Distributed Neural Network-Based DDoS Detection in Vehicular Communication Systems

Nicholas Jaton[1], Sohan Gyawali[2], Yi Qian[3]
[1] Department of Electrical and Computer Engineering, University of Nebraska-Lincoln, Omaha, NE, USA
[2] Department of Technology Systems, East Carolina University, Greenville, NC, USA
[3] Department of Electrical and Computer Engineering, University of Nebraska-Lincoln, Omaha, NE, USA
Emails: nickajaton@gmail.com, gyawalis22@ecu.edu, yi.qian@unl.edu

***Abstract*—As modern vehicular communication systems advance, the demand for robust security measures becomes increasingly critical. A misbehavior detection systems (MDS) is a tool developed to detect if a vehicular network is being attacked so that the system can take steps to mitigate harm from the attacker. Vehicular communication systems face significant risks from distributed denial of service (DDoS) attacks. During a DDoS attack, multiple nodes are used to flood the target with an overwhelming amount of communication packets. In this paper, we first survey the current MDS literature and how it is used to detect and mitigate DDoS attacks. We then propose a new distributed multilayer perceptron classifier (MLPC) for DDoS detection and evaluate the performance of the proposed detection scheme in vehicular communication systems. For the evaluations using simulations, two specific implementations of the attacks are conducted. Apache Spark is then used to create the distributed MLPC. The median F1-score for this MLPC method was 95%. The proposed method outperformed linear regression and support vector machines, which achieved 89% and 88% respectively, but is unable to perform better than random forests and gradient boosted trees which both achieved a 97% F1-score. Using Amazon Web Services (AWS), it is determined that model training and detection time are not significantly increased with the inclusion of additional nodes after three nodes including the master.**

***Index Terms*—Artificial neural networks, distributed computing, machine learning, vehicular communication networks, vehicular network security, vehicle safety, wireless sensor networks**

I. INTRODUCTION

Despite the acceleration in the deployment of cellular-based vehicular communication networks by automobile manufacturers, there remain significant challenges to be addressed before their widespread implementation can be realized. Vehicular networks are vulnerable to various attacks because of the wireless nature of vehicular communications. In this paper, we explore misbehavior detection systems (MDS) to detect and mitigate these attacks. A MDS utilizes an algorithm to identify whether incoming data packets may be indicative of a security breach. These algorithms are difficult to create due to the large number 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 techniques hold potential in MDS for both traditional networks and vehicular communication networks, but further research is necessary to advance the field. In this paper, we specifically focus on a distributed neural network-based DDoS detection scheme in vehicular communication systems.

A DDoS attack involves a malicious party utilizing multiple vehicles to produce large amount of network traffic in an attempt to prevent the regular vehicular communications. The vehicles used in a DDoS attack may not be aware that they are being utilized by a malicious attacker. For this reason, the devices used in DDoS attacks are often referred to as ”zombies”. Blocking the vehicular communication system could cause a vehicular accident and in a worse case scenario, the loss of human life. This kind of attack can be difficult to detect and recover due to the use of zombie vehicles.

In this paper, we propose a distributed neural network-based scheme - a new distributed multilayer perception classifier for DDoS detection in vehicular communication systems. We implement the proposed scheme in Objective Modular Network Testbed in C++ and OMNeT++, and evaluate the performance of the scheme. We use OMNeT+ as the simulation tool for network communications.

Our MDS model draws on insights from [1] which examined distributed MDS performance on real-world vehicular communication datasets. Although their findings were encouraging regarding the use of distributed random forest models, these were not tested on contemporary vehicular communication simulators like Network Simulator 2 (NS2) or OMNeT++. Furthermore, both [2] and [3] achieved impressive outcomes through the application of neural networks in NS2-based vehicular communication simulations. Hence, the study in this paper aimed to test the hypothesis that a distributed neural network could serve as an effective MDS approach, while also evaluating the potential benefits of employing distributed algorithms in contemporary network simulations.

II. BACKGROUND AND RELATED WORK

Statistical detecting approaches show great promising in MDS. Both [4] and [5] demonstrate the effectiveness of diverse machine learning models for detecting Sybil, DoS, and false alert attacks in vehicular communication networks. The authors studied the using of k-Nearest Neighbors, Logistic Regression, Decision Tree Classifier, Bagging, and Random

Forest in both the publications. The methods tested in [4] and [5] utilized the VeReMi datasets and Veins for network simulations. These papers use 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 measure 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 using the Dempster-Sharfer (DS) in [4]. By adding DS to the existing methods proposed in [5], the authors were able to produce higher F1-scores for the bagging and random forest models which were the top performers in both papers.

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 [6] utilized SVM in the analysis step of their design. In that model, various watchdog vehicles monitor communications and use the SVM model to analyze the data collected. The collection and exchange elements of the 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 then sends 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 revealed a significant variation in performance between CEAP, DS, and Averaging methods.

Neural Networks have also been considered in MDS design. The authors in [7] developed a convolutional neural network (CNN)-based scheme 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 the 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.

In [8] the authors also used neural network-based detection model. This paper described an artificial neural network (ANN) scheme 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.

In [2] the authors studied 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 [2], both SVM and feed forward neural network (FFNN) are utilized in an effort to detect these grey hole attacks. Much like in [8], 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 [8] and [5], etc.

In [3] the authors continue 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 method consists of 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 [9]. An 80/20 ratio of normal vs. misbehaving vehicles was used. The F1-score results for this model remained above .97 for each of the 4 vehicles analyzed. These results are comparable to the results found by [8] and [4]. Unfortunately, because the studies used different datasets, it is hard to make accurate comparisons.

Deep learning methods have also been utilized by MDS. The authors in [10] engineered a deep neural network (DNN) scheme to detect suspicious activities in a vehicular controller area network (CAN). 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 detect did not improve, however, normal packet detection increased by 4%. The expected training time and testing time increased as the layers increased.

An analysis was performed by [11] to determine how effective various machine learning algorithms 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 forging 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 detecting 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.

The application of big data techniques has proven to be valuable in contemporary misbehavior detection systems. For instance, in 2019, [1] published a study on the efficacy of combining distributed schemes with machine learning for detecting DDoS attacks in vehicular communication systems. The system described in the study transformed a communication 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 [12] and UNSW-NB15 [13] datasets. Although NSL-KDD is derived from KDDCUP99, it is considered outdated by contemporary standards. 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 GBDT performed well, the authors version of boosted random forest out performed all other algorithms.

III. Distributed Systems and Computation

While previous research has successfully developed machine learning-based detection methods for DoS attacks, the issue of computation cost has received limited attention. A potential solution could be to incorporate distributed system techniques into misbehavior detection systems to reduce the cost of detection on individual vehicles. This implementation is designed to enable parallel operations on a cluster, consisting of nearby vehicles in a vehicular communication system. The proposed scheme uses a neural network to analyze communication data and identify DoS attacks.

For the vehicular communication system, distributed computation 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. 1 shows an example of how this scheme works in 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.

Apache Spark is used in the proposed system for distributed computation. Spark functions by splitting the data into a type of dataset referred to as a resilient distributed dataset. The

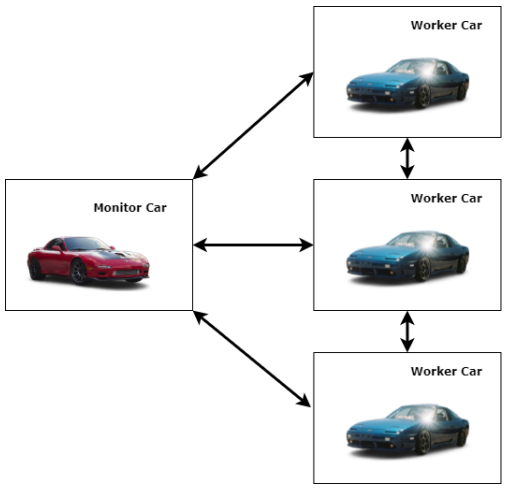

Fig. 1: Vehicular Distributed System Example

RDD structure allows adjustments to the data to be processed 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.

IV. 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 is to create a classifier that could work as effectively as the human brain. The single layer perceptron is developed to simulate a neurons operation in the brain. Researchers found that chaining these perceptrons together can create a neural network that simulates the operation of the human brain. The rest of this section gives an overview on how these models work using both single and multiple perceptron layers.

A. 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 is shown in Fig. 2. In this example, the input value is denoted as X and the weight value is represented by W.

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

$$z = f(\sum_{i=1}^{n} W_i * X_i) \tag{1}$$

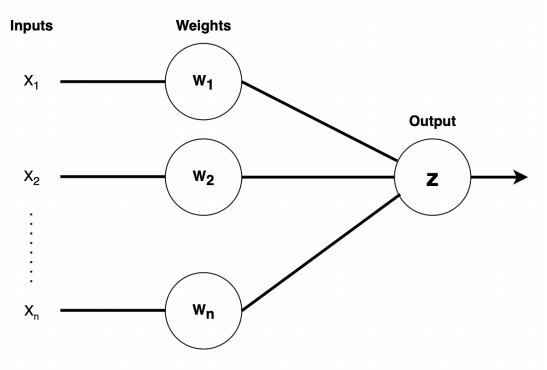

Fig. 2: Single Layer Perceptron

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

$$f(x) = \begin{cases} 1, & \text{if } z > \theta \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

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

B. 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 MLPC or a feed forward neural network. This scheme contains a series of layers. Each of these layers are effectively 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 contains a simple visual example of the MLPC scheme.

Mathematically speaking, the MLPC is just a more complex version of the single layer perceptron. Recall equations (1) and (2), 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:

$$H_j = f(\sum_{i=1}^{n} W_{ji} * X_i) \tag{3}$$

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:

$$O_j = f(\sum_{j=1}^{n} W_{kj} * H_j) \tag{4}$$

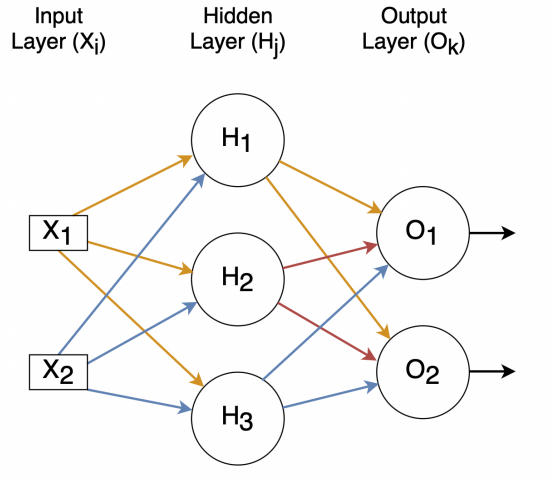

Fig. 3: Multilayer Perceptron

V. L-BFGS Optimization Routine

The Limited-memory Broyden–Fletcher–Goldfarb–Shanno algorithm (L-BFGS) is selected as the optimization routine for the proposed model. This algorithm is a variant of the Broyden-Fletcher-Goldfarb-Shanno algorithm that is specifically designed for optimizing 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 [14]. 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:

$$\rho_k = \frac{1}{y_k^T s_k} \tag{5}$$

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

$$H_k^0 = (\frac{s_k^T y_{k-1}}{y_k^T y_{k-1}})I \tag{6}$$

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 $H_k \nabla f_k$

$q = \nabla f_k$
for *i = k - 1, k - 2,..., k - m* **do**
  $\alpha_i = \rho_i s_i^T q$
  $q = q - \alpha_i y_i$
end
$r = H_k^0 q$
for *i = k - m, k - m + 1,..., k - 1* **do**
  $\beta = \rho_i y_i^T r$
  $r = r + s_i(\alpha_i - \beta)$
end
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:

$$f(x_k + \alpha p_k) \leq f(x_k) + c_1 \alpha \nabla f_k^T p_k \tag{7}$$

Please note that c is 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*

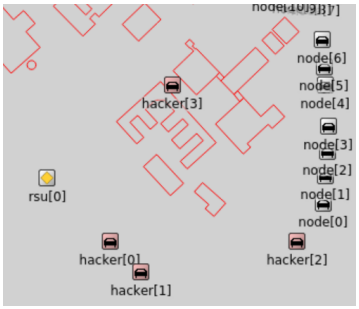

Fig. 4: Attack Simulation View in OMNeT++

VI. Simulation Settings and Datasets

In the related work section, 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 tool for the simulation evaluations in this paper. To enable these simulations to include mobile vehicles, Veins and SUMO are used. SUMO is a traffic simulation software that generates vehicular traffic and 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 for this study utilize 1-hop broadcasting between the vehicles and the roadside units. Therefore, each message will be distributed to all nodes in the range of the sender.

Our 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. We show a scenario that 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 attack detection.

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. The simulation was repeated for each of the following five vehicle numbers: 15, 20, 25, 30, and 35. In addition to the number of 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 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”.

To evaluate the performance of the MDS under varying attack densities, seven distinct versions of each attack simulation were created. 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 I 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 contains the vehicles' current road id that are then distributed to all nearby vehicles.

Attack %	1st Attack (s)	2nd Attack (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 I: Attack times used in simulations

Algorithm 3: 10% Attack Density Simulation

Result: Perform DDoS Attack

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

VII. Data Preparation and Scheme Design

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 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. A "if then" command was used in excel to create a column labeling the attack times. If an attack occurred 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 vehicles 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.

The MLPC scheme 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 models with less layers. This testing also determined that ranges just above the feature number worked well in the second layer. Further testing also indicated that smaller values performed best for the third and fourth layers. To narrow in on possible high-performing models, Algorithm 4 was ran on all simulations.

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 scheme 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 scheme, 76 is referring to the input layer and 2 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 4 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 the methods section, L-BFGS was used for the optimization routine. L-BFGS was chosen over minibatch gradient descent due to the latter's frequent underperformance

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
```

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.

VIII. Simulation Results and Discussions

A. Model Comparison During Various Attack Densities

To test how this method compares to other common machine learning models, the proposed MDS 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, RF, GBT, and 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 learning 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.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{8}$$

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

$$Precision = \frac{TP}{TP + FP} \tag{9}$$

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

$$Recall = \frac{TP}{TP + FN} \tag{10}$$

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.

$$F1 - score = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{11}$$

The implementation of these functions in PySpark.ML was used for consistency. 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. 5 shows the accuracy of each model at different attack densities. By selecting the median F1-score across all attack densities, the overall scheme 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 logistic regression and SVM models with the exception of the 40% attack density.

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. 6 shows the precision value for each model at various attack densities.

Recall was consistent across all models except for 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. 7 shows the recall value for each model at various attack densities.

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. 8.

B. Training and Detection Time Using Multiple Clusters

Since PySpark is based on a distributed model, 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 an Amazon S3 Bucket in Amazon Web Services (AWS). Amazon

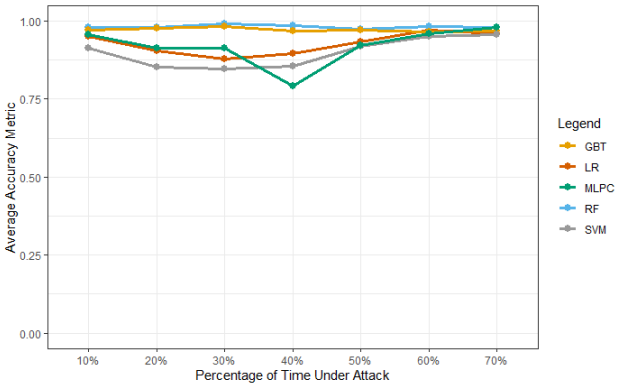

Fig. 5: Model Accuracy

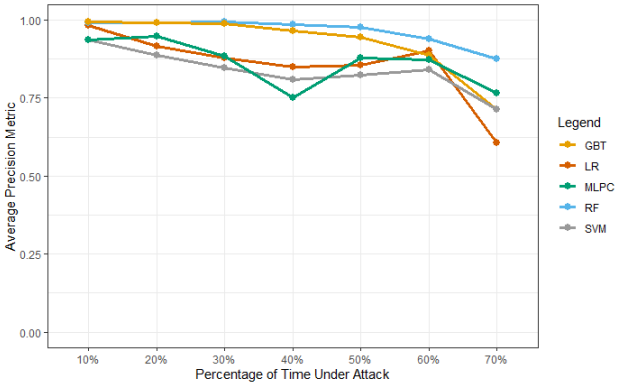

Fig. 6: Model Precision

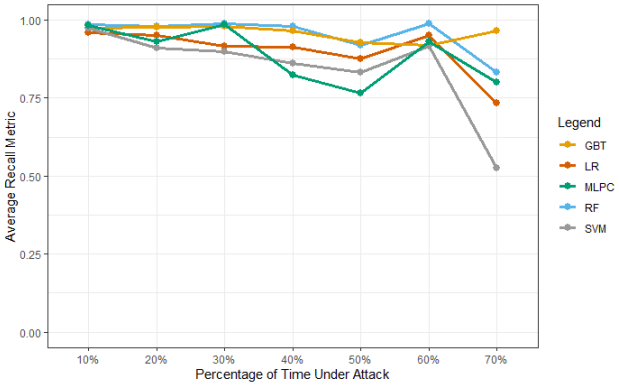

Fig. 7: Model Recall

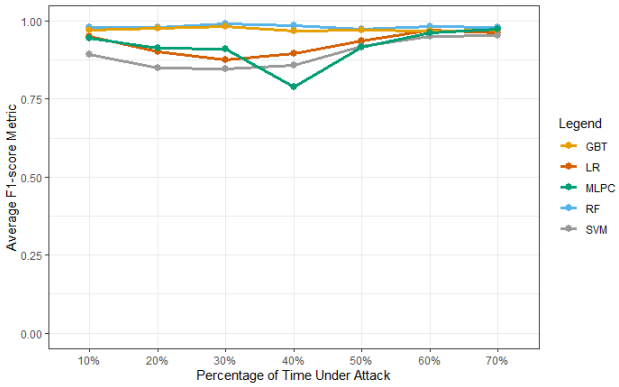

Fig. 8: Model F1-score

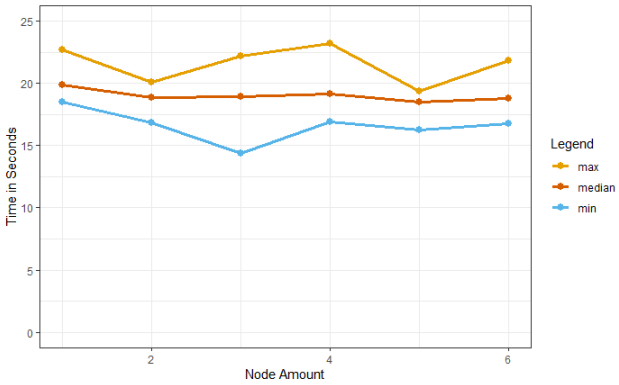

Fig. 9: MPLC Computation Speed vs Node Amount

elastic map reduce (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. Each EMR instance used release emr-6.1.0 and contained Hadoop 3.2.1, JupyterHub 1.1.0, Livy 0.7.0, Pig 0.17.0, Hive 3.1.2, Hue 4.7.1, and Spark 3.0.0. 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, and 64 Gib of storage. All logging was stored in an additional AWS S3 bucket.

To assess the performance variation across instance configurations, we conducted timed training and prediction on six distinct AWS instances, varying from a solitary master node to a total of six nodes. The simulation including 35 vehicles was used for this analysis. The Jupyter Notebook containing the algorithm was restarted and ran 10 different times for each instance. Fig. 9 shows the time required to run training and utilize the algorithm at each node count. The inclusion of extra nodes did not appear to significantly affect the overall computation time, although there was a slight reduction in median time with each additional node.

IX. CONCLUSION

The implementation of MDS systems is a crucial security measure for deploying vehicular communication networks. Without an MDS in place, all vehicles connected to the network would be susceptible to a range of attacks, including the likes of DDoS attacks. Such attacks can block communication channels by flooding the network with junk 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 [3], [2], and [10] showed promising results for the use of neural networks in MDS. Both [1] and [15] 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 primary objective of this study was to develop a MDS based on multilayer perceptron classifier, which is a type of feed forward neural network, for analyzing vehicular communications via simulations built in OMNeT++, Veins, and SUMO. Specifically, the study investigated the effects of DDoS attacks, which were carried out using parked zombie vehicles, on the simulated network. 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 node was meant to simulate a nearby vehicle.

Although multilayer perceptron classifier is a powerful predictive algorithm, it may not be the optimal choice for misbehavior detection in vehicular communication systems. Our OMNeT++ simulation involved only a limited number of vehicles and ran for a relatively short duration of 380 seconds. Despite this, both RF and GBT demonstrated reasonable accuracy, recall, precision, and F1-scores. In future research, it may be worthwhile to explore statistical learning methods that are less computationally demanding than RF and GBT. Doing so could improve the efficiency of misbehavior detection and reduce time delays in distributed computing applications.

ACKNOWLEDGMENT

This work was partially supported by National Science Foundation under grant CNS-2008145.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] J. Colyar, "Next generation simulation (ngsim) vehicle trajectories and supporting data: Department of transportation - data portal," 2016.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] J. Nocedal and S. J. Wright, *Numerical optimization*. New York, NY, USA: Springer, 2e ed., 2006.

[15] 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.