Intrusion Attacks on Automotive CAN and their Detection

Halley M. Paulson
Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA 56267
paul1098@morris.umn.edu

ABSTRACT

The main highway of communication in a vehicle is the Controller Area Network, commonly known by the acronym CAN. Any vulnerability in this network could allow bad actors to block communication between vehicle subsystems, risking the safety of the vehicle's occupants. With the ever growing list of vulnerabilities being exposed in the CAN, it is critical to address its safety. This paper looks at one of the known vulnerabilities in the data link layer of the CAN and an Intrusion Detection System that could detect attacks on this network. We detail a few processes of the CAN, arbitration and error states, and how they are leveraged during different attacks. We also explain the core component of the Intrusion Detection System, the Detection Engine, and discuss testing results.

Keywords

Controller Area Network, Intrusion Detection System, Fault Injection, Long Short-Term Memory Network

1. INTRODUCTION

Modern vehicles are comprised of a multitude of sensors, actuators, and controllers that all send and receive data, which is why they are considered Internet of Things (IoT) systems. Sensors take physical measurements, controllers receive sensor data and perform analysis, and actuators make physical changes according to the controller's instructions. Vehicles have multiple subsystems that each have their own sets of sensors, controllers, and actuators. An example of one of these subsystems would the the Automatic Braking System (ABS). The vehicle's subsystems must be able to send data back and forth in real-time for the car to function properly. The Controller Area Network (CAN) makes this possible. It allows all the subsystem's controllers to send data without a host computer. The issue with the CAN is the amount of vulnerabilities it has.

Having been created in 1985, a time before cybersecurity was even a consideration, the CAN's protocols are not designed to prevent, or even detect, attacks. In recent years, many studies have exposed different vulnerabilities and how they can be leveraged to manipulate vehicles. One vulnerability in the CAN that is especially dangerous involves CAN processes that are designed to organize the physical transmission of data across wires and manage subsequent errors. Researchers Murvay and Groza [3] show that this vulnerability can be manipulated to perform Denial of Service (DoS) attacks. These DoS attacks prevent subsystems of the vehicle from communicating with each other, which can risk the safety of the vehicle's occupants. Imagine if the ABS couldn't signal the car to stop; it could be extremely dangerous. Since this vulnerability involves many physical components and protocols, many of the currently proposed solutions for securing the CAN don't work. The CAN is also a low-resource system, meaning it doesn't have a lot of extra computing power or memory to be able to run heavy security protocols that typical networks can. So, there aren't any perfect patches for it, but there are proposed solutions.

An Intrusion Detection System (IDS) was proposed recently that holds some hope in sealing the CAN [7]. A system that monitors network traffic and picks up on anomalies may be the CAN's saving grace, especially since it claims to be resource efficient. This IDS would allow the CAN to at least identify when its being attacked, which is a huge step in the right direction. It's powered by a Long Short-Term Memory network, a type of machine learning model. This machine learning model has ensured success for a plethora of systems.

We begin background information with section 2, explaining the technologies involved with the CAN vulnerability [3] and with the proposed IDS [7]. From there we move to section 3, where we walk through DoS attacks performed on the system that leverage the CAN vulnerability in different ways. Next, we discuss how the proposed IDS works and detail the experiments done to try and prove the system's validity in section 4. Finally, we wrap up with conclusions in section 5.

2. BACKGROUND

2.1 The Controller Area Network

The *Controller Area Network (CAN)* is comprised of the vehicle's controllers which all run on protocols designed for limited-resource IoT systems. Micro-controllers are just one of the few different types of controllers that exist today, and the majority of controllers in the CAN are micro-controllers. Micro-controllers are mini computers integrated into chips that control one small part of a greater system. In a vehicle, a part of the system could be anything from the entertainment system to a section of the engine.

The CAN physically consists of a CAN bus and CAN

UMM CSci Senior Seminar Conference, November 2021 Morris, MN.

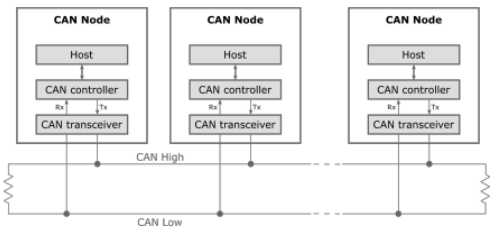

Figure 1: CAN Bus Topology [3]

nodes. The CAN bus is physical wiring allowing communication between CAN nodes and is typically referred to as either CAN High or CAN Low. Referring to Figure 1, a CAN High bus carries data faster than a CAN Low bus. Each node that participates in CAN communication requires a CAN interface; a low-speed interface to communicate with a CAN Low bus and a high-speed interface for a CAN High bus. Interfaces are comprised of a CAN controller and a CAN transceiver connected by 2 wires [3]. These interfaces are most commonly installed as a module of a micro-controller and is what allows communication between a CAN node and bus through implementation of CAN protocols.

Data shared in the CAN is logically organized and packaged as frames. Frames are patterns of bits, and bits are just bursts of electricity at specific voltage levels. The voltage levels for transmitting bits are different depending on the speed of communication, but the CAN frame is the same regardless.

CAN frames are visible to every node connected to the CAN bus since this is a broadcast network. There are a few mechanisms in place to organize the chaos of broadcasting and any errors caused from this method of communication. The main mechanisms are arbitration and error states.

2.1.1 Arbitration

Arbitration is the organization of nodes taking and releasing control of the bus. For nodes to transmit data on the CAN bus, they need control of it. Nodes will watch the bus to ensure that it is idle before they try to transmit data and, as soon as a node takes control, the bus will signal that it's active. In the case that multiple nodes try to transmit at the same time, the process of arbitration takes place. Arbitration depends on the message priorities of the competing frames. The message priority of a frame is a number, included in the beginning of said frame, that signals the importance of the data in the frame; the lower the number the higher the priority [8]. The message priority is calculated using the initial bits of the frame and typically more dominant bits (0's) means higher priority. If the frame the node is sending has a higher message priority, then it wins arbitration and can continue transmitting the frame without worrying about corruption. The loser with the lower message priority frame has to watch the bus until it sees the winner is finished transmitting before it can try again. This is the main way the CAN organizes the flow of communication.

2.1.2 Error Handling

To keep errors under control, CAN nodes switch between three different error states. The error state of a node is determined by the value of one of its error counters, the Transmission Error Counter (TEC). As previously stated, CAN nodes take control of the CAN bus to transmit data. A CAN node is constantly checking to make sure the intended frame is written to the bus. A transmission error is when the frame written to the bus is different than what the node intended to write. The TEC is incremented when the node observes a transmission error. It decrements the TEC when successful transmissions are observed.

When the TEC is below 127 the node is in the Error Active state, which is the default state. The node can send and receive data normally in this state, and is expected to signal the entire network when it notices errors.

A node will switch to an Error Passive state when the TEC goes above 127. Like the Error Active state, it can send and receive data normally, but it can't notify the entire network when it notices errors.

If the TEC goes past 256, the node switches to the Bus Off state and can no longer communicate until it sees 11 consecutive recessive bits (1's) on the CAN bus 128 times [8]. Since the idle state of the bus is indicated with recessive bits (1's), this just means the node has to wait for the bus to be idle for a specific amount of time before it can reset. Once it observes that reset condition, it can clear its counters and return to the Error Active state. This keeps problematic nodes from interrupting communication.

2.2 Fault Injection

A *fault* is an error or failure. *Fault injection* is a verification technique which induces artificial faults in a system to evaluate the behavior of the system in response to those faults [6]. Faults can be injected at the physical level by using a fault injector tool to inject bits into frames, thereby changing the intended message of the frame. Systems should have protocols in place to successfully identify and handle faults. While it is a testing technique, it is a tool bad actors can use to change or block system functions in cybersecurity attacks since fault injectors can be easily created either with physical parts and software, or by remotely reprogramming parts of an existing system.

2.3 Intrusion Detection Systems

Intrusion Detection Systems (IDSs) can either monitor hosts or entire networks. A Host IDS looks at traffic coming and going to a system. A Network IDS looks at traffic going to and from all systems in a network. Host IDSs are more effective and reliable, however Network IDSs have a greater scope and are more resource-friendly - perfect for a resource-limited system.

There are two IDS modes, online and offline. Online-mode IDSs will alert as soon as a threat is detected, while Offline-mode IDSs will collect and store data after an attack happens [2].

IDSs can be sorted into signature-based or anomaly-based by the way they detect threats. Signature-based IDSs detect threats by pattern-matching; comparing traffic to previously recorded threats to see if any of it is the same. They have low false alarms rates and are very efficient, but they are limited by the records they compare to [2]. Anomaly-based IDSs detect threats by identifying unusual network behavior. Once they are trained, they compare real-time traffic to the trained base-line and raise an alarm if the values are too different. While these are useful in keeping up with new or unknown threats, they tend to have high false alarm rates as it is quite difficult to predict user behavior. For example, if installed in a vehicle, an IDS could be used to a driver who usually goes between 30 and 40 mph, but then suddenly the driver is going between 80 and 100 mph. The IDS could view this as anomalous and raise an alarm, thus creating high false alarm rates.

2.4 Recurrent Neural Networks

To understand *Recurrent Neural Networks* (RNNs) it is helpful to understand *Feedforward Neural Networks* (FNNs) as they were the first and are the simplest type of Neural Network. A FNN is made up of an input layer, hidden layers, and an output layer. They can only move information forward from input to output layers.

Each *layer* is made up of cells connected to the next layer of cells through edges. Each *edge* contains a weight. *Weights*, typically called internal weights, are values that decide how much effect inputs have on outputs. Weights are updated during training. Training allows a model to map inputs to desired outputs through the use of a training dataset. Inputs from the dataset are fed through the model and predictions are returned as an output. During a process called back propagation, a loss function is used to calculate the error rate of the model. This involves comparing outputs of the model to the known expected results for the given inputs in the training dataset. Then an optimization function is used, starting from the output layer and working towards the input layer, to adjust the model's weights in an effort to lower the error rate. This is how the model "learns".

Each *cell* contains one or multiple activation functions. Activation functions introduce non-linearity, meaning it allows the network to predict data that doesn't fit into a linear model. In a FNN, cells take a set of inputs, pass the sum of inputs multiplied by weights through their activation functions, and then feed the output to the next layer of cells.

RNNs are similar to FNNs, but are special in the way they handle previous inputs. In the hidden layers, RNN cells can loop their previous inputs into the next through a collection of values called the hidden state. The hidden state is updated after each element in the input sequence. Since this allows past data to have an impact on the next output, RNNs are considered to have a memory [5].

Say, for example, we have sequential data in the form of a sentence - "I Fly A Plane" - and our model is trying to understand its meaning. Without memory, the model wouldn't know what any of the previous words were. The meaning of "Fly" depends on it coming after "I". Likewise, the meaning of "Plane" depends on it coming after "Fly". So, without memory, the model wouldn't be able to have a reliable understanding of the sentence's meaning. With memory, the model would be able to cataloging the sequence of the sentence with every new input, which allows it to remember the order of words. This makes RNNs a powerful learning model.

RNN's suffer from the Vanishing Gradient problem. The *Vanishing Gradient* problem causes a model to learn less as it accumulates layers. The nature of back propagation leads to changes in internal weights becoming smaller and smaller as the algorithm calculates from the output to the input layer [4]. The changes to the internal weights eventually become insignificant which means that those layers stop learning. This can lead models to become less accurate as it becomes more complex.

2.4.1 Long Short-Term Memory Networks

A *Long Short-Term Memory network (LSTM)* is a type of RNN that combats the Vanishing Gradient problem. This model functions similarly to basic RNNs. The big difference is the calculations done inside each cell which allow the model to keep information for longer.

Like RNNs, each cell has a hidden state, but LSTMs also have a *cell state.* Both retain information from previous inputs, but the cell state aggregates the entire sequence in some form while the hidden state typically emphasizes the most previous input. Gates are used to update the cell state, instead of updating after each input. *Gates* are calculated values. The *forget gate* specifically handles what data is forgotten or kept. The *input gate* handles adding new data to the cell state.

Both the previous hidden state and cell state values pass through activation functions to update them. Calculating the new cell state also involves both the forget and input gates. The previous cell state values and the forget gate go through an activation function to remove unwanted cell state values. A similar process happens, using the input gate instead of the forget gate, when determining what to add to the cell state.

This process allows the LSTM to control what it remembers and forgets, putting it above regular RNNs in performance. For example, lets take our previous thought experiment. Lets take the sentence - "I Fly A Plane". LSTMs are able to keep some of the information from processing "I", "Fly", and "Plane" because their gates and algorithms have decided they are absolutely critical in understanding the meaning of the sentence. Since the model has a separate process to determine and incorporate critical data, the Vanishing Gradient problem doesn't have as much of an impact on its continued learning. This makes LSTMs more accurate and reliable when handling longer and more complex sequences of data.

3. ATTACKING THE CAN

3.1 Attack Experiments

Murvay and Groza [3] conducted four different attacks on the CAN with their personal fault injector. These experiments were labeled Full Bus DoS, Directed DoS, Arbitration Denial and Disrupting Synchronization. We will be looking into the Full Bus DoS, Directed DoS and Arbitration Denial attacks to illustrate the core vulnerability these experiments reveal. The fault injector used in these attacks was a homemade CAN node built using inexpensive and widely available electronics, but it is possible to compromise and reprogram an existing node in the network as well [3]. This means there are multiple accessible entry points for bad actors. With the right resources and time to learn, anyone with malicious intent could create a fault injector and begin interfering with vehicle communication.

3.1.1 Full Bus DoS

This attack was designed to completely stop communication on the CAN bus; no node would be able to send or receive messages. It worked by configuring the fault injector to continuously send dominant bits (0's). The CAN transceiver built into the fault injector had its own fault detecting mechanism. The mechanism would trigger when it noticed anomalies, such as continuous transmission of the same bit, and interrupt the injector. To prevent this from triggering, recessive bits (1's) would occasionally have to be sent.

Results showed this to be the simplest attack on the CAN. Since the fault injector was constantly sending dominant bits (0's), the bus was constantly active and other CAN nodes were left waiting until the bus was idle. Since no other nodes could send or receive messages with the bus being used to transmit bogus data, CAN communication was completely denied and the attack was very successful.

3.1.2 Directed DoS and Arbitration Denial

The Directed DoS attack was designed to target frames and cause specific CAN nodes to switch to the Bus Off state. It worked by injecting a dominant bit at the start of the data portion of the frame. This then caused a transmission error which the target node promptly marked by incrementing its TEC. After that, the node would try to retransmit, but once again a bit would be injected. This cycle would continue until the node switched to the Bus Off state.

The tricky aspect of this attack was the timing and prior knowledge needed. Before the attack could even begin, Murvay and Groza [3] gathered information pertaining to the possible IDs of the frames that the target node sent so their fault injector could monitor the bus for them. The next hard part was the injection timing. The fault injector needed to have enough computing power to transmit with the same amount of time it takes the CAN to transmit a bit; it needed to fit into the CAN bit timing.

On low-speed CAN, the attack succeeded in injecting bits and causing enough errors that the node switched into the Bus Off state. During this attack, the target node tried retransmission 31 times, and each time a bit was injected to cause an error and start the process over again. Eventually, the TEC became high enough that it switched the node to the Bus Off state. Results were different with high-speed CAN. Murvay and Groza [3] succeeded with injecting bits, but only into a specific sequence at top speeds. They also needed a much more detailed understanding of the CAN frames being sent from the node in order to identify the sequence they could manipulate. With their setup, they didn't have the computing ability needed to flexibly inject at such speeds like they could at low-speed CAN, and couldn't switch nodes into the Bus Off state. In these experiments, they had better success at slower speeds, but upgrading the setup would be simple enough and in all likelihood achieve successful results at higher speeds.

With this same setup, Arbitration Denial attacks can be achieved. The only difference between Directed DoS and Arbitration Denial attacks is where the bit is injected. If it is injected into the arbitration area of the frame, the beginning bits, the message priority can be altered so the target node loses arbitration and has to wait until the winning node finishes transmitting. When this is done every time the node tries to arbitrate for the bus it will lose every time. If it loses every time it will always be waiting its turn and never be able to send messages, therefore service is successfully denied to the target node. While the Arbitration Denial experiment is labeled differently, the target and results are the same as a Directed DoS attack.

4. SECURING THE CAN

4.1 Proposed Intrusion Detection System

Tanksale [7] introduced an Anomaly-based IDS that accommodates the resource-limited CAN. The IDS has two main parts, an anomaly detection engine and a decision engine. The detection engine is powered by an LSTM, refer to 2.4.1, prediction algorithm. It's designed to predict values based on previous CAN measurements. Those predictions are compared to a threshold value to determine their labels. If a predicted value is higher than the threshold value, it's marked as an anomaly. All anomaly flags are sent to the decision engine, which decides if the anomalies make up an attack or not. We will be focusing on the anomaly detection engine and the prediction algorithm powering it.

4.2 Data Formulas

The IDS was designed to monitor multiple vehicle subsystems for anomalies. To make that possible, the LSTM had to be trained using values that represent multiple vehicle subsystems. This is where the two formulas fit in; we will call them Formula A and Formula B. These formulas bring together CAN measurements from different subsystems of the vehicle. Two different datasets were created from the two formulas. The Formula A and Formula B datasets were generated using the same CAN measurements that were collected from 10 different cars, which all drove the same route for a duration of 35 to 45 minutes [7].

Formula A was the first data formula. It depended on only two different CAN measurements, engine speed and accelerator pedal position.

$$F = \frac{EngineSpeed}{AcceleratorPedalPosition} \tag{1}$$

Each data point in the Formula A dataset was calculated using this ratio.

Formula B was much more detailed as it took into account 10 different CAN measurements as it generated the dataset. Let x_1: brake position, x_2: brake pressure, x_3: wheel speed, x_4: current gear, x_5: lateral acceleration, x_6: steering angle, x_7: accelerator pedal position, x_8: longitudinal acceleration, x_9: engine coolant temperature, x_{10}: intake air temperature. The Pearson Correlation Coefficient, which is used in statistics to determine the strength of a relationship between two variables, is represented as $corr(p, q)$.

$$\begin{split} F = \frac{x_1}{x_2} + \frac{x_3}{x_4} + corr(x_5, x_6) + corr(x_3, x_2) + \\ corr(x_7, x_8) + |x_9 - x_{10}| + corr(x_1, x_8) \end{split} \tag{2}$$

Essentially, Formula B is the summation of ratios and measured relationships between multiple CAN measurements. This formula more accurately reflects the nature of the CAN.

4.3 Preparing the Prediction Algorithm

To support the proposed IDS, the prediction algorithm had to go through three phases: Pre-Training, Formula A Testing, and Formula B Testing. In the 'Pre-Training' section, the LSTM was trained and tested numerous times to figure out the best hyperparameters, values used to control the learning process, to use in training. In the 'Formula A Testing' section, data from the Formula A dataset and the hyperparameters found in the 'Pre-Training' section, were used to test how well the algorithm could predict anomalies.

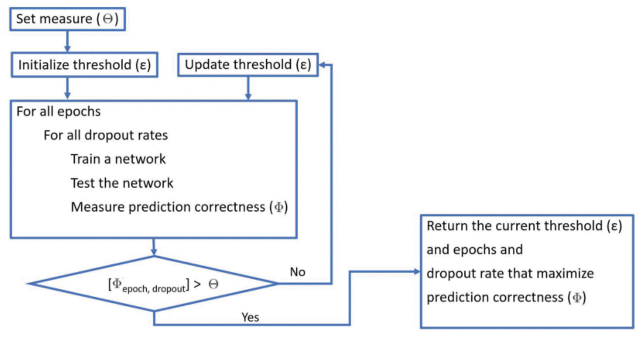
Set measure (Θ)
Initialize threshold (ε)
Update threshold (ε)
For all epochs
For all dropout rates
Train a network
Test the network
Measure prediction correctness (Φ)
$[\Phi_{epoch, dropout}] > \Theta$
No
Yes
Return the current threshold (ε) and epochs and dropout rate that maximize prediction correctness (Φ)

Figure 2: Pre-Training Algorithm Flowchart [7]

The ‘Formula B Testing’ section repeated this process, but with Formula B to test for any improvements in anomaly prediction.

4.3.1 Pre-Training

Recursion is used in this section to automate the process of finding the best hyperparameters, and Figure 2 illustrates this. In Figure 2, there are parameters labeled measure (θ), threshold (ϵ), epochs, dropout rates, and prediction correctness (Φ). These will all be explained shortly. We are actively trying to find the best combination of epochs and dropout rates. We also determine the threshold as a byproduct of this process. In order to understand the process, we must understand the parameters used.

An *epoch* is one full pass of an entire training dataset through the learning model. This means the entire set of data has had an opportunity to affect the model and update its internal weights. Typically, there are multiple epochs for training a learning model, so the full training dataset is passed through the model multiple times. This exposes the model to different combinations of data within the dataset and reduces error [1]. We pass an array of epochs to the recursive function.

The *dropout rate* is the percent of hidden nodes in the learning model that are randomly ignored during training. This is to combat over-fitting, predicting too close to a set of data which can cause a model to predict inaccurately. We pass an array of dropout rates to the recursive function.

The *threshold* (ϵ) is used to determine the labels of predictions.

The *prediction correctness* (Φ) is the percent of accurate predictions the algorithm has made. This is calculated for each combination of epochs and dropout rates.

The *measure* (θ) is how high the prediction correctness needs to be before the algorithm stops looping. This value doesn’t change once it’s set.

Now that we have explained the function’s parameters, let us explain the process. First, the measure (θ) is set to 0.93 and the threshold (ϵ) is initialized at 0.1. We pass the array {10,50,100} to epochs and the array {20%,30%,50%} to dropout rates. 80% of the Formula A dataset is used to train the LSTM and 20% is used to test. We use Formula A in this section because, at the time, Formula B didn’t exist. Then we run the function. For every combination of epochs and dropout rates, we train and test the LSTM prediction algorithm. Then, we take the calculated prediction correctness (Φ) and compare it to the measure (θ). If the prediction correctness (Φ) is less than the measure (θ), we increment the threshold by 0.1 and loop again with the next combination. The function loops until the prediction correctness (Φ) is greater than the measure (θ) for all combinations of epochs and dropout rates. When the loop is finished, the combination that maximizes the prediction correctness (Φ), as well as the final threshold (ϵ), are returned.

What Tanksale [7] found was that the best combination of epochs and dropout rates was 100 and 20%, and the final threshold (ϵ) was 0.3. It produced a maximum prediction correctness (Φ) of 0.9825, which is impressive considering the lowest was 0.9301. These three values played an important role in the next two sections.

4.3.2 Formula A Testing

Now, the hyperparameters and threshold (ϵ) from ’Pre-Training’ are used to test how well the prediction algorithm can predict anomalies. Remember, from section 4.2, that Formula A is a ratio of the vehicle’s engine speed and accelerator pedal position; anomalies are created by changing the engine speed. Three forms of anomalies are created, one form triples the engine speed, one form doubles the engine speed, and one form has 1.5 times the engine speed. For the array {1%,2%,5%,10%,15%,20%}, each value is a percent of the Formula A dataset that is manipulated to be anomalous. For each percent of anomalous data, tests are conducted on all 10 cars to predict the three different forms of anomalies.

Tanksale [7] documented the results of each test in tables such as Figure 3. Figure 3 shows the Accuracy and the False Positive Rate (FPR) for all 10 cars. The Accuracy is how often the prediction algorithm predicted results correctly. The FPR is how often the algorithm predicted an anomaly when there wasn’t one. Figure 3 reflects the test where 1% of the data is anomalous in the form of tripled engine speed.

A more detailed illustration of a car’s results can be shown through a Confusion Matrix, which is given in Figure 4. The Confusion Matrix of Car 5 is of the same test, where 1% of anomalous data is in the form of tripled engine speed. In Car 5, the prediction algorithm accurately predicted 14 anomalous, marked as malicious in the matrix, frames and predicted 1,642 normal frames. The algorithm predicted 28 anomalous frames when they were actually normal, and predicted 3 normal when they were actually anomalous. Accuracy from Figure 3 reflects the amount of correctly predicted normal frames, whereas FPR reflects the amount of predicted anomalous frames when they were actually normal.

Overall, the IDS performed well with a reasonably low FPR. Tanksale [7] did note a critical issue with Formula A; it was too simple and training the LSTM with Formula A data reflected that. An attacker could evade the IDS if they alter both the engine speed and accelerator pedal position while keeping the ratio the same. The anomalous frames would essentially be disguised as normal frames in a successful attack. The IDS also wouldn’t be able to detect attacks on other subsystems of the vehicle since Formula A only trained it to monitor two. Formula B was the next step in improving the IDS and making the CAN harder to penetrate.

4.3.3 Formula B Testing

Similar tests are conducted using Formula B data but with new goals. The main goal was to make sure Formula B data

	Accuracy	FPR
Car 1	0.9855	0.0140
Car 2	0.9864	0.0126
Car 3	0.9780	0.0209
Car 4	0.9789	0.0202
Car 5	0.9816	0.0168
Car 6	0.9864	0.0124
Car 7	0.9807	0.0189
Car 8	0.9868	0.0118
Car 9	0.9802	0.0188
Car 10	0.9803	0.0187

Figure 3: Results with 1% of x3 Engine Speed Anomalies [7]

improves the algorithm's accuracy and resiliency to attacks on multiple CAN measurements. There was also an effort to see if anomaly placement throughout the dataset effected prediction accuracy. Two sets of tests were done to ensure reliable results. The first and second set of tests both inserted anomalies into the dataset to test different placement patterns, but the CAN measurements manipulated were different in the first versus the second.

For the first set of tests, the longitudinal acceleration, accelerator pedal position and brake position were changed to create anomalies. The longitudinal acceleration was increased 1.5 times its original values, the accelerator pedal position was doubled, and the brake position was set to zero. For one test, the anomalies were inserted randomly throughout the test data; this was the pattern of insertion used in the previous section. In another test, the anomalies were inserted adjacent to each other in one random location. In a final test, 50% of anomalies were inserted adjacent to each other in a single random location while the rest where inserted adjacent to each other in a different random location.

In the second set of tests, the wheel speed, lateral acceleration and steering angle were changed. The wheel speed was changed to 0.7 times its original values, the lateral acceleration was changed to 1.05 times its original values, and the steering angle was increased by one degree. The same patterns of insertion from the first set of tests were also tested in this set.

Tanksale [7] found conflicting results in tests with random anomaly placement. In both sets of tests, inserting anomalies randomly in the dataset actually showed to decrease accuracy and increase FPR. While it only altered the values by about 0.1 on average, the increase in FPR could make a big difference when handling hundreds of thousands of frames, and it's concerning to move forward with Formula B if it decreases the algorithm's overall prediction accuracy. However, Formula B allowed the prediction algorithm to detect much more. With Formula A, the algorithm couldn't monitor more than two CAN measurements at a time and was easily manipulated. Formula B allowed the algorithm to monitor 10 different CAN measurements and detect more complicated anomalies with minor loss of accuracy.

Where randomized placement decreased accuracy and increased FPR, patterned anomaly placement boosted the accuracy and decreased FPR. In such tests, Formula B allowed the prediction algorithm to perform better. As seen in section 3, attacks on the CAN involve injection patterns in order to trick CAN mechanisms. So, training the prediction algorithm with such patterns or similar ones may reflect

		Actual	
		Malicious	Normal
Predicted	Malicious	14	28
	Normal	3	1642

Figure 4: Confusion Matrix of Car 5 with 1% Anomalies [7]

more realistic attack scenarios. With these results and considerations, Tanksale [7] concluded that Formula B did in fact train the prediction algorithm better than Formula A did.

5. CONCLUSIONS

The CAN has a data link layer vulnerability that can be leveraged in different ways with off-the-shelf technology. Experiments discussed in section 3 have shown that the attacks on the CAN are straightforward because CAN protocols have no way to differentiate between a node with faulty behavior and an attack. In section 4, an IDS was discussed that can detect anomalies with acceptable accuracy and FPR. While the CAN vulnerability is well known, the IDS is not fully finished or ready for deployment and has yet to be field tested. While the idea of the decision engine has been decided, the functionality still needs to be implemented and tested. Better data formulas for training the prediction algorithm in the detection engine also need to be created [7]. Despite the continued development needed, the IDS shows promising results. There exists a promising solution to combat this difficult problem and it will only get better with time.

6. ACKNOWLEDGMENTS

Thank you to Elena Machkasova for your support in this process. Thank you to Ariel Cordes for taking the time to read and review my paper.

7. REFERENCES

[1] J. Brownlee. Difference between a batch and an epoch in a neural network, July 2018.

[2] S. Hajj, R. El Sibai, J. Bou Abdo, J. Demerjian, A. Makhoul, and C. Guyeux. Anomaly-based Intrusion Detection Systems: The requirements, methods, measurements, and datasets. *Transactions on Emerging Telecommunications Technologies*, 32(4):e4240, 2021.

[3] P.-S. Murvay and B. Groza. DoS attacks on Controller Area Networks by fault injections from the software layer. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*, ARES '17, New York, NY, USA, 2017. Association for Computing Machinery.

[4] M. Phi. Illustrated guide to LSTMs and GRUs: A step by step explanation, Sep 2018.

[5] M. Phi. Illustrated guide to Recurrent Neural Networks, Jun 2020.

[6] G. Rodriguez-Navas, J. Jimenez, and J. Proenza. An architecture for physical injection of complex fault scenarios in CAN networks. In *EFTA 2003. 2003 IEEE Conference on Emerging Technologies and Factory Automation. Proceedings (Cat. No.03TH8696)*, volume 2, pages 125–128 vol.2, 2003.

[7] V. Tanksale. Anomaly detection for Controller Area Networks using Long Short-Term Memory. *IEEE Open Journal of Intelligent Transportation Systems*, 1:253–265, 2020.

[8] C. Watterson. *Controller Area Network (CAN) Implementation Guide*. Analog Devices, 10 2017. Rev. A.