Summer 7-27-2021

DISTRIBUTED NEURAL NETWORK BASED ARCHITECTURE FOR DDoS DETECTION IN VEHICULAR COMMUNICATION SYSTEMS

Nicholas Jaton
University of Nebraska-Lincoln, njaton@unomaha.edu

DISTRIBUTED NEURAL NETWORK BASED ARCHITECTURE FOR DDoS DETECTION IN VEHICULAR COMMUNICATION SYSTEMS

by

Nicholas A. Jaton

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfillment of Requirements

For the Degree of Master of Science

Major: Telecommunications Engineering

Under the Supervision of Professor Yi Qian

Lincoln, Nebraska

July, 2021

DISTRIBUTED NEURAL NETWORK BASED ARCHITECTURE FOR DDoS DETECTION IN VEHICULAR COMMUNICATION SYSTEMS

Nicholas Jaton, M.S.

University of Nebraska, 2021

Advisor: Yi Qian

With the continued development of modern vehicular communication systems, there is an ever growing need for cutting edge security in these systems. A misbehavior detection systems (MDS) is a tool developed to determine if a vehicle is being attacked so that the vehicle can take steps to mitigate harm from the attacker. Some attacks such as distributed denial of service (DDoS) attacks are a concern for vehicular communication systems. During a DDoS attack, multiple nodes are used to flood the target with an overwhelming amount of communication packets. In this thesis, we investigated the current MDS literature and how it is used to prevent DDoS attacks. Additionally, we proposed and developed a new distributed multilayer perceptron classifier (MLPC) and evaluated it using vehicular communication simulations generated using OMNeT++, Veins, and Sumo. These simulations contained a group of normal vehicles and some attacking vehicles. During the simulations, two different attacks were conducted. Apache Spark was then used to create the distributed MLPC. The median F1-score for this MLPC architecture was 95%. This architecture outperformed linear regression and support vector machines, which achieved 89% and 88% respectively, but was unable to better random forests and gradient boosted trees which both achieved a 97% F1-score. Using Amazon Web Services (AWS), it was determined that model training and detection time were not significantly increased with the inclusion of

additional nodes after three nodes including the master.

Acknowledgments

I would like to express my greatest gratitude to Dr. Yi Qian for his guidance and support throughout my research. Your feedback and encouragement has been tremendously important to me over the past year.

I would like to thank Dr. Sohan Gyawali for his insights and guidance in vehicular communication simulations and misbehavior detection systems. Your wisdom was paramount in understanding how to develop in OMNeT++.

I would like to thank Dr. Hamid Sharif and Dr. Kuan Zhang for taking the time to contribute on my thesis committee. Additionally, I would like to thank Dr. Hamid Sharif for his guidance and encouragement throughout my graduate degree.

A special thank you to my significant other Jessica Rodino, for your love, support, and patience as I spent my nights and weekends conducting research.

Additionally, thank you to my father Nick Jaton, mother Michele Dees, and stepfather Patrick Dees for your encouragement and support.

Contents

List of Tables

List of Figures

List of Algorithms

List of Acronyms

ACK acknowledgement.

ANN artificial neural network.

AODV ad hoc on-demand distance vector.

AWS Amazon Web Services.

CAN controller area network.

CEAP collection, exchange, analysis, and propagation.

CNN convolutional neural network.

CO2 Carbon Dioxide.

CPU Central Processing Unit.

CTS clear to send.

CUSUM Cumulative Sum.

D-FICCA Density-based fuzzy imperialist competitive clustering algorithm.

DARPA Defense Advanced Research Projects Agency.

DBSCAN density-based spatial clustering of applications with noise.

DCMD data-centric misbehavior detection.

DDoS distributed denial of service.

DFT discrete Fourier transform.

DNN deep neural network.

DoS denial of service.

DRL deep reinforcement learning.

DS Dempster-Sharfer theory.

DWT discrete wavelet transform.

EAD driver's emotional state algorithm.

ELIDV efficient and light-weight intrusion detection mechanism for vehicular network.

EMR Elastic MapReduce.

FFNN feed forward neural network.

FN false negative.

FP false positive.

G-CUSUM Generalized Cumulative Sum.

GAF geographic aware flooding.

GBDT gradient boosting decision trees.

GBT gradient boosted trees.

GEM Geometric Entropy Minimization.

GMM gaussian mixed model.

GUI graphical user interface.

H-IDS Hybrid Intrusion Detection System.

HTM hierarchical temporal memory.

ICA imperialist competitive algorithm.

IDS intrusion detection systems.

IP internet protocol.

K-MICA modify imperialist competitive algorithm and K-means.

KAREN Kiwi Advanced Research and Education Network.

L-BFGS Limited-memory Broyden–Fletcher–Goldfarb–Shanno algorithm.

Linear-CRF Linear chain conditional random field.

LR logistic regression.

MA misbehavior authority.

MANET mobile ad hoc network.

MDS misbehavior detection systems.

MITM man-in-the-middle.

MLPC multilayer perceptron classifier.

MOVE Mobility VEhicles.

MPR multipoint relay.

MVSA multivariant stream analysis.

NED network description.

NGSIM Next Generation Simulation.

NN neural network.

NS2 Network Simulator 2.

NS3 Network Simulator 3.

OCTANE Open Car Testbed and Network Experiments.

ODIT online discrepancy test.

OLSR optimized link state routing.

OMNeT++ Objective Modular Network Testbed in C++.

PCA principle component analysis.

RAM Random Access Memory.

RBS Reference Broadcast Method.

RDD resilient distributed datasets.

RF random forest.

RSA Rivest–Shamir–Adleman.

RSSI received signal strength indicator.

RSU road side unit.

RTS request to send.

SDVN software defined vehicular networks.

SUMO Simulation of Urban Mobility.

SVM support vector machines.

tcl tool command language.

TN true negative.

TP true positive.

TTL time to live.

V2V vehicle to vehicle.

V2X vehicle to everything.

VANET vehicular ad hoc network.

VANET QoS-OLSR vehicular ad hoc network quality of service link state routing.

VeReMi Vehicular Reference MisBehavior Dataset.

WSM WAVE short messages.

List of Symbols

R_{de}	Emotional recognition coefficient
X_i	Input value of perceptron
W_i	Weight value of perceptron
Y	Output value of perceptron
n	Number of perceptron inputs (X_i)
z	Output value of perceptron
θ	Threshold used to determine binary output of perceptron
H_j	Hidden Layer of multilayer perceptron classifier
O_k	Output Layer of multilayer perceptron classifier
x_k	optimal value estimate at k
∇f_k	gradient value in L-BFGS
y_k	Change of Gradients
s_k	Displacement
H_k	Inverse Hessian approximation
q	Current gradient value in L-BFGS

r	Search direction for minimization function
α_k	Step Length, which is used to satisfy Wolfe conditions
β	Variable used to represent $\rho_i y_i^T r$ when computing $H_k \nabla f_k$
H_k^0	Initial inverse Hessian approximation
ρ_k	Variable used to denote $\frac{1}{y_k^T s_k}$
I	Initial Hessian Approximation
m	Level of memory used in L-BFGS
p_k	Search direction
γ_k	Scaling factor
N	Number of features used in MLPC architecture

Chapter 1

Introduction

1.1 Introduction

With modern technologies such as Telsa's self driving vehicles entering the mass market, the demand for stronger, more robust vehicular communication systems increases. As the computational power and connectivity of these vehicles increases, malicious users will find ways to use these resources to generate harm. According to the United States Bureau of Transportation Statistics, there were 228,679,719 licensed drivers in the United States in 2019 [5]. As car manufactures produce 5G enabled vehicles, a significant amount of potential networked vehicles may be vulnerable to attackers. To prevent this, researchers developed misbehavior detection systems (MDS) to detect and mitigate these attacks. The MDS consists of an algorithm designed to determine if an incoming data packet could be the result of an attack. These algorithms are difficult to create due to the large amount of attacks a malicious user could produce including man-in-the-middle (MITM), grey hole, black hole, distributed denial of service (DDoS), and Sybil attacks. Statistical and machine

learning approaches show great promise in both traditional and vehicular communication networks MDS, but there is much work to be done in the field.

1.2 Distributed Denial of Service Attacks

A DDoS attack involves the malicious party utilizing multiple vehicles to produce high quantities of network traffic in an attempt to damage the networks integrity. The vehicles used in a DDoS attack are not aware that they are being utilized by the malicious user. For this reason, the computers used in DDoS attacks are often referred to as ”zombies”. Blocking the vehicles' communication systems could cause a vehicular accident and in a worse case scenario, the loss of human life. This attack can be difficult to detect due to its use of zombie vehicles. Fig. 1.1 shows how the attacks are conducted in vehicular communication systems. This thesis focuses on the detection and evaluation of DDoS in vehicular communication networks.

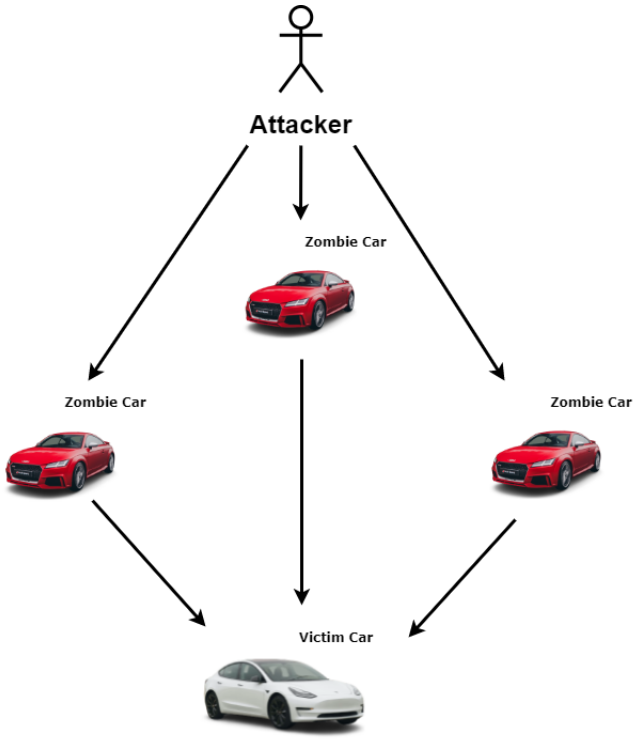

Figure 1.1: DDoS Attack Structure in Vehicular Communications [2] [3]

1.3 Proposed Solution

There are two major contributions of this thesis, the first being the development of a DDoS attack in Objective Modular Network Testbed in C++ (OMNeT++) and the second being the proposed MDS based on a distributed neural network (NN) design. OMNeT++ is a cutting edge simulation tool for network communications. Unlike most network tools, OMNeT++ allows for the implementation of vehicular mobility via Veins and SUMO. The implemented simulations contain a number of vehicles, allowing the MDS to utilize location and emission data from each vehicle. In a real world scenario, this information is valuable and easy to collect.

The proposed MDS was built using results found by [6] on distributed MDS on already existing vehicular communication datasets. Their research showed promising results using distributed random forest models, but this analysis was not conducted on modern vehicular communication simulators such as Network Simulator 2 (NS2) or OMNeT++. Additionally, [1] and [7] both generated very impressive results when utilizing neural networks on NS2 based vehicular communication simulations. Thus, this study investigated the hypothesis that a distributed neural network would prove to be an effective MDS method, but also that distributed technologies would also prove valuable in modern network simulations.

1.4 Thesis Organization

There are four additional chapters in this thesis. Chapter 2 introduces current research literature related to misbehavior detection in vehicular communication systems. The focus and interest of these systems is on the effectiveness of modern statistical learning methods that

are currently being implemented. Chapter 3 contains the proposed MDS and a discussion on the detection methods used in the MDS, including an introduction to neural networks and distributed computing. Chapter 4 contains the evaluations using the developed simulation models and the discussions on the simulation results. Chapter 5 gives conclusions and the future work.

Chapter 2

Literature Survey

2.1 An Overview for Security in Vehicular Networks

Before one can understand the importance of misbehavior detection in vehicular networks, it is important to understand general vehicular network security architecture. The authors in [8] developed a review of 114 research articles related to vehicular ad hoc network (VANET) security. As described in the aforementioned review, VANETs are targets to various attacks such as Bogus Information, Denial of Service, impersonation, eavesdropping, message suspension, and hardware tampering. The misbehavior detection systems described later in this thesis work to counter these attacks. As described in [8], other forms of security mechanisms such as public key cryptography, RSA, and Elliptic Curve Cryptography play a critical role in reducing the amount of intrusions that makes it into the

system.

The remainder of this survey focuses specifically at misbehavior detection systems (MDS) for vehicular communication networks. In the process of proposing a new reporting protocol, [9] managed to effectively detail the steps needed to build a MDS. These steps are as follows:

1. Misbehavior detection
2. Misbehavior reporting
3. Misbehavior investigation

Misbehavior detection is referred to as act of determining if a vehicle is suspicious for abnormal behaviors. Most of the research work referenced in the following section is involved with this first step of the MDS. After misbehavior is detected by this first step in the system, the misbehavior will be reported to a misbehavior authority (MA). The MA will then be able to complete an investigation into the accusation. In [9], the researchers developed a new misbehavior reporting message based on ASN.1. The message contains multiple containers of information on the misbehaving vehicle. Examples of these containers include the *EvidenceContainer*, which contains information or previous reports of vehicle in question, and the *DetectionReferenceContainer* which includes information regarding the type of error reported.

Because VANETs are a subset of mobile ad hoc network (MANET), it can be valuable to view the wider body of MANET literature. An example of such research is [10]. The authors proposed an clustering based intrusion detection systems (IDS). Detailed descriptions

of both the strengths and weaknesses of IDS architectures were included by the authors. The stand-alone IDS architecture is the first of 3 architectures described. Stand-alone IDS architectures are completely self contained systems. Cooperative architectures allow for communications between neighboring nodes in the system. Lastly, Hierarchical IDS architectures involve dividing the notes in clusters for use in detection. Specific designs within each of the described architectures are detailed in [10]. There was no statistical analysis completed by the authors to determine the effectiveness of their IDS architecture against the other architectures designed.

2.2 Vehicular Network Datasets and Simulation Software

To assist researchers in developing reproducible research, the authors in [11] generated the Vehicular Reference MisBehavior Dataset (VeReMi). The authors specifically pointed out the clear issue that many researchers are not using reproducible datasets and software in the study of Vehicular Networks. VeReMi contains message logs designed to be used when building misbehavior detection systems. The source code that generates this dataset is public and contains over 255 simulations. VeReMi also includes multiple different attack types, attacker densities, and different attack traffic densities. This gives the user many options when designing the dataset.

Heijden et al. claim that this is the first dataset of its kind. This is a great achievement, though the dataset is not perfect. VeReMi is only designed for detection algorithms and

a user is not able to analyze the effects of multiple attacks in a single simulation. After outlining VeReMi, the authors continued to use this data to conduct an analysis of the Gini index. The Gini index results seemed to carry less of an impact than the dataset itself.

A simulation tool called Simulation of Urban Mobility (SUMO) is frequently referenced in many vehicular research activities. SUMO is an open source traffic simulation software. As referenced by [12], SUMO is frequently used in the study of vehicular communications. Behrish et al. also stated that generally, this tool will be combined with another comminucation simulator such as NS2 or NS3. [13] is an example of SUMO's usage when combined with NS2. The work of Gruebler et al. is further discussed in the following sections. SUMO has the ability to import network models from other simulation software such as VISUM, Vissim and MATsim. [12] highlighted a handful of recent SUMO projects such as iTETRIS, VABENE, and CityMobil. Of the tools discussed, iTETRIS shows the most practicality for researchers of vehicular networks. iTETRIS was developed with the goal of simulating V2X communications at the whole city scale. This was done by connecting NS2 and SUMO with the iTETRIS control system. Berish et al. stated that a graphical user interface used to design road networks has been created but, this tool is not available to the public at the time of publication (2011).

OMNeT++ is another simulation tool that is frequently utilized in vehicular communications research. This tool allows for a combination of graphical user interface (GUI), C++, and network description (NED) based options for simulation development. Mohd et al. utilized this software to show a distributed denial of service (DDoS) on vehicular networks [14]. The study was based around the communications of 4 simulated vehicles. A DDoS attack must come from more than one attacker. In this study, groups of 3, 6, 9,

and 12 attacking vehicles were used in an attempt to disrupt the transmission of packets from the 4 normal cars. Packets of either 1 KB, 10 KB, or 100 KB at 0.1 ms intervals were sent by the attacking vehicles. This implementation in OMNeT++ was successful as the 12 vehicle DDoS attack was able to cause a 13.98% loss of packets between the 4 cars when sending 100 KB packets.

Similarly, Kaur et al. utilized OMNet++ & ReaSE to develop DDoS attacks on Web Servers [15]. This DDoS attack was generated using random distributed zombie hosts. There were 119 devices used in this simulation. Three different routers and a variety of web based servers were all included in this total. As the duration of the DDoS attack increased, the bandwidth required for the web server increased. The amount of packets dropped also increased with the attack duration. When adjusting for buffer size, the study showed that smaller buffers increase the impact of the DDoS attack. Additionally, various researchers have used these simulation techniques to generate the datasets used to design misbehavior detection software [16][17].

OMNeT++ has proven to be a popular tool in modern network security research. The authors in [18] compared the software to a testbed environment using a D-Link wireless access point and two wireless clients. The testbed was replicated completely in OMNeT++. Both the testbed and the simulation were hit by RTS, CTS, and ACK DoS attacks. For the attacks on the real networks, a Wireshark network analyzer was used. The results given by this study favored the use of OMNeT++ as a network simulator. During all three attacks, the OMNeT++ simulation was able to give results within 1% of the real network. Though this paper was not studying vehicular communication networks, its results do help to validate a substantial amount of research that relies on OMNeT++.

2.3 Simulations for Misbehavior Detection in Vehicular Communication Systems

MITM attacks are a clear threat in vehicular communication networks. To accomplish this attack, the hostile party must intercept valuable information from a normal vehicle. Giving the attacker the ability to listen in, delay, drop, or modify packets crucial to vehicular safety. The researchers in [19] showed how these attacks can be created using OMNeT++, SUMO, and VEINS. This simulation used the default map given in Veins. A total of 100 cars and 5 RSUs were used for this simulation along with either 10%, 20%, 30%, 40%, or 50% malicious nodes. With the study simulation being fairly well documented and being based in OMNeT++, it should not be prohibitive to recreate it if needed for verification. Both distributed and fleet based attacks were compared in the study. Between each of the three aforementioned MITM attacks, The distributed nodes produced more harmful results to the network than the fleet based design. This proved more apparent as the percent of attackers increased to 20% or more.

A comparison between the affect of pure flooding, multipoint relay (MPR) flooding, and geographic aware flooding (GAF) techniques on vehicular communication systems was conducted by [20]. GAF works similarly to a pure flooding method if the sender is in front of the recipient but it will drop packets if sent from behind. This is because of the

assumption that if the vehicle is behind the recipient it is no longer a threat. MPR is based on a protocol called optimized link state routing (OLSR). This technique works through the idea of each vehicle relaying the message to only a small amount of nearby vehicles. The message will have to then move through each section of vehicles. The authors proposed that flooding can be used as an effective method of forced communications between vehicles in a safety critical situation. To test this hypothesis, accident simulations were created in NS2. Each method was studied using a flooding rate of 5, 10, and 20 packets per second with each vehicle moving at 130 kph. The pure flooding method proved more effective at delaying background packets while maintaining a high packet delivery ratio. It was mentioned that pure flooding did cause a larger amounts of network traffic overhead than the other flooding methods.

2.4 Statistical Approaches to Misbehavior Detection in Vehicular Communication Networks

Statistical approaches show great promise in the topic of MDS. [16] and [21] show how various machine learning models can be used in the detection of Sybil, DoS, and false alert attacks in vehicular communication networks. Gyawali et al. studied the use cases of k-Nearest Neighbors, Logistic Regression, Decision Tree Classifier, Bagging, and Random Forest in both the publications. The methods tested in [16] and [21] utilized the VeReMi

datasets and Veins for network simulations. This leads to more reproducible research. These papers all the use of precision, recall, and F1-score as the performance metrics. Precision is the ratio of correctly predicted attacks vs. the amount of total predicted attacks. Recall refers to the predicted attacks vs. the number of actual attacking vehicles. The F1-score is the weighted average of the precision and the recall of the model.

Precision and Recall are common metrics used to discuss the efficiency of a statistical algorithm. F1-score is the weighted average of the precision and recall values. A key difference between the papers is the addition of the Dempster-Sharfer theory (DS) in [16]. By adding DS to the existing methods proposed in [21], Gyawali et al. were able to produce higher F1-Scores for the bagging and random forest models which were the top performers in both papers. Table 2.1 contains the results for the proposed false alert verification scheme at a 30% attacker density.

Model	F1-score (Non-DS)	F1-score (DS Applied)
K-Nearest Neighbor	.94	.841
Logistic Regression	.78	.687
Decision Tree Classifier	.94	.941
Bagging	.94	.986
Random Forest	.95	.986

Table 2.1: Effect of DS on F1-score

Other classification methods such as support vector machines (SVM) have also been utilized in misbehavior detection. The collection, exchange, analysis, and propagation (CEAP) algorithm written by [22] utilized SVM in the analysis step of their design. In

this model, various watchdog vehicles monitor communications and use the SVM model to analyze the data collected. The collection and exchange elements of this system are based on the vehicular ad hoc network quality of service link state routing (VANET QoS-OLSR) protocol. Various kernel designs such as linear, polynomial, and Gaussian Radial Basis Function kernels were compared prior to the final design of the SVM algorithm. Though the performance difference was not extreme, the Gaussian Radial Basis function outperformed the others. After using the SVM model for detection, the system will then send out the necessary information to other watchdog vehicles. This MDS was compared to systems using only SVM, DS and averaging on a dataset generated using VanetMobiSim and MATLAB. CEAP outperformed all models to which it was compared in terms of accuracy, attack detection rate, false positive rate, and packet delivery ratio. The less complex SVM model was not far behind CEAP, but this was not the case for the other two models. The simulation did produce a large performance difference between CEAP, DS, and Averaging.

Neural Networks have also been considered in MDS design. The authors in [23] developed a convolutional neural network (CNN) architecture to detect traffic anomalies.The proposed CNN included 8 hidden layers, half of the hidden layers being convolutional and half being sub-sampling layers. The authors compared this model to a system designed using principle component analysis (PCA). The authors of this paper did not use an open source dataset, instead utilizing their own network testbed for their analysis. When comparing the CNN and PCA based models, the authors found that the CNN model produced higher true positive values. The paper also indicated that the CNN model produced less bias than the PCA equivalent.

The authors of [13] also found use in neural networks. This paper described an artificial

neural network (ANN) architecture designed to detect Black Hole attacks in VANETs. A Black Hole attack is a specific form of DoS attack in which important packet information is discarded by the attacker. Fuzzification was applied to the dataset prior to the training of the ANN. The fuzzification process helped to create a more distinct boarder between each feature. SUMO and Mobility VEhicles (MOVE) were used to generate the NS2 simulations ultimately used. In the end, the ANN based MDS had a classification rate of 99% for both the normal and abnormal classes. Since the authors included the true positive, true negative, false negative, and false positive rates, the following values can be found. The precision of the model is .998, recall is .99 and the F1-score is .998.

Ali et al [1] have also produced research on the detection of grey hole attacks using similar strategies. A grey hole attack differs from the previously mentioned black hole attack in the frequency that it attempts to drop packets. The black hole attack will drop all information where the grey hole will only discard some of the information. In [1], both SVM and feed forward neural network (FFNN) are utilized in an effort to detect these grey hole attacks. The attack simulation was conducted using NS2, SUMO, and MOVE, all common tools in vehicular network research. Much like in [13], fuzzification was used prior to the model training process. Both models proved effective at detecting the grey hole attacks produced. In fact, both models produced an accuracy score of over 99.7%. Precision, recall, and f1-scores were calculated so this research can be more easily compared to [13] and [21], etc. Please refer to Table 2.2 for more information.

Model	FFNN	SVM
Precision	.998	.999
Recall	.999	.997
F1-score	.998	.997

Table 2.2: Derived Precision, Recall, and F1-score from [1]

Ghaleb et al.'s research conducted in [7] continues with the theme of artificial neural networks . The authors of this study designed an ANN model based MDS to detect simulated hidden vehicles and illusion attacks. The proposed architecture was broken into 4 stages including Information Acquisition, Information Sharing, Data Analysis and Features Driving, and Misbehavior Detection. In the end, the model contained 7 features and 1 hidden layer with 15 neurons. Next Generation Simulation (NGSIM) was used for the testing and validation of this model. This set was generated using synchronized digital video cameras on real world highways [24]. An 80 / 20 ratio of normal vs. misbehaving vehicles was used. The F-score results for this model remained above .97 for each of the 4 vehicles analyzed. These results are comparable to the results found by [13] and [16]. Unfortunately, because the studies used different datasets, it is hard to make accurate comparisons.

Deep learning methods have also been utilized by MDS researchers. Researchers at Ewha W. University engineered a deep neural network (DNN) architecture to detect suspicious activity in a vehicles controller area network (CAN) [25]. The DNN model focused on detecting false information targeting the vehicles tire pressure light. Open Car Testbed and Network Experiments (OCTANE) was used to generate the dataset used to train and test the DNN model. The model was analyzed with 3 layers, 5 layers and 7 layers. All

three models succeeded in predicting both the attacking packets and normal packets. At 7 layers the ability to detection did not increase but, normal packet detection did by 4%. As expected training time and testing time increased as the layers increased.

The authors in [17] utilized hypothesis testing to detect rogue nodes in VANET systems. The data was generated by a simulation utilizing OMNET++, SUMO, and VACaMobil. For traffic flow modeling the greenshield model was used. False Information and Sybil attacks were both considered and tested in this IDS simulation. If the test statistic is not within a 99% confidence interval, the parameter will be rejected. The proposed methods true positive and false positive values were compared to data-centric misbehavior detection (DCMD) and efficient and light-weight intrusion detection mechanism for vehicular network (ELIDV). Even as the number of attackers increased up to 40%, the proposed IDS managed to get as much if not more true positives than the other two models. Though ELIDV was scored very well, the same is true for the false positives test, with ELIDV being marginally better when the attack rate is over 30%.

Researchers at Dalian University of Technology attempted to locate anomalies based on the driver's emotional state algorithm (EAD) [26]. The derived algorithm called EAD makes use of an emotional recognition coefficient R_{de} to determine if the driver is a cautious, normal, or an aggressive driver. A gaussian mixed model (GMM), that takes the R_{de}, acceleration, velocity, and distance as inputs to detect the outliers. To test this new model, the authors used the NGSIM dataset. This is a publicly available dataset maintained by the U.S. Department of Transportation. The proposed algorithm was compared to the hierarchical temporal memory (HTM) algorithm using precision, recall, and F1-scores. On both simulations, the EAD algorithm outperformed HTM. HTM gave comparable, often better,

precision scores but had very low recall scores. Due to this, the F1-scores remained lower than EAD. These scores in general do not seem to compare well with the models produced in [16] [25] . In order to determine this, the models would have to be applied to the same dataset.

Linear chain conditional random field (Linear-CRF) algorithms have also been analyzed for their effectiveness in misbehavior detection systems by [27]. This anomaly detection model aims to determine missing and false alarm data in the in-vehicle CAN communication system. The CANoe simulation software was used on an Win7 machine to generate the CAN bus network data analyzed in this report. Randomly injected anomaly data was added to this data manually by the authors. The proposed method included two models a normal model based on a Linear-CRF and an anomaly model based on Veterbi's algorithm. The detection rate when using either independent model achieved over 90% and grew as the number of states grow. Misreporting rate declined in a similar fashion. When combined the authors noticed better detection and misreporting rates overall. Ideally the authors would have compared this model to more common models to give a better representation of the increase in detection.

An analysis was performed by [28] to determine how effective various machine learning method were in misbehavior detection. The network simulation was developed using NCTUns-5.0. This simulation included 4528 samples with 1427 of the samples being malicious. The authors used multiple attacks such as packet detention, suppression, and identity fording in the testing sample. Variants of Decision trees, random forest, K-means, and naive bayes were all used to detect these attacks via a data mining software called Weka. Random forest and a variant of decision trees were shown to be the most successful at de-

tecting attacks. The authors did not split the results to show how effective each statistical method was for each attack. Instead the reader is only given the results against all attacks.

Big data technique has also been found valuable in modern misbehavior detection systems. In 2019, [6] published research on the value of distributed schemes in combination with machine learning to detect DDoS attacks in vehicular communication systems. The presented system converted a communications dataset into resilient distributed datasets (RDD) using Spark. Once the data is in the RDD, each of the databases will be used to train a decision tree. After the individual trees are trained, they are merged using the boostrap sampling method commonly used in random forests. The proposed method was trained on both the NSL-KDD [29] and UNSW-NB15 [30] datasets. NSL-KDD is derived from KDDCUP99 but still remains rather out of date. Thus UNSW-BN15, a dataset developed from the Australian Centre for Cyber Security, was also used. The resulting Random Forest algorithm was compared to SVM, gradient boosting decision trees (GBDT), XGBoost and Naive Bayes. Though both XGBoost and gradient boosting decision trees (GBDT) performed well, the authors version of boosted random forest out performed all other algorithms. Table 2.3 shows the F1-score of the top performing models on the NSL-KDD dataset [6].

Model	RF	GBDT	XGBoost
F1-Score	.965	.959	.960

Table 2.3: Top Performing F1-scores on NSL-KDD dataset

2.5 More Misbehavior Detection Systems

Non-machine learning approaches also have their place in misbehavior detection. One such example of this is Müter and Asaj's entropy-based system [31]. Much like in [25], this study focused on determining outliers in the vehicles CAN messages. Instead of including accuracy statistics of the proposed model, the paper detailed how the entropy can be used to detect common attacks. Attacks such as increased frequency, message flooding, and plausibility attacks were all studied. Unfortunately, the dataset used for this study was collected by connection straight to a production vehicles CAN network. Ideally, the authors would have included statistics toward the accuracy of this method on a publicly available dataset.

Haydari et al. presented a method of anomaly detection that was developed to detect DDoS attacks on road side unit (RSU) [32]. The method proposed is based on the online discrepancy test (ODIT), which itself is based on the Cumulative Sum (CUSUM) and Geometric Entropy Minimization (GEM). A practical version of CUSUM called Generalized Cumulative Sum (G-CUSUM) was compared to the developed model in testing. OMNET++, SUMO, and Veins were all used to generate the simulation data for the analysis of the detection algorithms. Simulation parameters are included in the paper. The authors simulated the traffic based on a street in the University of South Florida's campus while complying with IEEE 802.11p. When applying low rate DDoS attacks to this dataset, it was found that proposed method had a lower detection delay vs. false alarm probability than Generalized Cumulative Sum (G-CUSUM) based algorithms.

A modified version of the Reference Broadcast Method (RBS) was introduced by [33]

to prevent DoS attacks in vehicular communications. RBS is based on the idea of having an additional reference node that is used as an intermediary between the nodes in question. The presented algorithm called IP-CHOCK combines the RBS model with clock synchronization. The actual implementation of IP-CHOCK was conducted in C++. Detailed pseudocode was referenced in this article. The Network Simulator (NS-2.34) was used along with tool command language (tcl) to run the simulations. Simulations were completed using either 20 or 50 nodes with 2 of the nodes being attackers. When compared to an IP-trackback approach, the RBS based model produced better packet delivery ratio and throughput. The proposed model did have an increased delay time initially, but this delay decreased over time.

Greedy Geographic Routing is a GPS based method presented in [34]. This algorithm uses the GPS information to update a trust routing table (TRT). The TRT will hold a trust value and a progress value for each neighbor which can be used for misbehavior detection. This table needs to be updated and monitored regularly to indicate when ACK messages are received from the destination node. If a node packet is not from the destination or the source, it will be dropped. The authors tested this method using data that was generated using OMNeT++, SUMO, and VaCAMobil. Ideally, more descriptive statistics such as accuracy and false positives would have been presented in this paper. Without these statistics, the efficiency of the method cannot be easily compared to the other algorithms referenced.

Unlike in wired systems, vehicular communication systems are also vulnerable to jamming attacks. This form of DoS involves the use of a malicious signal used to disrupt normal communication at the radio frequency level. In other words, attackers would use artificial noise to muddy a healthy vehicular communication system. The authors in [35]

presented a novel way to detect these attacks using an system based on k-means. This method was analyzed against Interference, Smart Attack, and Constant Attack Scenarios. Though the authors' data visualizations support their claim that their unsupervised learning method was successful, they used an entirely custom simulation. The study would be much more trustworthy if they used a modern simulation tool or published their simulation code for other researchers to review.

The authors in [36] focused on the use of a received signal strength indicator (RSSI) for misbehavior detection. Their algorithm, referred to as INTERLOC, is specifically used to detect Sybil attacks during times when GPS is not available. Since the RSSI values can be collected from neighboring vehicles, INTERLOC does not require any RSUs to function. INTERLOC works by creating a circle around each participating vehicle. By matching up these circles, a localization polygon will encompass the vehicle in question. If the vehicle is not within this range it is at risk of being a Sybil attacker. A combination of OMNeT++, Veins, and SUMO were used to simulate this technique. The simulations ranged between 900 and 7200 vehicles. One vehicle was chosen at random to be the attacker. The chosen vehicle would periodically be changed over time. The results of the simulations showed that the INTERLOC algorithm was able to outperform both FRIIS and the misbehavior detection algorithm developed by [37]. Unfortunately, the authors did not provide in-depth details for the simulation software. This lack of information makes replication and validation of the study more difficult to accomplish.

Sentinel is a misbehavior detection algorithm that was proposed by [38]. This approach was specifically developed for software defined vehicular networks (SDVN) to protect against flooding attacks. The two primary steps of this method are detection and miti-

gation. The detection step looks at both the new packet and flow to determine if a flooding attack has occurred. The cut offs for misbehavior detection were determined manually by reviewing their affects on detection rate and false positive rate. This could troublesome in the fast moving world of vehicular communication networks. If the algorithm decides that an attack was found, a tree building algorithm will search through all of the primary node's neighbors to determine the mischievous node. If a malicious node is found, the algorithm will instruct the victim to drop any packets addressed from the attacker. To test the algorithm, the authors used simulations in OMNeT++, Veins, & SUMO. The algorithm was found to perform fairly well at attack mitigation in environments with under 200 vehicles but declined as the volume of vehicles increased.

Another approach for DDoS detection in vehicular communications systems was proposed by [39]. This method called multivariant stream analysis (MVSA) involves collecting statistics from the current traffic to generate rules. Each new packet will be compared against the rules developed by the algorithm. Average payload, hop count, time to live (TTL) value, and packet frequency are all used as the primary indicators in determining multivariate stream weight. Once the multivariant stream weight is calculated from the algorithm, the packet will be classified as a normal packet or an attacker. This method was tested using simulations built in Network Simulator 2.34. The DDoS attacks were conducted over the AODV routing protocol for each simulation. MVSA performed well against the baseline algorithms used in regarding to throughput, accuracy, detection time, and delivery ratio. The baseline algorithms used for this comparison are Hybrid Intrusion Detection System (H-IDS), Multi filter, and trilateral trust. Ideally, a statistical standard such as logistic regression could have been included in the baseline so that the method

could be compared more easily to other approaches.

2.6 Research Challenges

Density-based fuzzy imperialist competitive clustering algorithm (D-FICCA) is a clustering algorithm developed by [40] to detect DDoS attacks in WSM. D-FICCA utilizes a modified version of density-based spatial clustering of applications with noise (DBSCAN) and the imperialist competitive algorithm (ICA). ICA is described using a metaphor where the clusters are referred to as empires and the data points are referred to as colonies. Each of the empires will attempt to take control of a given colony. Fuzzy logic controller was built into the ICA algorithm to avoid the algorithm from assigning colonies based on poorly performing features. The data used to compare D-FICCA was collected at Intel Berkeley Research Lab via 54 different sensors. The authors did not give DDoS attack algorithm used, but stated it was create using Matlab data generation algorithms. It was determined that D-FICCA had the highest silhouette coefficient when compared to methods such as K-means, DBSCAN, and modify imperialist competitive algorithm and K-means (K-MICA). Since this algorithm was built for stationary wireless sensor networks, it still needs to be improved in vehicular communication networks due to the frequent mobility.

Distributed system applications are currently being investigated further in the field of web based misbehavior detection systems. [41] shows an example of how Hadoop and

HBase can be utilized along with neural networks for misbehavior detection. HBase is a non-relational database developed for use in a Hadoop process. Once the data is collected using Hadoop, it can be feed into the neural network to determine if a DDoS attack is occurring. Unfortunately, this method was not developed to determine individual attackers. The researchers also did not run its efficiency over a large sample. Only one example is given for a normal case, an attack, a busy traffic day, and a single factor detection approach. As vehicular traffic in major cities continues to grow, so could the demand for distributed architectures in vehicular communication systems. Studies comparing this method to the random forest based method in [6] on vehicular communications systems could prove valuable.

Additional research in distributed system based misbehavior detection systems has been conducted by Mizukoshi and colleagues. In [42] Apache Spark was combined with the use of genetic algorithms for misbehavior detection. Since attackers will often use fake IPs, an entropy based detection function was utilized. The authors compared results from both the DARPA Intrusion Detection Evaluation and KAREN's WITZ publicly available datasets. These datasets contained communication data during DDoS attacks. This study utilized 16 nodes on the Hokkaido University Academic Cloud. Increasing these nodes made a clear difference in the time required to complete the genetic algorithm. Detection rates on both DARPA's and KAREN's dataset remained over 95% with false positive and false negative rates under .02%. Unfortunately, the authors did not provide more reliable statistics such as F1-scores. Ideally, this method would have been directly compared to other statistical algorithms such as random forests, logistic regression, or naive bayes.

Traditional signal processing methods have also been applied to detection of DDoS

attacks on web servers in combination with machine learning techniques. The authors in [43] applied discrete Fourier transform (DFT) and discrete wavelet transform (DWT) to real attack data collected from SURFnet. After the transformations were applied to the training and testing datasets, the data was feed into naive bayes. By combining both the DFT and DWT, Fouladi et al. was able to achieve attack detection accuracy of 95%. When only one transformation was applied, DWT outperformed DFT in regards to its ROC curve but not in terms of misbehavior detection accuracy. Unfortunately, statistics such as F1-score, was not given by the authors. Applying DFT prior to running statistical algorithms could reduce the need for more computationally advanced methods such as neural networks in vehicular communication systems. Though, these methods would need to be thoroughly analyzed against methods using statistics such as false positives, false negatives, and F1-scores.

Though the following paper does not focus on misbehavior detection or security, [44] does show an interesting example of how graph theory and deep reinforcement theory can be used in telecommunication systems. The authors suggested system is utilized to maximize the total throughput of the vehicle to vehicle (V2V) system. There are two stages to this proposed system. The first is based on a bipartite graph of cellular users. Max N-cut is then used for channel allocation of the users. The second half of the system is a deep reinforcement learning (DRL) system that is used for power control. The proposed resource allocation method shows promising results in comparison to the random and hybrid models. Simulation parameters are included in this paper for replication purposes. As indicated by the authors the training time is a concern for this method, but it was not an issue during execution.

Chapter 3

Misbehavior Detection of DoS Attacks in Vehicular Networks

3.1 MDS Architecture

Previous researchers have been successful in developing detection methods for DoS attacks using machine learning methods, but the issue of computation cost is not frequently discussed. To improve this, the misbehavior detection system could incorporate distributed system technology to reduce the cost of detecting on each individual vehicle. One such technology that solves this issue is Apache Spark. Apache Spark is an open-source distributed system technology that is designed for use with common computational languages such as Python, R, and Scala. The primary function of this tool is to force parallel operations on a cluster. In the case of a vehicular communication system, this cluster is made up of a group of nearby vehicles. At this point, the communication data is fed into a neural network to determine if a DoS attack is occurring.

3.2 Distributed Systems and Computation

A distributed system is a group of machines that are working together to appear as a single system by an end user. For use in vehicular communication systems, this is achieved by considering each vehicle as a machine in the system. Each of these vehicles coordinates with each other to develop and use the neural network model. This system should reduce the total resource allocation for each vehicle and increase the speed of misbehavior detection.

Fig. 3.1 shows an example of how this architecture works at a high level. One vehicle monitors the safety of the specified location. This vehicle then makes contact with the neighboring vehicles to collaborate in misbehavior detection. The worker vehicles are only to be used as additional computational resources. Each role should be rotated between neighboring vehicles for added security.

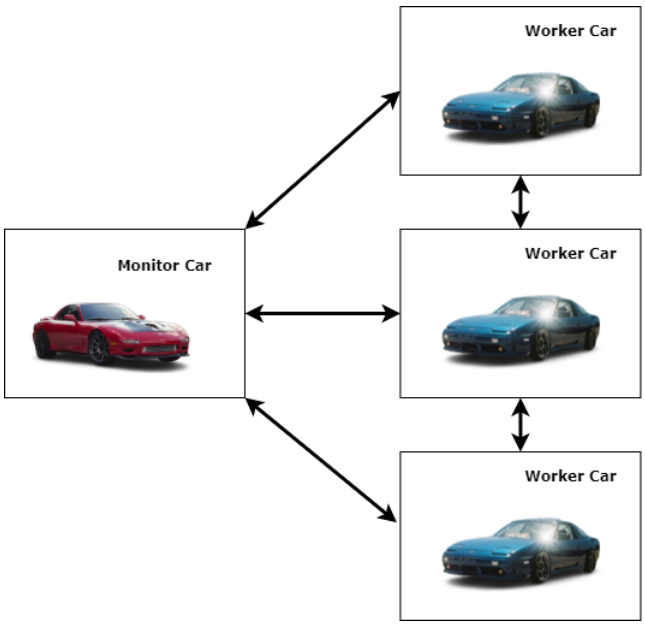

Figure 3.1: Vehicular Distributed System Example

To achieve a distributed architecture, Apache Spark is used in the proposed system. Spark functions by splitting the data into a type of dataset referred to as a resilient distributed dataset. The RDD structure allows adjustments to the data to be performed in parallel. The parallel computation gives the monitoring car the ability to use the workers to speed up the misbehavior detection process. The monitoring car runs the driver program used to utilize the resources on the other vehicles. Inside of the driver program, an object called ”SparkContext” connects to the cluster managers. A resource manager is a tool that determines the resource allocation for each node in the cluster. In this solution, Spark’s own cluster managers were used. A visual representation of the cluster operation is shown in Fig. 3.2.

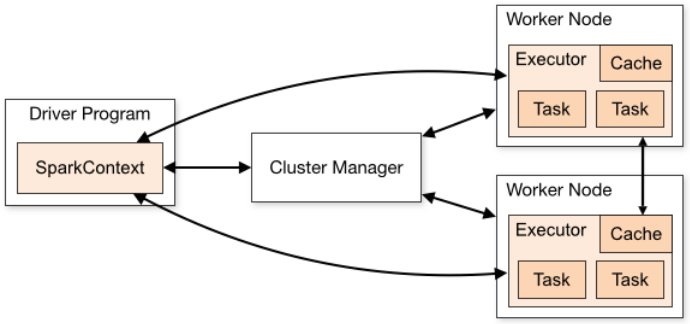

Figure 3.2: Cluster Operation Diagram [4]

3.3 Introduction to Neural Networks

A neural network is an algorithm that is heavily influenced by the operation of neurons in the brain. The goal for the initial model development was to create a classifier that could work as effectively as the human brain. The single layer perceptron was developed to simulate a neurons operation in the brain. By chaining these perceptrons, researchers discovered that could create a network that closely resembles operations of a human brain. Section 3.3.1 details the design of a single layer perceptron and Section 3.3.2 shows how these perceptrons can be used to develop a neural network.

3.3.1 Single Layer Perceptron Classifier

A single layer perceptron is a binary classification algorithm designed to mimic a single human neuron. The algorithm begins by multiplying each input by the weight associated with the input value. A visual representation of this process can is shown in Fig. 3.3. For this example, X is written as a given input and W is used to signify the weight value.

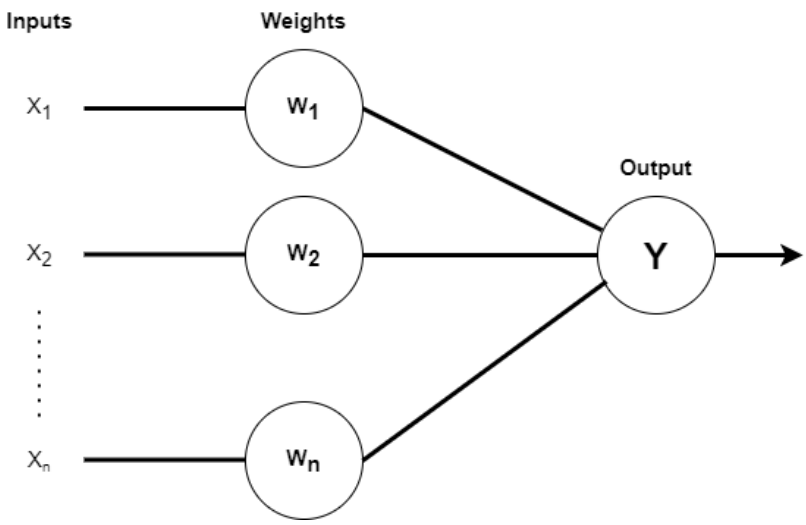

Figure 3.3: Single Layer Perceptron

All of the multiplicative values determined in the last step must then be added together, resulting in the following equation:

$$z = \sum_{i=1}^{n} W_i * X_i \tag{3.1}$$

Once the output z is determined, a comparison against the threshold value θ can be conducted. This can be shown mathematically as:

$$f(x) = \begin{cases} 1, & \text{if } z > \theta \\ 0, & \text{otherwise} \end{cases} \tag{3.2}$$

At this step the algorithm has now made its predictions on the dataset given.

3.3.2 Multilayer Perceptron Classifier

By grouping together multiple single layer perceptrons, researchers determined that they could create a model with a higher predictive output. This new model is often refered to as a multilayer perceptron classifier (MLPC) or a feed forward neural network. This architecture contains a series of layers. Each of these layers are essentially a column of single layer perceptrons. The first of these layers is an input layer that must be the same size as the number of features used in the given dataset. The layer often called the output layer contains just enough perceptrons to achieve the required classification. The middle layers, often referred to as ”hidden layers” are much more nebulous in nature. Unlike the input and output layers, there is no strict formula for design. These layers are often determined by trail and error or by a brute force algorithm. Fig. 3.4 contains a simple visual example of the MLPC architecture.

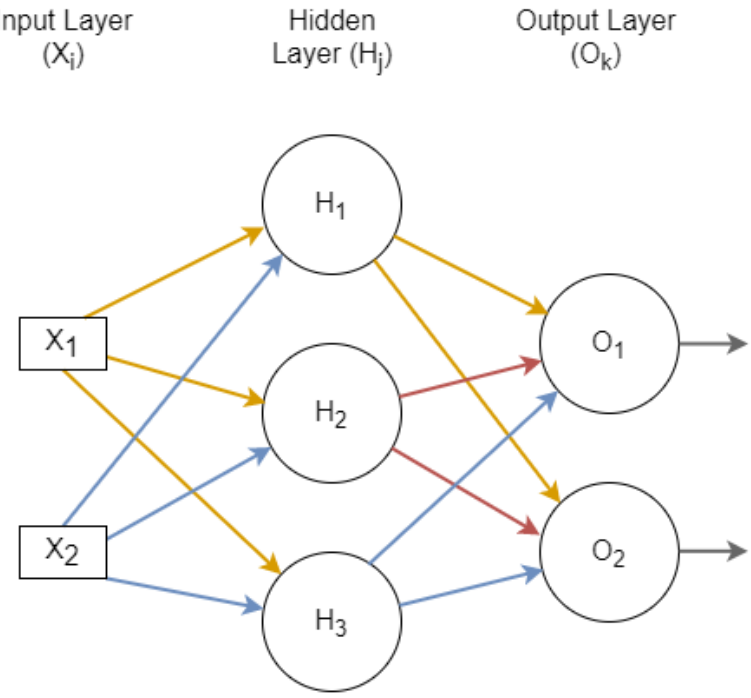

Figure 3.4: Multilayer Perceptron

Mathematically speaking, the MLPC is just a more complex version of the single layer perceptron. Recall equation 3.1 and 3.2 from section 3.3.1, this formula is to be applied at each layer of the MLPC with X_i being the previous output. For example to calculate H_j, the following equation can be used:

$$H_j = f(\sum_{i=1}^{n} W_{ji} * X_i) \tag{3.3}$$

To find O_k, the weight values need to be updated and H_j is used instead of X_i. With these adjustment, the following equation is derived:

$$O_j = f(\sum_{j=1}^{n} W_{kj} * H_j) \tag{3.4}$$

3.4 L-BFGS Optimization Routine

The Limited-memory Broyden–Fletcher–Goldfarb–Shanno algorithm (L-BFGS) was selected as the optimization routine for the proposed model. This algorithm is a derivation of Broyden-Fletcher-Goldfarb-Shanno that is optimized for larger datasets. Two different algorithms are used to explain the operation of this method. Algorithm 1 is a recursive algorithm that is used to determine $H_k \nabla f_k$, where H_k represents the inverse Hessian approximation value and ∇f_k is the gradient [45]. It is important to know that the use of an approximate Hessian instead of a true Hessian is the reason that L-BFGS is a Quasi-Newton method, not a true Newton Method. The determined $H_k \nabla f_k$ value is used to determine the search direction p_k in Algorithm 2.

Before diving into the nuance of L-BFGS, please understand that s_k refers to the displacement and y_k refers to the change in gradients. In Algorithm 1 there are two for loops that are used to update the final Hessian matrix. The first loop calculates the current gradient q and the step length α_i. One of the variables used to determine α_i is ρ_k, which came from the Davidon-Fletcher-Powell formula. ρ_k is calculated using the following formula:

$$\rho_k = \frac{1}{y_k^T s_k} \tag{3.5}$$

Prior to starting the second loop, the new q is multiplied by a initial inverse Hessian matrix. Normally the initial inverse Hessian is found using the following formula, where I is an initial Hessian approximation:

$$H_k^0 = (\frac{s_k^T y_{k-1}}{y_k^T y_{k-1}})I \tag{3.6}$$

The matrix found by multiplying q and H_k^0 is referred to as r. r is then repeatedly updated using the derived β value on line 8 and the α_i value derived on line 3. Once the r value is equivalent to $H_k \nabla f_k$, there is no need to continue and the algorithm stops.

Algorithm 1: L-BFGS Two-Loop Recursion to Compute $H_k \nabla f_k$

1 $q = \nabla f_k$

2 **for** i = k - *1*, k - *2*,..., k - m **do**

3 $\quad \alpha_i = \rho_i s_i^T q$

4 $\quad q = q - \alpha_i y_i$

5 **end**

6 $r = H_k^0 q$

7 **for** i = k - m, k - m + *1*,..., k - *1* **do**

8 $\quad \beta = \rho_i y_i^T r$

9 $\quad r = r + s_i(\alpha_i - \beta)$

10 **end**

11 **stop** when $H_k \nabla f_k = r$

The full computation of L-BFGS is shown in Algorithm 2. To start the algorithm must have a starting estimated optimal value x_0, memory m greater than zero, and an initial inverse Hessian H_k^0. The same method for determining the initial inverse Hessian matrix for Algorithm 1 is applied here. At this point, the algorithm can calculate the search direction p_k and update x_{k+1}. Notice that to update x_{k+1}, the step length α_i must satisfy Wolfe conditions. The Wolfe condition is used to verify that a_k gives a reasonable decrease to the objective function f. The Wolfe condition can written as:

$$f(x_k + \alpha p_k) \leq f(x_k) + c_1 \alpha \nabla f_k^T p_k \tag{3.7}$$

z Please note that c is used as a constant between zero and one. At this point the algorithm will remove the vector pair $\{s_{k-m}, y_{k-m}\}$ if k is large than the memory m. Recall that s_k refers to the displacement and y_k refers to the change in gradients. Otherwise, the new values for s_k are calculated. This process repeats until the algorithm converges.

Algorithm 2: L-BFGS

Choose starting point x_0

Set integer $m > 0$

Choose H_k^0

repeat

  $p_k = -H_k \nabla f_k$ from Algorithm 1

  $x_{k+1} = x_k + \alpha_k p_k$ where α_k is used to satisfy Wolfe Conditions

  if $k > m$ **then**

    Discard vector pair $\{s_{k-m}, y_{k-m}\}$

  end

  Compute vector pair $s_k = x_{k+1} - x_k, y_k = \nabla f_{k+1} - \nabla f_k$

  $k = k + 1$

until *algorithm converges*

Chapter 4

Performance Evaluations and Results Discussions

4.1 OMNeT++ Overview

In chapter 2, NSL-KDD, UNSW-NB15, NS2, and OMNeT++ were all introduced as viable options for vehicular communication systems performance evaluations. Though all of these datasets and tools can be used to evaluated MDS, OMNeT++ was used as the backbone of the simulations discussed in this chapter. OMNeT++ is designed to handle advanced communication simulations. To enable these simulations to include mobile vehicles, Veins and SUMO must be used. SUMO is a traffic simulation software that generates the normal vehicles and all for vehicular mobility. Veins is then used as glue to allow OMNeT++ and SUMO to communicate in order to generate vehicular communication systems. The OMNeT++ simulations developed in this thesis utilize 1-hop broadcasting between the vehicles and the roadside units. Therefor, each message will be distributed to all nodes in

the range of the sender.

4.2 Simulation Settings and Background Information

All vehicular network simulations were developed on an Ubuntu Virtual Machine. Upon installation, Veins is equipped with a built in map and mobile vehicular communication system simulation. This is a general simulation in which 194 vehicles are all driving in the same direction when an accident suddenly occurs 73 seconds into the simulation. This accident lasts 50 seconds in total. This causes the vehicles to react and alert the other vehicles of the upcoming traffic jam. The total simulation lasts 200 seconds and contains a single road side unit. This simulation was used as a starting point for the DDoS simulation.

A series of modifications were made to the default simulation to fit the requirements of this study. The simulation time was increased to 380 seconds. A simulation was included for each of the following five vehicle amounts: 15, 20, 25, 30, and 35. In addition to the vehicles previously described, four parked vehicles were included to preform the DDoS attacks on the remaining vehicles and road side unit. Though these vehicles were utilized in the attack, they would otherwise be considered the same as the mobile vehicles. These vehicles were used as zombies by the attacker, which means that they communicated normally outside of the attack. There was no change to the accident that was included in the default program at 73 seconds. Fig. 4.1 is a screen capture of a simulation taking place in OMNeT++. In this image, the attackers are labeled as ”hacker” and the normal vehicles are labeled as ”node”.

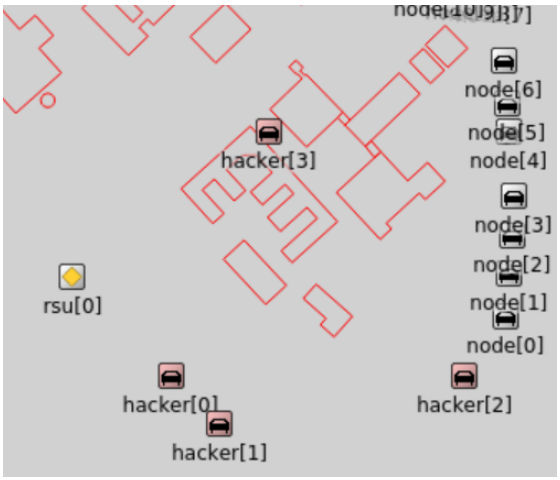

Figure 4.1: Attack Simulation View in OMNeT++

Seven different versions of each attack simulation were developed to determine how the MDS would handle different attack densities. The first simulation started at an attack density of 10%. This means that an attack occurred for 10% of the total simulation time. Attack density was then increase by 10% for each of the following simulations. Table 4.1 shows the time that each attack occurs during each simulation. For example, those that occurred at the 10% attack density, ran from 50 seconds to 74 seconds and 210 seconds to 224 seconds. The first attack started at 50 seconds for each attack density. The second attack started at 210 seconds for each density, except for 70%, where it started at 174 seconds. During this time period, 25,000 WAVE short messages (WSM) were sent to every vehicle. The code used to generate the WSM is based on the communication code found in traCIDemo11p.cc of the Veins demo previously described. The malicious changes were made in the handlePositionUpdate() method. Algorithm 3 describes how the attack operates. This algorithm repeatedly creates WSM that contain the vehicles' current road id that are then distributed to all nearby vehicles. All code used for these simulations can be

found in Appendix A.

Attack %	**1st Attack Duration (s)**	**2nd Attack Duration (s)**
10	50 - 74	210 - 224
20	50 - 74	210 - 262
30	50 - 112	210 - 262
40	50 - 150	210 - 262
50	50 - 150	210 - 300
60	50 - 150	210 - 337
70	50 - 150	174 - 340

Table 4.1: Attack Times used in Simulations

Algorithm 3: 10% Attack Density Simulation

Result: Perform DDoS Attack

```
1  if (simulation time between 50 & 74) && (simulation time between 210 & 224)
     then
2      for i:Range 1 to 25,000 do
3          sentMessage = true
4          wsm = new TraCI Demo Message
5          populate wsm
6          set wsm data to road id
7          send(wsm)
8      end
9  else
10     time last drove = simulation time;
```

4.2.1 Data Preparation

By default, the Veins simulation in OMNeT++ collected each vehicles x coordinate, y coordinate, speed, acceleration, and CO2 emissions. A new data point for each of the previously mentioned factors was collected during each second of the simulation. This data was than exported to an excel file so it could be cleaned and analyzed in a Jupyter Notebook. Before the data could be used to train models, all periods and number signs were removed from the column names. The time column generated by OMNeT++ was changed from ”t” to ”Time in Seconds” for readability. Than an ”if then” command was used in excel to create a column labeling the attack times. If an attack occured during this

time, the excel formula would output a 1. Otherwise, the result would be a 0. This column was named ”Attack Bool” for all simulations. Each simulation was 380 rows. The column size differed based on the number of vehicles used. The total columns for each simulation can be found by multiplying the number of cars by five and then adding two columns for the time and attack label.

The cleaned csv was then loaded into a Jupyter Notebook using Spark’s read method. Before the data could be used in PySpark.ML models, all of the columns were converted to floats. Additionally, the ”Attack Bool” column was changed ”Label” to match PySpark coding norms. The last step prior to breaking the data into testing and training datasets was to create a features vector. All of the models built in PySpark.ML used a vector representation of the data instead of multiple columns like other common libraries. This conversion was done using the vectorAssembler class and the transform function in PySpark.ML.

4.3 MLPC Architecture Development

The MLPC architecture was developed to handle all previously mentioned attack percentages. All available features were used in model development. For example, the simulation that contained 15 vehicles gave 76 features that were fed into the machine learning model. The first layer of the MLPC must be the same size as the features being fed. Since the model is attempting to determine if a DDoS attack is occurring, the result is boolean. Hence, two is used as the value of the final layer. Through trial and error, it was determined that five layers seemed to outperform smaller architectures. This testing also determined that ranges just above the feature number worked well in the second layer. Further testing also indi-

cated that smaller values performed best for the third and fourth layers. To narrow in on possible high-performing architectures, Algorithm 4 was ran on all simulations.

Algorithm 4: Determine candidate layers via brute force methods

Result: Print F1-score of all tested layer combinations

```
for i in Range 80 to 100 do
    for j in Range 5 to 10 do
        for k in Range 2 to 10 do
            layers = [76,i,j,k,2]
            Design model using layers
            Fit model using train_data
            Get predictions for test_data
            Determine F1-score using correct testing labels
            if F1-score > .95 then
                print layers & F1-score
        end
    end
end
```

The high performing results were then collected and moved into an excel report. If a layer combination appeared multiple times, it was then ran on all simulations. This was done to produce a model that would be universally valuable instead of being only effective for a single attack density. The mean and median F1-scores were used to determine the best fit model. The MLPC architecture that had the highest mean and median F1-scores

was [N, 87, 9, 4, 2], where N is the number of features given. In this architecture, 76 is referring to the input layer and two is the output layer. All columns of the dataset were used as features with the exception of the labels. The middle values of 87, 9, and four are the hidden layers of the neural network. The F1-score average for this layer design was 91.5% and the median was 95.9%.

As mentioned in section 3.4, L-BFGS was used for the optimization routine. L-BFGS was chosen over minibatch gradient descent due to the latter's frequent underperformance on the training data. PySpark.ML's MLPC is trained using backpropagation, which is a fairly standard training algorithm for neural networks. PySpark.ML uses logistic loss function for optimization.

4.4 Simulation Results

4.4.1 Model Comparison During Various Attack Densities

To test how this architecture compares to other common machine learning models, the design was implemented in a Jupyter Notebook on an Asus Zenbook. This machine contained a Intel core i7-8565U CPU and 16 GB of RAM. The MLPC model was compared against logistic regression (LR), random forest (RF), gradient boosted trees (GBT), and support vector machines (SVM). A 70/30 train test split was used for all models. Additionally, 100 training iterations were used for all models with the exception of RF. PySpark.ML's RF does not give the option to adjust the iteration value. RF was set to a max tree depth of 3. All of the models used were built into the PySpark.ML module. PySaprk.ML is a

machine library produced by Apache Spark. This library was developed specifically for use on Spark clusters. Accuracy, precision, recall, and F1-score are used to compare these models. The following formulas show how these statistics are calculated.

To find the accuracy of a model, the sum of the true positive (TP) and true negative (TN) values must be divided by all possible outcomes. In the following formulas, FP stands for false positive and FN stands for false negative.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{4.1}$$

For precision, the TP is divided by the summation of the TP and the FP.

$$Precision = \frac{TP}{TP + FP} \tag{4.2}$$

To determine the recall of a model, the TP must be divided by the summation of the models TP and FN.

$$Recall = \frac{TP}{TP + FN} \tag{4.3}$$

The F1-score is a weighted average of the precision and recall. This is calculated by multiplying the precision by the recall, then dividing by the summation of the same values. The previously derived value is then multiplied by 2 to calculate the F1-score.

$$F1 - score = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{4.4}$$

The implementation of these functions in PySpark.ML was used for consistency. Appendix B contains the code used to generate the accuracy, precision, recall, and F1-score for each model. Additionally, these statistics were collected for each of the five simulation

designs in respect to the number of vehicles. The average of each metric was used. Fig. 4.2 shows the accuracy of each model at different attack densities. Without further graphs, we can see the weak point of the MLPC model. By selecting the median F1-score across all attack densities, the overall architecture was not as effective at each density. It can be seen that RF and GBT both produced higher accuracy scores on all attack densities. The MLPC outperformed the LR and SVM models with the exception of the 40% attack density.

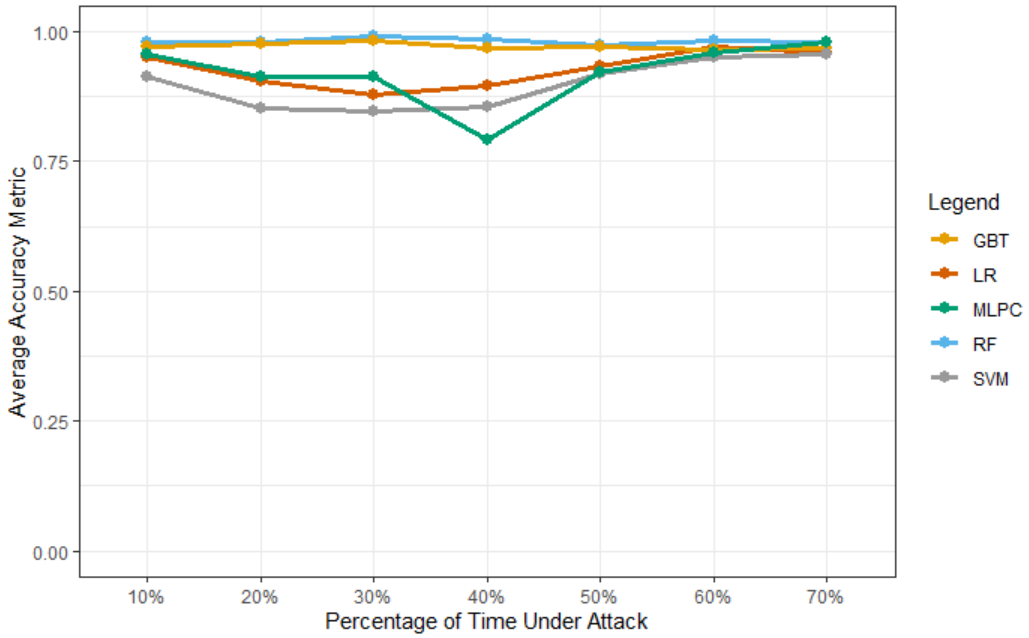

Figure 4.2: Model Accuracy

The precision of each model followed a similar trend to the accuracy. GBT and RF outperformed the other models in all attack densities with the exception of 60% and 70% attack densities for GBT. The MLPC performed comparably to the trained regression model. This was partially due to the MLPC producing low scores at 40% regardless of the number of vehicles used in the simulation. SVM generally did not perform at the same level as the other algorithms. Fig. 4.3 shows the precision value for each model at various attack densities.

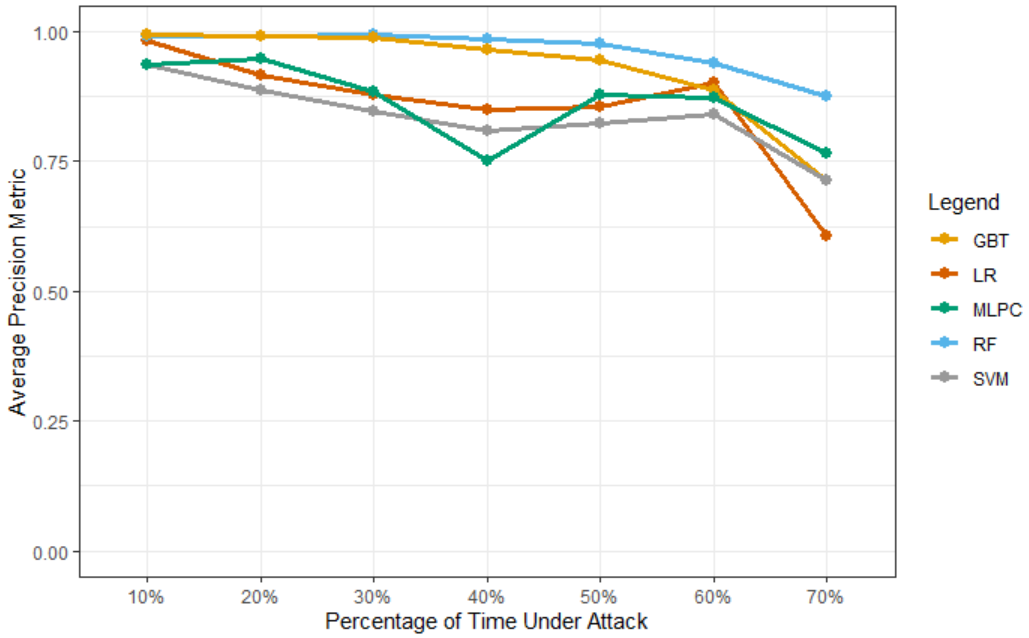

Figure 4.3: Model Precision

Recall was consistent across all models except the MLPC. RF, GBT, and MLPC all performed well until the MLPC dropped at the 40% attack density simulation. Interestingly, all of the models had much higher recall at 60% attack density than the 50% attack density. The only exception to this was GBT, which dropped slightly before increasing at 70%. LR performed fairly well, often better than the MLPC. SVM followed a similar trend to LR, but producing weaker results. Fig. 4.4 shows the recall value for each model at various attack densities.

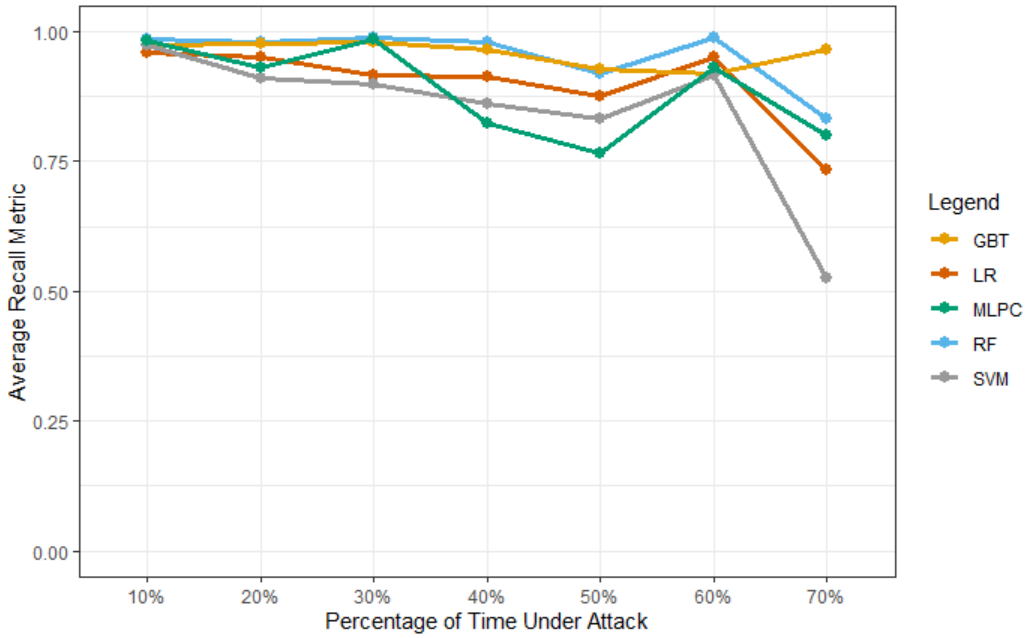

Figure 4.4: Model Recall

RF and GBT both produced very high F1-scores across all simulations. MLPC produced equivalent or lower metrics than RF and GBT. The MLPC had a lower score at a 40% attack density than the other models analyzed. Both the LR and SVM F1-scores were greater than the proposed MLPC at 40% attack density, but otherwise had lower scores. These results are shown in detail in Fig. 4.5.

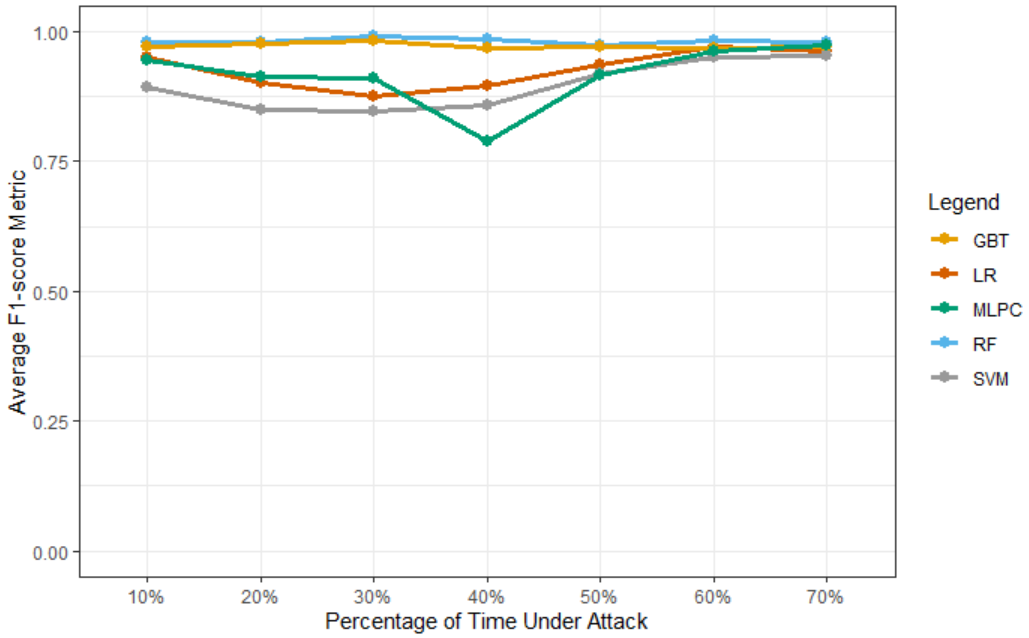

Figure 4.5: Model F1-score

4.4.2 Training and Detection Time Using Multiple Clusters

Since PySpark is based off a distributed technology, it is important to see if any value is gained from spreading the computation onto multiple vehicles. To simulate this process, the 50% attack percentage simulation was stored on in an Amazon S3 Bucket in Amazon Web Services (AWS). Amazon Elastic MapReduce (EMR) is a tool used to set up servers with multiple nodes that already contain all the schemes needed to run a PySpark cluster. AWS also contains additional support for Jupyter Notebooks to run on an EMR spark cluster. These clusters are designed to have a single master node and additional core nodes to spread the computational overhead. Fig. 4.6 shows the software configuration on each EMR instance. For this analysis, each EMR node (including the master node) was set up using the m5.large type in AWS EMR. This server contained a 4vCore, 16 Gib memory,

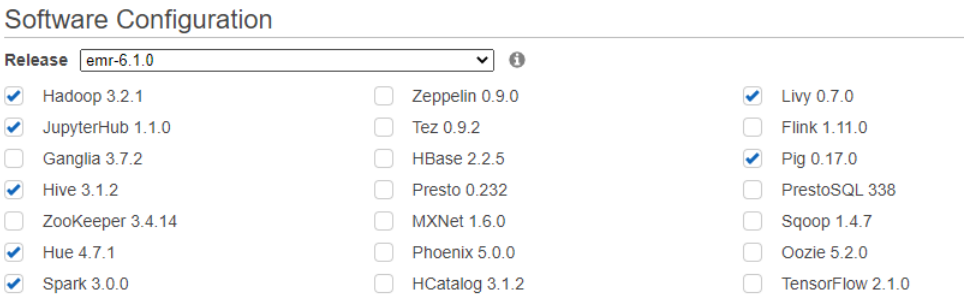

Figure 4.6: EMR Software Selection

and 64 Gib of storage. All logging was stored in an additional AWS S3 bucket.

To determine the performance difference of each instance configuration, both training and prediction was timed running on 6 different AWS instances ranging from a single master node to 6 nodes in total. The simulation including 35 cars was used for this analysis. The Jupyter Notebook containing the algorithm was restarted and ran 10 different times for each instance. Fig. 4.7 shows the time required to run training and utilize the algorithm at each node count. Adding additional nodes did not prove to have a large impact on the overall computation time. The median time did decrease marginally as nodes were included. The code used for this analysis can be found in Appendix C.

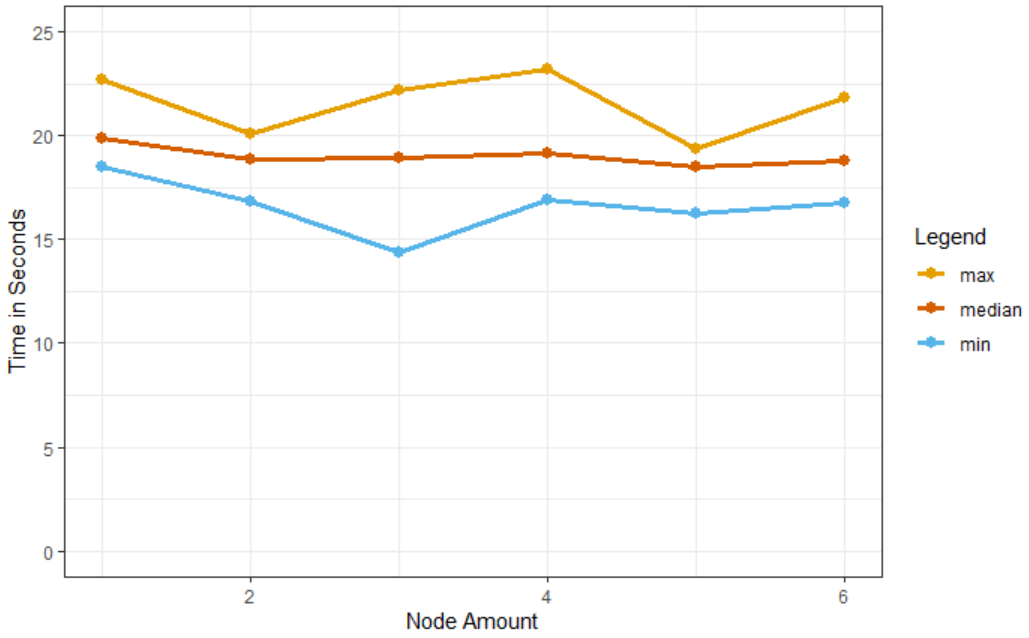

Figure 4.7: MPLC Computation Speed vs Node Amount

Chapter 5

Conclusions and Future Work

Misbehavior detection systems is a paramount tool for the continued development of vehicular communication networks. Without the implementation of a MDS, all vehicles connected to the network would be vulnerable to attacks such as the distributed denial of service attack. Such attacks can block communication channels by flooding the network with communication packets, causing confusion and potentially accidents. Machine learning methods have proved to be effective in MDS, with recent efforts focused on methods such as neural networks and distributed computing. In particular, publications such as [7], [1], and [25] showed promising results for the use of neural networks in MDS. Both [6] and [42] showed that distributed computing can greatly reduce the time required in a MDS. In time critical systems such as a MDS, a tenth of a second in detection time could be all that is needed to prevent an accident.

The logical next step was to investigate neural networks and distributed computing methods in combination to see if they would continue to be performant on modern mobile network simulators. This thesis focused on designing a multilayer perceptron classifier,

also known as a feed forward neural network on vehicular communication simulations built in OMNeT++, Veins, and SUMO. These simulations contained the distributed denial of service (DDoS) attack conducted via the use of a few parked zombie vehicles. The proposed MDS was built to utilize the Apache Spark distributed computing framework for data processing and attack detection. By using this framework, we were able to analyze the benefit of spreading the resource demand of model training and prediction against multiple nodes. Amazon Web Services was used to run the Spark clusters, each Elastic MapReduce (EMR) node was meant to simulate a nearby vehicle.

By only collecting x coordinate, y coordinate, speed, acceleration, and CO2 emissions of each vehicle, RF was able to predict an DDoS attack with very high accuracy. If vehicles are able to monitor and communicate this information in real time, it may not be necessary to use big data tools. The exception to this may be large scale cities such as Seattle or Denver that contain large amounts of vehicular traffic. It may be valuable to compare the results on a large city wide simulation against a smaller simulation such as the one presented in this research. Further investigation into the dataset such as DARPA and WITZ a dataset with the part of the study of KAREN (Kiwi Advanced Research and Education Network) should be considered before the widespread use of these methods.

Though a multilayer perceptron classifier (MLPC) is a powerful predictive algorithm, it is evident that it may not be the most effective algorithm for application in misbehavior detection in vehicular communication systems. The simulation generated from OMNeT++ only contained up to vehicles and ran for 380 seconds. From this small amount of data, random forest (RF) and gradient boosted trees (GBT) were able to produce reasonable accuracy, recall, precision, and F1-scores. In future studies, it may be worth investigating

statistical learning methods that are less computationally expensive than RF and GBT. This could help increase the speed of misbehavior detection and reduce the time delay on distributed computing applications.

Bibliography

[1] K. M. Ali Alheeti, A. Gruebler, and K. D. McDonald-Maier, "On the detection of grey hole and rushing attacks in self-driving vehicular networks," in *2015 7th Computer Science and Electronic Engineering Conference (CEEC)*, pp. 231–236, 2015.

[2] C. Deets, "White sedan parked beside mountain during daytime," Jun 2019.

[3] E. Flores, "Red audi coupe on road near trees at daytime," Jun 2018.

[4] "Cluster mode overview."

[5] "State highway travel."

[6] Y. Gao, H. Wu, B. Song, Y. Jin, X. Luo, and X. Zeng, "A distributed network intrusion detection system for distributed denial of service attacks in vehicular ad hoc network," *IEEE Access*, vol. 7, pp. 154560–154571, 2019.

[7] F. A. Ghaleb, A. Zainal, M. A. Rassam, and F. Mohammed, "An effective misbehavior detection model using artificial neural network for vehicular ad hoc network applications," in *2017 IEEE Conference on Application, Information and Network Security (AINS)*, pp. 13–18, 2017.

[8] F. Qu, Z. Wu, F. Wang, and W. Cho, "A security and privacy review of vanets," *IEEE Transactions on Intelligent Transportation Systems*, vol. 16, no. 6, pp. 2985–2996, 2015.

[9] J. Kamel, I. Ben Jemaa, A. Kaiser, and P. Urien, "Misbehavior reporting protocol for c-its," in *2018 IEEE Vehicular Networking Conference (VNC)*, pp. 1–4, 2018.

[10] S. Chadli, M. Emharraf, M. Saber, and A. Ziyyat, "The design of an IDS architecture for manet based on multi-agent," in *2014 Third IEEE International Colloquium in Information Science and Technology (CIST)*, pp. 122–128, 2014.

[11] R. W. van der Heijden, T. Lukaseder, and F. Kargl, "VeReMi: A dataset for comparable evaluation of misbehavior detection in vanets," in *SecureComm*, 2018.

[12] M. Behrisch, L. Bieker-Walz, J. Erdmann, and D. Krajzewicz, "Sumo – simulation of urban mobility: An overview," vol. 2011, 10 2011.

[13] A. Gruebler, K. D. McDonald-Maier, and K. M. Ali Alheeti, "An intrusion detection system against black hole attacks on the communication network of self-driving cars," in *2015 Sixth International Conference on Emerging Security Technologies (EST)*, pp. 86–91, 2015.

[14] T. K. Mohd, S. Majumdar, A. Mathur, and A. Y. Javaid, "Simulation and analysis of ddos attack on connected autonomous vehicular network using omnet++," in *2018 9th IEEE Annual Ubiquitous Computing, Electronics Mobile Communication Conference (UEMCON)*, pp. 502–508, 2018.

[15] R. Kaur, A. L. Sangal, and K. Kumar, "Modeling and simulation of ddos attack using omnet++," in *2014 International Conference on Signal Processing and Integrated Networks (SPIN)*, pp. 220–225, 2014.

[16] S. Gyawali, Y. Qian, and R. Q. Hu, "Machine learning and reputation based misbehavior detection in vehicular communication networks," *IEEE Transactions on Vehicular Technology*, vol. Vol.69, pp. 8871–8885, August 2020.

[17] K. Zaidi, M. B. Milojevic, V. Rakocevic, A. Nallanathan, and 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, 2016.

[18] M. Malekzadeh, A. Ghani, S. Shamala, and J. Desa, "Validating reliability of omnet++ in wireless networks dos attacks: Simulation vs. testbed," *Int. J. Netw. Secur.*, vol. 13, pp. 13–21, 2011.

[19] F. Ahmad, A. Adnane, V. Franqueira, F. Kurugollu, and L. Liu, "Man-in-the-middle attacks in vehicular ad-hoc networks: Evaluating the impact of attackers' strategies," vol. 11, 2018.

[20] P. Muhlethaler, A. Laouiti, and Y. Toor, "Comparison of flooding techniques for safety applications in vanets," in *2007 7th International Conference on ITS Telecommunications*, pp. 1–6, 2007.

[21] S. Gyawali and Y. Qian, "Misbehavior detection using machine learning in vehicular communication networks," in *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*, pp. 1–6, 2019.

[22] O. A. Wahab, A. Mourad, H. Otrok, and J. Bentahar, "Ceap: Svm-based intelligent detection model for clustered vehicular ad hoc networks," *Expert Systems with Applications*, vol. 50, p. 40–54, 2016.

[23] L. Nie, Y. Li, and X. Kong, "Spatio-temporal network traffic estimation and anomaly detection based on convolutional neural network in vehicular ad-hoc networks," *IEEE Access*, vol. 6, pp. 40168–40176, 2018.

[24] "Next generation simulation (ngsim) vehicle trajectories and supporting data: Department of transportation - data portal," Aug 2018.

[25] M. Kang and J. Kang, "A novel intrusion detection method using deep neural network for in-vehicle network security," in *2016 IEEE 83rd Vehicular Technology Conference (VTC Spring)*, pp. 1–5, 2016.

[26] N. Ding, H. Ma, C. Zhao, Y. Ma, and H. Ge, "Driver's emotional state-based data anomaly detection for vehicular ad hoc networks," in *2019 IEEE International Conference on Smart Internet of Things (SmartIoT)*, pp. 121–126, 2019.

[27] T. Kai, L. Zhongwei, J. Wenqi, G. Yadong, and T. Weiming, "In-vehicle can bus anomaly detection algorithm based on linear chain condition random field," in *2019 IEEE 19th International Conference on Communication Technology (ICCT)*, pp. 1153–1159, 2019.

[28] J. Grover, N. K. Prajapati, V. Laxmi, and M. S. Gaur, "Machine learning approach for multiple misbehavior detection in vanet," Springer, Berlin, Heidelberg, Jul 2011.

[29] M. Tavallaee, E. Bagheri, W. Lu, and A. A. Ghorbani, "A detailed analysis of the kdd cup 99 data set," in *2009 IEEE Symposium on Computational Intelligence for Security and Defense Applications*, pp. 1–6, 2009.

[30] N. Moustafa and J. Slay, "Unsw-nb15: a comprehensive data set for network intrusion detection systems (unsw-nb15 network data set)," in *2015 Military Communications and Information Systems Conference (MilCIS)*, pp. 1–6, 2015.

[31] M. Müter and N. Asaj, "Entropy-based anomaly detection for in-vehicle networks," in *2011 IEEE Intelligent Vehicles Symposium (IV)*, pp. 1110–1115, 2011.

[32] A. Haydari and Y. Yilmaz, "Real-time detection and mitigation of ddos attacks in intelligent transportation systems," in *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*, pp. 157–163, 2018.

[33] K. Verma, H. Hasbullah, and H. K. Saini, "Reference broadcast synchronization-based prevention to dos attacks in vanet," in *2014 Seventh International Conference on Contemporary Computing (IC3)*, pp. 270–275, 2014.

[34] H. Almutairi, S. Chelloug, H. Alqarni, R. Aljaber, A. Alshehri, and D. Alotaish, "A new black hole detection scheme for vanets," in *Proceedings of the 6th International Conference on Management of Emergent Digital EcoSystems*, MEDES '14, (New York, NY, USA), p. 133–138, Association for Computing Machinery, 2014.

[35] D. Karagiannis and A. Argyriou, "Jamming attack detection in a pair of rf communicating vehicles using unsupervised machine learning," Elsevier, May 2018.

[36] M. T. Garip, P. H. Kim, P. Reiher, and M. Gerla, "Interloc: An interference-aware rssi-based localization and sybil attack detection mechanism for vehicular ad hoc networks," in *2017 14th IEEE Annual Consumer Communications Networking Conference (CCNC)*, pp. 1–6, 2017.

[37] B. Xiao, B. Yu, and C. Gao, "Detection and localization of sybil nodes in vanets," *Proceedings of the 2006 workshop on Dependability issues in wireless ad hoc networks and sensor networks - DIWANS 06*, 2006.

[38] G. de Biasi, L. F. M. Vieira, and A. A. F. Loureiro, "Sentinel: Defense mechanism against ddos flooding attack in software defined vehicular network," in *2018 IEEE International Conference on Communications (ICC)*, pp. 1–6, 2018.

[39] R. Kolandaisamy, R. M. Noor, I. Ahmedy, I. Ahmad, M. R. Z'Aba, M. Imran, and M. Alnuem, "A multivariant stream analysis approach to detect and mitigate ddos attacks in vehicular ad hoc networks," *Wireless Communications and Mobile Computing*, vol. 2018, p. 1–13, 2018.

[40] S. Shamshirband, A. Amini, N. B. Anuar, M. L. Mat Kiah, Y. W. Teh, and S. Furnell, "D-ficca: A density-based fuzzy imperialist competitive clustering algorithm for intrusion detection in wireless sensor networks," *Measurement*, vol. 55, pp. 212–226, 2014.

[41] T. Zhao, D. C. Lo, and K. Qian, "A neural-network based ddos detection system using hadoop and hbase," in *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium*

on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems, pp. 1326–1331, 2015.

[42] M. Mizukoshi and M. Munetomo, "Distributed denial of services attack protection system with genetic algorithms on hadoop cluster computing framework," in *2015 IEEE Congress on Evolutionary Computation (CEC)*, pp. 1575–1580, 2015.

[43] R. F. Fouladi, C. E. Kayatas, and E. Anarim, "Frequency based ddos attack detection approach using naive bayes classification," in *2016 39th International Conference on Telecommunications and Signal Processing (TSP)*, pp. 104–107, 2016.

[44] S. Gyawali, Y. Qian, and R. Q. Hu, "Resource allocation in vehicular communications using graph and deep reinforcement learning," in *2019 IEEE Global Communications Conference (GLOBECOM)*, pp. 1–6, 2019.

[45] J. Nocedal and S. J. Wright, *Numerical optimization*. Springer, 2006.

Appendix A

OMNeT++ Simulation Code

The following sections show the modification to Veins used to generate all DDoS simulations used in this research. The following sections contain .ini, .ned, .h, and .cc files. Please refer to the section title for the type of file being shown.

A.0.1 Simulation Set Up

Modified omnetpp.ini File

```
[General]

cmdenv-express-mode = true

cmdenv-autoflush = true

cmdenv-status-frequency = 1s

experiment-label = ${configname}

parallel-simulation = false
```

```
parsim-synchronization-class = omnetpp::cNullMessageProtocol

simtime-resolution = ps

**.cmdenv-log-level = info


ned-path = .

image-path = ../../images


network = RSUExampleScenario


############################################################

# Simulation parameters #

############################################################

debug-on-errors = true

print-undisposed = true


sim-time-limit = 380s


**.scalar-recording = true

**.vector-recording = true


*.playgroundSizeX = 2500m

*.playgroundSizeY = 2500m
```

```
*.playgroundSizeZ = 50m



############################################################
# Annotation parameters #
############################################################
*.annotations.draw = true


############################################################
# Obstacle parameters #
############################################################
*.obstacles.obstacles = xmldoc("config.xml", "//AnalogueModel[@type='
    ↪ SimpleObstacleShadowing']/obstacles")


############################################################
# TraCIScenarioManager parameters #
############################################################
*.manager.updateInterval = 1s
*.manager.host = "localhost"
*.manager.port = 9999
*.manager.autoShutdown = true
*.manager.launchConfig = xmldoc("erlangen.launchd.xml")
```

```
##########################################################

# RSU SETTINGS #

# #

# #

##########################################################

*.hacker[0].mobility.y = 1500

*.hacker[1].mobility.y = 2150

*.hacker[1].mobility.x = 2150

*.hacker[0].mobility.x = 1500

*.rsu[0].mobility.x = 2000

*.rsu[0].mobility.y = 2000

*.rsu[0].mobility.z = 3


*.rsu[*].applType = "TraCIDemoRSU11p"

*.rsu[*].appl.headerLength = 80 bit

*.rsu[*].appl.sendBeacons = false

*.rsu[*].appl.dataOnSch = false

*.rsu[*].appl.beaconInterval = 1s

*.rsu[*].appl.beaconUserPriority = 7

*.rsu[*].appl.dataUserPriority = 5

*.rsu[*].nic.phy80211p.antennaOffsetZ = 0 m
```

```
##########################################################
# 11p specific parameters #
# #
# NIC-Settings #
##########################################################
*.connectionManager.sendDirect = true
*.connectionManager.maxInterfDist = 2600m
*.connectionManager.drawMaxIntfDist = false

*.**.nic.mac1609_4.useServiceChannel = false

*.**.nic.mac1609_4.txPower = 20mW
*.**.nic.mac1609_4.bitrate = 6Mbps
*.**.nic.phy80211p.minPowerLevel = -89dBm # was -110

*.**.nic.phy80211p.useNoiseFloor = true
*.**.nic.phy80211p.noiseFloor = -98dBm

*.hacker[3].mobility.y = 1850
*.hacker[2].mobility.y = 2100
*.**.nic.phy80211p.decider = xmldoc("config.xml")
```

```
*.**.nic.phy80211p.analogueModels = xmldoc("config.xml")
*.**.nic.phy80211p.usePropagationDelay = true

*.**.nic.phy80211p.antenna = xmldoc("antenna.xml", "/root/Antenna[@id
    ↪ ='monopole']")
*.node[*].nic.phy80211p.antennaOffsetY = 0 m
*.node[*].nic.phy80211p.antennaOffsetZ = 1.895 m

##########################################################
# App Layer #
##########################################################
*.node[*].applType = "TraCIDemo11p"
*.node[*].appl.headerLength = 80 bit
*.node[*].appl.sendBeacons = false
*.node[*].appl.dataOnSch = false
*.node[*].appl.beaconInterval = 1s

##########################################################
# Mobility #
##########################################################
*.node[*].veinsmobility.x = 0
*.node[*].veinsmobility.y = 0
```

```
*.node[*].veinsmobility.z = 0

*.node[*].veinsmobility.setHostSpeed = false

**.mobility.z = 3

*.node[*0].veinsmobility.accidentCount = 1

*.node[*0].veinsmobility.accidentStart = 73s

*.node[*0].veinsmobility.accidentDuration = 50s


**.hacker[*].applType = "AttackSim"

*.hacker[3].mobility.x = 2200

*.hacker[2].mobility.x = 1700


[Config WithBeaconing]

*.rsu[*].appl.sendBeacons = true

*.node[*].appl.sendBeacons = true


[Config WithChannelSwitching]

*.**.nic.mac1609_4.useServiceChannel = true

*.node[*].appl.dataOnSch = true

*.rsu[*].appl.dataOnSch = true


##########################################################

# DDoS #
```

```
##########################################################

*.hacker[*].mobility.x = 2050

*.hacker[*].mobility.y = 2050

*.hacker[*].mobility.z = 3

*.hacker[*].applType ="AttackSim"

*.hacker[*].appl.headerLength = 80 bit

*.hacker[*].appl.sendBeacons = true

*.hacker[*].appl.dataOnSch = false

*.hacker[*].appl.beaconInterval = 1s

*.hacker[*].appl.beaconUserPriority = 7

*.hacker[*].appl.dataUserPriority = 5

*.hacker[*].nic.phy80211p.antennaOffsetZ = 0 m
```

Adding Attacking Vehicles to Communication System NED File

```
//

// Copyright (C) 2017 Christoph Sommer <sommer@ccs-labs.org>

//

// Documentation for these modules is at http://veins.car2x.org/

//

// SPDX-License-Identifier: GPL-2.0-or-later

//
```

```
// This program is free software; you can redistribute it and/or
   ↪ modify
// it under the terms of the GNU General Public License as published
   ↪ by
// the Free Software Foundation; either version 2 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program; if not, write to the Free Software
// Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307
   ↪  USA
//


import org.car2x.veins.nodes.Car;
import org.car2x.veins.nodes.RSU;
import org.car2x.veins.nodes.Scenario;
```

```
// the next line was added by Nick Jaton 2/16/2021

import org.car2x.veins.modules.application.DoS.HackerSim;



network RSUExampleScenario extends Scenario

{

    submodules:

        rsu[1]: RSU {

            @display("p=150,140;i=veins/sign/yellowdiamond;is=vs");

        }

        // hacker was added by Nick Jaton 02/16/2021

        hacker[4]: HackerSim {

                        @display("p=150,140;i=veins/node/car,#EF2929;is=

                            ↪ vs");

        }


}
```

Hacker NED File

```
package org.car2x.veins.modules.application.DoS;


import org.car2x.veins.base.modules.*;
```

```
import org.car2x.veins.modules.nic.Nic80211p;


module HackerSim
{
    parameters:
        string applType; //type of the application layer
        string nicType = default("Nic80211p"); // type of network
            ↪ interface card


        // Add mobility element 03-02-2021
        string veinsmobilityType = default("org.car2x.veins.modules.
            ↪ mobility.traci.TraCIMobility"); //type of the mobility
            ↪ module
    gates:
        input veinsradioIn; // gate for sendDirect
    submodules:
        appl: <applType> like org.car2x.veins.base.modules.
            ↪ IBaseApplLayer {
            parameters:
                @display("p=60,50");
        }
```

```
        nic: <nicType> like org.car2x.veins.modules.nic.INic80211p {
            parameters:
                @display("p=60,166");
        }

        mobility: BaseMobility {
            parameters:
                @display("p=130,172;i=block/cogwheel");
        }

    connections:
        nic.upperLayerOut --> appl.lowerLayerIn;
        nic.upperLayerIn <-- appl.lowerLayerOut;
        nic.upperControlOut --> appl.lowerControlIn;
        nic.upperControlIn <-- appl.lowerControlOut;

        veinsradioIn --> nic.radioIn;
}
```

A.0.2 DDoS Implementation

Attacker NED File

```
package org.car2x.veins.modules.application.DoS;

import org.car2x.veins.modules.application.ieee80211p.
    ↪ DemoBaseApplLayer;

simple AttackSim extends DemoBaseApplLayer
{
    @class(veins::TraCIDemoRSU11p);
    @display("i=block/app2");
}
```

Attacker Hex File

```
#pragma once

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

namespace veins {
```

```
/*************************************************************************/
    ↪
/**
 * Small RSU 11p that is used for flooding
 */
/* This was the first attempt with stationary RSU based attacker
class VEINS_API AttackSim : public DemoBaseApplLayer {
protected:
    void onWSM(BaseFrame1609_4* wsm) override;
    void onWSA(DemoServiceAdvertisment* wsa) override;
};
*/


/*************************************************************************/
    ↪
/* this is a second attack based off of car communication (stationary)
    ↪  */
class VEINS_API AttackSim : public DemoBaseApplLayer {
public:
    void initialize(int stage) override;


protected:
```

```
    simtime_t lastDroveAt;
    bool sentMessage;
    int currentSubscribedServiceId;

protected:
    void onWSM(BaseFrame1609_4* wsm) override;
    void onWSA(DemoServiceAdvertisment* wsa) override;

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

}
```

Attacker C++ File

```
#include "veins/modules/application/DoS/AttackSim.h"

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

#include <unordered_map>
using namespace veins;
```

```
Define_Module(veins::AttackSim);


/* This attack is based off the idea of parked cars attacking
 * Using the communication methods given.
 */



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


void AttackSim::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 AttackSim::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);

    // End testing code
    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 AttackSim::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);
    }
}

// Handle Position Update contains the attack.
void AttackSim::handlePositionUpdate(cObject* obj)
{
    DemoBaseApplLayer::handlePositionUpdate(obj);

    if((simTime() >= 50 && simTime() < 74) ||
            (simTime() >= 210 && simTime() < 224)
            ) {

        for (int i = 0; i < 25000; i++){

            sentMessage = true;
            TraCIDemo11pMessage* wsm = new TraCIDemo11pMessage();
```

```
            populateWSM(wsm);
            wsm->setDemoData(mobility->getRoadId().c_str());


            sendDown(wsm);
        }
    }
    else {
        lastDroveAt = simTime();
    }
}
```

Appendix B

Predictive Modeling Code

The following program was used on each simulation to determine the efficiency of each predictive algorithm. This is written in Python, primary making use of PySpark.

```
# Load the data into Spark
from pyspark.sql import SparkSession


spark = SparkSession.builder.appName('Test Vec Data').getOrCreate()


vecData = (spark.read.format("csv" \
                          ).option('header','true' \
                                 ).load("batch6/AttackSim.csv"))


# convert the data to floats
from pyspark.sql.functions import col
```

```
from pyspark.sql.types import IntegerType


vecDataFloats = vecData.select(*(col(c).cast("float").alias(c) \

                               for c in vecData.columns))

vecDataFloats = vecDataFloats.withColumn("Attack Bool",

                                      col("Attack Bool" \

                                         ).cast(IntegerType()))

vecDataFloats = vecDataFloats.withColumnRenamed("Attack Bool", "label
    ↪ ")


#########################################################


# Set up model scoring methods

from pyspark.ml.evaluation import MulticlassClassificationEvaluator

eval_acc = MulticlassClassificationEvaluator(labelCol = 'label',

                                          metricName ='accuracy')


eval_precision = MulticlassClassificationEvaluator(labelCol="label",

                                                 predictionCol="

                                                    ↪ prediction",

                                                 metricName="

                                                    ↪ precisionByLabel")
```

```
eval_recall = MulticlassClassificationEvaluator(labelCol="label",
                                                predictionCol="prediction",
                                                metricName="recallByLabel")

eval_f1 = MulticlassClassificationEvaluator(labelCol="label",
                                            predictionCol="prediction",
                                            metricName="f1")

#######################################################

from pyspark.ml.feature import VectorAssembler

ignore = ['Simulation', 'label']
assembler = VectorAssembler(
    inputCols=[str(x) for x in vecDataFloats.columns if x not in
        ↪ ignore],
    outputCol='features')

output = assembler.setHandleInvalid("skip").transform(vecDataFloats)
# output.printSchema()
```

```
# Train test Split

train_data, test_data = output.randomSplit([.70,.30], seed=1)


#######################################################


# Attempt to build regression model

from pyspark.ml.classification import LogisticRegression


lr = LogisticRegression(featuresCol = 'features',

                        labelCol = 'label',

                        maxIter=100)


lrModel = lr.fit(train_data)

lr_preds = lrModel.transform(test_data)


acc = eval_acc.evaluate(lr_preds)

precision = eval_precision.evaluate(lr_preds)

recall = eval_recall.evaluate(lr_preds)

f1score = eval_f1.evaluate(lr_preds)


print("LR Acc: "+ str(acc))

print("LR Prec: " + str(precision))
```

```
print("LR Recall: " + str(recall))

print("LR F-score: " + str(f1score))


#####################################################


# Build Random Forest Model

from pyspark.ml.classification import RandomForestClassifier

rf = RandomForestClassifier(featuresCol = 'features',

                            labelCol = 'label',

                            maxDepth=3)


rf_model = rf.fit(train_data)

rf_preds = rf_model.transform(test_data)


acc = eval_acc.evaluate(rf_preds)

precision = eval_precision.evaluate(rf_preds)

recall = eval_recall.evaluate(rf_preds)

f1score = eval_f1.evaluate(rf_preds)


print("RF Acc: "+ str(acc))

print("RF Prec: " + str(precision))

print("RF Recall: " + str(recall))
```

```
print("RF F-score: " + str(f1score))


#####################################################


# Build GBT
from pyspark.ml.classification import GBTClassifier
GBT = GBTClassifier(featuresCol = 'features',
                    labelCol = 'label',
                    maxIter=100)


GBT_model = GBT.fit(train_data)
GBT_preds = GBT_model.transform(test_data)


acc = eval_acc.evaluate(GBT_preds)
precision = eval_precision.evaluate(GBT_preds)
recall = eval_recall.evaluate(GBT_preds)
f1score = eval_f1.evaluate(GBT_preds)


print("GBT Acc: "+ str(acc))
print("GBT Prec: " + str(precision))
print("GBT Recall: " + str(recall))
print("GBT F-score: " + str(f1score))
```

```
#####################################################


# Build Support Vector Machines
from pyspark.ml.classification import LinearSVC
SVC = LinearSVC(featuresCol = 'features',
                labelCol = 'label',
                maxIter=100)


SVC_model = SVC.fit(train_data)
SVC_preds = SVC_model.transform(test_data)


acc = eval_acc.evaluate(SVC_preds)
precision = eval_precision.evaluate(SVC_preds)
recall = eval_recall.evaluate(SVC_preds)
f1score = eval_f1.evaluate(SVC_preds)


print("SVC Acc: "+ str(acc))
print("SVC Prec: " + str(precision))
print("SVC Recall: " + str(recall))
print("SVC F-score: " + str(f1score))
```

```
#####################################################


# Build MLPC
from pyspark.ml.classification import MultilayerPerceptronClassifier
layers_1 = [76,81,5,5,2]


mpc = MultilayerPerceptronClassifier(featuresCol='features',
                                     labelCol ='label',
                                     maxIter=100,
                                     layers=layers_1,
                                     blockSize=16,
                                     solver = "l-bfgs",
                                     seed=111)


mpc_model = mpc.fit(train_data)
mpc_preds = mpc_model.transform(test_data)




# MPC accuracy calcs
pred_and_label = mpc_preds.select("prediction", "label")


acc = eval_acc.evaluate(pred_and_label)
```

```
precision = eval_precision.evaluate(pred_and_label)

recall = eval_recall.evaluate(pred_and_label)

f1score = eval_f1.evaluate(pred_and_label)



print("MPC Acc: "+ str(acc))

print("MPC Prec: " + str(precision))

print("MPC Recall: " + str(recall))

print("f1: " + str(f1score))
```

Appendix C

AWS Node Testing Code

The following program contains the code used to test the computation time on the AWS EMR cluster. This program is entirely written using PySpark. Please refer to section 4.4.2 for more information.

```
# AWS Pyspark code


# Load the data into Spark

from pyspark.sql import SparkSession


# Create spark session

spark = SparkSession.builder.appName('AWS Cluster Testing').
    ↪ getOrCreate()


input_bucket = 's3://attack-sim/batch9_35Cars_50pct.csv'
```

```
vecData = (spark.read.format("csv").option('header',
                                          'true' \
                                         ).load(input_bucket))


from pyspark.sql.functions import col
from pyspark.sql.types import IntegerType


vecDataFloats = vecData.select(* \
                               (col(c).cast("float").alias(c) \
                                for c in vecData.columns))


vecDataFloats = vecDataFloats.withColumn("Attack Bool",
                                         col("Attack Bool" \
                                            ).cast(IntegerType()))


vecDataFloats = vecDataFloats.withColumnRenamed("Attack Bool",
                                                "label")


# Set up metrics
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
eval_acc = MulticlassClassificationEvaluator(labelCol = 'label',
                                             metricName ='accuracy')
```

```
eval_f1 = MulticlassClassificationEvaluator(labelCol="label",
                                            predictionCol="prediction",
                                            metricName="f1")


# Get data ready for MLPC
from pyspark.ml.feature import VectorAssembler


ignore = ['Simulation', 'label']
assembler = VectorAssembler(
    inputCols=[str(x) for x in vecDataFloats.columns \
               if x not in ignore],
    outputCol='features')


output = assembler.setHandleInvalid("skip").transform(vecDataFloats)


# Split data into training and testing datasets
train_data, test_data = output.randomSplit([.70,.30], seed=1)




# Build a NN using pysparks built in library.
from pyspark.ml.classification import MultilayerPerceptronClassifier
```

```
layers = [176,87,9,4,2]


import time

start_time = time.time()


# Build MLPC

mpc = MultilayerPerceptronClassifier(featuresCol='features',

                                     labelCol ='label',

                                     maxIter=100,

                                     layers=layers,

                                     blockSize=16,

                                     solver = "l-bfgs",

                                     seed=111)


# Fit model and get results

mpc_model = mpc.fit(train_data)

mpc_preds = mpc_model.transform(test_data)


end_time = time.time()

print("Total execution time: {} seconds".format(end_time - start_time)
    ↪ )
```

```
# MPC accuracy calcs

pred_and_label = mpc_preds.select("prediction", "label")



acc = eval_acc.evaluate(pred_and_label)

f1score = eval_f1.evaluate(pred_and_label)



print("MPC Acc: "+ str(acc))

print("MPC F-score: " + str(f1score))
```