;
    lastDroveAt = simTime();
    DemoSafetyMessage* becon = new DemoSafetyMessage();
    populateWSM(becon);
    ChannelMobilityPtrType const mobility = check_and_cast<ChannelMobilityPtrType>(obj);
    Coord pos = mobility->getPositionAt(simTime());
    becon->setSenderPos(pos);
    becon->setPsid(id);
    BaseApplLayer::sendDown(becon);
    steps = 0;
  }
}
```

# Appendix F

# GreenMDS Source

```
/*
 * FeatureFactory.h
 *
 *  Created on: Feb 23, 2022
 *      Author: tmgreen
 */

#ifndef API_INCLUDE_FEATUREFACTORY_H_
#define API_INCLUDE_FEATUREFACTORY_H_

#include "greenMds/include/data/Training.h"
#include "greenMds/include/data/Testing.h"
#include "greenMds/include/data/TrainingManager.h"
#include "greenMds/include/features/features.h"
#include "greenMds/include/dt/Rf.h"
#include "greenMds/include/data/Log.h"
#include "greenMds/include/dt/Ann.h"
#include "greenMds/include/dt/Stack.h"

namespace api {
   class FeatureFactory {
   public:
      FeatureFactory();

      void addTraningData(std::string, features::feature dataType);
      void generateModel();

      features::feature getFeature(std::string data);

   private:
      ml::Stack* stack;
   };
}

#endif /* API_INCLUDE_FEATUREFACTORY_H_ */
```

```
/*
 * Log.h
 *
 *  Created on: Mar 7, 2022
 *      Author: tmgreen
 */

#ifndef LOG_H
#define LOG_H

#include <string>
#include <regex>

namespace data {

    const std::string TYPE_MATCH = "\"type\":(.*?),\"";
    const std::string REC_TIME_MATCH = "\"rcvTime\":(.*?),\"";
    const std::string TIME_MATCH = "\"time\":(.*?),\"";
    const std::string SEND_TIME_MATCH = "\"sendTime\":(.*?),\"";
    const std::string SENDER_MATCH = "\"sender\":(.*?),\"";
    const std::string SENDER_PUDO_MATCH = "\"senderPseudo\":(.*?),\"";
    const std::string MSG_ID_MATCH = "\"messageID\":(.*?),\"";
    const std::string POS_MATCH = "\"pos\":\\[(.*?)\\]";
    const std::string POS_NOISE_MATCH = "\"pos_noise\":\\[(.*?)\\]";
    const std::string SPEED_MATCH = "\"spd\":\\[(.*?)\\]";
    const std::string SPEED_NOISE_MATCH = "\"spd_noise\":\\[(.*?)\\]";
    const std::string ACL_MATCH = "\"acl\":\\[(.*?)\\]";
    const std::string ACL_NOISE_MATCH = "\"acl_noise\":\\[(.*?)\\]";
    const std::string HED_MATCH = "\"hed\":\\[(.*?)\\]";
    const std::string HED_NOISE_MATCH = "\"hed_noise\":\\[(.*?)\\]";

    class Log {
    public:
        Log(){};
        Log(std::string log);
        virtual ~Log() = default;

        std::string minLog();

        std::string getRecTime();
        std::string getType();
        std::string getPos();
        std::string getPosNoise();
        std::string getSpeed();
        std::string getSpeedNoise();
```

```
  private:
    std::string type;
    std::string recTime;
    std::string time;
    std::string sendTime;
    std::string sender;
    std::string senderPudo;
    std::string msgId;
    std::string pos;
    std::string posNoise;
    std::string speed;
    std::string speedNoise;
    std::string acl;
    std::string aclNoise;
    std::string hed;
    std::string hedNoise;
  };
}

#endif /* LOG_H */
```

```
/*
 * Testing.h
 *
 *  Created on: Feb 21, 2022
 *      Author: tmgreen
 */

#ifndef INCLUDE_DATA_TESTING_H_
#define INCLUDE_DATA_TESTING_H_

#include <string>
#include "Log.h"

namespace data {
   class Testing {
   public:
      Testing();
      Testing(const Log& data);
      virtual ~Testing();

      Log getData();
      long getNormalizedData() const;
      static long getNormalizedData(const std::string& data);

   private:
      Log data;
      long normalizedData;
   };
}
#endif /* INCLUDE_DATA_TESTING_H_ *
```

```
/*
 * Testing.h
 *
 *  Created on: Feb 21, 2022
 *      Author: tmgreen
 */

#ifndef INCLUDE_DATA_TRAINING_H_
#define INCLUDE_DATA_TRAINING_H_

#include "Testing.h"
#include "greenMds/include/features/features.h"

namespace data {
   class Training : public Testing {
   public:
      Training();
      Training(Log data, features::feature dataType);
      virtual ~Training();

      void setDataType(features::feature dataType);
      features::feature getDataType();
      bool operator==(const data::Training& other);
      bool operator<(const Training& other);

   private:
      features::feature dataType;
   };
}
#endif /* INCLUDE_DATA_TRAINING_H_ */
```

```
/*
 * TrainingManager.h
 *
 *  Created on: Feb 21, 2022
 *      Author: tmgreen
 */

#ifndef INCLUDE_DATA_TRAINING_MANAGER_H_
#define INCLUDE_DATA_TRAINING_MANAGER_H_

#include <algorithm>
#include <list>
#include <iterator>

#include "Training.h"

namespace data {
   class TrainingManager {
   public:
      static TrainingManager* getInstance();

      void addData(data::Training data);
      data::Training getData(int index);
      void removeData(data::Training data);

      void sort();
      void eliminate();
      int getCount();
   private:
      std::list<data::Training> dataList;

      TrainingManager();
   };
}
#endif /* INCLUDE_DATA_TRAINING_MANAGER_H_ */
```

```
/*
 * Ann.h
 *
 *  Created on: Mar 24, 2022
 *      Author: tmgreen
 */

#ifndef SRC_DT_ANN_H_
#define SRC_DT_ANN_H_

#include "greenMds/include/dt/MlNode.h"
#include "greenMds/include/data/Training.h"
#include "greenMds/include/features/features.h"

namespace ml {
   class Ann {
   public:
      Ann(data::Training* data, int numData);
      ~Ann() = default;

      features::feature getFeature(data::Testing data);

   private:
      MlNode* input;
      MlNode* output;

      int numLayerOne;
      int numLayerTwo;
      MlNode* hLayerTwo;
      MlNode* hLayerOne;
   };
}

#endif /* SRC_DT_ANN_H_ */
```

```
/*
 * DtNode.h
 *
 *  Created on: Feb 24, 2022
 *      Author: tmgreen
 */

#ifndef INCLUDE_DT_DTNODE_H_
#define INCLUDE_DT_DTNODE_H_

#include "greenMds/include/data/Training.h"

namespace dt {
    class DtNode {
    public:
        DtNode();
        virtual ~DtNode() = default;

        bool operator==(DtNode& other);
        bool operator<(DtNode& other);

        void setParent(DtNode* parent);
        void setLeftChild(DtNode* leftChild);
        void setRightChild(DtNode* rightChild);

        DtNode* getParent();
        DtNode* getLeftChild();
        DtNode* getRightChild();

        void setData(data::Training data);
        data::Training getData();

        bool isLeaf();

        int good;
        int bad;
    private:
        DtNode* parent;
        DtNode* leftChild;
        DtNode* rightChild;

        data::Training data;
        bool leaf = false;
    };
}
#endif /* INCLUDE_DT_DTNODE_H_ */
```

```
/*
 * DtTree.h
 *
 *  Created on: Feb 24, 2022
 *      Author: tmgreen
 */

#ifndef INCLUDE_DT_DTTREE_H_
#define INCLUDE_DT_DTTREE_H_

#include "greenMds/include/dt/DtNode.h"
#include "greenMds/include/features/features.h"
#include "greenMds/include/data/Testing.h"

namespace dt {
   class DtTree {
   public:
      DtTree(){}

      DtTree(DtNode* nodes, uint32_t numNodes);
      virtual ~DtTree() = default;

      features::feature getFeature(data::Testing data);
      DtNode getRoot();

   private:
      DtNode root;

      DtNode* splitTree(DtNode* nodeList,
            int index,
            int lowerBound,
            int upperBound,
            bool starting,
            bool leftSplit,
            bool rightSplit);
   };
}

#endif /* INCLUDE_DT_DTTREE_H_ */
```

```
/*
 * MlNode.h
 *
 *  Created on: Mar 24, 2022
 *      Author: tmgreen
 */

#ifndef INCLUDE_DT_MLNODE_H_
#define INCLUDE_DT_MLNODE_H_

#include <math.h>
#include <time.h>

namespace ml {
  class MlNode {
  public:
    MlNode();
    ~MlNode() = default;

    //Hidden LayerNode
    void setInNode(MlNode* inNodes, int numInNodes, int numLayerNodes);

    float getValue();
    void setValue(float value);
    void calValue(bool isLayerOne);

    int getError();
    void setError(int error);

    bool activate();
    void adjutWeight(float value);

  private:
    struct InLink {
      MlNode* node;
      float weight;
      float error;
      float weightOld;
    };

    InLink* inNodes;
    int numInNodes;

    float value;
    float error;
  };
```

```
}

#endif /* INCLUDE_DT_MLNODE_H_ */
```

```
/*
 * DtTree.h
 *
 *  Created on: Feb 24, 2022
 *      Author: tmgreen
 */

#ifndef INCLUDE_DT_RF_H_
#define INCLUDE_DT_RF_H_

#include "greenMds/include/dt/DtTree.h"
#include "greenMds/include/dt/DtNode.h"
#include "greenMds/include/features/features.h"
#include "greenMds/include/data/Testing.h"

#include <stdlib.h>
#include <time.h>
#include <list>

namespace dt {
   class Rf {
      public:
         Rf(dt::DtNode* nodes, uint32_t numNodes);
         virtual ~Rf() = default;

         features::feature getFeature(data::Testing data);

      private:
         const int NUM_TREES = 128;
         const int NODES_PER_TREE = 200;

         DtTree* trees;
         DtNode** nodes;
   };
}

#endif /* INCLUDE_DT_RF_H_ */
```

```
/*
 * Stack.h
 *
 *  Created on: Apr 6, 2022
 *      Author: tmgreen
 */

#ifndef INCLUDE_DT_STACK_H_
#define INCLUDE_DT_STACK_H_

#include "greenMds/include/dt/Ann.h"
#include "greenMds/include/dt/Rf.h"
#include "greenMds/include/data/TrainingManager.h"

namespace ml {
   class Stack {
   public:
      Stack();
      ~Stack() = default;

      features::feature getFeature(data::Testing data);
   private:
      Ann* ann0;
      data::Training* annNodes0;

      dt::Rf* rf0;
      dt::DtNode* rfNodes0;

      dt::DtTree* tree;
      dt::DtNode* dtNodes0;

      dt::Rf* rf1;
      dt::DtNode* rfNodes1;
   };
}

#endif /* INCLUDE_DT_STACK_H_ */
```

```
/*
 * features.h
 *
 *  Created on: Feb 21, 2022
 *      Author: tmgreen
 */

#ifndef INCLUDE_FEATURES_FEATURES_H_
#define INCLUDE_FEATURES_FEATURES_H_

namespace features {
   enum feature {
      NORMAL,
      BAD
   };
}

#endif /* INCLUDE_FEATURES_FEATURES_H_ */
```

```
#include "greenMds/api_include/api/FeatureFactory.h"

api::FeatureFactory::FeatureFactory() {

}

void api::FeatureFactory::addTraningData(std::string data, features::feature dataType) {
   data::Log log(data);
   data::Training trainingData(log, dataType);

   data::TrainingManager::getInstance()->addData(trainingData);
}

void api::FeatureFactory::generateModel() {
   //Sort list and apply feature elimination
   data::TrainingManager::getInstance()->sort();
   data::TrainingManager::getInstance()->eliminate();

   stack = new ml::Stack();
}

features::feature api::FeatureFactory::getFeature(std::string data) {
   data::Log log(data);
   data::Testing testing(log);

   return stack->getFeature(testing);
}
```

```
/*
 * Log.cpp
 *
 *  Created on: Mar 7, 2022
 *      Author: tmgreen
 */

#include "greenMds/include/data/Log.h"

data::Log::Log(std::string log) {
   std::smatch typeMatch;
   std::smatch recTimeMatch;
   std::smatch timeMatch;
   std::smatch sendTimeMatch;
   std::smatch senderMatch;
   std::smatch senderPudoMatch;
   std::smatch msgIdMatch;
   std::smatch posMatch;
   std::smatch posNoiseMatch;
   std::smatch speedMatch;
   std::smatch speedNoiseMatch;
   std::smatch aclMatch;
   std::smatch aclNoiseMatch;
   std::smatch hedMatch;
   std::smatch hedNoiseMatch;

   std::regex_search(log, typeMatch, std::regex(TYPE_MATCH));
   std::regex_search(log, recTimeMatch, std::regex(REC_TIME_MATCH));
   std::regex_search(log, timeMatch, std::regex(TIME_MATCH));
   std::regex_search(log, sendTimeMatch, std::regex(SEND_TIME_MATCH));
   std::regex_search(log, senderMatch, std::regex(SENDER_MATCH));
   std::regex_search(log, senderPudoMatch, std::regex(SENDER_PUDO_MATCH));
   std::regex_search(log, msgIdMatch, std::regex(MSG_ID_MATCH));
   std::regex_search(log, posMatch, std::regex(POS_MATCH));
   std::regex_search(log, posNoiseMatch, std::regex(POS_NOISE_MATCH));
   std::regex_search(log, speedMatch, std::regex(SPEED_MATCH));
   std::regex_search(log, speedNoiseMatch, std::regex(SPEED_NOISE_MATCH));
   std::regex_search(log, aclMatch, std::regex(ACL_MATCH));
   std::regex_search(log, aclNoiseMatch, std::regex(ACL_NOISE_MATCH));
   std::regex_search(log, hedMatch, std::regex(HED_MATCH));
   std::regex_search(log, hedNoiseMatch, std::regex(HED_NOISE_MATCH));

   type = typeMatch[1];
   recTime = recTimeMatch[1];
   time = timeMatch[1];
   sendTime = sendTimeMatch[1];
```

```
    sender = senderMatch[1];
    senderPudo = senderPudoMatch[1];
    msgId = msgIdMatch[1];
    pos = posMatch[1];
    posNoise = posNoiseMatch[1];
    speed = speedMatch[1];
    speedNoise = speedNoiseMatch[1];
    acl = aclMatch[1];
    aclNoise = aclNoiseMatch[1];
    hed = hedMatch[1];
    hedNoise = hedNoiseMatch[1];

    if(time != "" && recTime == "") {
        recTime = time;
    }
}

std::string data::Log::minLog() {
    return type
            + pos
            + posNoise
            + speed
            + speedNoise;
}

std::string data::Log::getType() {
    return type;
}

std::string data::Log::getPos() {
    return pos;
}

std::string data::Log::getPosNoise() {
    return posNoise;
}

std::string data::Log::getSpeed() {
    return speed;
}

std::string data::Log::getSpeedNoise() {
    return speedNoise;
}

std::string data::Log::getRecTime() {
```

```
    return recTime;
}
```

```
/*
 * Testing.cpp
 *
 *  Created on: Feb 21, 2022
 *      Author: tmgreen
 */

#include "greenMds/include/data/Testing.h"

data::Testing::Testing() {

}

data::Testing::Testing(const Log& data) {
   this->data = data;
   std::string logStr = this->data.minLog();
   this->normalizedData = getNormalizedData(logStr);
}

data::Testing::~Testing() {

}

data::Log data::Testing::getData() {
   return this->data;
}

long data::Testing::getNormalizedData() const {
   return this->normalizedData;
}

long data::Testing::getNormalizedData(const std::string& data) {
   //Only allow alphanumeric numbers. Ignore everything else.
   std::string allowedLetters =
            "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789.,";
   int normalizedData = 0;
   for(unsigned int ii=0; ii<data.length(); ++ii) {
      if(allowedLetters.find(data.c_str()[ii]) == std::string::npos) {
         continue;
      } else {
         normalizedData += data.c_str()[ii];
      }
   }

   return normalizedData;
}
```

```
/*
 * Training.cpp
 *
 *  Created on: Feb 21, 2022
 *      Author: tmgreen
 */

#include "greenMds/include/data/Training.h"

data::Training::Training() : data::Testing() {

}

data::Training::Training(Log data, features::feature dataType) :
       data::Testing(data){
   this->dataType = dataType;
}

data::Training::~Training() {

}

void data::Training::setDataType(features::feature dataType) {
   this->dataType = dataType;
}

features::feature data::Training::getDataType() {
   return this->dataType;
}

bool data::Training::operator==(const data::Training& other) {
   return getNormalizedData() == other.getNormalizedData();
}

bool data::Training::operator<(const Training& other) {
   return getNormalizedData() < other.getNormalizedData();
}
```

```
/*
 * TrainingManager.cpp
 *
 *  Created on: Feb 21, 2022
 *      Author: tmgreen
 */

#include "greenMds/include/data/TrainingManager.h"

data::TrainingManager* data::TrainingManager::getInstance() {
    static data::TrainingManager manager;

    return &manager;
}

void data::TrainingManager::addData(data::Training data) {
    dataList.push_front(data);
}

data::Training data::TrainingManager::getData(int index) {
    return *std::next(dataList.begin(), index);
}

void data::TrainingManager::removeData(data::Training data) {
    this->dataList.remove(data);
}

void data::TrainingManager::sort() {
    dataList.sort();
}

int data::TrainingManager::getCount() {
    return dataList.size();
}

void data::TrainingManager::eliminate() {
    long long average = 0;
    for(data::Training data : dataList) {
        average += data.getNormalizedData();
    }

    std::list<data::Training>::iterator itr = dataList.begin();

    std::list<data::Training>::iterator last;
    int badVote = 0;
    int goodVote = 0;
```

```
    itr = dataList.begin();
    last = itr;
    ++itr;
    while(itr != dataList.end()) {
        if(itr->getNormalizedData() == last->getNormalizedData()) {
            if(itr->getDataType() == features::BAD) {
                ++badVote;
            } else {
                ++goodVote;
            }
            itr = dataList.erase(itr);
            continue;
        } else {
            if(goodVote > badVote) {
                last->setDataType(features::NORMAL);
            } else {
                last->setDataType(features::BAD);
            }
            badVote = 0;
            goodVote = 0;
        }
        last = itr;
        ++itr;
    }
}

data::TrainingManager::TrainingManager() {

}
```

```
/*
 * Ann.cpp
 *
 *  Created on: Mar 24, 2022
 *      Author: tmgreen
 */

#include "greenMds/include/dt/Ann.h"

ml::Ann::Ann(data::Training* data, int numData) {
   //Build the model
   numLayerOne = 60;
   numLayerTwo = 30;

   this->input = new MlNode[5];
   this->output = new MlNode();

   hLayerTwo = new MlNode[numLayerTwo];
   hLayerOne = new MlNode[numLayerOne];

   for(int index=0; index<numLayerOne; ++index) {
      MlNode* node = &this->hLayerOne[index];
      node->setInNode(this->input, 5, numLayerOne);
   }

   for(int index=0; index<numLayerTwo; ++index) {
      MlNode* node = &this->hLayerTwo[index];
      node->setInNode(hLayerOne, numLayerOne, numLayerTwo);
   }

   this->output->setInNode(hLayerTwo, numLayerTwo, 1);

   for(int index=0; index<numData; ++index) {
      data::Training training = data[index];

      //We were right dont bother adjusting
      if(getFeature((data::Testing)training) == training.getDataType()) {
         continue;
      }

      int actualValue = 0;
      if(training.getDataType() == features::BAD) {
         actualValue = 1;
      }

      input[0].setValue(data::Training::getNormalizedData(training.getData().getType()));
```

```
      input[1].setValue(data::Training::getNormalizedData(training.getData().getPos()));
      input[2].setValue(data::Training::getNormalizedData(training.getData().getPosNoise()));
      input[3].setValue(data::Training::getNormalizedData(training.getData().getSpeed()));
      input[4].setValue(data::Training::getNormalizedData(training.getData().getSpeedNoise()));

      for(int itr=0; itr<1000 && getFeature((data::Testing)training) != training.getDataType(); ++itr) {
        float guessValue = 0;

        //feed forward

        for(int index=0; index<numLayerOne; ++index) {
          MlNode* node = &this->hLayerOne[index];
          node->calValue(true);
          guessValue += pow((node->getValue() - actualValue), 2);
        }

        for(int index=0; index<numLayerTwo; ++index) {
          MlNode* node = &this->hLayerTwo[index];
          node->calValue(false);
          guessValue += pow((node->getValue() - actualValue), 2);
        }

        output->calValue(false);
        guessValue += pow((output->getValue() - actualValue), 2);

        guessValue = guessValue / (numLayerOne + numLayerTwo + 1 + 5);

        //back propagate

        output->adjutWeight(guessValue);

        for(int index=0; index<numLayerTwo; ++index) {
          MlNode* node = &this->hLayerTwo[index];
          node->adjutWeight(guessValue);
        }

        for(int index=0; index<numLayerOne; ++index) {
          MlNode* node = &this->hLayerOne[index];
          node->adjutWeight(guessValue);
        }
      }
    }

}

features::feature ml::Ann::getFeature(data::Testing data) {
```

```
    features::feature guess = features::NORMAL;

    input[0].setValue(data::Training::getNormalizedData(data.getData().getType()));
    input[1].setValue(data::Training::getNormalizedData(data.getData().getPos()));
    input[2].setValue(data::Training::getNormalizedData(data.getData().getPosNoise()));
    input[3].setValue(data::Training::getNormalizedData(data.getData().getSpeed()));
    input[4].setValue(data::Training::getNormalizedData(data.getData().getSpeedNoise()));

    for(int index=0; index<numLayerOne; ++index) {
        MlNode* node = &this->hLayerOne[index];
        node->calValue(true);
    }

    for(int index=0; index<numLayerTwo; ++index) {
        MlNode* node = &this->hLayerTwo[index];
        node->calValue(false);
    }

    this->output[0].calValue(false);
    if(this->output[0].activate()) {
        guess = features::BAD;
    }

    return guess;
}
```

```
/*
 * DtNode.cpp
 *
 *  Created on: Feb 24, 2022
 *      Author: tmgreen
 */

#include "greenMds/include/dt/DtNode.h"

dt::DtNode::DtNode() :
        parent(NULL),
        leftChild(NULL),
        rightChild(NULL),

                data(data::Training(data::Log("{\"type\":4,\"time\":25299.986984130697,\"sender\":1429,\"atta
                ckerType\":1,\"messageID\":6403019,\"pos\":[4431.179821821522,5322.041109381504,1.895],\
                "pos_noise\":[0.0,0.0,0.0],\"spd\":[-5.390899823618186,-2.0799435132659438,0.0],\"spd_noise
                \":[0.0,0.0,0.0]}"), features::NORMAL)),
        leaf(true) {

}

bool dt::DtNode::operator==(DtNode& other) {
    return this->getData() == other.getData();
}

bool dt::DtNode::operator<(DtNode& other) {
    return this->getData() < other.getData();
}

void dt::DtNode::setParent(DtNode* parent) {
    this->parent = parent;
}

void dt::DtNode::setLeftChild(DtNode* leftChild) {
    this->leftChild = leftChild;
    if(this->leftChild == NULL && this->rightChild == NULL) {
        leaf = true;
    } else {
        leaf = false;
    }
}

void dt::DtNode::setRightChild(DtNode* rightChild) {
    this->rightChild = rightChild;
    if(this->leftChild == NULL && this->rightChild == NULL) {
```

```
        leaf = true;
    } else {
        leaf = false;
    }
}

dt::DtNode* dt::DtNode::getParent() {
    return parent;
}

dt::DtNode* dt::DtNode::getLeftChild() {
    return leftChild;
}

dt::DtNode* dt::DtNode::getRightChild() {
    return rightChild;
}

void dt::DtNode::setData(data::Training data) {
    this->data = data;
}

data::Training dt::DtNode::getData() {
    return data;
}

bool dt::DtNode::isLeaf() {
    return leaf;
}
```

```
/*
 * DtTree.cpp
 *
 *  Created on: Feb 24, 2022
 *      Author: tmgreen
 */

#include "greenMds/include/dt/DtTree.h"

dt::DtTree::DtTree(DtNode* nodes, uint32_t numNodes) {
   root = *splitTree(nodes, -1, 0, numNodes-1, true, false, false);
}

features::feature dt::DtTree::getFeature(data::Testing data) {
   dt::DtNode* runningNode = &root;
   dt::DtNode* lastNode = NULL;

   while(runningNode != NULL) {
      lastNode = runningNode;
      if(data.getNormalizedData() > runningNode->getData().getNormalizedData()) {
         runningNode = runningNode->getRightChild();
      } else if(data.getNormalizedData() < runningNode->getData().getNormalizedData()) {
         runningNode = runningNode->getLeftChild();
      } else {
         break; //Exact match, exit loop
      }
   }

   return lastNode->getData().getDataType();
}

/**********************************
 *          5
 *        /  \
 *       2    7
 *      / \  / \
 *     1  3 6   8
 *         \     \
 *          4     9
 *                 \
 *                 10
 */
dt::DtNode* dt::DtTree::splitTree(DtNode* nodeList,
      int index,
      int lowerBound,
      int upperBound,
```

```
    bool starting,
    bool leftSplit,
    bool rightSplit) {
  DtNode* node;

  if(starting) {
    index = upperBound;
  }

  if(!starting && ((leftSplit && index == lowerBound) || (rightSplit && index == upperBound))) {
    node = nodeList+index; //Last node in tree
  } else if((!starting && ((leftSplit && index < lowerBound) || (rightSplit && index > upperBound)))) {
    node = NULL; //End of tree
  } else { //Otherwise continue creating tree
    int newIndex = (upperBound + lowerBound - 1) / 2;
    node = nodeList + newIndex;
    //Split list between left and right
    node->setLeftChild(splitTree(nodeList, newIndex-1, lowerBound, newIndex-1, false, true, false));
    node->setRightChild(splitTree(nodeList, newIndex+1, newIndex+1, upperBound, false, false, true));
  }

  return node;
}

dt::DtNode dt::DtTree::getRoot() {
  return root;
}
```

```
/*
 * MlNode.cpp
 *
 *  Created on: Mar 24, 2022
 *      Author: tmgreen
 */

#include "greenMds/include/dt/MlNode.h"
#include <math.h>

ml::MlNode::MlNode() {

}

void ml::MlNode::setInNode(MlNode* inNodes, int numInNodes, int numLayerNodes) {
   srand( (unsigned)time( NULL ) );

   this->numInNodes = numInNodes;
   this->inNodes = new InLink[numInNodes];
   for(int index=0; index<numInNodes; ++index) {
      this->inNodes[index].node = &inNodes[index];
      float randVal = rand();
      float factor = (float)sqrt((float)2/(float)numLayerNodes);

      this->inNodes[index].weight = ((float)randVal * (float)factor)/(float)RAND_MAX;
   }

   error = 0;
}

float ml::MlNode::getValue() {
   return this->value;
}

void ml::MlNode::setValue(float value) {
   this->value = value;
}

void ml::MlNode::calValue(bool isLayerOne) {
   const int BIAS = 0;
   this->value = 0;
   for(int index=0; index<numInNodes; ++index) {
      if(isLayerOne || inNodes[index].node->activate()) {
         this->value += inNodes[index].weight * inNodes[index].node->getValue();
      }
   }
```

```
    this->value += BIAS;

    int temp = this->value;
    if(temp < 0) {
        temp = 0;
    }
}

int ml::MlNode::getError() {
    return this->error;
}

void ml::MlNode::setError(int error) {
    this->error = error;
}

// Rectified Linear Activation
bool ml::MlNode::activate() {
    return this->value > 0;
}

void ml::MlNode::adjutWeight(float value) {
    const float LEARNING_RATE = .01; //.00135
    const float MC = 0;

    //Calulate total weight
    float totalWeight = 0;
    for(int index=0; index<numInNodes; ++index) {
        totalWeight += inNodes[index].weight;
    }

    for(int index=0; index<numInNodes; ++index) {
        inNodes[index].error = inNodes[index].weight/totalWeight;
    }

    for(int index=0; index<numInNodes; ++index) {
        float dWeight = (MC*inNodes[index].weightOld) - (LEARNING_RATE*(tanh(inNodes[index].error)));
        inNodes[index].weight += dWeight;
        inNodes[index].weightOld = dWeight;
    }
}
```

```
#include "greenMds/include/dt/Rf.h"

dt::Rf::Rf(dt::DtNode* nodes, uint32_t numNodes) {
  trees = new DtTree[NUM_TREES];
  this->nodes = new DtNode*[NUM_TREES];

  srand (time(NULL));
  for(int tree=0; tree<NUM_TREES; ++tree) {
    std::list<DtNode> treeNodes;
    for(int node=0; node<NODES_PER_TREE; ++node) {
      treeNodes.push_front(nodes[rand()%(numNodes-1)]);
    }
    treeNodes.sort();

    std::list<DtNode>::iterator last;
    int badVote = 0;
    int goodVote = 0;
    std::list<DtNode>::iterator itr = treeNodes.begin();
    last = itr;
    ++itr;
    while(itr != treeNodes.end()) {
      if(itr->getData().getNormalizedData() == last->getData().getNormalizedData()) {
        if(itr->getData().getDataType() == features::BAD) {
          ++badVote;
        } else {
          ++goodVote;
        }
        itr = treeNodes.erase(itr);
        continue;
      } else {
        if(goodVote > badVote) {
          last->getData().setDataType(features::NORMAL);
        } else {
          last->getData().setDataType(features::BAD);
        }
        badVote = 0;
        goodVote = 0;
      }
      last = itr;
      ++itr;
    }

    this->nodes[tree] = new DtNode[NODES_PER_TREE];
    std::copy(treeNodes.begin(), treeNodes.end(), this->nodes[tree]);
    trees[tree] = dt::DtTree(this->nodes[tree], NODES_PER_TREE);
  }
```

```
}

features::feature dt::Rf::getFeature(data::Testing data) {
  int normal = 0;
  int bad = 0;

  for(int tree=0; tree<NUM_TREES; ++tree) {
    switch(trees[0].getFeature(data)) {
    case features::NORMAL:
      ++normal;
      break;
    case features::BAD:
      ++bad;
      break;
    }
  }

  features::feature guess;
  if(bad > normal) {
    guess = features::BAD;
  } else {
    guess = features::NORMAL;
  }

  return guess;
}
```

```
/*
 * Stack.h
 *
 *  Created on: Apr 6, 2022
 *      Author: tmgreen
 */

#include "greenMds/include/dt/Stack.h"

ml::Stack::Stack() {
   int numData = data::TrainingManager::getInstance()->getCount();
   const int NUM_FOLDS = 1;
   const int NODES_PER_FOLD = numData/NUM_FOLDS;

   annNodes0 = new data::Training[numData];
   for(int nodeIndex=0; nodeIndex<numData; nodeIndex+=NUM_FOLDS) {
      annNodes0[nodeIndex] = data::TrainingManager::getInstance()->getData(nodeIndex);
   }
   ann0 = new ml::Ann(annNodes0, NODES_PER_FOLD);

   rfNodes0 = new dt::DtNode[numData];
   for(int nodeIndex=0; nodeIndex<numData; nodeIndex+=NUM_FOLDS) {
      rfNodes0[nodeIndex] = dt::DtNode();
      rfNodes0[nodeIndex].setData(data::TrainingManager::getInstance()->getData(nodeIndex));
   }
   rf0 = new dt::Rf(rfNodes0, NODES_PER_FOLD);

   dtNodes0 = new dt::DtNode[numData];
   for(int nodeIndex=0; nodeIndex<numData; nodeIndex+=NUM_FOLDS) {
      dtNodes0[nodeIndex] = dt::DtNode();
      dtNodes0[nodeIndex].setData(data::TrainingManager::getInstance()->getData(nodeIndex));
   }
   tree = new dt::DtTree(dtNodes0, NODES_PER_FOLD);

   int secondIndex=0;
   rfNodes1 = new dt::DtNode[3*NODES_PER_FOLD];
   for(int nodeIndex=0; nodeIndex<numData; nodeIndex+=NUM_FOLDS) {
      data::Training trainingData = data::TrainingManager::getInstance()->getData(nodeIndex);

      rfNodes1[secondIndex].setData(data::Training(trainingData.getData(),
            ann0->getFeature((data::Testing) trainingData)));
      ++secondIndex;

      rfNodes1[secondIndex].setData(data::Training(trainingData.getData(),
            rf0->getFeature((data::Testing) trainingData)));
      ++secondIndex;
```

```
    rfNodes1[secondIndex].setData(data::Training(trainingData.getData(),
        tree->getFeature((data::Testing) trainingData)));
    ++secondIndex;
  }
  rf1 = new dt::Rf(rfNodes1, 3*NODES_PER_FOLD);
}

features::feature ml::Stack::getFeature(data::Testing data) {
  return rf1->getFeature(data);
}
```