April 2018

# Implementing Cryptographic Hash Functions on CAN Bus

Joseph B. Asante
*Worcester Polytechnic Institute*

Tasnim Rahman
*Worcester Polytechnic Institute*

Timothy Ignatius Reuter
*Worcester Polytechnic Institute*

Valerie Moore
*Worcester Polytechnic Institute*

MQP-ASEC-AW1

# Implementing Cryptographic Hash Functions on CAN Bus

A Major Qualifying Project

Submitted to the Faculty of

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree in Bachelor of Science

in

Electrical and Computer Engineering

By


Tasnim Rahman

Tim Reuter

Joseph Asante

Valerie Moore


Date: 04/26/18

Project Advisor:

Professor Alexander Wyglinski, Advisor

# ABSTRACT

This report focuses on documenting the process and results of efforts made to use light weight encryption to secure the Controller Area Network (CAN) in automotive vehicles. As vehicles have become more complex they have opened the CAN to a multitude of different attacks. These attacks range from simple malfunctioning hardware to attacks across vehicle to vehicle communication. These new opportunities for attack have created a need in the automotive world for an easily implementable and cheap security option for the CAN.

This project focused on using light weight encryption to secure the network by allowing for message verification which mitigates many different types of network attacks. To evaluate the effectiveness of the encryption a test bench was created using the CAN harness from a 2015 Chevy Impala. This contained four microcontrollers, or Electrical Control Units (ECU), to emulate the behavior of a vehicle's CAN. The test bench allowed for the development of message verification using Secure Hashing Algorithms (SHA). A Graphical User Interface was also developed to track messages being sent across the CAN, and to compare how many attacks on the system have been prevented. The results of this project support the idea that using simple cryptography both message ID spoofing attacks and timing-based attacks can be mitigated. Additionally, through a series of performance tests with specific controls for the test set it was determined that the use of simple cryptography did not impact the systems performance. This report contains the information regarding the CAN, ECUs, SHA and the methodology used to develop and test a secure network using these resources.

# ACKNOWLEDGEMENTS

We would like to thank our advisor Professor Alexander Wyglinski, as well as Hristos Giannopoulos, Kyle Robert Scaplen, Jacob Thomas Grycel, and MITRE, whom without this MQP would have been impossible.

# TABLE OF CONTENTS

# LIST OF FIGURES

# List of Tables

# List of Acronyms

- **CAN** – Control Area Network
- **ECU** – Electrical Control Unit
- **SHA** – Secure Hashing Algorithm
- **GUI** – Graphical User Interface
- **LIN -** Local Interconnect Network
- **OBD-II** – On-Board Diagnostics
- **OSI** – Open Systems Interconnection
- **RSA** – Rivest-Shamir-Adleman
- **DSA** – Digital Signature Algorithm
- **PRNG** – Pseudorandom Number Generator
- **LED** – Light Encryption Device are
- **COM –** Communication
- **V2V –** Vehicle-to-vehicle
- **SAE –** Society of Automotive Engineers

# EXECUTIVE SUMMARY

Automotive vehicles are going through a technological change as microcontrollers and real-time devices have become more available. More automotive vehicles than ever before have complex systems that allow for internet connection and vehicle-to-vehicle (V2V) communication. While these advancements have opened many doors for the development of new features in cars ranging from safety to convenience, they also allow for more potential attacks as the systems get ever more complicated. The CAN is a main communication standard in automotive vehicles that has increased its complexity with each new electrical system added to a vehicle. The importance of studying ways to secure the CAN system in vehicles is important to prevent the manipulation of vehicle data and ensure that critical safety features are not compromised. The proposed approach of using light-weight encryption to implement message verification allows for existing system designs to be more secure without the cost of adding additional hardware in a production line.

The goal of this project was to implement a light-weight cryptographic method that would allow for the mitigation of multiple types of attacks across the CAN system. The attacks included message ID spoofing and a bit copying message timing attack. These attacks were chosen because they are the most common attacks that happen on communication networks and can have large impacts on the performance of vehicles if left unmitigated. Once the attacks to mitigate were chosen the success of mitigating those attacks needed to be measured in a test bed that replicated the behavior of a vehicles CAN system. This replication of CAN behavior was created using a complete CAN harness from a working vehicle and the implementation of multiple microcontrollers that would send and receive messages. The visualization of this project was a key

aspect in determining the effectiveness of the study. These project goals would allow for the development of a mitigation solution that has been thoroughly vetted.

To achieve these goals the project was broken down into multiple milestones that needed to be completed to ensure each stage of the project was correct. The first major milestone for this project was to complete the functional programming of a simple CAN network using microcontrollers to create behavioral attacks on the simple network. The next stage of this project was to implement an encryption method that would allow for message verification. Finally, the implementation of software devices onto a real CAN harness and the use of a Graphical User Interface showed that the implemented system does ensure that attacks are mitigated.

Each milestone of this project was successful with only minor changes to the projected timeline. The implementation of microcontrollers was quickly sped along by MITRE Cooperation providing use with Teensy controllers that rested on pre-spun boards with useful materials such as the MCP2551 CAN transceiver and a multitude of user interface devices. This added help greatly sped up the process of reaching a point of implementing programmable controllers. The project goal of creating a simple CAN network was implemented quickly with the use of open source Arduino libraries, the provided hardware, and a series of trouble shooting sessions. The next goal that was met was the implementation of SHA1 and SHA256 as light weight encryption methods for message verification. Implementing attacks was done by creating a controller that copied a message bit for bit and then sent the message after a time delay and creating a controller that sent a message with either incorrect data or an incorrect message ID. These attacks were shown to be mitigated on a full CAN harness with multiple controllers creating network behavior. Finally, a

Graphical User Interface was implemented using Python that allowed for transparency in the electrical system.

The following images display the proof of results, both by verifying the CAN messages being sent and showing the performance impact on the CAN system. The first success for the development of the results was the completion of a simple CAN system that could send one message from one ECU to another. The following image, Figure 1, shows the successful transmission of a CAN High and CAN Low message as a real-time signal.

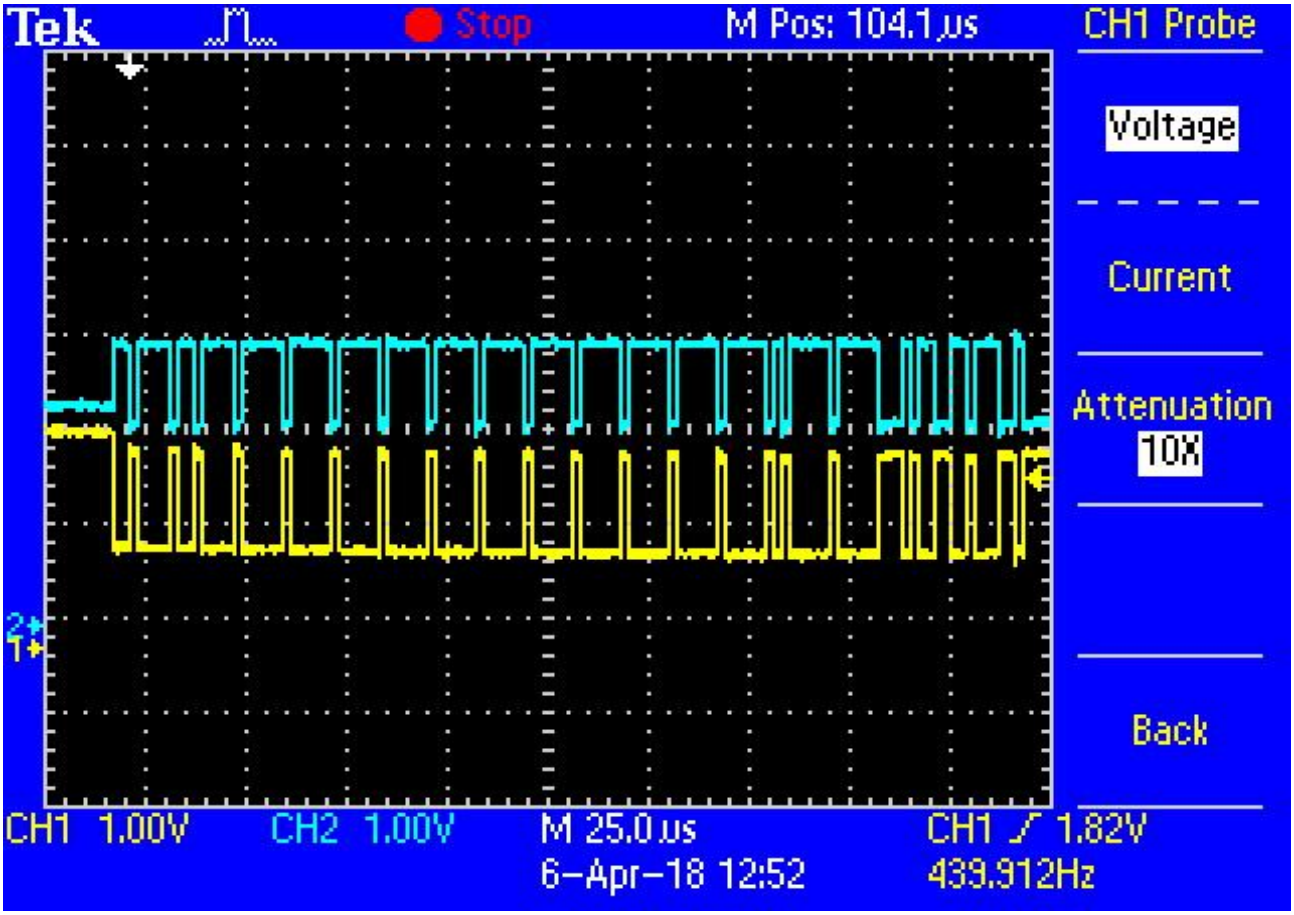



*Figure 1: Image of Successful CAN message being sent across the CAN High (Blue) and CAN Low (Yellow) signals. The message was sent across 20 Gauge wire and wass captured using an*

*oscilloscope. CAN High and CAN Low are opposite waveforms, as CAN High is pulled to a plus 3.3 Volts and CAN low is pulled to ground*

The next step was to create a system that handed messages from one ECU to the next ECU, and then responded. The following image, Figure 2, displays how the ECUs in a system created a real-time environment that handed multiple messages across the CAN system successively.

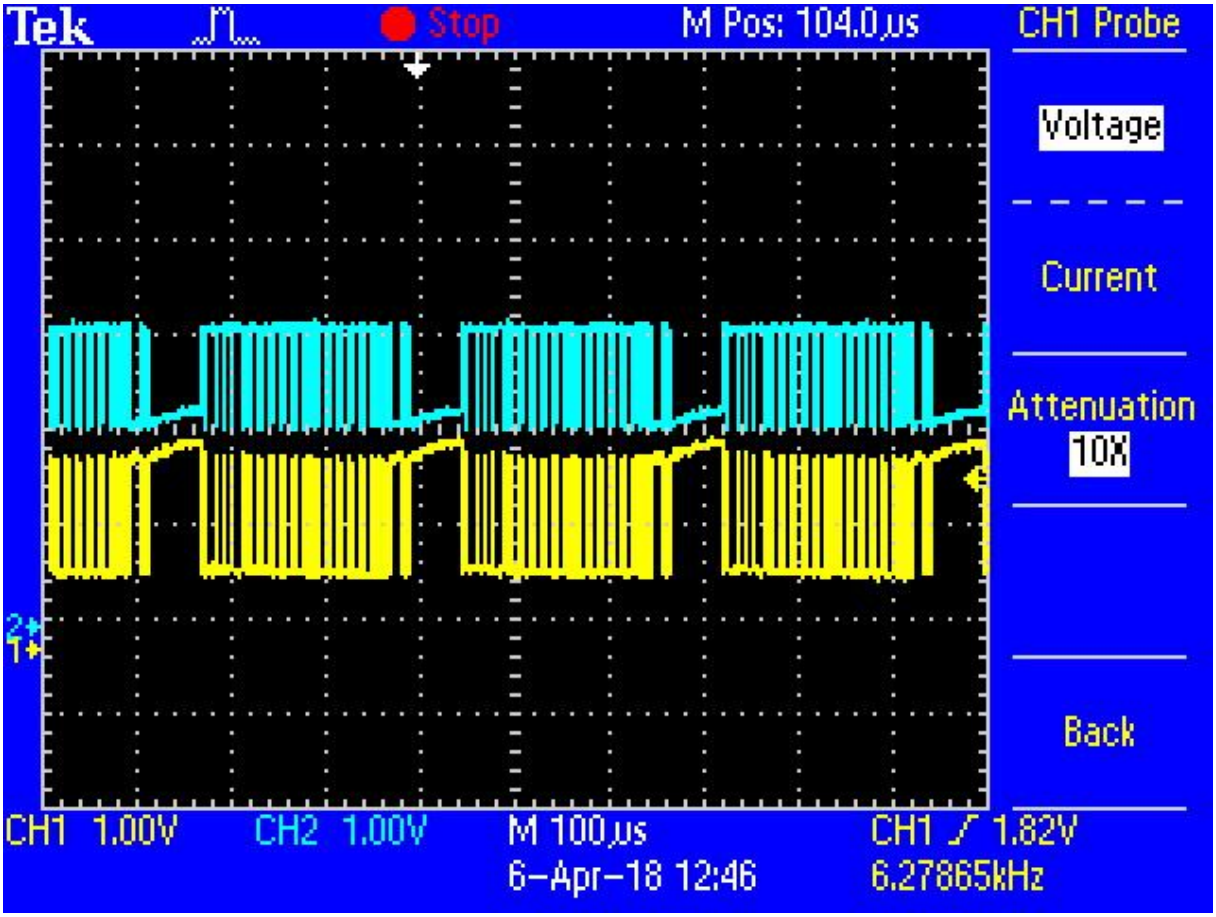



*Figure 2: Multiple CAN Message System Displayed signals in real time, CAN High (Blue) and CAN Low (Yellow) show matching waveforms and are sent at regular timing intervals*

The results of this project show that implementing low cost light-weight encryption methods is possible on a vehicle's CAN system. While the project has yielded good results in

mitigating the implemented attacks more work can be done to further verify the usefulness of light-weight encryption. The future work for this project should focus on the development of more robust CAN behavior with a multitude of controllers emulating different systems in a vehicle. The continued investigation into different implemented attacks will provide a test bench that can anticipate more problem scenarios. Finally, the implementation of more encryption methods will allow a comparison that will help find the most effective and secure message verification process.

# CHAPTER 1: INTRODUCTION

According to the US Department of Transportation (Federal Highway Administration), there was an estimated 268 million cars in the U.S in the year 2016 which is about 1.8 million more than there was in 2010 [1]. Additionally, Figure 3 shows that the total number of cars keeps increasing from year to year. With this increase, vehicles also continue evolving with new technologies. These changes have allowed for the continued changes to cars in the various ways it communicates: vehicle-to-vehicle, vehicle-on-infrastructure, vehicle-to-internet and communication with external devices like smartphones and tablets. These forms of vehicular communication are carried out using network technologies such as Local Interconnect Network (LIN), Control Area Network (CAN), FlexRay and Local Interconnect Network (LIN). CAN is popular for medium speed applications, FlexRay is used in high speed applications, and LIN is mostly used in low speed applications [2]. However, these technologies are vulnerable to security issues that can threaten the networking systems, the vehicle and the vehicle's safety.

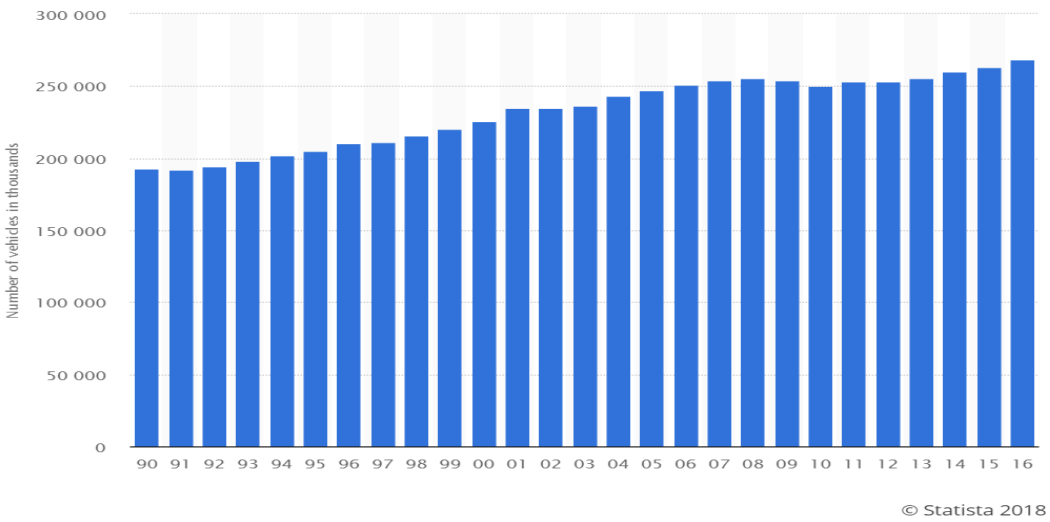



*Figure 3: Number of vehicles registered in the United States from 1990 to 2016 (in 1,000s). Graph shows an increasing pattern in the number of cars which is projected to increase.*

CAN is the most widely used form of communication technology in vehicles [3]. It was developed by Bosch as a “multi-master, message broadcast system that specifies a maximum signaling rate of 1 megabit per second (bps)” [4]. Additionally, CAN is a serial communications bus defined by the International Standardization Organization (ISO) that was originally introduced to the automotive industry so that wiring harnesses can be replaced with a two-wire bus [5]. Although CAN bus can transmit and receive messages between their controllers, or Electronic Control Units (ECUs), with fault confinement and error containment, such as error detection, error signaling and self-monitoring, it still is vulnerable to security threats [6]. Studies have shown that accessing a single node on the network can allow the vehicular network to be compromised [7].

Imagine your vehicle braking while you are stepping on the accelerator on the highway due to someone hijacking the CAN system of your vehicle. Imagine that your windows roll down when you step on the brakes instead of your car stopping. These are just some unfortunate events that could happen if the CAN in vehicles is compromised [8] [9] [10]. Hence, there is an increasing research into ways to make it more secure with better performance. Vehicular CAN system can be compromised either wirelessly or through direct contact with the vehicle system [6]. Many researchers have demonstrated attacks that proves that one can gain access to a vehicle’s internal communication system [11]. With the increase in wireless connectivity, wireless attacks on the CAN could very soon be on the increase. Attacks through direct contact with the system may be unintentional on the part of the driver which makes it even more dangerous. A driver may buy a car accessory which may cause the system to be compromised once it is installed in the vehicle. There is also the potential of an attacker installing a hardware device in the vehicle to affect the correct functioning of the CAN system.

The purpose of this project was to find means of implementing security for the CAN that is used in vehicles. ECUs are the main devices used in the CAN system of most modern vehicles. To make CAN in vehicles more secure, the functionality of the ECU was analyzed to see if it could be altered so that it is able to either accept or reject messages being transmitted. Since the CAN should respond as quickly as possible, the most optimized algorithm was sought so that the response time of the ECUs will not be drastically affected.

This project had a series of challenges faced when attempting to create the secure CAN network. The fist challenged faced on this project was a lack of proper documentation for hardware provided to the team for both an ECU and a CAN Harness. The lack of documentation delayed the project goals and development slightly past what was initially proposed. Additionally, challenges were faced when first trying to implement a simple CAN system. At first the setup of the terminating resistors in the CAN system caused some problems with the expected digital waveform. Additionally, the team experienced problems implementing the Flexcan library from TeachOP because the programming platform already had a Flexcan library defined [12]. While the project had challenges and setbacks the results show that these challenges did not impede the creation of a secure CAN system.

The results of this project indicate that the use of cryptographic methods is capable of mitigating network level attacks on the CAN network. Specifically, the project was able to develop a test bed that implemented ECUs throughout a physical CAN harness and used Secure Hashing Algorithms (SHA) to develop a message verification procedure on the CAN network. This verification process was able to stop ID spoofing attacks and time delayed attacks. After running a series of tests to determine the behavior of the network under an attack using SHA it was

necessary to measure the impact of using this message verification process. After running the performance test, it was determined that the use SHA in the system had little to now performance impact on the system. It was also determined that in a system that was under attack the use of SHA improved the performance of the system, allowing less time to be spent processing faulty messages. Over all the use of SHA as a message verification process promises to be a useful tool in the automotive security environment.

This report describes our approach to implementing cryptographic hash functions to secure the CAN bus. Chapter 2 of this report provides an overview of the CAN bus, its ECUs and their networking functionalities. This chapter also explains cryptographic methods that could be implemented to provide security on the CAN. Chapter 3 defines how we determined the final design for our approach to implementing a security algorithm on the CAN and ECUs. Chapter 4 describes the implementation of this proposal. The results of the implementation, conclusions and suggestions for other research areas that could be explored in the future are discussed in Chapter 5.

# CHAPTER 2: Overview of ECUs and Cryptography

This chapter gives insight into what ECUs and the CAN System are and how they create an in-vehicle network that controls different functions. In addition, the two main types of message security or encryption in cryptography, traditional methods such as RSA and DSA and lightweight methods such as PRNG and Hash Functions are also compared and discussed in this chapter.

## 2.1 In-Vehicle Network: CAN Bus System and ECUs

The CAN system is the most widely used bus standard for vehicular communication technology. It allows a way of connecting the digital systems, which includes controls, actuators, and other nodes, in real-time programs within a vehicle for effective communication [13]. Figure 4 shows a diagram of the basic wiring of a two-wire (high and low) and half-duplex CAN system with different nodes, or ECUs [5].

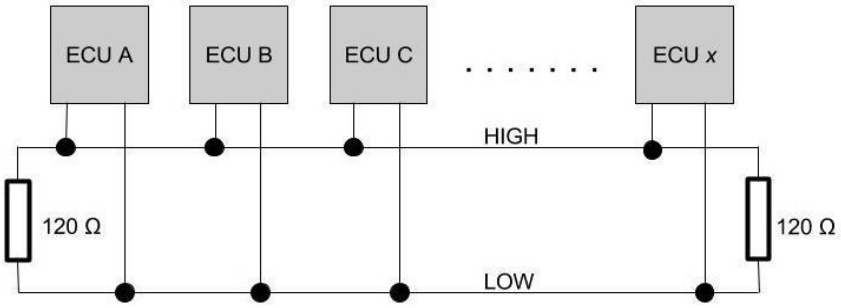



*Figure 4: Basic CAN System: ECUs are wired together from high end to low end in a circuit with 120 Ohm resistors on either end.*

The CAN bus system is a network for ECUs allowing them to send and receive messages. It is a "serial networking technology for embedded solutions" and allows the use of high levels of security for protection [5]. The CAN system is real-time and has a baud rate of 1 Mb/s.

Additionally, messages are sent by priority where a node with a lower message ID has a higher priority.

Figure 5 displays the ECUs and gateways (which connect to other buses and wireless access) within an in-vehicle network. The CAN bus is connected to other in-vehicle systems that can include the Local Interconnect Network (LIN) and FlexRay. The On-Board Diagnostics (OBD-II) port allows users to read and write data and install software to the ECUs. Additionally, the CAN bus enhances the OBD-II system by reducing wiring requirements and implementing fast and more efficient communication between data bus [13].

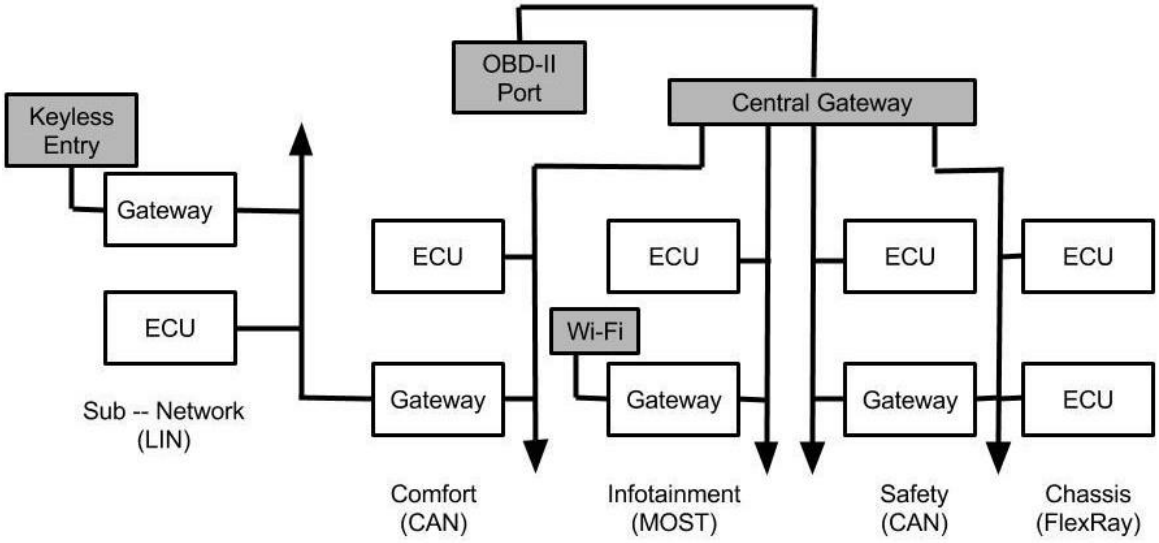



*Figure 5: CAN System within a Vehicle: The CAN System contains ECUs and Gateways that are in network with the rest of the vehicles network which include the LIN Network, MOST network and Flex Ray network.*

CAN communications data follows the Open Systems Interconnection (OSI) model [13]. The CAN System specification is divided into different layers which include: object layer, transfer

layer, and physical layer. The object layer filters messages and statuses. The transfer layer carries out transfer protocols such as fault confinement; error detection and signaling; message validation; transfer rate and timing; and more. The physical layer handles actual transfer of the bits between the different mediums through signal level and bit representation [14] and the communication between ECUs within a CAN system uses this specification. Figure 6 shows the different layers and the function of each layer for the CAN System specification.

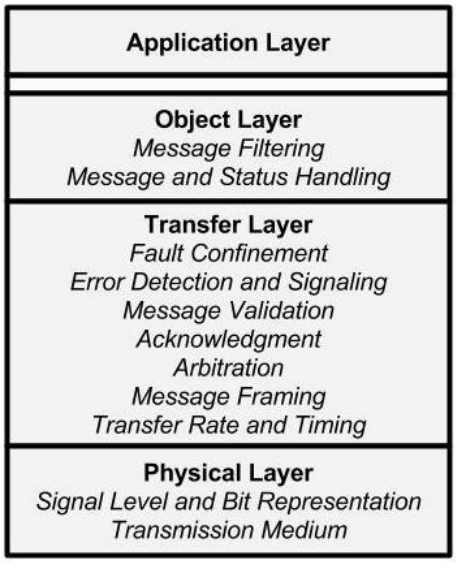



*Figure 6: Layers of CAN System Specification: The CAN System has an Application Layer, an Object Layer, a Transfer Layer and a Physical Layer that are all required within the network.*

ECUs are microcontrollers that require real-time data transmission and information collection and processing. Modern luxury cars contain about 70 ECUs with up to 2,500 messages transferred between them [13]. The microcontrollers used within a CAN system can be programmed for communication using the following platforms: Arduino, Teensyduino, Raspberry Pi, and Intel Galileo.

The Arduino platform is widely used and low in cost. It is efficient and readily available for use with many libraries. However, its weaknesses include execution speed, memory resources, and it has a low tolerance for data traffic. Teensyduino shares the same properties as Arduino but is also compatible with Teensy controllers that contain Teensy ports. The Raspberry Pi platform is also readily available and low in cost. It is easy to use with an extensive library and is efficient with more RAM and storage compared to other microcontrollers. The Intel Galileo, also compatible with the Arduino platform, is more for industrial use and "expand[s] native usage and capabilities beyond the Arduino shield ecosystem" [5]. It is more expensive compared to the other platforms mentioned in this section and is not compatible with all boards. However, it is easy to use with a high-level library.

The microcontrollers that can be used for the ECUs in the CAN system include the MCP2551, MIKROE-2228, ADM00617 or a controller integrated with Teensy. The MCP2551 requires an integrated transceiver and is compatible with the Arduino, Raspberry Pi, and Intel Galileo platforms. It comes with Arduino CAN Shields, which are needed when setting up a CAN network with ECUs. It has an SPI interface and its average cost is $2.50. The MIKROE-2228 controller interfaces immediately with Arduino. It is a controller integrated with Arduino Shield 1 and costs $24.00 per unit. ADM00617 is also a controller integrated with Arduino Shield and interfaces immediately with Arduino. However, it costs $40.00 per unit. The Teensy integrated controller is of no cost because it can be obtained from MITRE. The controller of it is connected to Teensy. Table 1 below provides information on each of the CAN controllers.

*Table 1: Comparison of four CAN controllers that can be used for the ECUs, which are the MCP2551 controller, the MIKROE-2228 controller, the ADM00617 controller, and the Teensy-integrated controller. All of them are compatible with Arduino.*

| Controller | Controller integrated with Arduino CAN Shields | Controller integrated with Arduino Shield 1 | Controller integrated with Arduino Shield 2 | Teensy integrated controller |
|---|---|---|---|---|
| **Part ID** | MCP2551 | MIKROE-2228 | ADM00617 | N/A |
| **Cost** | $2.50 per unit | $24.00 per unit | $40.00 per unit | Free |
| **Setup** | Requires integrated transceiver | Controller integrated with Arduino Shield 1 | Controller integrated with Arduino Shield 2 | Controller integrated with Teensy |
| **Interface** | SPI Interface with Arduino, Intel Galileo or Raspberry Pi | Arduino | Arduino | Arduino and Teensyduino |
| **Resources** | Technical Data Sheet | Technical Data Sheet and Company Support | Technical Data Sheet, Application examples and paper, extensive Company Support | Source codes and libraries, Technical Data and extensive Company Support [MITRE] |

The CAN bus system is vulnerable to security issues. ECUs can be attacked to allow a hacker to take control of the ECU and the in-vehicle network. These attacks can occur through different points which include the OBD-II Port and wireless communication protocols [6]. The OBD-II Port allows access to all the ECUs within the in-vehicle system. Wireless access includes weak security implementations within keyless entry systems and other modes of wireless access. However, there are measures that can be taken to mitigate these attacks.

## 2.2 Cryptography: Mitigation of Attacks

Types of attacks on the CAN bus system include man-in-the-middle attack, data interception, timing-based attacks and message spoofing attacks. These attacks can be mitigated

using traditional as well as lightweight cryptographic methods for security. Common examples of traditional cryptographic methods are Rivest-Shamir-Adleman (RSA) and Digital Signature Algorithm (DSA). Both RSA and DSA are also public-key cryptography methods, which means they require asymmetric algorithms for encryption [15]. Hash functions, Pseudorandom Number Generator (PRNG), Butterfly and Light Encryption Device (LED) are examples of lightweight cryptographic methods. These methods are public-key cryptography methods as well, but they are lighter and require less resources to run and implement, but still have almost the same quality of security, specifically for low power applications [16]. To find out which one of these two to use, it was necessary to compare them. This comparison was done in terms of cost, complexity of implementation, platform, performance and availability of resources as shown in Table 2.

*Table 2: Comparison of Traditional Cryptography and Lightweight Cryptography based on cost/resource requirement, complexity of implementation, performance, platforms on which they can be implemented and the availability of resources to help implementation.*

| **Category/Type** | **Traditional** | **Lightweight** |
|---|---|---|
| **Cost/Resource requirement of platforms** | High | Low |
| **Complexity** | High | Low |
| **Platforms** | Servers/desktops, tablets and smartphones | Embedded devices, RFID and sensor Networks |
| **Performance** | High | High |
| **Availability of Resources** | Many years of resources and research papers available | IoT provides lots of resources and case studies. |

Cost refers to the resource requirements that must be met to implement the algorithm as far as cryptography is concerned [17]. Examples of these resources in hardware implementations are

gate area, gate equivalents, or logic blocks while register, ROM and RAM usage are example of software resources. An implementation of the traditional cryptographic methods requires more of these resources than lightweight cryptographic methods. Traditional cryptography focuses on providing high levels of security without really considering the resource requirements [18]. Hence, traditional cryptography is costlier than lightweight cryptography.

A traditional cryptographic method, such as RSA and DSA, uses complex operations through asymmetric algorithms, making it difficult for unauthorized users to have access to the keys [9]. It is due to this that traditional cryptographic algorithms require more resources as stated above and become "more demanding in terms of computing power" [16]. The complexity of its operations and algorithm makes it computationally expensive and means the encryption will require many rounds to encrypt [19]. However, traditional public-key cryptography methods are very secure and are still used in most modern technologies and applications. Thus, lightweight cryptographic methods are introduced because require a less resources and computing power demand by requiring less complex operations while still providing the public-key secure encryption. It should be noted that although a less complex may mean compromising the integrity of security, there is continued research on lightweight cryptography from the academic community to make it better and produce more efficient algorithms. For example, the Secure Hash Algorithm has many different renditions of it which is the result of improvement upon each one that is breached. Like most modern technologies and applications, it is being updated every day.

The traditional cryptographic methods, due to their complexity and resource requirements are mostly used on servers, desktops, tablets and mobile phones. Each of these platforms have, to some extent, the ability to handle complex algorithms and will perform better compared to a

platform such as a microcontroller which has limited resources, such as low power to the controller. To be able to implement security on platforms or devices such as embedded systems, microcontrollers, RFIDs, and sensor networks, lightweight cryptography is used. This type of cryptography method uses less resources but has good performance because it uses the resources efficiently.

In cryptography, there is a tradeoff between performance and amount of resources needed. Most of the time, the amount of resources required may be reduced to have greater performance [20]. Since the traditional cryptographic methods are used on different platforms (which have more resources) than lightweight cryptography, they both perform very well on their respective platforms. If traditional cryptography were used on the platforms that require lightweight cryptography, such as a microcontroller, there will be an issue of performance because the platform is resource constrained. For example, such a platform would not have the required processing power to process all the complex algorithms that the traditional cryptography uses. On the other hand, using lightweight cryptography for platform that requires traditional cryptography such as a server will lead to a compromise of the desired integrity [19].

The final comparison done involved the availability of existing resources, such as source code and algorithms that will aid in implementing the cryptographic method. Since traditional cryptographic methods have been around for years now, there have been many years of research into it. Lightweight cryptography is more new than traditional cryptography and does not have as much research, but there is also a wide range of resources out there.

## 2.3 Chapter Summary

In this chapter, the CAN system and the ECUs within the network were explored. Essentially, the CAN system is a network that wires and connects ECUs, allowing them to send and receive messages. The controllers require real-time data transmission and information collection and processing. Controllers were also compared based on cost, setup, interfacing and resources required. ECUs need to be programmed using a development platform or testbed. These interfacing platforms include Intel Galileo, Raspberry Pi and Arduino.

This chapter also discussed the potential hazards to a CAN system and its ECUs Two cryptographic methods were considered: the traditional methods and lightweight methods. They are both public-key methods of encryption, but traditional cryptographic methods are geared towards devices or platforms that require a lot more resources and demand in computing power due to the complex asymmetric algorithm. Lightweight cryptographic methods are used for resource constrained devices or platforms, such as microcontrollers, because they reduce the complexity of asymmetric algorithms.

# CHAPTER 3: PROPOSED APPROACH

This chapter proposes the implementation of a replica of the CAN system in a vehicle and adding a cryptographic algorithm that will provide security of the network. The ECUs on the CAN system in vehicles are prone to be corrupted through methods such as supply chain attacks or through external media that can compromise safety since corrupted non-critical systems have access to the CAN. The proposal outlines: The CAN system and its ECUs, attack implemented on the system and mitigation methods for those attacks and GUI platforms designed to monitor the CAN system, attacks and mitigation of those attacks. Additionally, this chapter shows the timeline and Gantt chart for completion of the project.

## 3.1 In-Vehicle Network: Proposal for CAN System

The ECUs chosen for our CAN system are the Teensy integrated controllers that were provided by MITRE at no cost. No further hardware work would be required besides of connecting the controllers because it comes integrated with an ARM controller and Teensy ports. The controller interfaces with Arduino and Teensyduino, which is a software add on for Arduino development studios that is used to program Teensy integrated controllers. It also utilizes the Flexcan library. Additionally, support is easily available from MITRE when working with them. Fortunately, there are many open source documentation and libraries to use with Arduino and Teensyduino.

A total of four Teensy controllers will be used and called ECU A, ECU B, ECU C and ECU D. Figure 7 below shows the basic circuit design of the CAN system. The ECUs are to be connected high end to low end using jumpers. The CAN system and ECUs will connect to the

computer and Teensyduino test bed through a micro USB. All programming and cryptography will go to the ARM controller of the ECUs. To terminate each ends of the CAN system, a 120 Ohm resistor is used on either end.

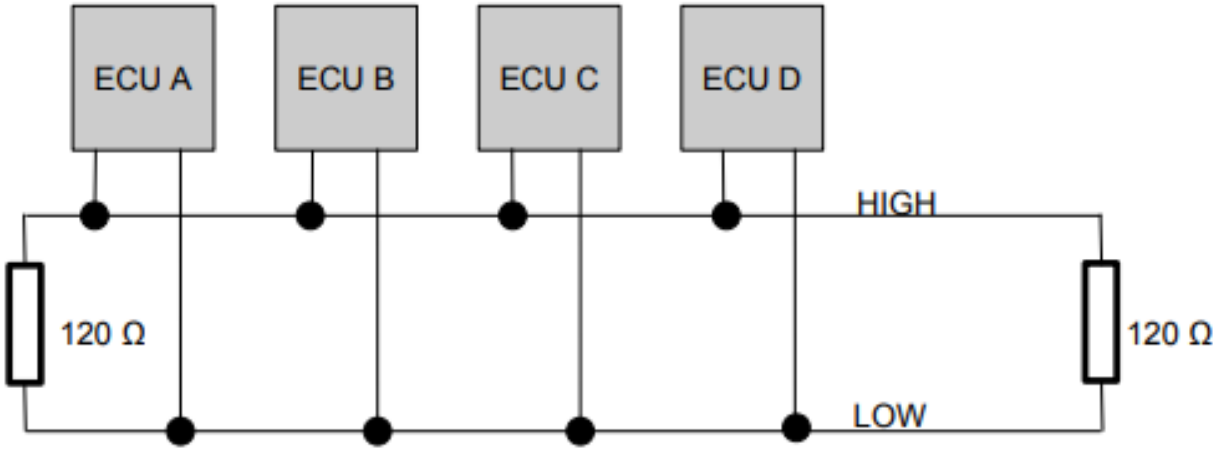



*Figure 7: Proposed CAN System: The CAN System will have four ECUs that are wired high end to low end with 120 Ohm resistors on either end.*

The ECUs will be sending and receiving messages as shown in Figure 8. ECU A will be sending normal (not spoofed) messages to ECU B. ECU C will be copying and spoofing messages from ECU A and sending them to ECU B as well. Finally, ECU D will be copying the messages of ECU A and sending it to ECU B without spoofing the messages. Ideally, ECU B will not be able to receive spoofed messages in the CAN system due to cryptography that will be targeting the ARM processor of the Teensy controller.

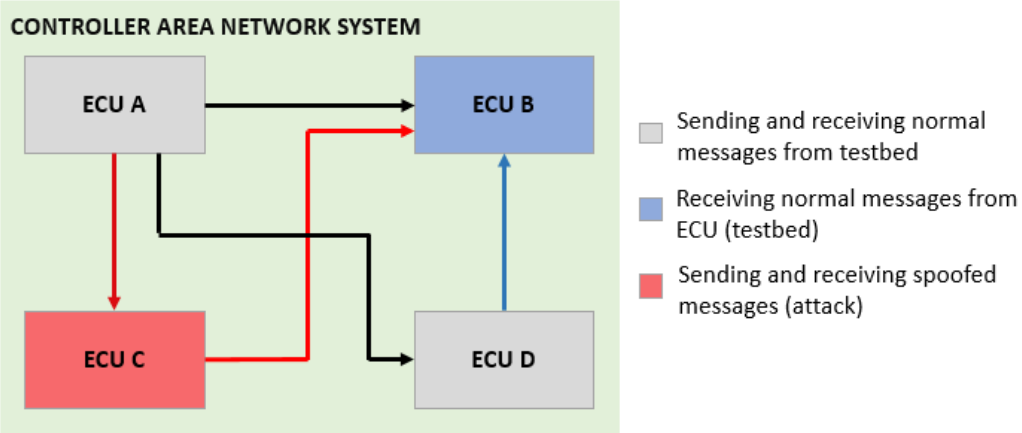



*Figure 8: Proposed ECU Network: Connections between ECU A, ECU B, ECU C and ECU D. Arrows indicate direction of messages being sent and received. The red component (ECU C) is the controller that is attacked and sends/receives spoofed messages. The blue component (ECU B) received normal messages. ECU A and ECU D both receive and send messages to and from the testbed.*

## 3.2 Cryptography: Proposal for Mitigation of Attacks

ECUs within the CAN system are vulnerable to supply chain attacks or attacks through external media. Therefore, these attacks must be mitigated to protect the CAN system and the vehicle. By using a message verification scheme for the message sent across CAN systems in cars we can implement some form of security to mitigate damage from potential intruders. This solution could be used against message spoofing and timing-based attacks. To create such a solution, traditional and lightweight cryptographic methods were considered as discussed in Chapter 2. Out of the two, lightweight cryptography was selected because it is designed to provide security for microcontrollers which is what was used to implement the ECUs. Using traditional cryptographic methods will require a lot of processing power and a lot of resources which the microcontroller does not have. In order words, lightweight cryptography provides efficiency of end-to-end

communication in that it allows lower energy consumption for end devices [21]. Hence, lightweight cryptography was chosen as the preferred method of ensuring security on the ECUs.

As stated in Chapter 2, there many types of lightweight cryptography methods. PRNG, LED and SHA cryptographic methods were compared to find one that will be convenient to implement. These algorithms were compared for their cost, complexity, efficiency and implementation as shown in Table 2 below:

*Table 3: Lightweight cryptography methods PRNG, LED and SHA are compared by cost, complexity, efficiency and implementation. PRNG and SHA are both free in terms of cost compared to LED but are not as efficient as LED. Although, all three methods are less complex to implement.*

| **Category/Type** | **PRNG** | **LED** | **SHA** |
|---|---|---|---|
| **Cost** | Free | Price per unit varies | Free |
| **Complexity** | Low | Low | Low |
| **Efficiency** | Low | High | Low |
| **Implementation** | Simplistic design allows for easy decryption. | Independent hardware allows for large complex encryption schemes to be used. | Simple implementation and a lot of open source libraries available |

After consideration of each of these lightweight cryptographic methods, Secure Hash Algorithm (SHA) was proposed as the preferred method of implementing security on the ECUs. SHA was chosen because it is free, simple to implement and has a lot of open source libraries. It was picked over LED mostly because LED is not free although it is somewhat more efficient that

SHA, and because LED may involve large complex encryption. SHA was also picked over PRNG because it does not require a decryption method. Since the ECUs are supposed to be responding at a very fast rate after receiving a message, a decryption stage after receiving the message would lead to lagging. SHA was used as a message verification tool so that the expected SHA digest (the hashed message) would be hard-coded into the ECU and checked against the message digest of an incoming message. This way, when the ECU receives a message, it will only allow the message whose digest matches the hard-coded digest in and disallow the message whose digest does not match.

SHA is an algorithm that is computationally infeasible to find a message which corresponds to a given message digest. It is also almost impossible for two different messages to have the same digest [22]. This occurrence is called collision attack and to achieve this will require a lot of work and machines with huge computational power. In 2007, CWI Amsterdam and Google successfully performed a collision attack against SHA-1 by publishing two different PDF files which produced the same hash. This successful effort required over 9,223,372,036,854,775,808 computations [23]. However, SHA-256 has not been compromised yet.

### 3.3 Data Analysis: Proposal for GUI

It was proposed that a GUI be designed to visualize and aid in tracking the performance of the ECUs, attacks and mitigation methods. The GUI proposed has two main parts and must be written in the Python programming language. The first part of the GUI must display message transmissions and indicate when an ECU/transmitted message has been corrupted. When the attack happens, the mitigation technique will be implemented so that the original message can be

transmitted. The second part of the GUI must have indicators for the type of the message being transmitted (whether correct or wrong message). Green will indicate a correct message and red will indicate that a wrong message is being transmitted and the ECU is being attacked. These two parts of the GUI come together to display the output of CAN interactions, messages between ECUs and the status of any ECUs/messages being attacked.

### 3.4 Expected Timeline: Gantt Chart

The project was completed in two academic semesters, from August 24 to May 1. This period was divided into four quarters and different goals were set for each quarter. Work on the ECUs communicating with each other and the implementation of SHA were mostly completed in the first two quarters and designing the GUI and connecting the system to the harness were completed in the last two quarters.

For the first quarter of the two semesters, the first step was to find the best solution to the problem at hand. This was done by the entire team after which the solutions obtained were discussed and a common one agreed on. The next thing done was to complete implementation of communication between at least two ECUs. At this same time, an attempt was made to implement a Secure Hash Algorithm (SHA) to suit the project.

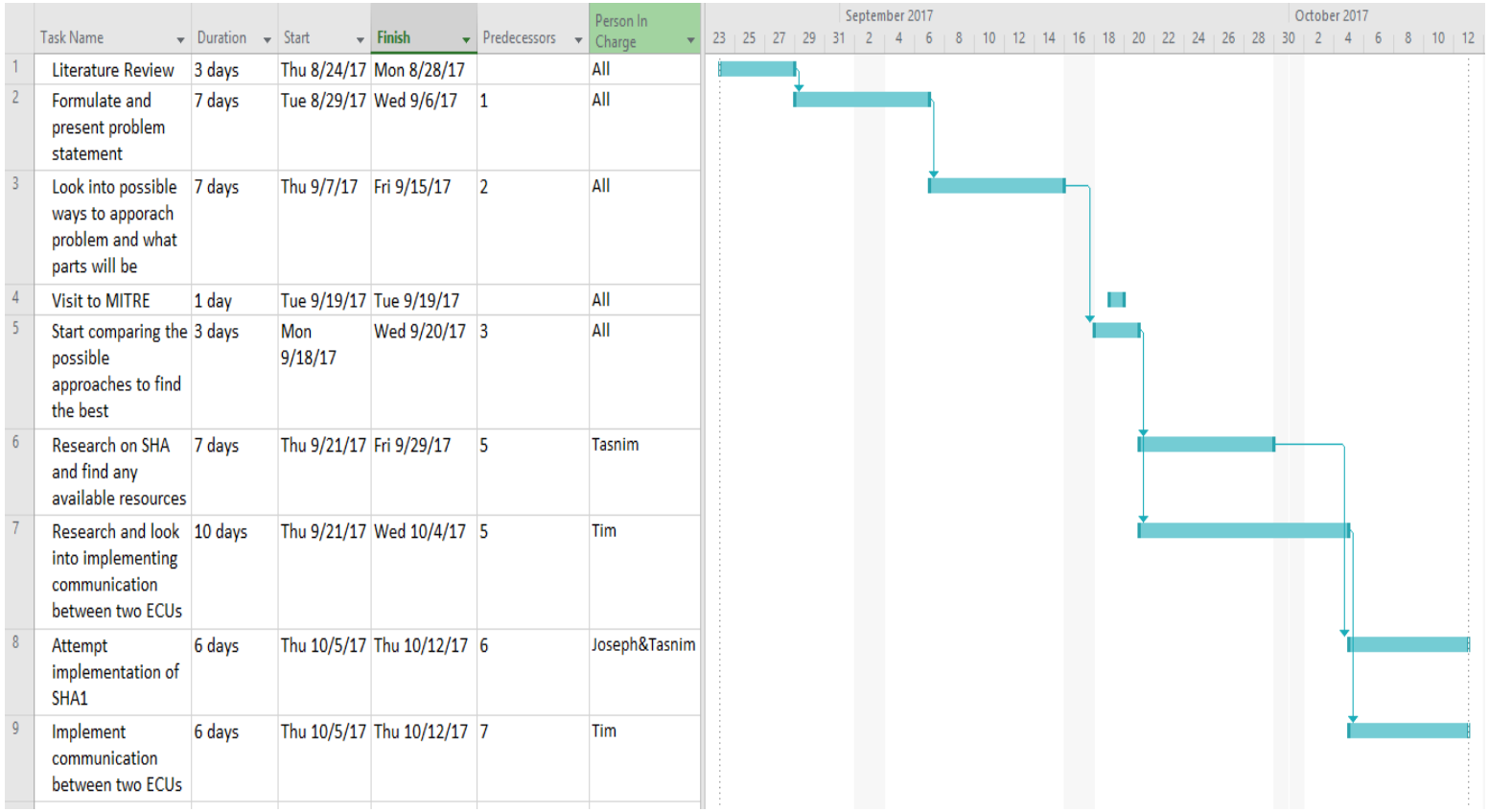
In the second quarter, SHA-1 was completely implemented and the implementation of SHA-256 was also done. An attempt was also made to add a third ECU to the network. This also involved implementing the message spoofing attack. The drafting of the project report was started in the quarter.

The third ECUs functionality was completed in the third quarter as well as the GUI. The harness was also explored and added to the network. After this, extensive testing of the system was done. At this same time, the draft of the report was being written.

In the fourth and final quarter, the fourth ECU was added to the network which means a timing-based attack was implemented. The rest of the quarter was dedicated to presentations, finishing up the report and writing an IEEE conference paper.

## 3.5 Chapter Summary

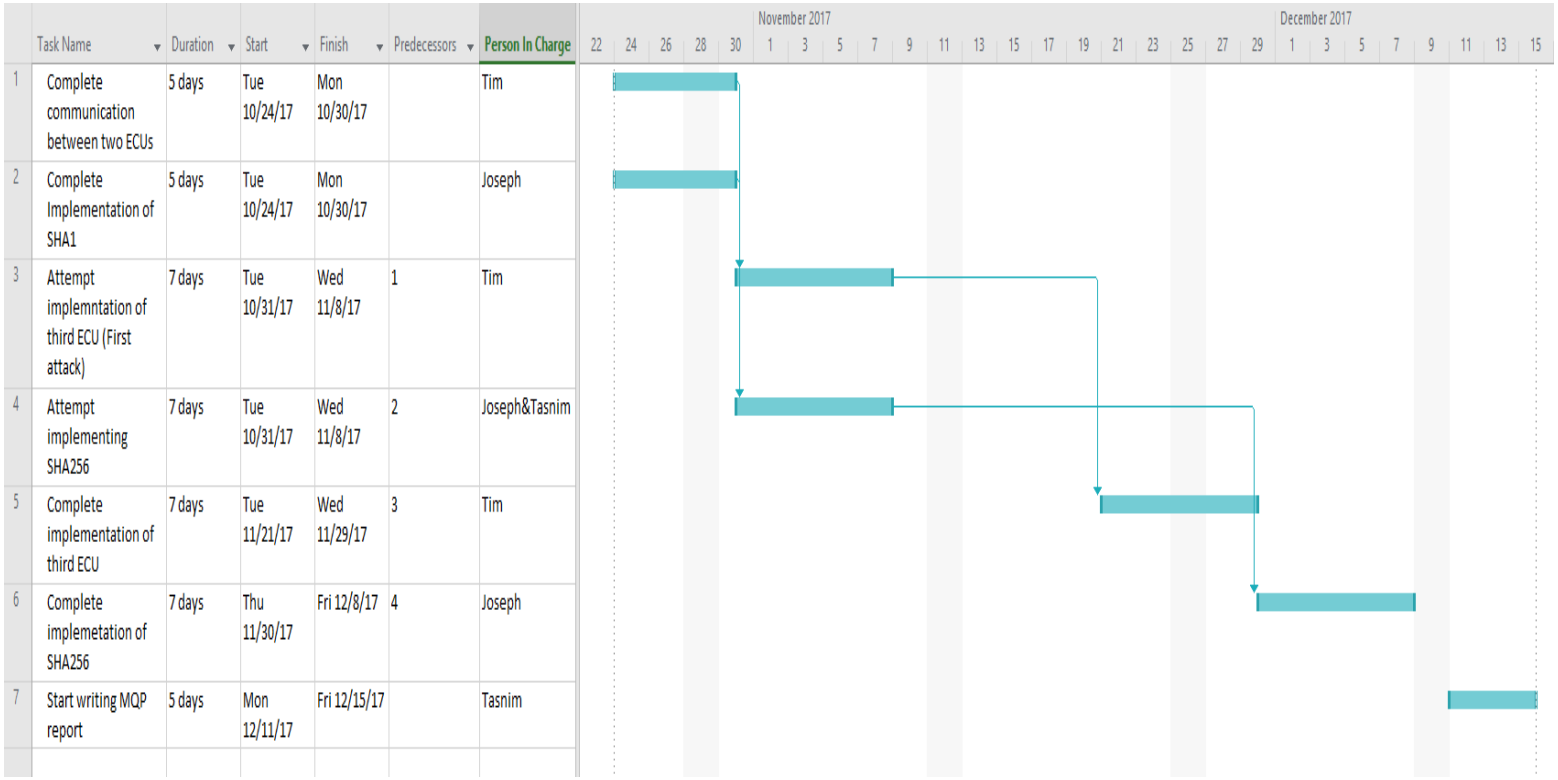
The proposal for the In-Vehicle Cyber Security project is to use SHA to provide some form of security for the CAN in vehicles. This will be done by hashing the messages that are passed along the CAN so that their message digests will be compared to a hard-coded expected message digest. The two types of SHA that will be used are SHA-1 and SHA-256 and it was expected that this solution would be used against message spoofing attacks and timing-based attacks. The cryptographic method will be implemented in a proposed CAN system that will consist of four Teensy integrated controllers that are controlled though an Arduino and Teensyduino testbed. A GUI was also proposed to be created and used to monitor the ECUs and the messages being sent.



| | Task Name | Duration | Start | Finish | Predecessors | Person In Charge |
|---|---|---|---|---|---|---|
| 1 | Literature Review | 3 days | Thu 8/24/17 | Mon 8/28/17 | | All |
| 2 | Formulate and present problem statement | 7 days | Tue 8/29/17 | Wed 9/6/17 | 1 | All |
| 3 | Look into possible ways to apporach problem and what parts will be | 7 days | Thu 9/7/17 | Fri 9/15/17 | 2 | All |
| 4 | Visit to MITRE | 1 day | Tue 9/19/17 | Tue 9/19/17 | | All |
| 5 | Start comparing the possible approaches to find the best | 3 days | Mon 9/18/17 | Wed 9/20/17 | 3 | All |
| 6 | Research on SHA and find any available resources | 7 days | Thu 9/21/17 | Fri 9/29/17 | 5 | Tasnim |
| 7 | Research and look into implementing communication between two ECUs | 10 days | Thu 9/21/17 | Wed 10/4/17 | 5 | Tim |
| 8 | Attempt implementation of SHA1 | 6 days | Thu 10/5/17 | Thu 10/12/17 | 6 | Joseph&Tasnim |
| 9 | Implement communication between two ECUs | 6 days | Thu 10/5/17 | Thu 10/12/17 | 7 | Tim |



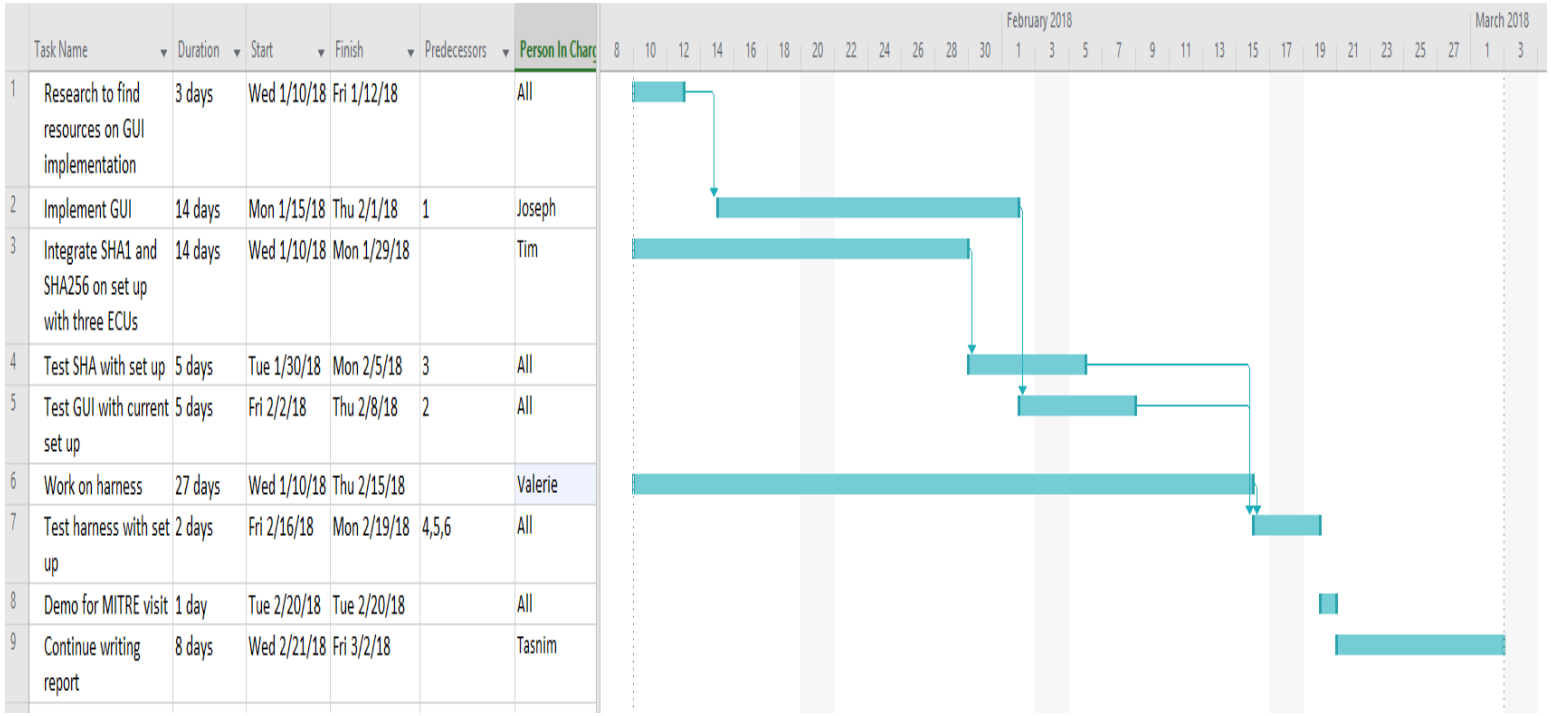
*Figure 9 Gantt Chart for A term. This shows the expected timeline of events leading to the implementation of SHA1 and the communication between the ECUs.*



| | Task Name | Duration | Start | Finish | Predecessors | Person In Charge |
|---|---|---|---|---|---|---|
| 1 | Complete communication between two ECUs | 5 days | Tue 10/24/17 | Mon 10/30/17 | | Tim |
| 2 | Complete Implementation of SHA1 | 5 days | Tue 10/24/17 | Mon 10/30/17 | | Joseph |
| 3 | Attempt implemntation of third ECU (First attack) | 7 days | Tue 10/31/17 | Wed 11/8/17 | 1 | Tim |
| 4 | Attempt implementing SHA256 | 7 days | Tue 10/31/17 | Wed 11/8/17 | 2 | Joseph&Tasnim |
| 5 | Complete implementation of third ECU | 7 days | Tue 11/21/17 | Wed 11/29/17 | 3 | Tim |
| 6 | Complete implemetation of SHA256 | 7 days | Thu 11/30/17 | Fri 12/8/17 | 4 | Joseph |
| 7 | Start writing MQP report | 5 days | Mon 12/11/17 | Fri 12/15/17 | | Tasnim |



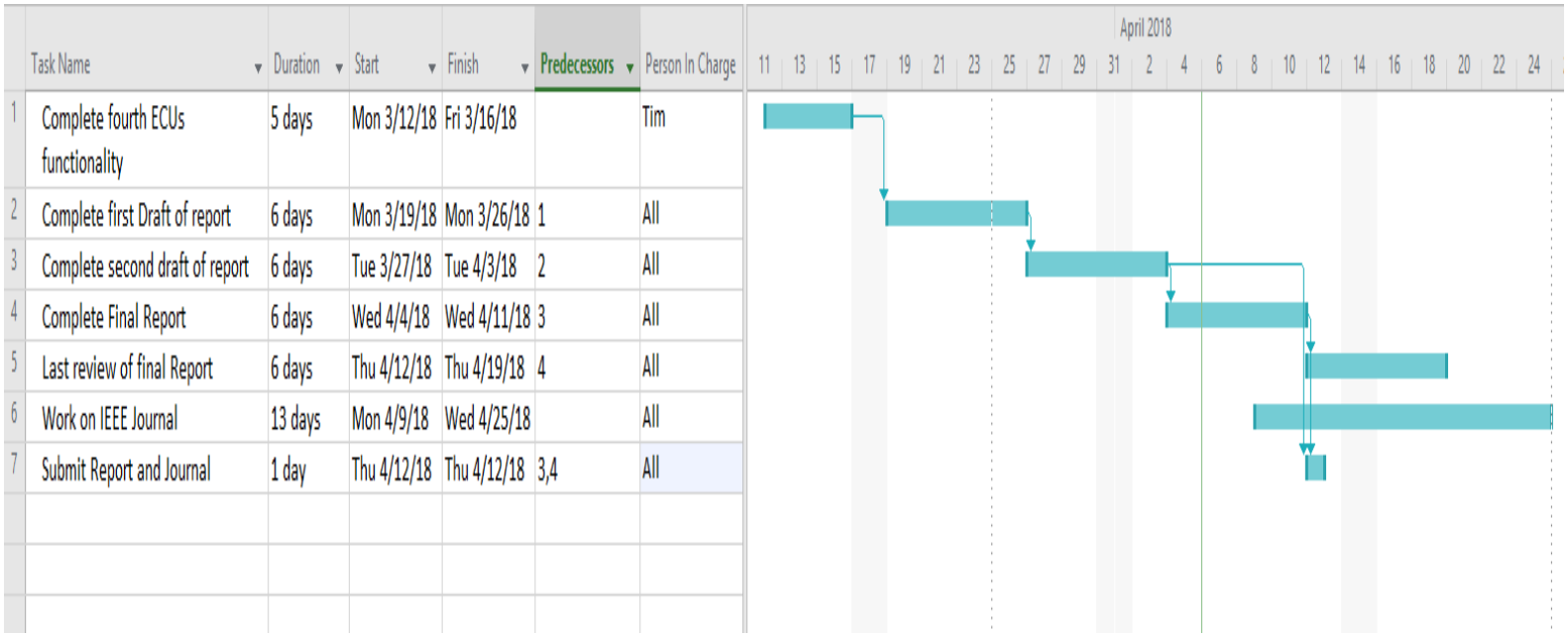
*Figure 10:Gantt chart for B term. This shows the expected timeline of events leading to the implementation of SHA256 and the third ECU. Writing of report was continued.*



| | Task Name | Duration | Start | Finish | Predecessors | Person In Charg |
|---|---|---|---|---|---|---|
| 1 | Research to find resources on GUI implementation | 3 days | Wed 1/10/18 | Fri 1/12/18 | | All |
| 2 | Implement GUI | 14 days | Mon 1/15/18 | Thu 2/1/18 | 1 | Joseph |
| 3 | Integrate SHA1 and SHA256 on set up with three ECUs | 14 days | Wed 1/10/18 | Mon 1/29/18 | | Tim |
| 4 | Test SHA with set up | 5 days | Tue 1/30/18 | Mon 2/5/18 | 3 | All |
| 5 | Test GUI with current set up | 5 days | Fri 2/2/18 | Thu 2/8/18 | 2 | All |
| 6 | Work on harness | 27 days | Wed 1/10/18 | Thu 2/15/18 | | Valerie |
| 7 | Test harness with set up | 2 days | Fri 2/16/18 | Mon 2/19/18 | 4,5,6 | All |
| 8 | Demo for MITRE visit | 1 day | Tue 2/20/18 | Tue 2/20/18 | | All |
| 9 | Continue writing report | 8 days | Wed 2/21/18 | Fri 3/2/18 | | Tasnim |



*Figure 11: Gantt chart for C term. This shows the expected timeline of events leading to the start and finish of the harness work, the completion of SHA256, and testing the set up.*



| | Task Name | Duration | Start | Finish | Predecessors | Person In Charge |
|---|---|---|---|---|---|---|
| 1 | Complete fourth ECUs functionality | 5 days | Mon 3/12/18 | Fri 3/16/18 | | Tim |
| 2 | Complete first Draft of report | 6 days | Mon 3/19/18 | Mon 3/26/18 | 1 | All |
| 3 | Complete second draft of report | 6 days | Tue 3/27/18 | Tue 4/3/18 | 2 | All |
| 4 | Complete Final Report | 6 days | Wed 4/4/18 | Wed 4/11/18 | 3 | All |
| 5 | Last review of final Report | 6 days | Thu 4/12/18 | Thu 4/19/18 | 4 | All |
| 6 | Work on IEEE Journal | 13 days | Mon 4/9/18 | Wed 4/25/18 | | All |
| 7 | Submit Report and Journal | 1 day | Thu 4/12/18 | Thu 4/12/18 | 3,4 | All |



*Figure 12: Gantt chart for D term. This shows the expected timeline of events leading to the implementation of the timing-based attack, writing and submission of the final report as well as IEEE Journal report.*

# CHAPTER 4: IMPLEMENTATION AND METHODOLOGY

This chapter details the method used to complete the goals in the proposed approach and discuss technical details of how the project was implemented. The methodology was broken down into six distinct steps each one with specific process details on how the steps were implemented. The first step of the project was designing the Hardware and Software requirements for encrypting a CAN system which lead nicely into the development of a simple CAN system. The next process that was completed in this project was the development of the ECU's behaviors, including the ability to implement and mitigate different types of attacks. To complete the behavior of each ECU the encryption methods of SHA-1 and SHA-256 were implemented to allow for attack mitigation. The last steps of this project were to create a complex CAN system on a vehicle's CAN harness, which involved extensive research and troubleshooting processes, and the development of a GUI so that the results of the project could be easily visualized.

## 4.1 Software and Hardware

To set up and implement our CAN system and mitigation methods, the controllers need to be acquired and set up with the appropriate test bed on the computer. The controllers used for the project are four Teensy integrated ECUs that are connected from high end to low end with 120 Ohm resistors on either end. The test bed to be downloaded and used for implementation are the Arduino platform and the Teensyduino software add on. All programming and encryption will go to the ARM controller of the ECUs. Additionally, all ECU/message transmission and status on whether they are being attacked or not will be displayed on the GUI platform on the computer. Figure 13 shows the connections between the computer and the CAN systems and the components

and network. The gray components which include the GUI, the Arduino and Teensyduino testbed, ECU A and ECU D are receiving and/or sending normal messages. The red component, ECU C, sends and received spoofed (corrupted) messages and is under attack. The blue component, ECU B, receives normal messages from ECU A and ECU D, which send and receive normal messages from the testbed.

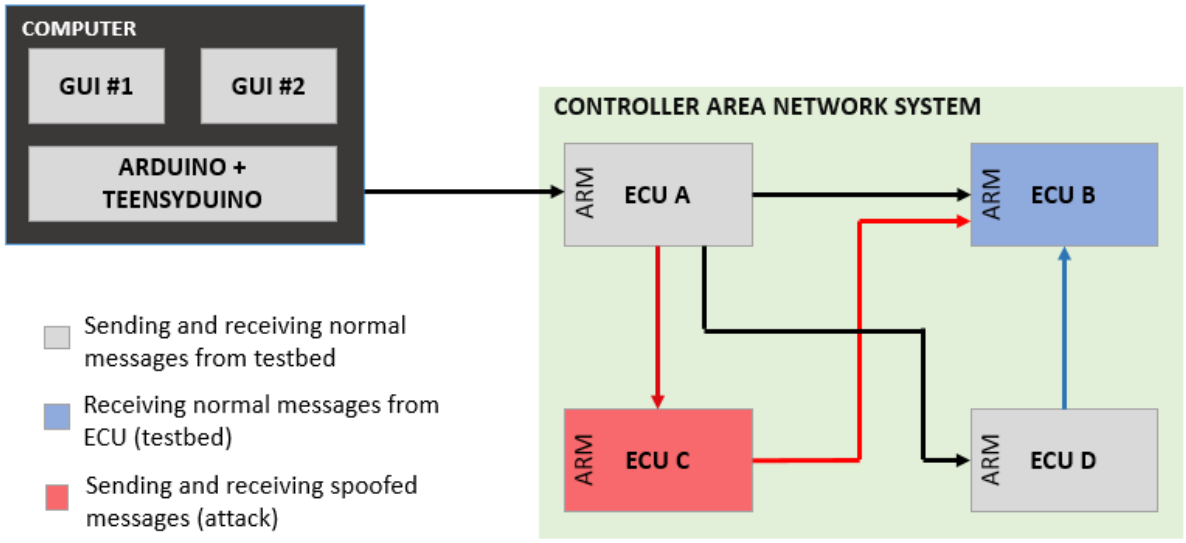



*Figure 13: Block Diagram of Computer connected to CAN System Network: Computer contains GUI #1, GUI #2 and the Arduino and Teensyduino testbed; CAN System contains ECU A, ECU B, ECU C and ECU D which are programmed and encrypted through their ARM controllers*

## 4.2 Design Methodology: CAN System

To build the CAN system, ECU A and ECU B were wired together and connected to the testbed first and can send and receive messages to each other. Figure 14 displays the preliminary stage of the CAN system, which is the connection between ECU A and ECU B. They are wired together from high end to low end and are programmed through the ARM controller. ECU A is

the only one connected to the computer and ECU B is wired to ECU A. This allows control of the preliminary CAN system using the Arduino and Teensyduino testbed.

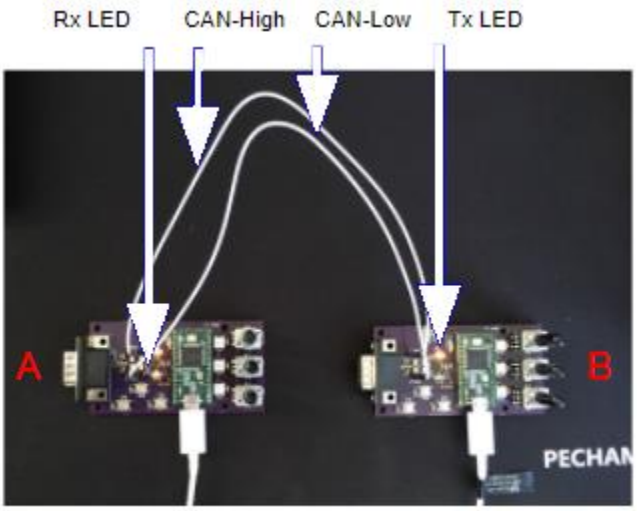



*Figure 14: : Preliminary CAN System: ECU A and ECU B are wired and connected for a preliminary network for building the rest of the CAN System (ECU A, ECU B, ECU C and ECU D); the LEDs indicate if messages are being sent and received (Rx and Tx LEDs).*

To properly implement this preliminary stage of the network and control it, Hristos Giannopoulos from MITRE, supplied the libraries that are used to implement the CAN communication between at least two Teensy CAN controllers.

### 4.3 Design Methodology and Implementation: ECUs

To implement an ECU that properly emulates the behavior of a CAN network a multiple chip system was implemented. The controller used in this chip is a Teensyduino with an ARM compiler that provided ease of programming by using the Arduino IDE as the development platform. The second microchip in this system was a MCP2551 CAN Transceiver, using a five

Volt digital logic it is capable of 500 kilobytes per second transfer speed. These two chips and the printed circuit board that the chips were built on were provided by the MITRE Corporation. Being provided the Chips and board greatly advanced the speed at which an ECU was successfully working in the CAN environment. The boards provided by MITRE also had a multitude of user interfaces such as potentiometers, switches and light emitting diodes. The final item included on the board is an output to standard DB-9 connector that allowed the ECU to connect to the Chevy CAN harness.

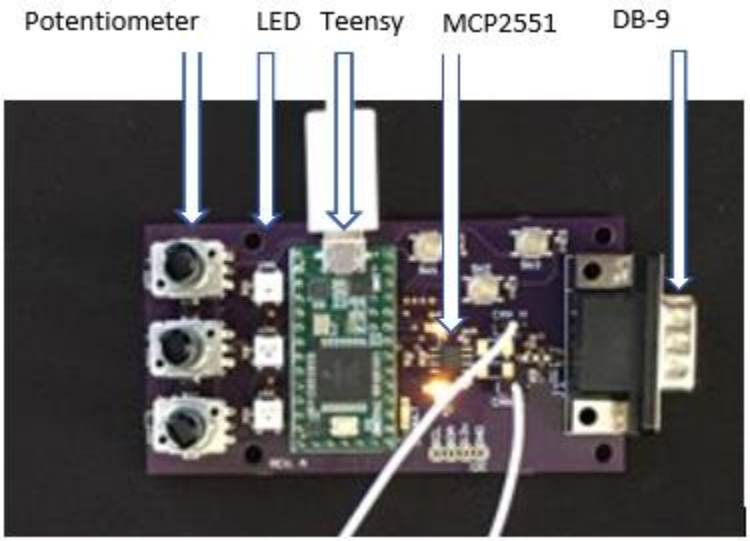



*Figure 15: ECU Layout: ECU contains a potentiometer, LED, Teensy integrated ports, MCP2551 controller and a DB-9 port*

Once the hardware was setup the next step was to test it by sending simple CAN messages. The Flexcan library provided by Teachop quickly allowed for the development of a simple CAN network by using the supported CAN library in Arduino for the MCP2551 [20]. Flexcan tied pins 3 and 4 to RX and TX pins respectively on the MCP2551. The software on the Teensyduino packs

a message with a three-byte ID and eight bytes of information. After the message is saved the write function provided by Flexcan sends the information to the MCP2551 where it sits in a FIFO buffer until the message has a chance to send over CAN based on the message ID [24]. Respectively the read function provided by Flexcan allows the Teensyduino to read from the FIFO buffer on the MCP2551, returning zero if no value has been saved. Once we had the Teensyduino setup with a proper packaged message two of the ECUs were connected to a properly terminating CAN system that allowed one ECU to send a message that was received by the second ECU.

The next step in development of the behavior of the ECUs was to use an encryption method to securely send messages between ECUs. Secure Hashing Algorithm-1 (SHA1) was implemented first because it takes up a small portion of programmable memory on the Teensyduino and number of instructions was low, so it could be handled by the ARM compiler on the Teensyduino. The final product of the SHA1 was a larger message then could be sent across CAN in a single message, so the message could not be directly encrypted. Instead a message that is received by an ECU is read from the FIFO buffer and the values in the message are used to determine what the message ID is and creates a value using SHA-1 that is based started from the predicted message ID and message values. An expected message digest which created from an expected message ID and message values is hard-coded into the receiving ECU. Comparing the two message digests of SHA1 allows the message to be accepted if they match and rejected if they do not match.

The next development for the ECUs was to utilize their user interface mechanisms. First was to implement the LEDs on the ECU to indicate the type of message received by the ECU. Once the pins for each LED on the ECU was verified and saved as static variables the FASTled library was used [25]. This allowed the ECU to make the LEDs red when rejecting a message and

green when receiving a message. The switches on the ECUs were also implemented by determining pins for each switch and then using Arduino interrupts to provide real time response when a user clicks the switch [26]. The switch was used to turn on and off the message verification system so that an attack could be demonstrated later.

The next step in the user display process was to attach to the CAN system a 2008 Rx Mazda instrument cluster [27]. This instrument cluster was expecting specific CAN messages with specific IDs to be received to display information on different interments such as a speedometer, fuel gage, and warning lights. Using open source tutorial and information from AutoZone the instrument cluster was determined to receive three can messages, with each bit in each message representing a different value to the display instruments [26]. Once the instrument cluster was running it provided an exciting way to include observers as to what is happening in the CAN system.

The next behavior for each ECU was to program the message spoofing and timing-based attacks. The message spoofing ECU simply took known values that could be represented and attached to the message the incorrect message ID. This message spoofing was prevented from sending data to the receiving ECU by the message verification process that utilizes SHA. The next development was the ECU with the timing-based attack. This ECU would wait for a message with a specific ID and then copy the entire message bit for bit. The message would then be sent out at a time delay which allowed it to successfully send messages across our CAN system. To prevent this type of attack it was necessary to develop in the SHA function to also check the output by adding a global timer variable to the bits being run through SHA. This allowed for any message

that was copied and then sent a few seconds later to be filtered out by the message verification process as well.

The final step for the development of the ECUs was to implement all four onto the CAN harness and attempt to run them together in the test bed. This step required the least work because of the preparation done before, once connected to the harness a few bits for the message to the instrument cluster had to be changed and some timing parameters in each ECU needed to be changed to get the full functionality expected of each ECU.

## 4.4 Design Modules and Implementations: Encryption Methods

Two versions of the Secure Hash Algorithm were implemented unto the ECUs: SHA-1 and SHA-256. Both algorithms were implemented so that they could be compared to see which one is faster and more efficient. SHA was implemented as a message verification algorithm on the ECU that was receiving messages from the other three. In this ECU, an expected value for the hashed message (message digest) is hard-coded. When the ECU receives a message, it uses this algorithm to hash the message and the created message digest is compared with the stored digest to see if it is a match. If it is a match, then the message is received and processed by the ECU. If the message digest of the hashed message does not match the expected digest, then there has been an attack and the message is refused. Figure 16 shows a flow diagram of how an ECU receives or does not receive a message depending on whether it is corrupted or not.

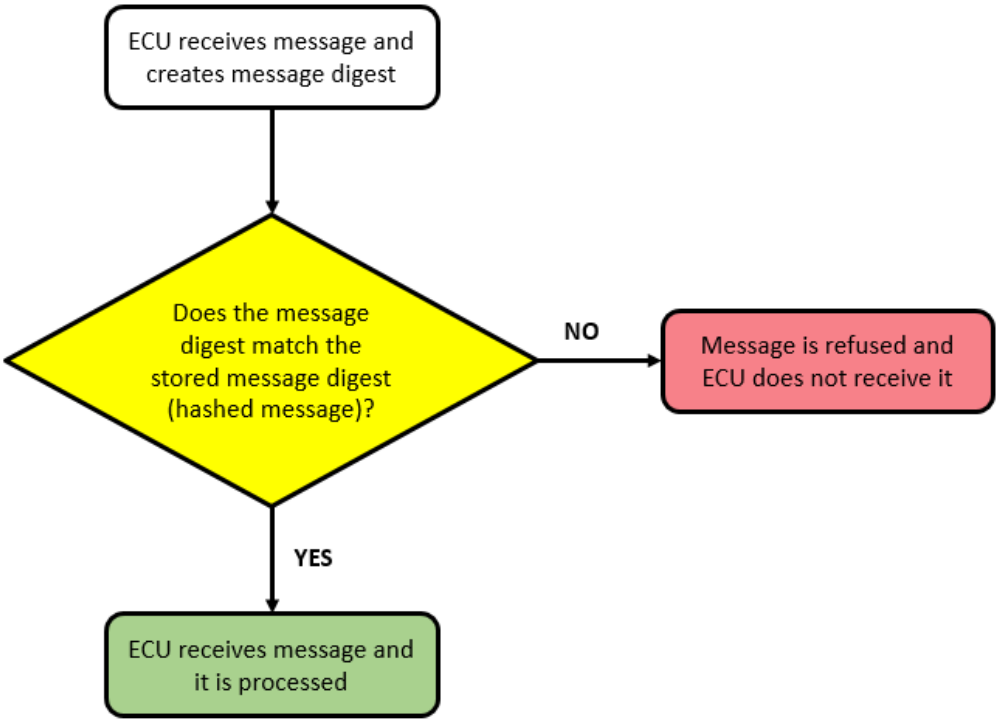



*Figure 16: Flow Diagram of Message sent to an ECU. If the message received is corrupted because it does not match the hashed message that is in the stored digest, the message will be refused and not processed by the ECU. If the message does match, it will be processed.*

The Secure Hash Algorithm takes in an input of any length within a range and gives out and output of a common length (message digest). Every message that is input in the algorithm will produce a totally different message digest, even if it is a single letter that was changed in the message.

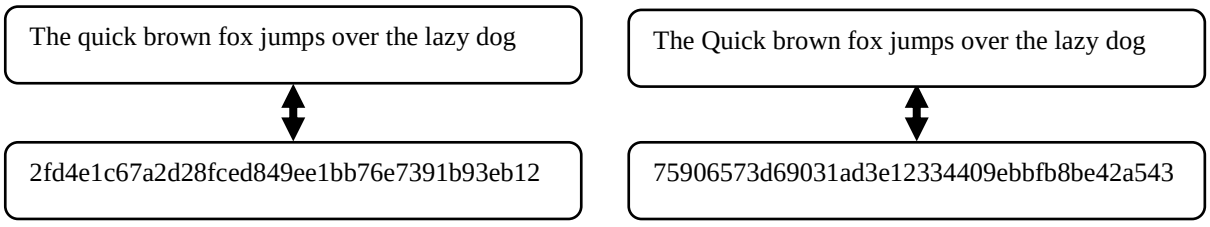



*Figure 17: An example of how a single character change results in a drastically different hash. Here, the letter q in quick was changed from lower to upper case, and the resulting hashes are significantly different.*

Figure 17 shows an example of a message and its hash value (SHA-1), and then a similar message with just one letter changed and its hash value (SHA-1). As can be seen, a change of a letter from lowercase to uppercase changed not just one character in the hash but the entire hash value.

The Simple Hash Algorithm-1 (SHA-1) is a type of cryptographic hash function that takes in an input of any length less than $2^{64}$ bits and gives an output of 160 bits long called a message digest. The Simple Hash Algorithm-256 (SHA-256) on the other hand takes an input of any length less than $2^{64}$ bits and gives an output of 256 bits long. The rendition of SHA-1 and SHA-256 used on the ECUs was based on a 2001 implementation of SHA-1 by Eastlake and Jones from Cisco systems and an implementation of SHA-256 by Brad Conte respectively. Modifications were made to their algorithms to suit the context of this project. The original code implemented just the hashing of a given message, but an additional step of converting that hash value to a binary was added to make it harder for anyone to predict the hard-coded expected hash value of the receiving ECU. Other changes were also made to make the code so that it would receive the type of message the ECU would receive. The functionality on the ECU was set up such that the received message is just passed through the SHA-1 algorithm and hashed. After this, the message digest is converted to binary and compared to the expected message (which is in binary form), and the message is either accepted or rejected.

The hashing of the message by SHA-1and SHA-256 is done in blocks. Each block of the message consists of 512 bits which are represented as a sequence of sixteen 32-bit words. Each word is a w-bit string that may be represented as a sequence of hex digits [28]. The algorithm takes the first 512 bits of the message, processes it and hashes it and then moves on to the next 512 bits. Hence, the length of a message received must be a multiple of 512. If this is not true for the message

received, the algorithm pads the message by appending it with a “1”, followed by a number of “0” s and then a 64-bit representation of the original length of the message. After the padding of the message, it is parsed into an N 512-bit blocks. Each block is processed through hash computations using operations such as bitwise logical word operations, addition modulo $2^W$, right shifting, rotate right, and rotate left to produce the message digest.

### 4.5 Design Module: Graphical User Interface (GUI)

A Graphical User Interface (GUI) was designed to visualize the sending and receiving of messages by the ECUs. It was designed using Python tkinter, which is Python’s most commonly used GUI package [21]. The GUI consists of two main sections as can be seen in Figure 18: the first section (left) displays information about messages being transmitted from the sending ECUs to the receiving ECU: a history of how many messages each has sent as well how many good and bad messages have been received by the receiving ECU. The other section (right) shows a historical graph of how many good, bad or no messages that the receiving ECU has received.

The left section of the GUI consists of the three sending ECUs: B, C and D. Each is followed by vertical arrows which show if an ECU sent a message a time or not. After a total of forty-eight messages have been sent, these arrows are deleted and recreated based on the new messages being received. There is also a counter at the bottom which shows how many good versus bad messages that has been received by ECU A. The counts should match the vertical arrows for how many messages were sent by ECU B, C and D. A good message means ECU B has sent a message and hence, will have a green arrow to indicate with the good messages count incremented by one. A bad message means that either ECU C or D has sent a message and will have a red arrow

to indicate with the bad messages count incremented by one. In Figure 20, a total of seventeen messages have been received. Out of the seventeen, six of them are good messages and eleven are bad messages.

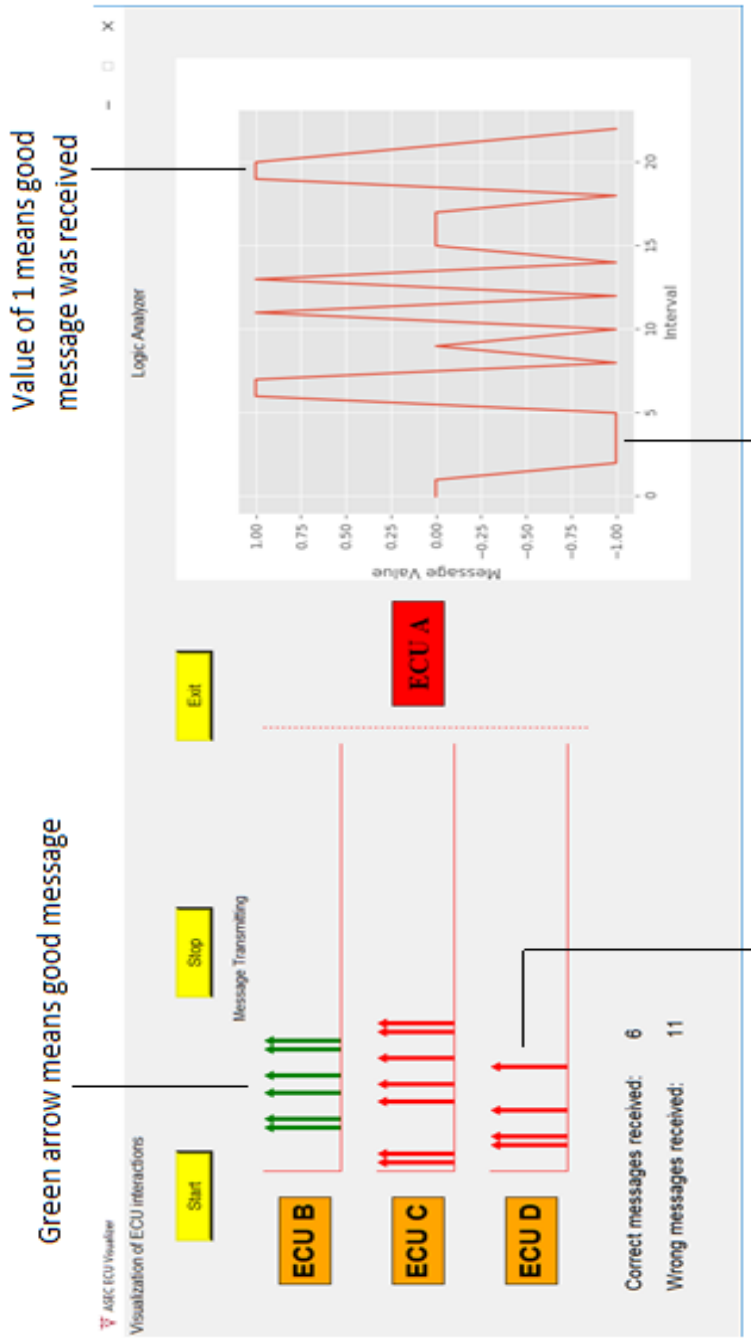



*Figure 18 Graphical User Interface for Visualizing and Analyzing Message Transmission: Left side shows upward facing arrows which represent the transmission of messages by the three ECUs. A green arrow and red arrow indicate whether the correct or incorrect message is being send respectively.*

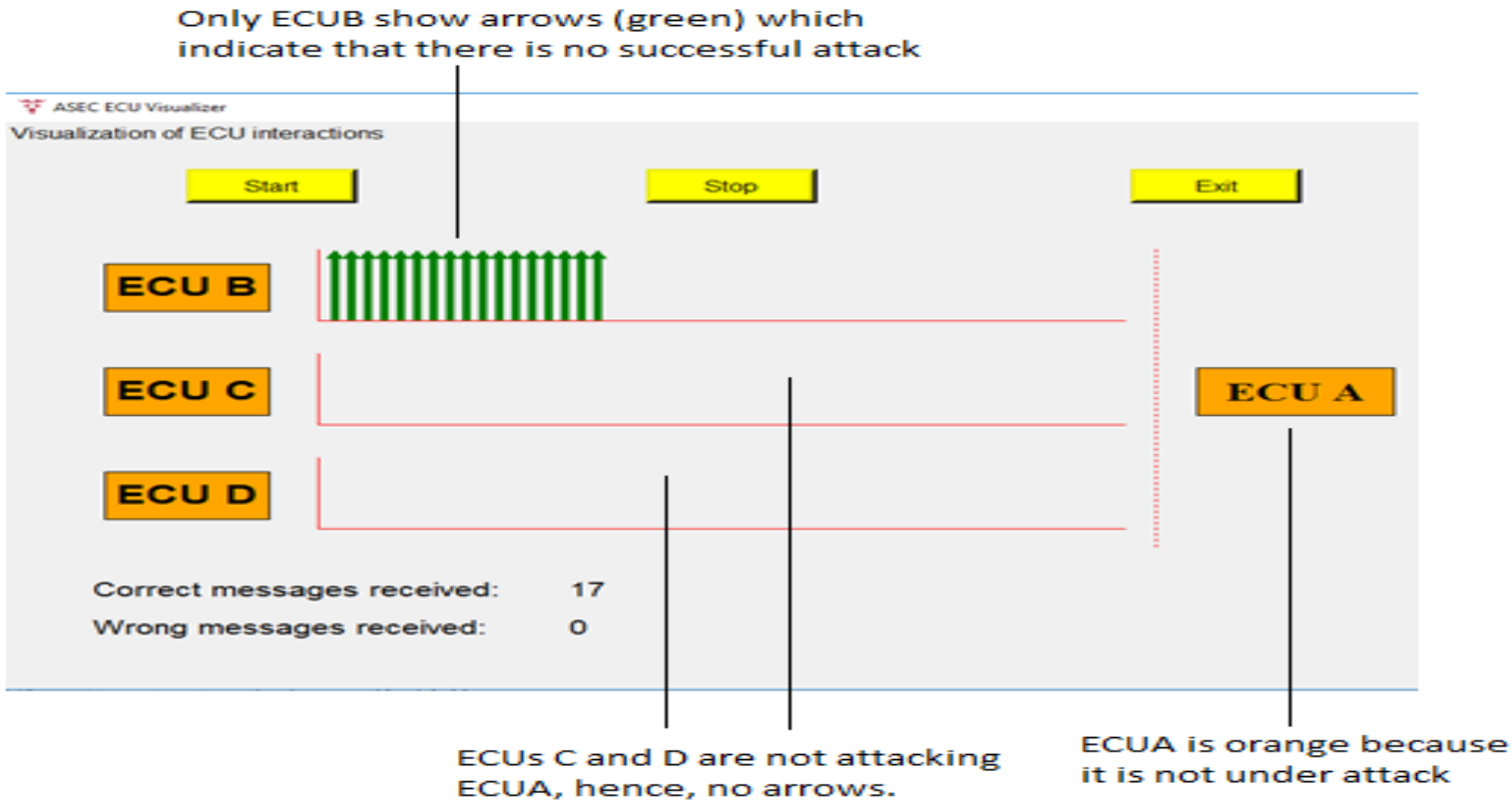



*Figure 19 GUI showing only good messages being transmitted across CAN system. Only correct messages are being sent so only green arrows can be seen for ECU B. ECU C and ECU D have no arrow indication because the message verification algorithm has been activated.*

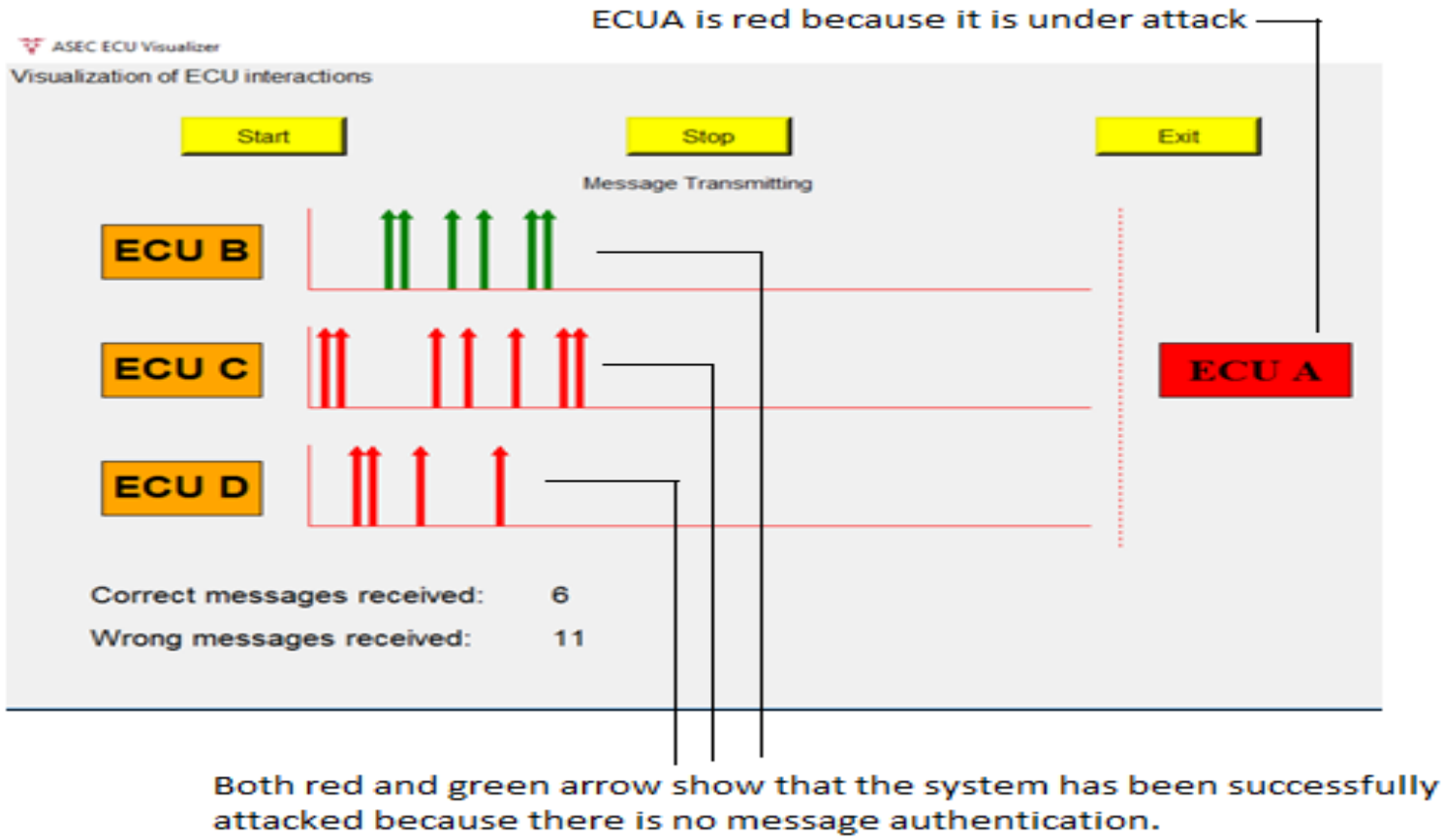



*Figure 20 GUI displaying good and bad messages being transmitted across the CAN system. This is the result of the absence of a message authentication algorithm which means that any kind of message can be transmitted.*

The right section shown in Figure 21 and 22, which has the graph, has three values for the vertical axis (one, zero, and negative one). A value of one means that a correct message has been received by ECU A, a value of zero means that no message has been received and a value of negative one means that a wrong message has been received. When the system is not under attack, the graph has vertical axis values of either zero or one to represent no message being transmitted or a message being transmitted respectively as shown in Figure 21. On the other hand, the graph

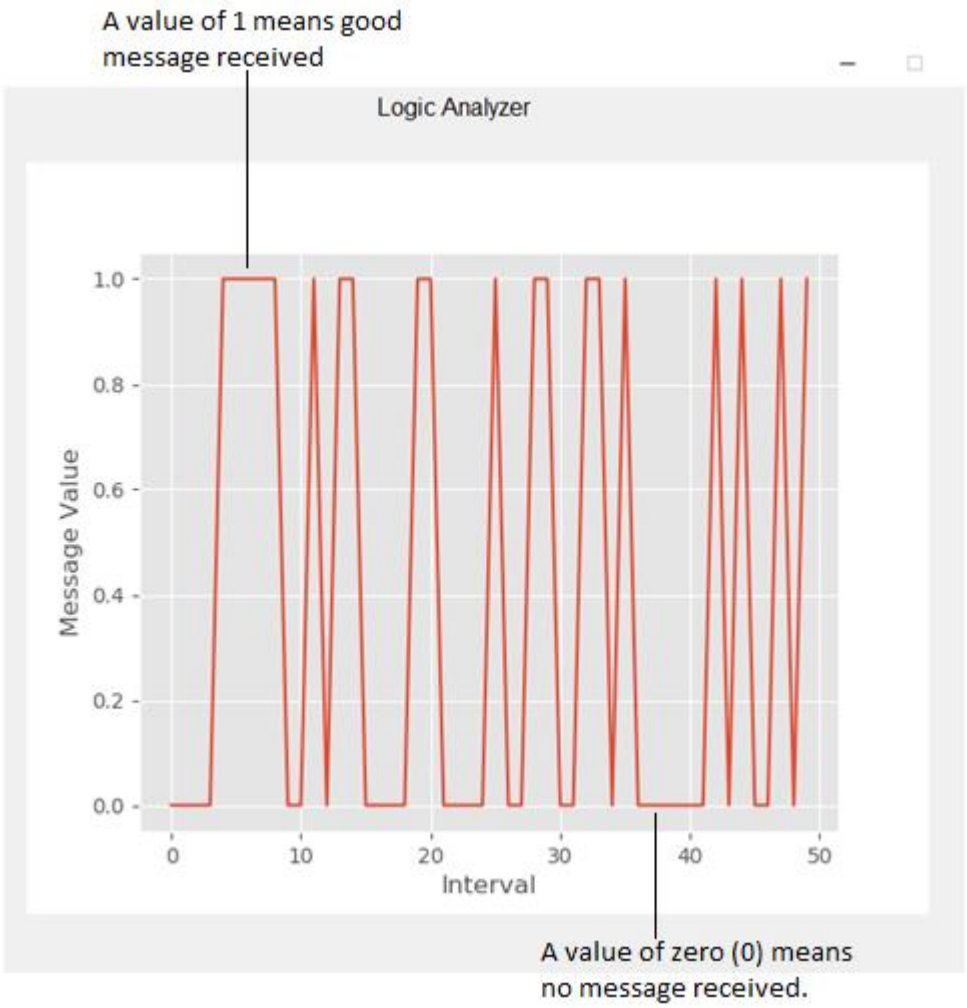



*Figure 21 Graph showing ECU A not under attack. As can be observed, the vertical axis has values of either 0 or 1 to indicate no message being transmitted and a correct message being transmitted. This happens when the message authentication algorithm is activated.*

has values of either zero, one or negative one when the system is under attack as shown in Figure 22. The value of negative one means that ECUA is receiving incorrect messages. The horizontal axis has intervals that change based on the number of messages that have been received. In other words, the horizontal axis increases in interval as more messages are received.

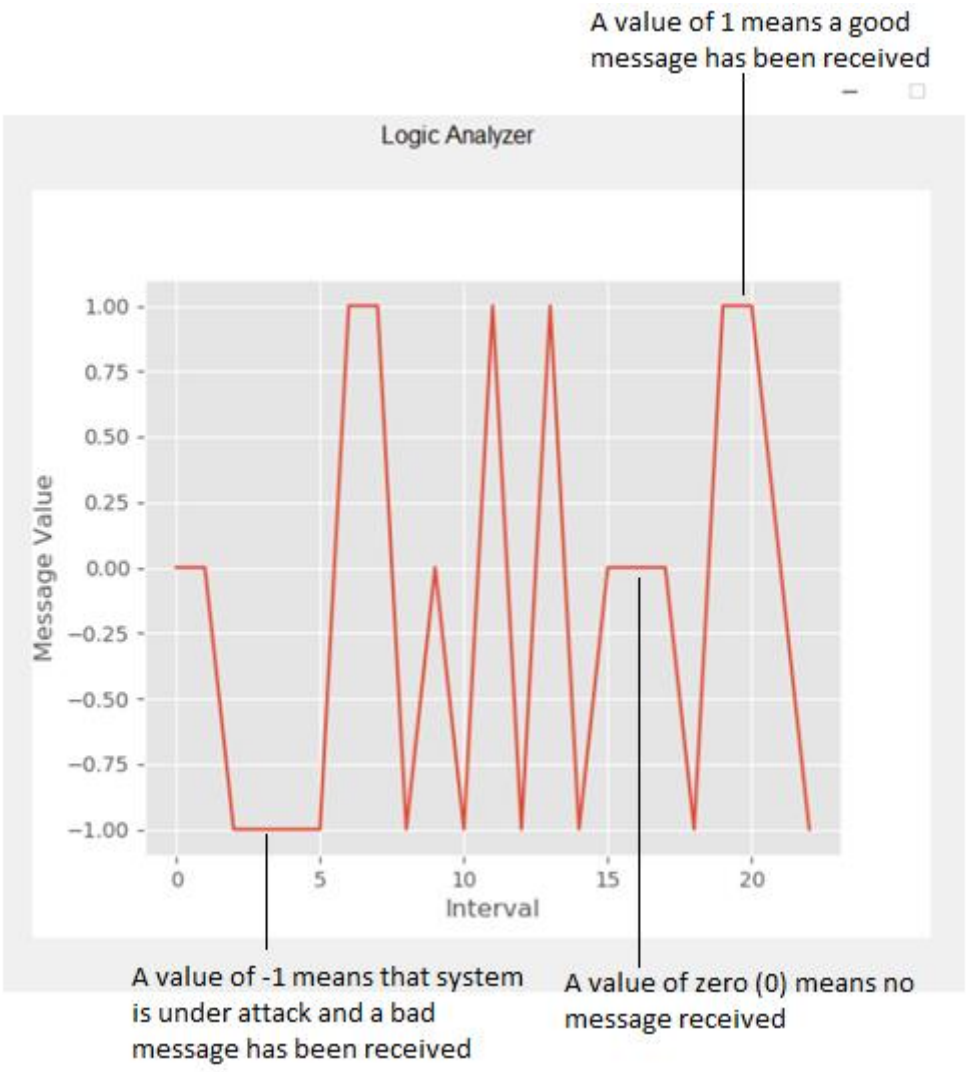



*Figure 22: Graph showing ECU A under attack because of the absence of a message authentication algorithm. It shows values of 0, 1 and -1 for the vertical axis.*

Below are the cautions to take for the GUI to work correctly with the ECU set up.

1. Download and install the following python packages: pyserial, matlibplot.
2. The Teensy should be connected to the computer on which the code for the GUI will be compiled and run.
3. If running both code from teensy and GUI, first run the code for the teensy and then run that of the GUI. If you just have the code for the teensy already compiled and loaded to the teensy, just plug in the teensy to the computer and compile and run the code for the GUI.
4. Make sure you set the value of the Communication (COM) port and the baud rates to the corresponding values that the teensy uses. The code snippet below shows teensy COM port "COM5" and baud rate of 9600.

```
ser = serial.Serial('COM5', 9600)
```

5. In case you need to copy the code for the GUI to another computer, be sure to copy "sampleText.txt" as well and make sure they are in the same directory. If you decide not to copy this file, you can also create a text file and replace any reference to "sampleText.txt" in the code with the name of the text file you create.

## 4.6 The CAN Harness

Preliminary development and testing of the ECUs was conducted over jumper wires. However, the plan was to use a physical CAN harness for final testing. The electrical wiring harness from a 2015 model year Chevy Impala was generously provided to the WPI MITRE Collaboratory by MITRE. Additionally, there was an instrument cluster from an unrelated vehicle.

For ease of use, the harness was stored attached to a metal grate, which gave some support to the

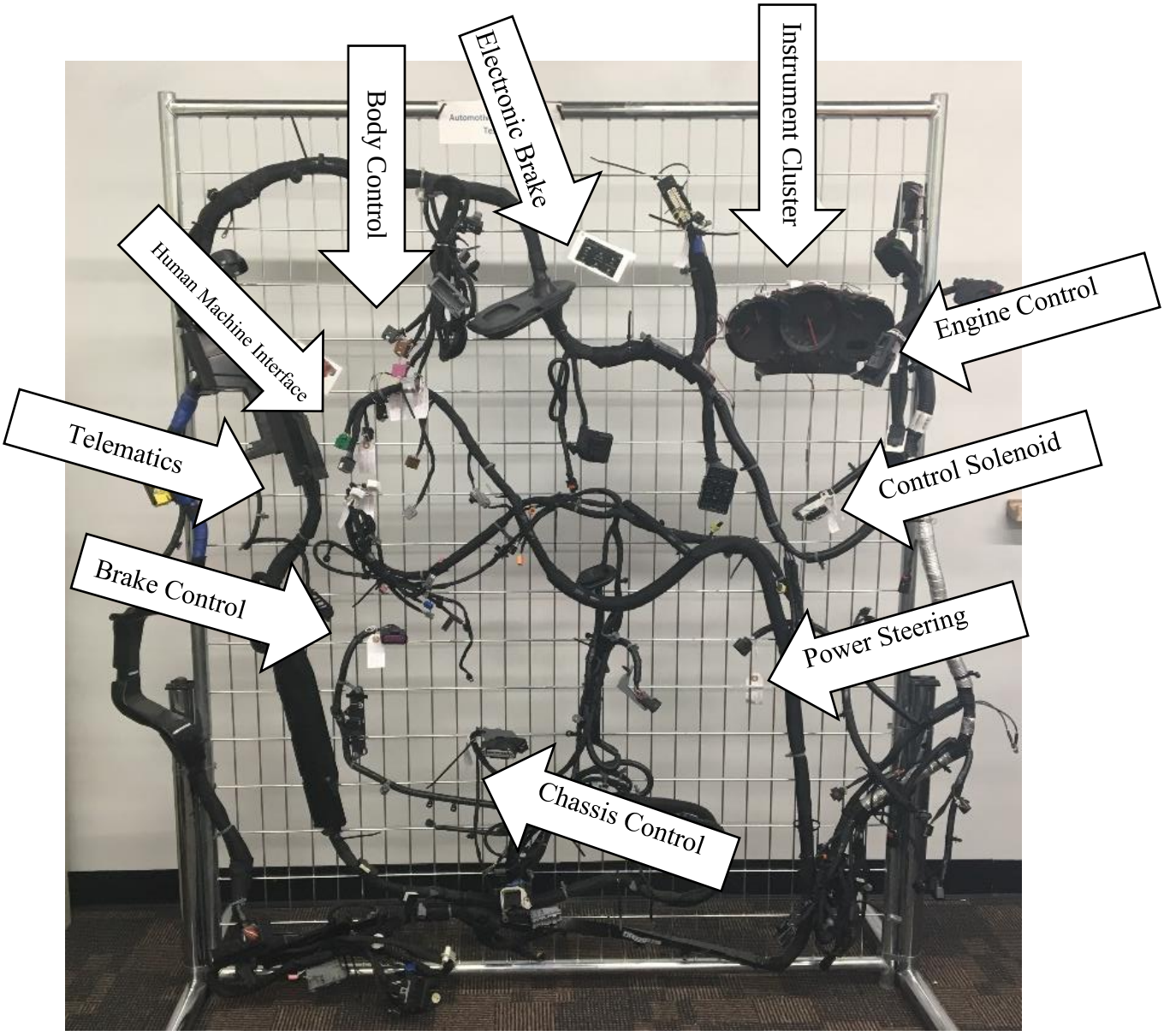



*Figure 23: Wiring harness from a 2015 Chevy Impala provided by MITRE. It is supported by a metal grate. The harness includes CAN bus connectors for brake control, telematics, human machine interfaces, body control, electronic, engine control, control solenoid, chassis control, and power steering. Additionally, there was an instrument cluster used for visual indicators in our setup.*

otherwise spaghetti like structure.

The CAN harness had been used by previous MQP teams and some of the connectors already had paper tags attached [29]. The tag indicated the connector type and what it was intended for as seen in Figure 24.


*Figure 24: Connector on the 2015 Chevy Impala wiring harness. The connector has been marked by the previous MQP team with a paper tag that identifies it as the K83 connector that is used by the parking brake.*

There were also scans of the CAN bus documentation for a 2014 Chevy Impala electrical system [30]. Unfortunately, the documentation did not match the 2015 harness. In some cases, one of the labeled connectors would have an entirely different physical shape. This can be seen with the K43 connector intended for power steering in Figure 25. The 2014 model year documentation showed a connector with a total of 10 bins arranged in 2 rows, while the connector on the harness featured just 5 pins in a single row.


*Figure 25: K43 Power Steering connector in the documentation for a 2014 Chevy impala and on the 2015 Harness. The documentation indicates a 10-pin layout in 2 rows of five, while the connector on the harness only has 5 pins in a single row.*

There were also other cases where the physical shape of the connector would match that of the documentation, but the pinout would not. For example, the K74 Human Machine Interface Control Module documentation indicated 16 pins arranged in two rows of eight, which the connector on the physical 2015 CAN harness had as shown in Figure 26. However, the pinout does not match. The documentation indicates that the CAN high is located on pins 11 and 13 and CAN low on pins 12 and 14. The physical harness connector does not have anything connected to pin 11, so it cannot possibly be CAN low. If we look at just the not connected pins as an indicator of discrepancies, we can see a big mismatch between the documentation and the physical harness.

Some of the not connected pins do line up between the documentation and physical harness, but the clear majority of them do not. In fact, the number of conductors going to the connector

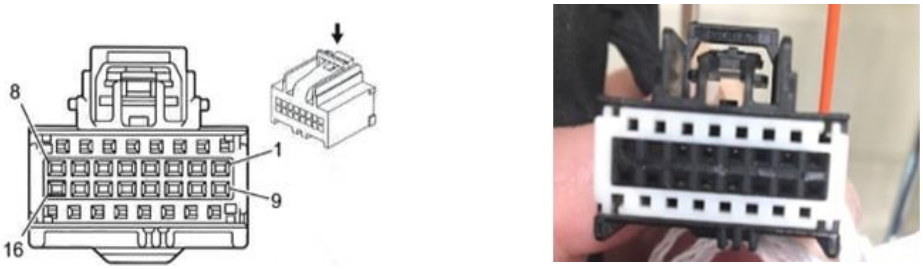



*Figure 26: On the left is the K74 Human Machine Interface Control Module as pictured in the 2014 documentation, which has 16 pins in two rows of eight. On the right is the same connector on the 2015 harness.*

does not even match between the two. The documentation indicates that there should be 10 conductors, while the actual harness only has eight conductors connected to it.

*Table 4: Pins that were not connected according to the 2014 documentation and on the 2015 harness.*

| Pin | Documentation | Harness |
|---|---|---|
| 1 | | |
| 2 | | |
| 3 | Not Occupied | Not Occupied |
| 4 | Not Occupied | |
| 5 | | |
| 6 | Not Occupied | |
| 7 | | Not Occupied |
| 8 | | Not Occupied |
| 9 | | Not Occupied |
| 10 | Not Occupied | Not Occupied |
| 11 | | Not Occupied |
| 12 | | |
| 13 | | |
| 14 | | |
| 15 | Not Occupied | Not Occupied |
| 16 | Not Occupied | Not Occupied |

With the high number of discrepancies, the existing 2014 documentation was of limited use for the 2015 harness. To do physical testing, the pin out of the harness had to be determined using other methods.

The first step to determining the pin out was to try to use the existing documentation. As described above, the existing documentation had limited usefulness. Therefore, efforts moved onto the second method of trying to find additional existing documentation. A rather extensive internet search was initiated, and a member of the WPI student branch of the Society of Automotive Engineers (SAE) was contacted to see if they had any resources. The search resulted in finding documents about the 2015 Chevy Impala, but they did not go into sufficient detail about

the electrical systems to be helpful [30]. The search also resulted in the discovery of a document that looked to have the information we needed, but it cost over $500 and was not inside of the budget of the project [31].

Method 3 was based upon the existing documentation, which was known to be poor. Even though some of the documentation was known to be incorrect, the expectation was at least a few of the connections would be accurate, and that it could be used as a starting point from. Using the existing documentation, looking only at the connectors with physical pinouts that matched the documentation, a continuity tester was used to test each documented connection. Using this method only two pairs of connections were found as can be seen in Figures 27 and 28.

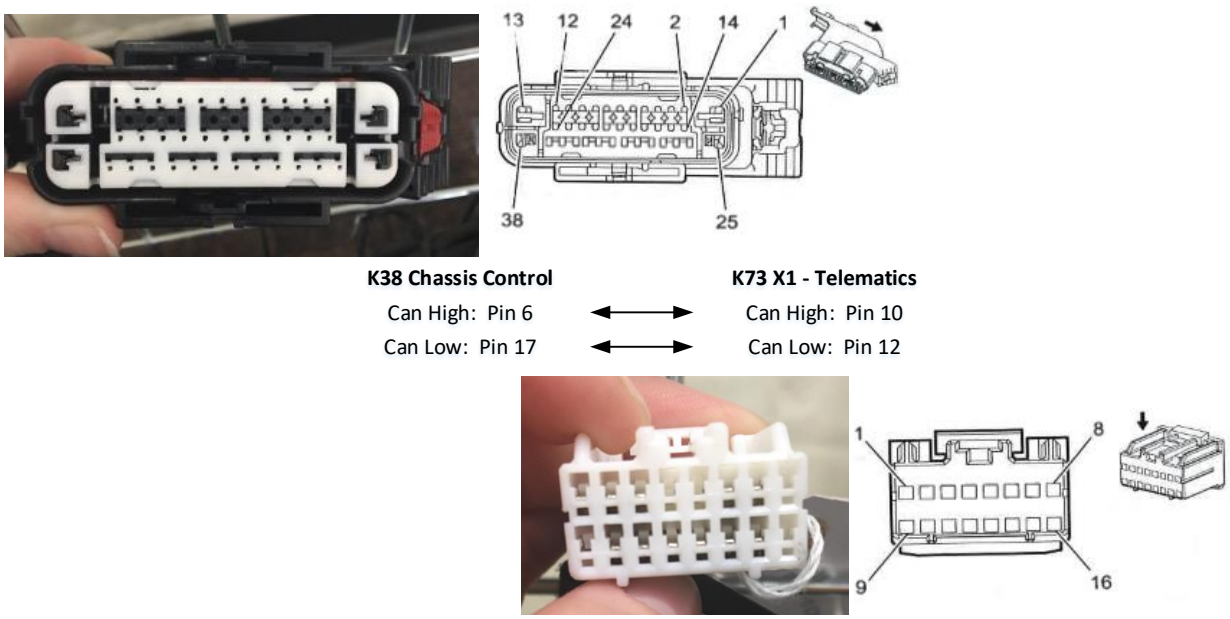



*Figure 27: The CAN high and CAN low connections between the K38 Chassis Control node and the K73 X1 Telematics connector forming a two node CAN bus. Pin 6 on the Chassis control module pictured at the top left connects to pin 10 on the K73 X1 telematics connector pictured on bottom right and makes up the CAN high portion of the bus. Pin 17 on the K38 Chassis Control connector and Pin 12 on the K73 Telematics connector makes up the CAN low connection.*

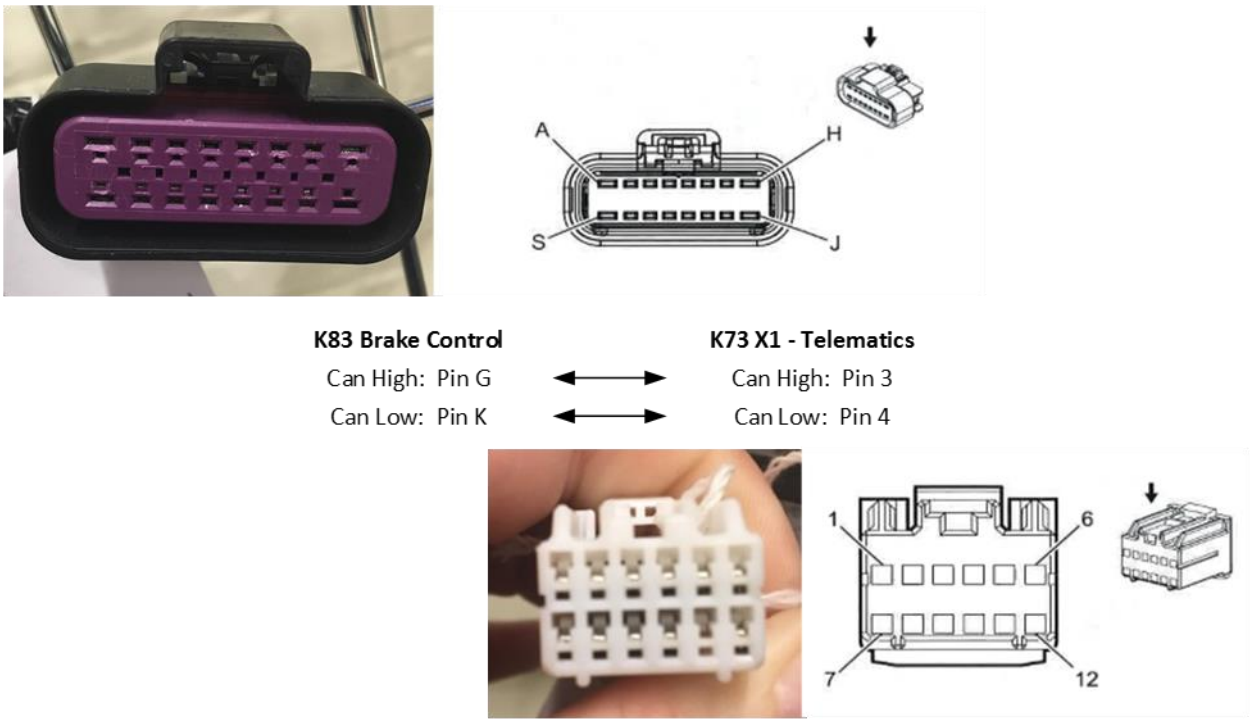



*Figure 28: The CAN high and CAN low connections between the K83 Brake Control node and the K73 X1 Telematics connector forming a two node CAN bus. Pin G on the Brake control module pictured at the top left connects to pin 3 on the K73 X1 telematics connector pictured on bottom right and makes up the CAN high portion of the bus. Pin K on the K83 Brake Control connector and Pin 4 on the K73 Telematics connector makes up the CAN low connection.*

With the fully accurate documentation exhausted, a fourth method of finding the connections was implemented. This used the documentation as a place to start but did not assume that it was a hundred percent accurate. This required spending quality time with a continuity tester [32]. While based upon documentation, this method was significantly more brute force than method three. One lead of the multi meter would be plugged into a pin on one connector, and the other lead of the multi meter would be used to test every pin on a different connector. Once all the pins on the second connector had been tested, the first lead would be incremented by 1 pin and the

process would start over. To increase efficiency, testing would begin with the pins marked as CAN high and CAN low in the documentation.

According to the 2014 documentation, the first node on one side of the CAN bus was the K20 Engine Control Module, which was connected to the Q8 Control Solenoid Valve Assembly. The 2014 documentation indicated the same physical pinout as was found on the 2015 harness K20 Connector. This did not guarantee that the electrical pinout would match, but it made sense to start with the pins listed in the documentation. Unfortunately, the Q8 connector had mismatching physical pin layout.

Since there was more accurate documentation of what the CAN pins should be on the K20 connector the process started from there, plugging one lead of the multi meter into pin 39 which was documented as the CAN high pin. The other lead of the millimeter was used to systematically test every pin on the Q8 connector for continuity. Eventually, it was found that there was a connection between K20 pin 39 and Q8 pin 9. This method was continued to find that the CAN low pin 40 on the K20 connector was connected to pin 1 on the Q8 connector. With these two connections known, there were three different buses, each with 2 nodes that we could create.

The next goal was adding a third node to the newest identified two node bus. According to the 2014 documentation the next node on the bus was K43 Power Steering. The K43 documentation did not line up with the physical pin layout of the connector. The documentation indicated 10 pins in 2 rows of 5, but the actual connector was a single row of 5 pins. With two connectors with unknown pin outs, a brute force method was applied. The expected results were two pins of the five on the Q43 connector being connected to two of the 12 empty pins on the Q8

connector. Eventually the expected connections were discovered, and our known CAN bus included three nodes.

Using the same method from nodes 1 through 3, the pins for nodes 4 and 5 were determined. The pinouts for the first five nodes can be seen in Figure 29.

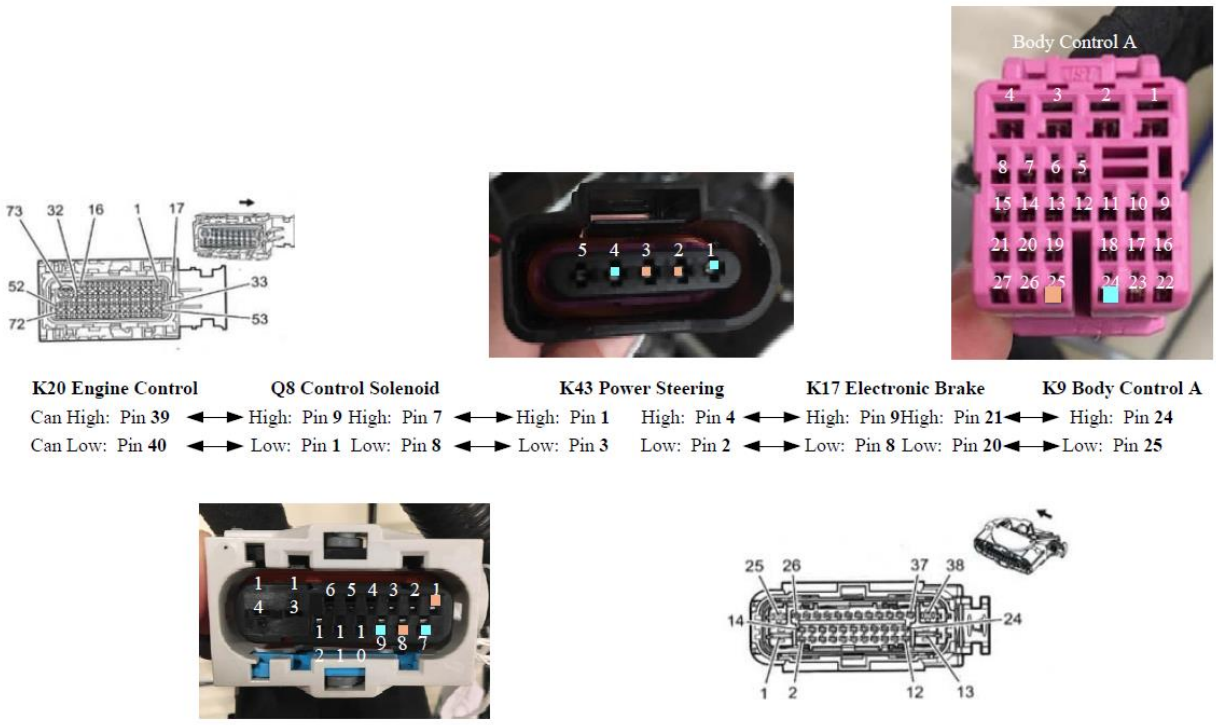



*Figure 29: The CAN bus pin out of the 5 node bus. The K20 Engine Control connector is connected to the Q8 Control Solenoid connector. The Q8 Control Solenoid is also connected to the K43 Power Steering Connector. The K43 Power Steering Connector is connected to the K17 Electronic Brake connector. Lastly, the K17 Electronic Brake Connector is connected to the K9 Body Control A connector.*

At this point the method reached a metaphorical wall. The last connector in the chain was the pink K9 Body Control Module. There were four body control connectors, and three body control connector types in the documentation. All four of the connectors on the harness were different, and none of them matched any of the physical pin layouts found in the documentation

as seen in Figure 30. The difficulty of this challenge came from the sheer number of possible connections that could have been made. The documentation indicated that there was a connection between the K9 Body Control Module and the K74 Human Machine Interface module, but it did not indicate how many of the K9 Body Modules were connected to the CAN bus. Consequently, it was possible that the next connection could start on any of the K9 Modules and could end on one of the 4 remaining K9 modules or the K74 HMI connector.

Additionally, because there were only a maximum of four ECUs and the instrument cluster, only five nodes were needed on the CAN bus for testing. Due to the large amount of effort required to connect a sixth node and the low increase of helpfulness, the pinout determination of the CAN

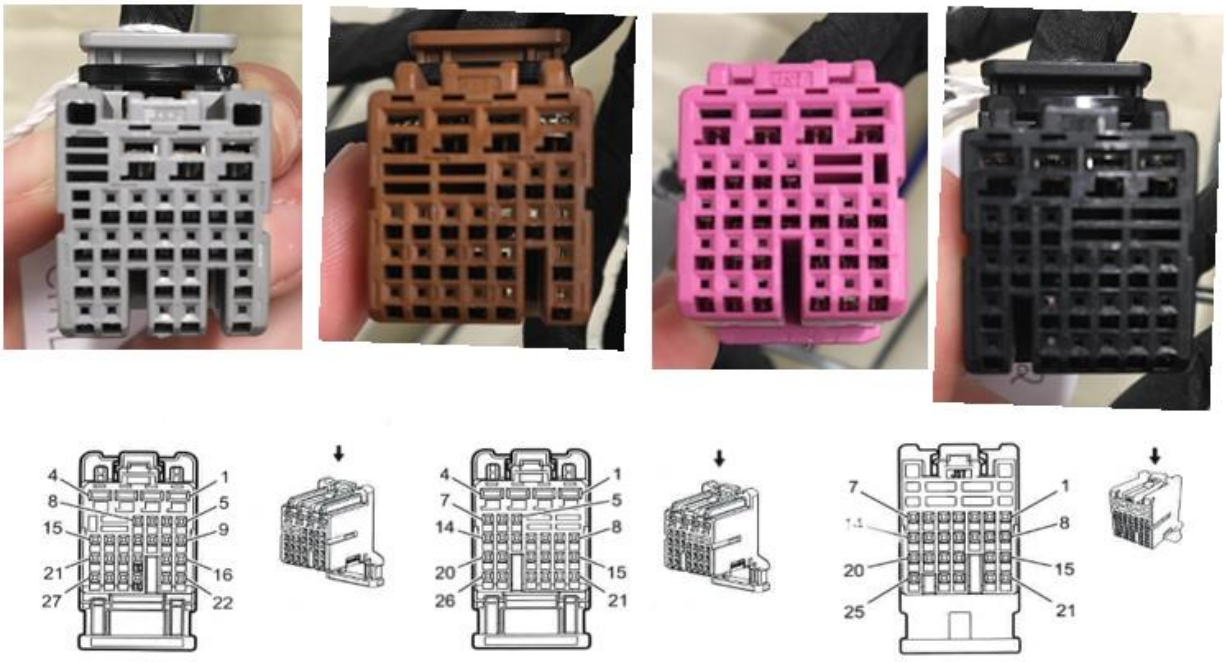



*Figure 30: K9 body control modules. There were 4 connectors on the 2015 harness, and 3 connectors in the 2014 documentation. Given the seven different connectors, none of them share a physical layout.*

harness stopped after determining 5 nodes. The connections determined on the harness can be seen in Figure 31.

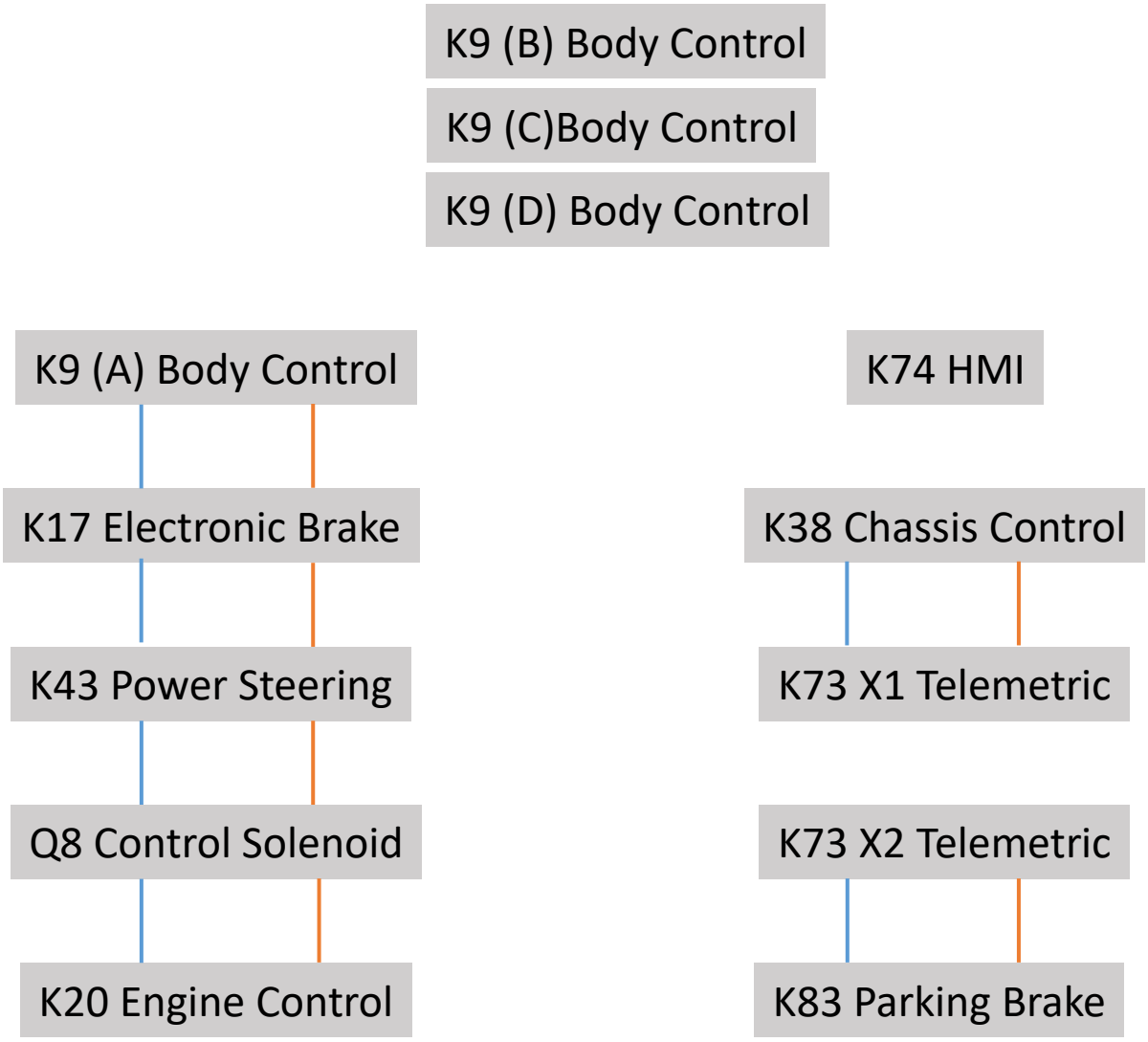



*Figure 31: High level diagram of the connectors on the harness and their known connections. There is a five-node chain, and two different two node chains that do not interconnect.*

The pinout of the CAN bus on the harness had been determined, but the ECUs still needed to be wired in. The CAN harness does not use taps off the network, but rather relies on

the connected ECU to bridge the gap. This means that each connector on the bus had two CAN high pins and two CAN low pins, as can be seen in Figure 32.

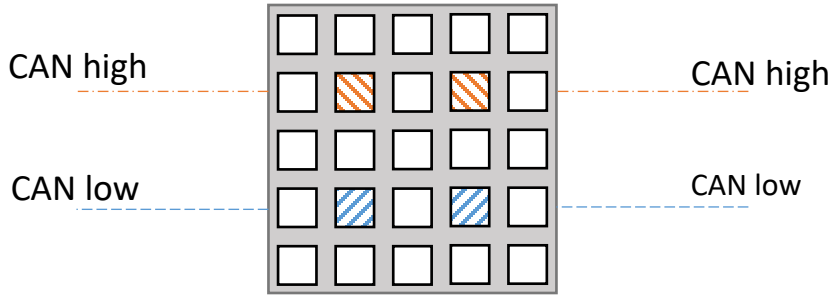



*Figure 32: Example of CAN pinout in Connector. There are pins that are connected to different segments of the CAN low or CAN high bus.*

This means that the CAN bus could be terminated at any desired node by simply placing a 120 Ω resistor between the CAN high and CAN low pins. However, it also means that every connector on the bus had to have a jumper from one CAN high pin to the other and another jumper between the two CAN low pins. As depicted in Figure 33, if any of these jumpers were not properly in place, it would have caused a short circuit in the bus and it would not have functioned.

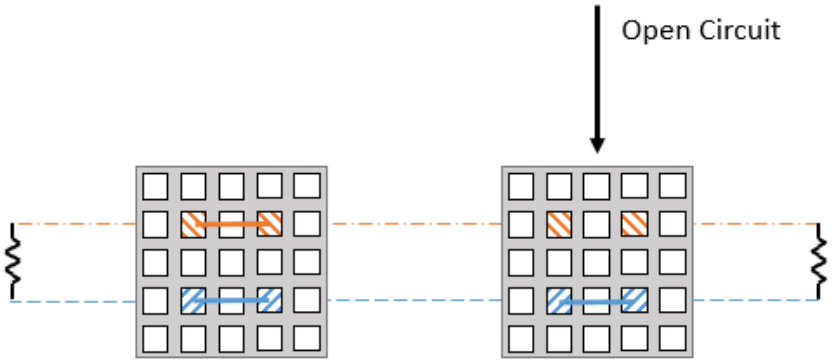



*Figure 33: Example of a sort circuit when 2 pins on a connector are not bridged. The harness connector does not automatically bridge the gap between the two pins, and so it is up to the device connector to do so.*

The boards used as ECUs output to a standard DB-9 Connector. Using the continuity tester function of a standard multi meter the connections between the CAN_L and CAN_H pins on the PCB and the DB9 Connector were determined. CAN high was located on pin 2 and CAN low was located on pin 7 as can be seen in Figure 34.

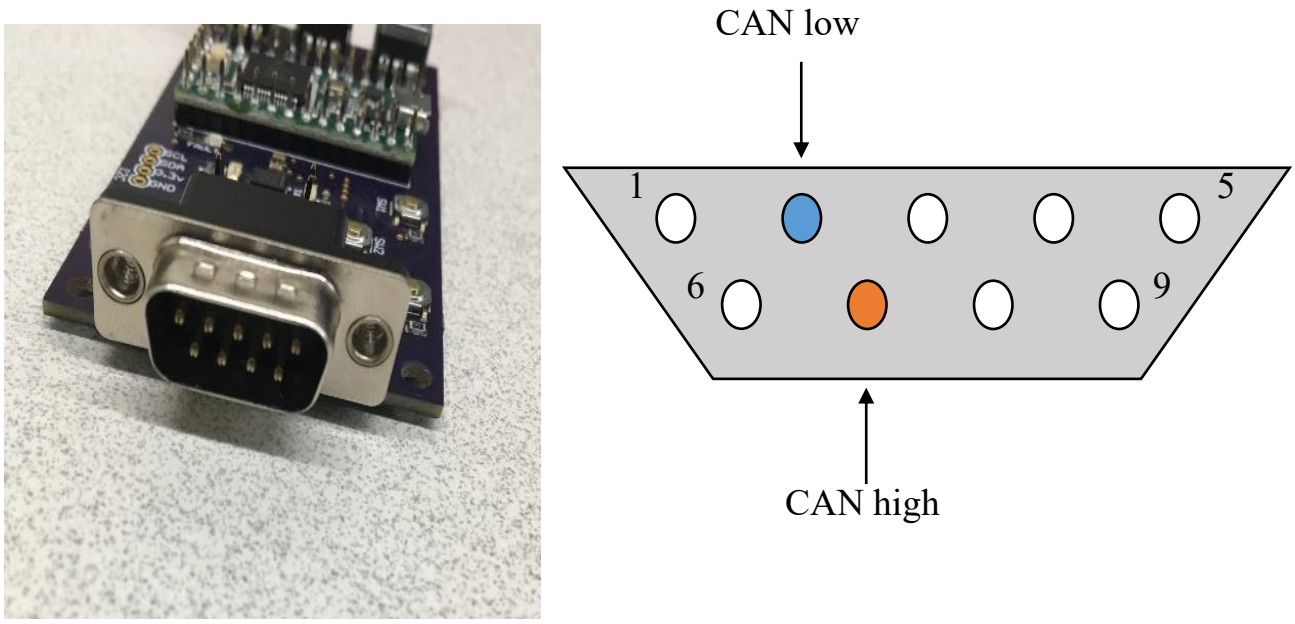



*Figure 34: PCB DB9 connector and pinout. The Can low pin of the PCB is tied to pin 2 and CAN high is tied to pin 7.*

The previous MQP team had left DB9 breakouts, which went from female DB9 to male pins. There were two variations of this, the standard breakout and the terminator which were labeled as such. Using a multi meter, the pinouts of the breakouts were determined, as can be seen in Figures 35 and 36.

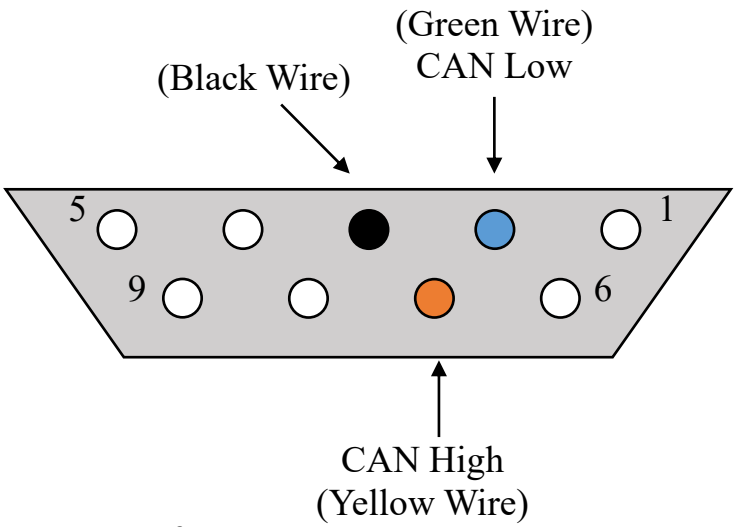



*Figure 35: The pinout of the standard DB9 breakout. Pin 2 is tied to the green wire, and when connected to the device is tied to that device's CAN low. Pin 7 is tied to the yellow wire and when connected to a device, is tied to the CAN high pin. Pin 3 is tied to the black wire.*

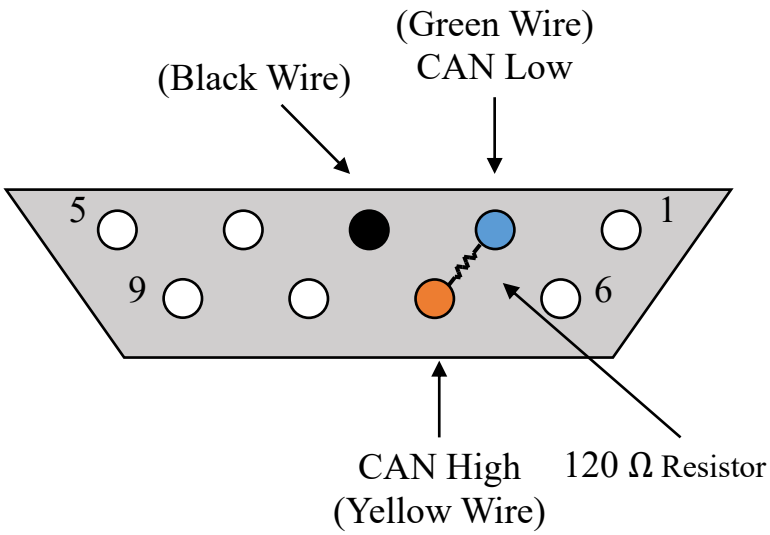



*Figure 36: The pinout of the terminator variation of DB9 Breakout. There is an internal 120 Ohm resistor between pin 2 and pin 7 but otherwise has the same connections as the standard variation.*

The standard variation wires broke out to three wires as seen in Figure 37. On a standard connector, one of these wires would go to one of the CAN pins and a second wire would go to the second CAN pin of that type. This effectively acted as the necessary bridge for that connector.

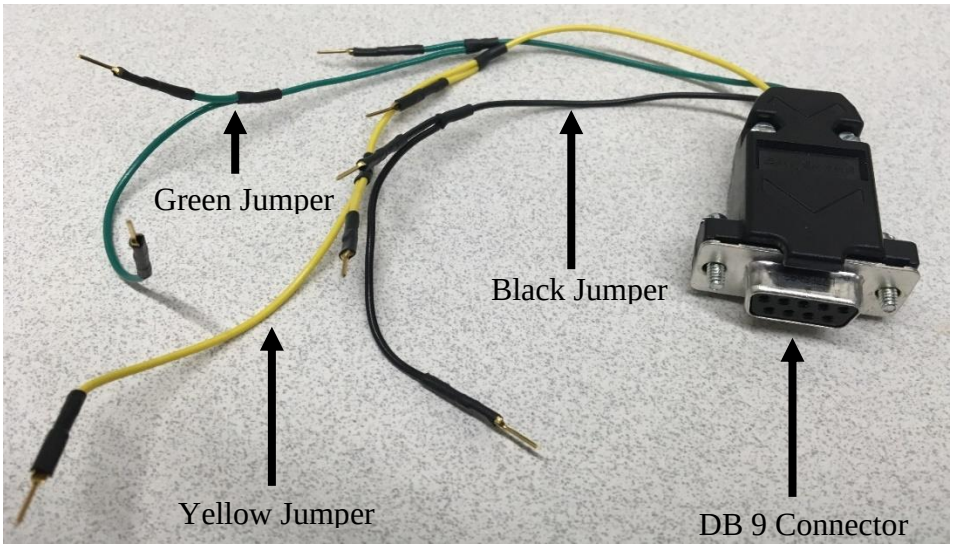



*Figure 37: Photograph of the standard DB9 breakout. Each wire is broken out to 3 pins so that the connector can bridge the CAN high and*

The terminator variation featured a built in 120 Ω resistor between the CAN high and CAN low pins. However, the connector only had one wire for each pin as seen in Figure 38.

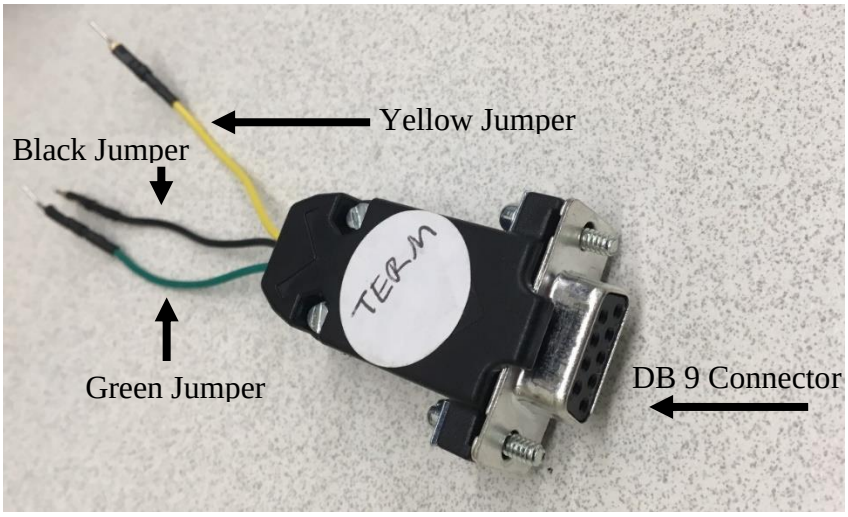



*Figure 38: Photograph of the terminator DB9 breakout. Each wire only goes to 1 pin because the internal resistor handles the bridging of the CAN High pin to the CAN low pin.*

So, to connect an ECU to the CAN harness the following steps would be taken:

1. Determine pinout of harness connector.
2. Determine if standard or terminated breakout needed.
3. Plug in the yellow wire of the breakout into the high pin(s) of the connector.
4. Plug in the green wire of the breakout into the low pin(s) of the connector.
5. Plug in the DB9 breakout into the ECU DB9 connector.

Unfortunately, one of the problems discovered during testing was that the pins on the breakout cables were not particularly durable. If the pins got bent even a little and an attempt was made to bend them back, they would break off. A quick fix for the connectors was created by soldering on a piece of jumper wire to where the pin had broken off as seen in Figure 39.

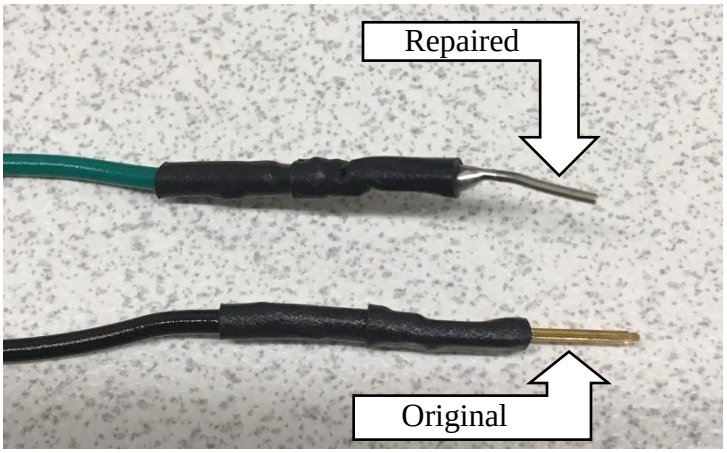



*Figure 39: Example of a repaired pin on a DB9 breakout compared to an original pin. A jumper wire was directly soldered to where the pin broke off.*

Additionally, two DB9 to CAT 5 converters (one male DB9, one female DB9) were acquired as shown in Figure 40. One Connector would plug into the PCB male DB9 Connector, and the other would plug into the DB9 breakout. The two converters could then be linked using a

CAT5 cable of any desired length. This meant that the ECU could be located a manageable distance away from the harness connector. For example, if someone was programming the ECUs using a laptop on a table, the ECU could be placed on the table next connected via USB to the computer, while simultaneously connected to the CAN harness connector. These did not end up being used in testing.

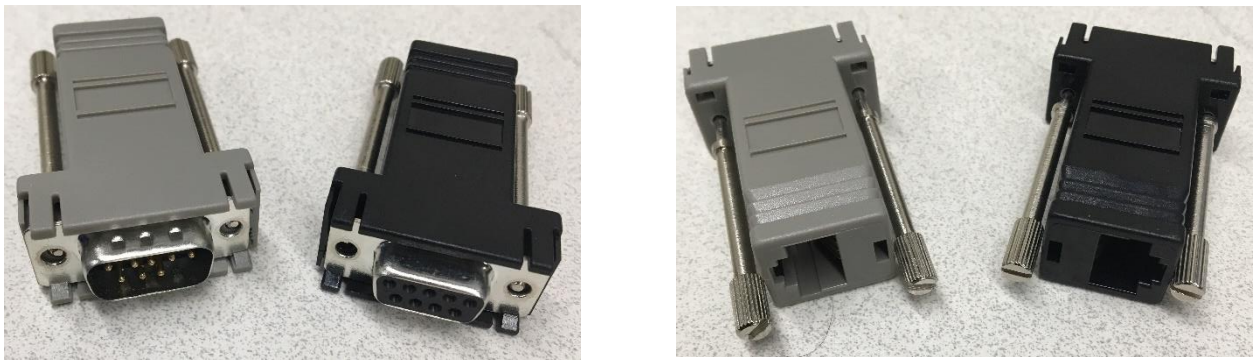


*Figure 40: DB9 to CAT5 converters. Any CAT5 cable can be inserted to lengthen the distance between the harness connector and the ECU.*

During preliminary testing with the ECUS, the boards were balanced on top of the grate holding up the harness. This was not an ideal position as it resulted in a high chance of an ECU being knocked off or the cables becoming load bearing. To fix this problem, shelves to hold the ECUs were designed and manufactured.

Using a pair of calipers, the dimensions of the grate and ECU board were taken. Using Solid works, a 3D model of a section of the grate was created as can be seen in Figure 41.

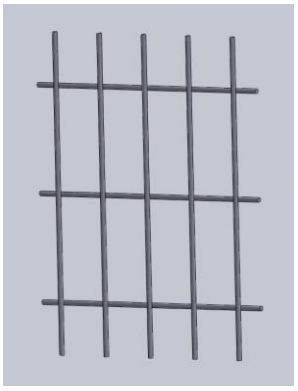


*Figure 41: 3D Solid works Model of a section of the grate holding up the CAN harness*

Using the measurements of the ECU board and the grate, a simple shelf was modeled. The shelf was made from two pieces, a horizontal piece that held the ECU and a vertical piece that acted as a support shown in Figure 42. Each piece was modeled as a 3D part, and then the two parts were combined into an assembly shown in Figure 43.

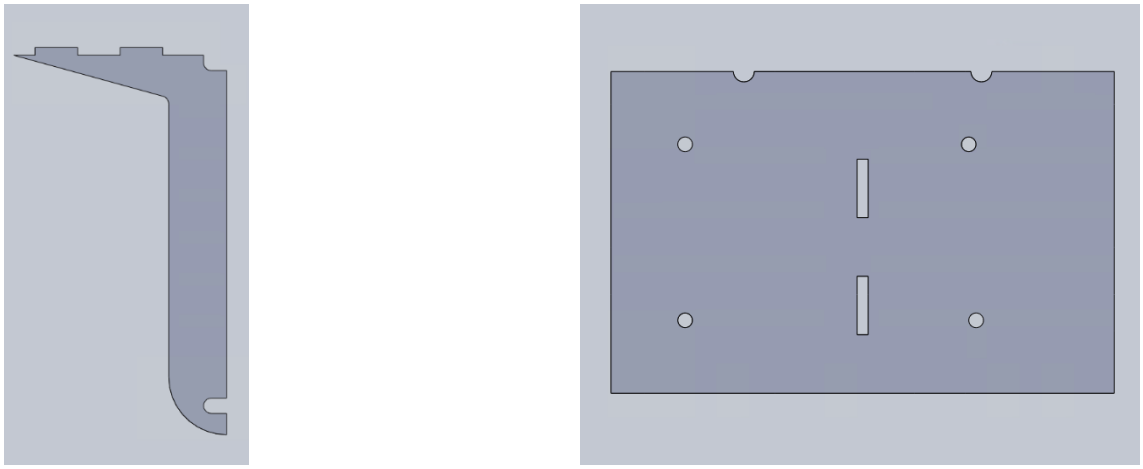


*Figure 42: Solid Works 3D Model of the first prototype shelf pieces. On the left is the supporting piece that rests up against the grate. The piece on the right is the horizontal piece that holds the ECU.*

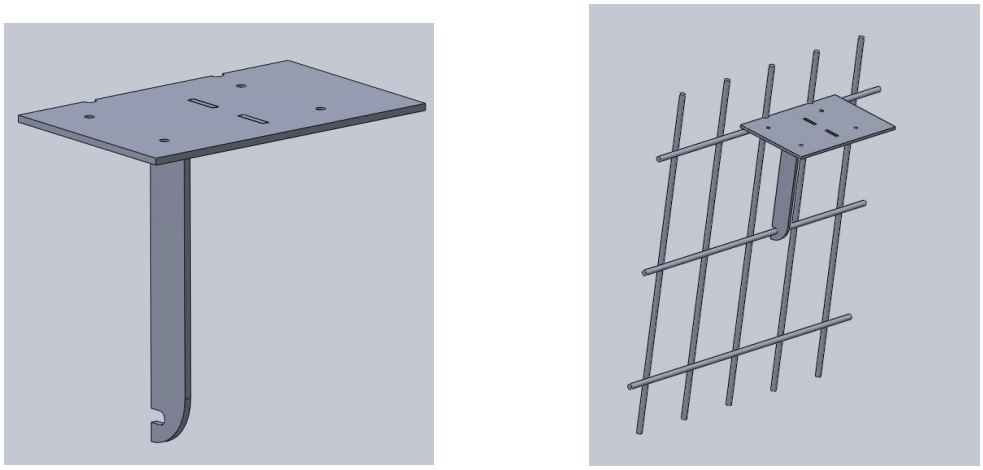


*Figure 43: The first prototype shelf assembly and that assembly on the grate model. As can be seen on the left, the pieces fit snuggly together. On the right the assembly can be seen as it would be attached to the grate.*

Space on WPI's Washburn Shop's laser cutter was booked. Using the laser cutter user guide provided by Washburn shops, the files were formatted to be cut out on the laser cutter. They were then cut out of nominal 1/8$^{th}$ inch Acrylic. The parts were then taken to the harness for brief functionality testing. At this point in time two problems were discovered: one of the holes was in the incorrect spot and there was nothing keeping the entire shelf from just swinging down off the supportive grate.

With these two problems in mind, the shelving unit design was updated as shown in Figures 44 and 45. The hole was moved to correct position, and the vertical piece was modified to give better rotational support. Once again, the laser cutter was used to cut out the pieces of the shelf. The parts were brought back to the lab for preliminary testing. Initially, no major issues were discovered, so the remaining three shelves were cut out. The shelves were then glued together.

It was now that issues were discovered. Although the grate appeared to have nominally the same spacing throughout, that was not the case. This difference in spacing caused it to be difficult to

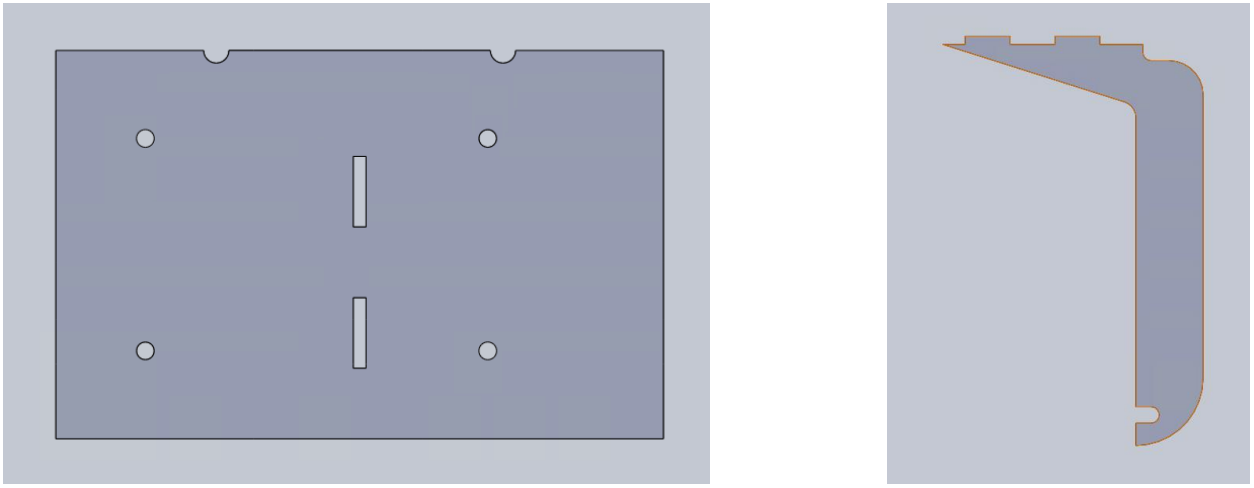


*Figure 44 The second prototype of the shelf vertical and horizontal pieces. The piece on the left shows the updated hole placement, and the piece on the right shows the updated attachment mechanism which comes from the back.*

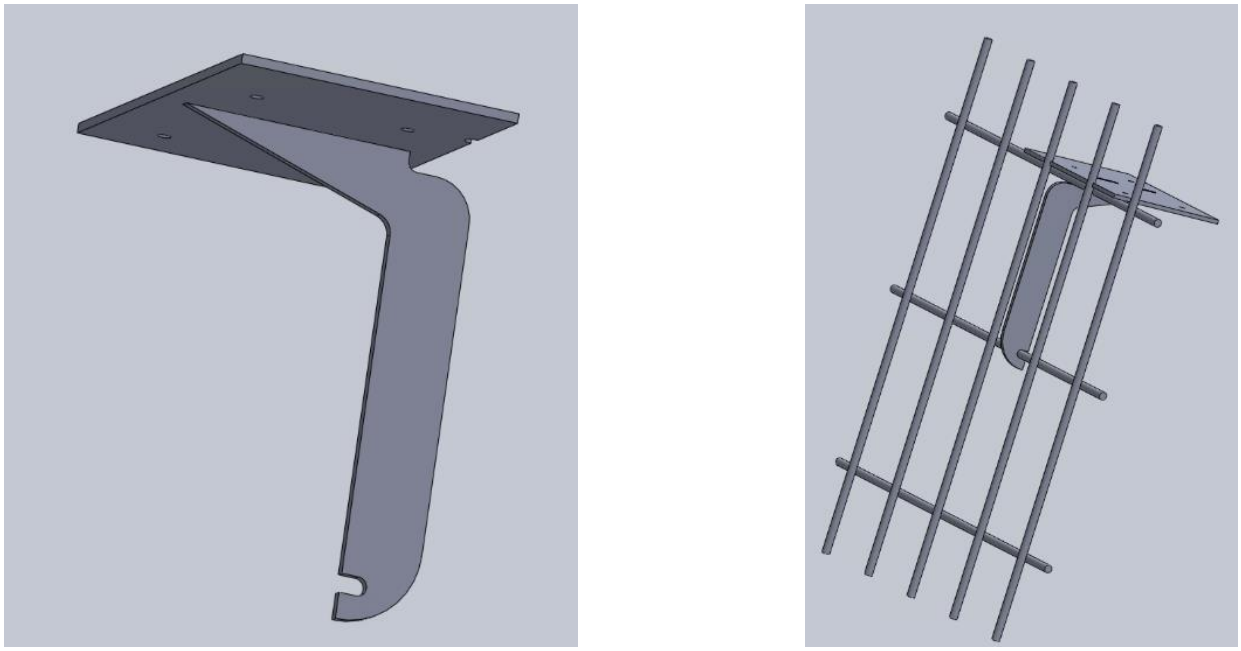


*Figure 45 The Solid Works 3D assembly of the second prototype shelf and the shelf on the g grate. On the left the pieces are fit snugly together, and on the right the attachment to the grate is demonstrated.*

install the shelves in some areas because there was not enough room, while in other areas the shelves would just fall off because the bars were spaced too far apart. While the shelfs did have issues, they were deemed satisfactory for the purposes of testing. The ECUs could be mounted to the shelves using stand offs and were no longer precariously balanced.

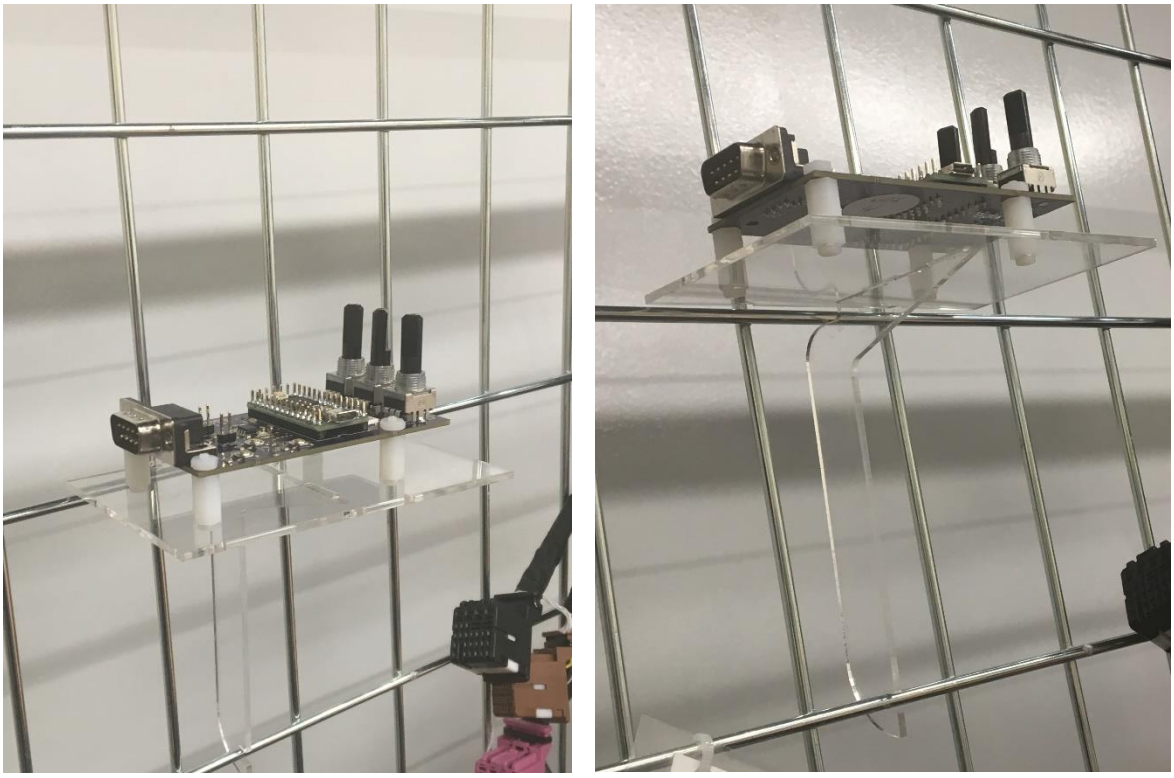


*Figure 46: ECU mounted on a shelving unit attached to the grate. The ECU is attached using plastic standoffs and nuts. The shelving unit uses a combination of its own parts, gravity, and friction to remain in place.*

## 4.7 Chapter Summary

The methodology for the development of this project was extensive in the details required to create a functional CAN system. The development of the software and hardware for a simple CAN network was completed with the use of MITRE supplied ECUs that were programmed for specific behavior of the team. With the use of the open source FLEXCAN library and the MCP2551 can messages were able to be sent across a simple terminated system. Message

verification was completed with the use of SHA1 and SHA256 in a complex message receiving system. The pinout for a five-node and two-node CAN buses on the harness was determined. Shelves to hold the ECUs were also developed. Using this physical support, the ECUs were able to be used in conjunction with the physical CAN harness. After the ECUs were implemented on the physical CAN harness the development of the GUI was completed to add additionally visibility to the CAN system. The methodology of this project showed the precise development of each proposed goal and the details of how each goal changed and was completed as the project went on.

# CHAPTER 5: RESULTS

## 5.1 Methodology for Obtaining Results

The results of this project can most effectively be broken into the following two categories, implementation of message security and results from running the complete CAN system with the GUI. To obtain the first set of results it was a matter of building a robust test bench that successfully rejected messages based on the use of SHA. As seen in our methodology the process was taken step by step, setting up each individual ECU and developing the message spoofing attack and timing based attack individually. After testing both individual types of attacks we collected performance information of the entire CAN system. This process of testing and verifying allowed for accurate representation of the results of using SHA to secure a CAN system.

Verifying that an attack was mitigated required an understanding of how the attack was being implemented and then understanding how the behavior in the CAN system changed under the attack. Once the attack and behavior change were observed it was then the process to implement a solution to the attack and determine if the behavior is still that of a system under attack.

The first attack to be mitigated by using SHA was an ID message spoofing attack. An ECU would send a message with an improper message ID for the data being carried in the CAN message. This type of attack was observed to both slow down the CAN system by filling FIFO buffers with bad messages but also created strange behavior in the instrument cluster on the CAN system, such as warning lights coming on and off or rapidly changing the speedometer without changing the RPM. When the message verification process using SHA was implemented and then the attack was run again it solved a few of the problems with the ID spoofing attack. The first change in

behavior under message verification was that instrument cluster no longer was displaying false information, showing that data with bad message IDs was no longer being saved. The second noticeable advancement while running the message verification process was that the number of messages processed on a given ECU went up, indicating that using message verification lowered the amount of time spent on processing information from bad messages. The use of SHA to secure the CAN system against this type of attack proved to be effective at filtering out ID spoofed messages and making the system run more efficiently.

The second attack that was mitigated was the timing-based attack. This attack would copy a message sent from an ECU and send the same message bit for bit after a brief time delay. This type of attack was shown to create a display error on the instrument cluster, however unlike the ID spoofing attack it did not cause any significant delays to the CAN system. To mitigate this attack SHA was used with a global based Arduino timer that allowed for the messages to be time stamped. While this type of message verification allowed for the display error corrected and the rejected the

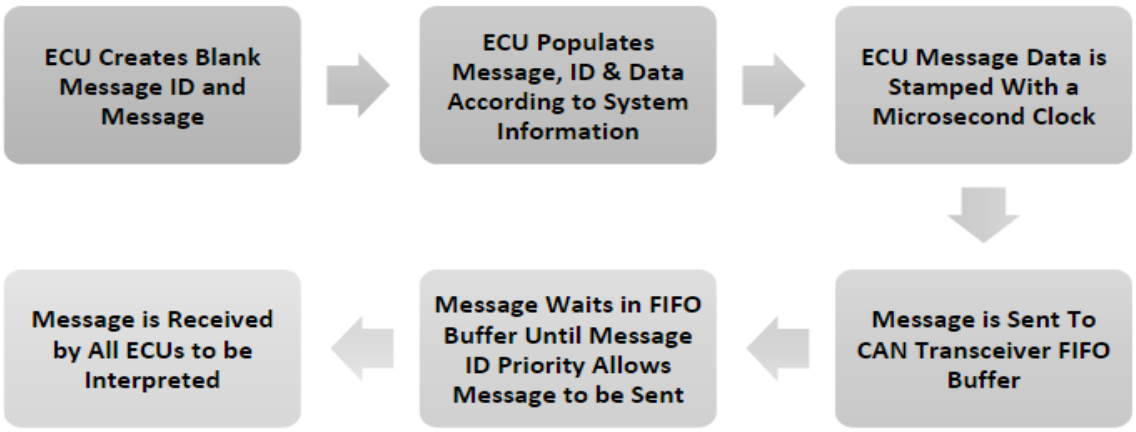



*Figure 47: The above diagram displays the process of creating CAN message and sending across the CAN system to other ECUs. This process includes creating the message, packing it with data and timing signature, and then sending it into the CAN transceiver to be sent to other ECUs.*

delayed attack message, it also caused problems for some low priority messages that took a long time to be sent across the CAN system. While the mitigation of this attack caused a few messages to be rejected, it did not cause any performance problems with the instrument cluster or the CAN system. The use of SHA and global timers to mitigate this timing-based attack was effective when rejecting time delayed messages that could cause dangerous behavior.

The above diagram shows the process of message packing to ensure the message has the proper detail to be properly verified. The first step of this process is to have the ECU create a memory space of 19 hex characters. Once the message space has been created the message is populated with an ID and the desired data for the message. After all the data values have been assigned the message is then sent to the FIFO buffer using the Flexcan library. Once the message is queued in the FIFO buffer the system waits until it can send the message across CAN according to the ID. Once the message is sent it is saved in the FIFO buffer of all the ECUs to be processed.

Before the message is received by other ECUs it needs to be physically transmitted across the CAN harness as a CAN High and CAN low message. Figure 48 shows that CAN High and CAN Low both have matching digital wave forms that verify each transmission line works as well as each ECU sends message properly formatted.

The next step in verification of a working CAN system was to implement a message being sent across CAN with full data values added to the message. Figure 49 displays a full CAN message, with an assigned message ID and message data in each section of the message. This was important to verify that he system was able to handle the creation of digital waveforms that had the accuracy to transmit messages.

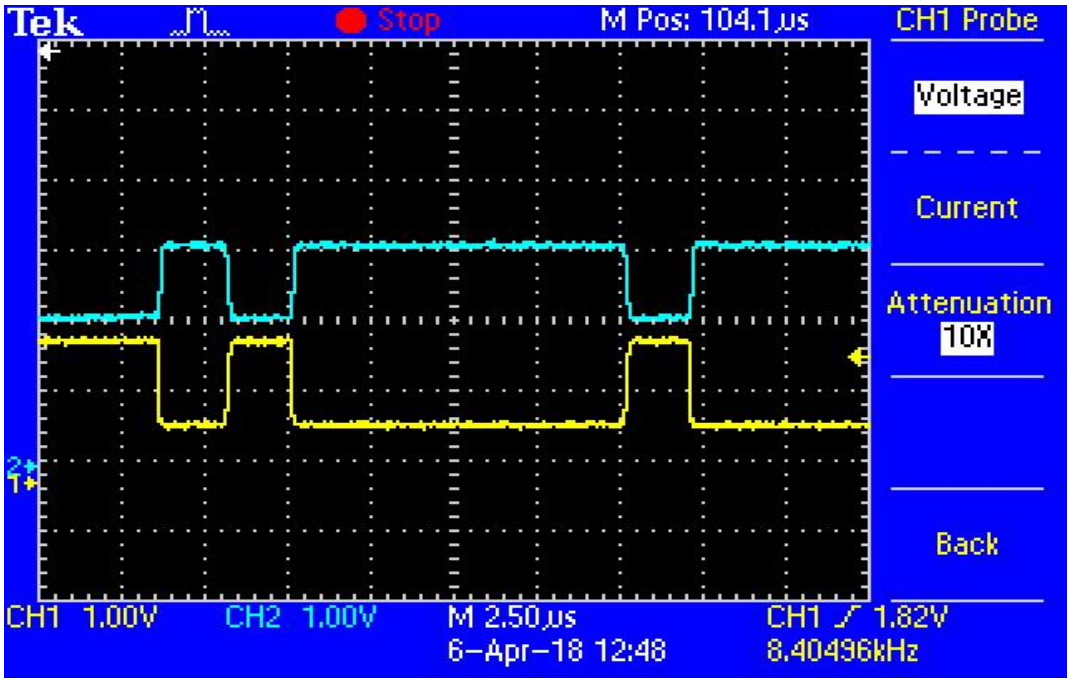



*Figure 48 Image of CAN High and CAN Low Signals being verified to match each other on the physical harness*

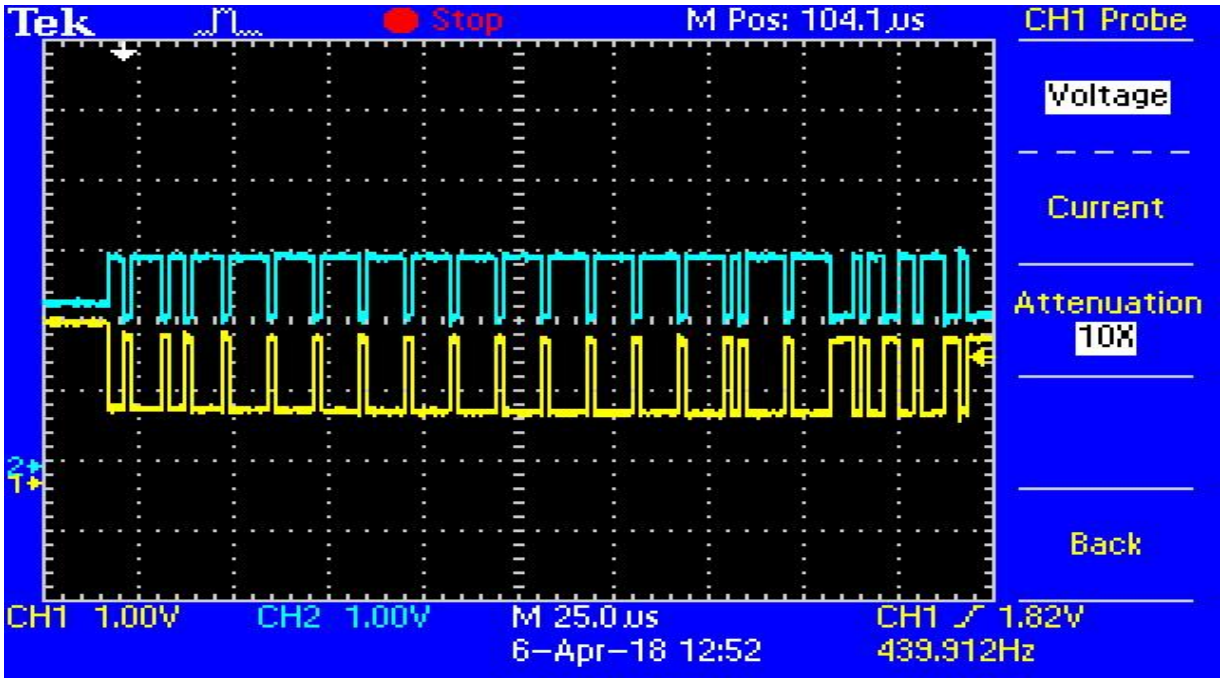



*Figure 49 Real-time image of CAN High and CAN Low transmitting data with full message ID And verified message data*

The last results of the tests to determine if the CAN transmissions were working properly was to send a message from multiple ECUs at the same time. This test was run to determine that the message being sent would not overwrite each other and send at times determined by their message ID. Figure 50 shows the result of this test where each message is sent at an individual time allowing for multiple devices to communicate across the network.

The verification that CAN High and CAN Low transmissions were successful and followed all the CAN standards was very important to the results of this project. Without the verification that the physical CAN network the rest of the results in this project would be under

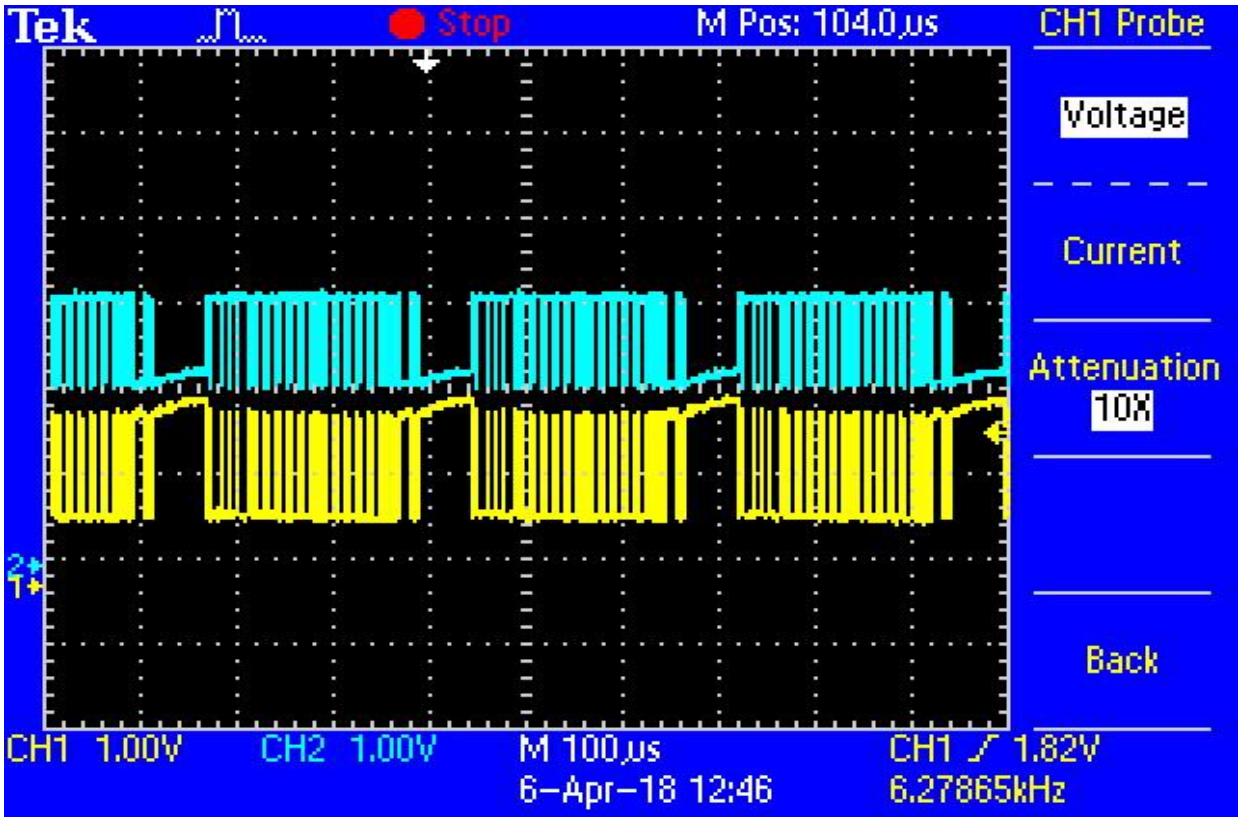



*Figure 50 Image of multiple messages being sent across CAN High and CAN Low in real-time, with each message waiting till the completion of the previous message to send across the network based on message ID priority.*

speculation. Once the system was verified to work properly the results of the attack mitigation could be investigated by analyzing the process of a message after it is received by an ECU.

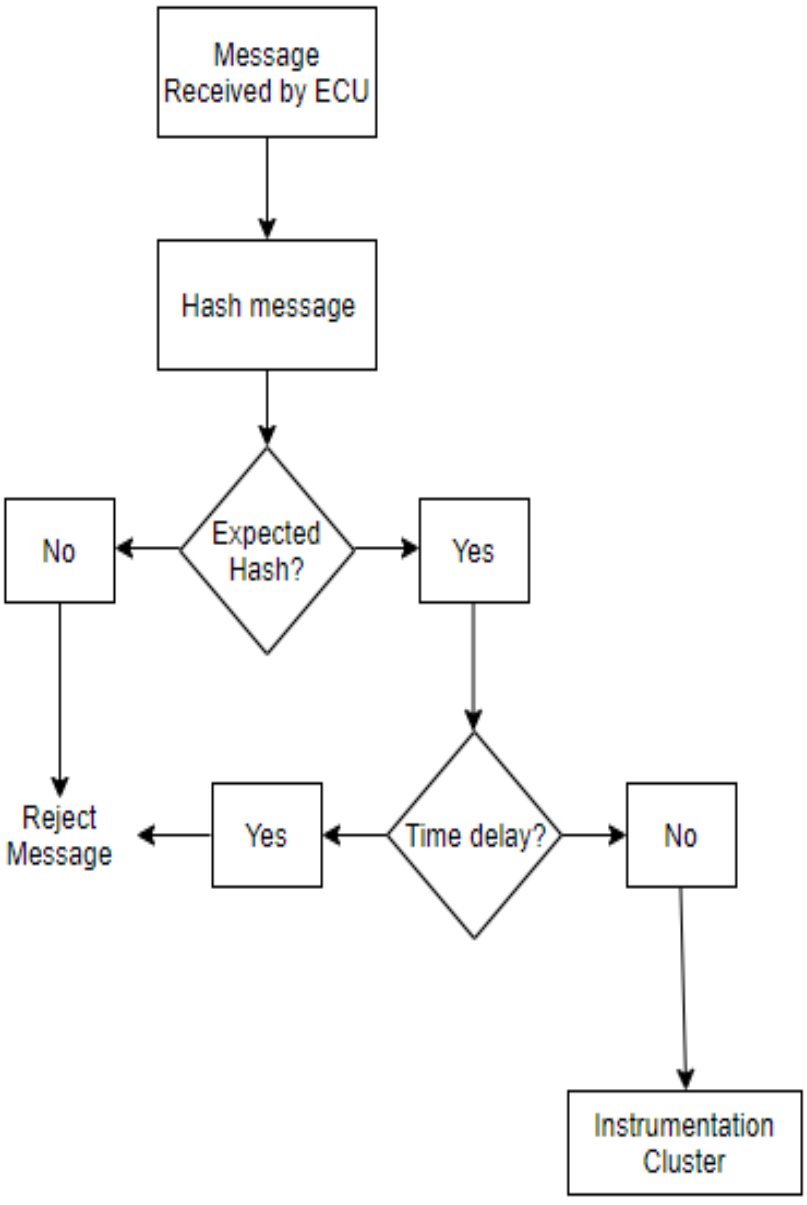



*Figure 51 The figure shows the process of a message after it is received by an ECU that is using SHA for the encryption method. The table highlights how a message would be rejected first if it does not have a recognized ID for the data.*

The first step in setting up message verification using SHA was to have the ECU begin to create a message digest using SHA. The input to make the message digest is the message ID plus an array of bits that show which hex values in the original message were populated, as well as the clock the time stamp on the message. Once the message digest is created it is checked against a list of stored hash values to determine if the message is valid. The stored hash values are seeded with a global timer to have them change based on time. This use of the timer allows for only messages that were sent recently to be received. This detailed process is the result of working with the developed CAN system to implement the most attack mitigations possible. The next step after verifying the process for message verification was to run a test on the network to determine how the system was impacted.

The last results to be observed was the performance impact that using SHA256 to mitigate attacks on the CAN system caused. This observation of the performance was measured by running two, one-minute tests of the can system. The first test was run without SHA to mitigate attacks to determine a control and set a base for what the number of messages sent across the system in a minute was. The second test run was implemented with SHA to mitigate attacks and the number of messages received and rejected. The first test produced 2,160 messages in the run time and the second run produced 2,810 messages, 980 of which were rejected messages and 1,830 were regular messages. The observation is that the use of SHA256 to verify messages increased the performance of the system by reducing the time each ECU spends processing attacks.

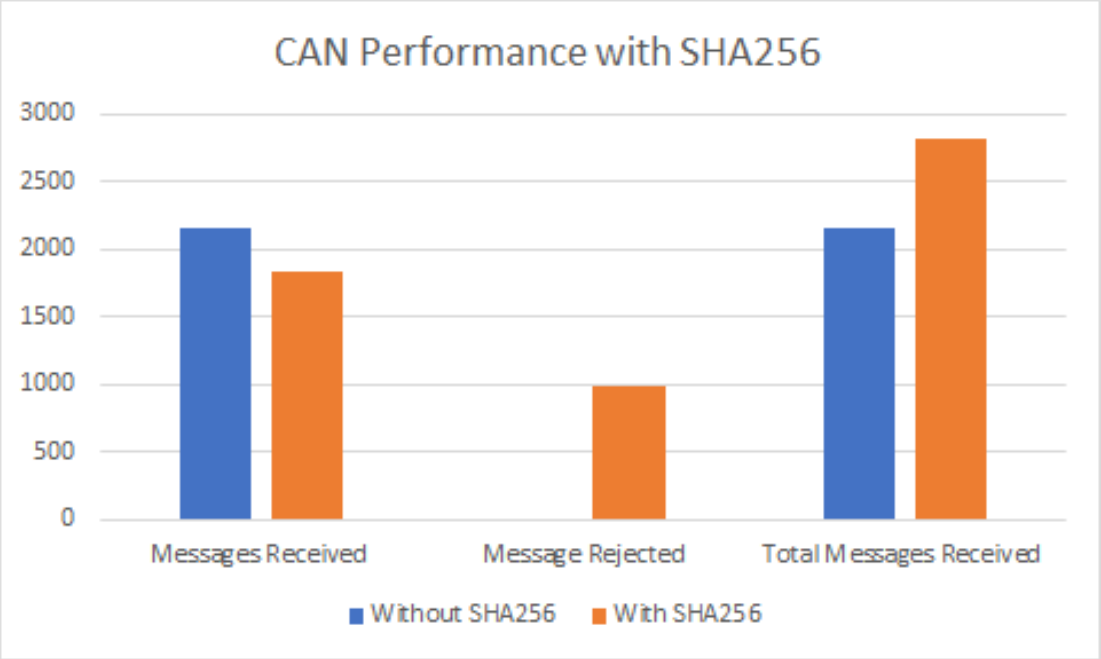



*Figure 52 The figure shows the results of a performance test run on SHA256 where a system was run with message verification, and without message verification to determine significant performance impacts on the system.*

*Table 5: Table showing results of Total Messages sent across CAN, without and with SHA256 encryption. The table highlights the total messages received, as well as messages rejected and accepted to determine if SHA256 impacts the performance of a CAN system.*

| Test | Messages Received | Messages Rejected | Total Messages | Run Time |
|---|---|---|---|---|
| Without SHA256 | 2,160 | 0 | 2,160 | 30:00 minute |
| With SHA256 | 1,830 | 980 | 2,810 | 30:00 minute |

The two, one-minute tests were run on the can systems while the ECUs ran SHA1 instead of using SHA256. These tests were run to see if there was a comparable difference in performance between SHA1 and SHA256. Since SHA1 is an older hashing algorithm and uses a bit size output of 160 instead of the 256-bit size SHA256 used it is a slightly less secure method. However, the smaller bit size output means that SHA1 uses slightly less inscriptions then SHA256 to reach a final output. The test showed slightly lower total messages being sent across CAN then the SHA256 test. The difference in the number of messages for the SHA1 test and SHA256 test is most likely due to human error in stopping the tests at the same time. The number of messages received was still close to the number of messages received previously so the limiting factor of the message total was the baud rate of the CAN system. The SHA1 implementation of message verification also had trouble accepting all the correct messages, rejecting a few properly formatted messages because the message was too low of a priority and did not reach the receiving ECU until after the timing stamp had expired.

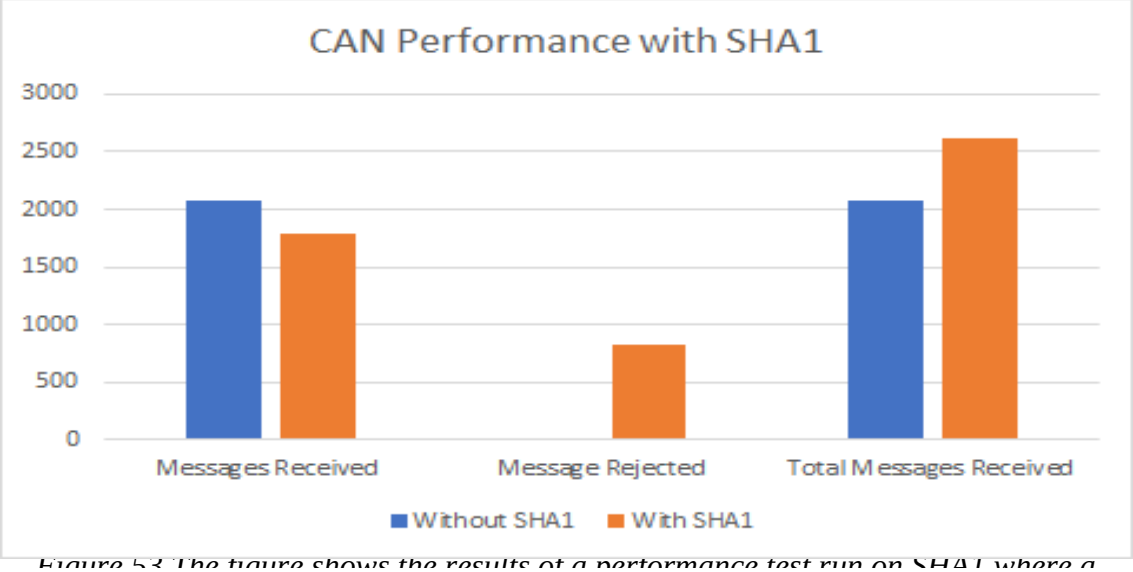



*Figure 53: The figure shows the results of a performance test run on SHA1 where a system was run with message verification, and without message verification to determine significant performance impacts on the system.*

*Table 6: Table showing results of Total Messages sent across CAN, without and with SHA1 encryption. The table highlights the total messages received, as well as messages rejected and accepted to determine if SHA1 impacts the performance of a CAN system.*

| Test | Messages Received | Messages Rejected | Total Messages | Run Time |
|---|---|---|---|---|
| Without SHA1 | 2,080 | 0 | 2,080 | 30:00 minute |
| With SHA1 | 1,790 | 825 | 2,615 | 30:00 minute |

## 5.2 Expected Results

This project produced some interesting results that did not perfectly align with the expected outcome. The expected results of this project were determined by what the goals of the project, trying to create a secure CAN system using SHA. The expected results were to have a software solution to stop ID spoofing attacks and timing-based attacks. The performance of the CAN system was expected to be reduced while running the message verification system. Finally, the use of SHA was expected to have a large impact on the resources of a Teensyduino. While these expected results were based in research and experience they did end up differing from the completed results of this project.

The first milestone of results was mitigating attacks on a CAN system, and the expected results were very similar to the results. Timing based attacks and ID spoofing attacks were both

shown to stop successfully sending messages when the SHA solution was implemented. The ECU required resources, processing speed and on-board memory, were not impacted as drastically as predicted and each ECU had memory to spare for additional programming. The ID spoofing attack was mitigated without any odd behavior being created in the CAN system but the implementation of SHA to stop timing-based attacks did have some performance errors. Some of the low priority messages being sent across CAN would also be rejected because it took the message longer to be sent across CAN then the timestamp on the message was valid for. This was not an expected outcome for how mitigating the timing-based attack would affect the CAN system. While the expected results did differ slightly from the determined outcome the use of SHA to mitigate attacks aligned with the project goals to create a secure CAN system.

The most unexpected result of implementing SHA for securing CAN was a slight performance increase on the system. The initial test without SHA was run and 72 messages were verified in a minute. The test with SHA received 95 messages total, which was a 31% increase in performance. The performance decrease was initially anticipated because of the additional time it would take to run the SHA implementation, delaying each message slightly. The performance increase happened because the bad message received on the network took less time to process when rejected allowing any ECU to process more messages in a smaller period.

## 5.3 Challenges

This project was not without challenges and setbacks that affected the development time and results. The first challenge for this project was when MITRE provided us with the hardware for each ECU. While receiving the hardware gave an advantage by saving time it also required a

significant amount of time to fully understand the hardware. Using time, research and trouble-shooting of pins provided us with all the information needed about the ECU.

The next challenge faced was attempting to implement the Flexcan library onto the Teensyduino. Implementing this library proved difficult because a secondary Flexcan library was defaulted into the Arduino IDE and a custom library had to be created to implement the proper version of Flexcan.

The development of a working CAN system was hindered by hardware restrictions and bugs as well. If attempting to use the Teensyduino with the MCP2551 a lower frequency for the baud rate was required. While both systems are capable of speeds up to 1 Mb/s together they can only communicate at 500 Kb/s. This change in described behavior caused a few problems with

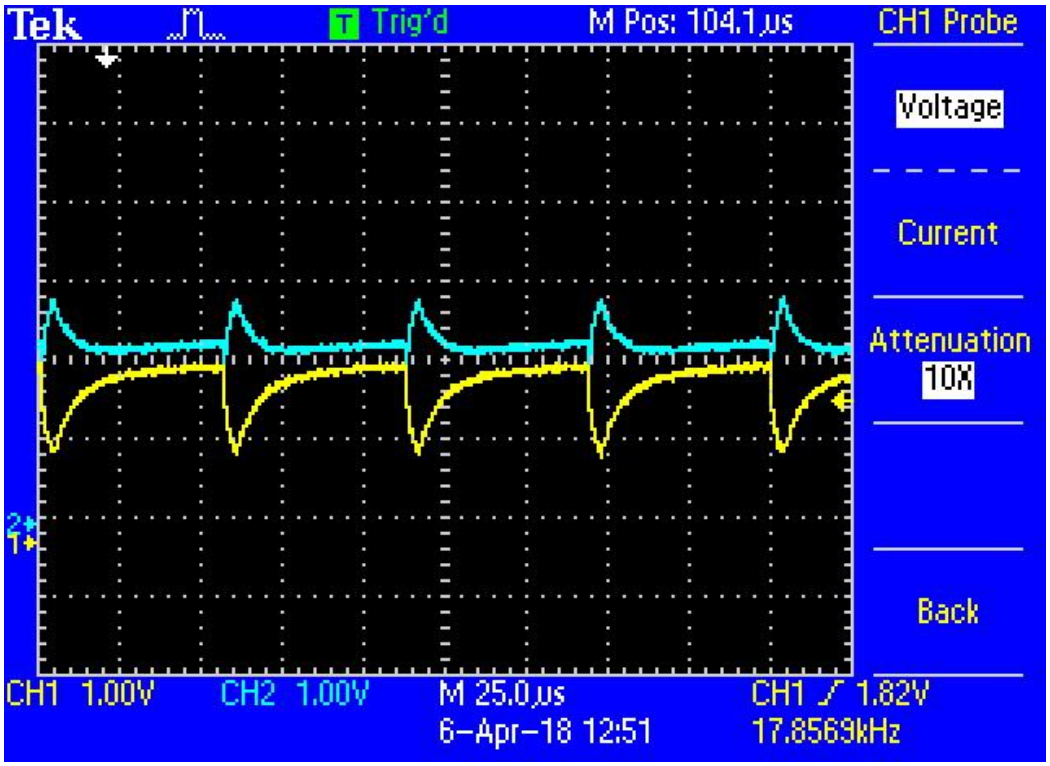



*Figure 54: Real-time signal of Multimode ECU cluster not displaying digital waveforms on either CAN High or CAN Low.*

trouble shooting and creating a working CAN system but was quickly resolved after information was provided by MITRE about this malfunction.

Implementing a multiple node CAN system with the instrument cluster also challenged the progress of this project. When initially connecting an ECU to the instrument cluster it appeared that no messages were being received, and an oscilloscope we had attached to monitor CAN high and CAN low was not displaying digital waveforms, as seen in Figure 54. Eventually it was discovered that the instrument cluster was not seen to accept the same ID values as referenced in an open source tutorial [20]. Correcting the message IDs allowed for the ECU to correctly communicate to the instrument cluster and additional ECUs. Additionally, the ringing effect where a digital waveform was expected was resolved by recreating the system with smaller connecting wires which may have changed the impedance of the system. Overall the challenges faced in the project were educational and did not impact the results of the project greatly.

## 5.4 Chapter Summary

The results of this project show clearly that the use of light weight encryption can be implemented into a CAN system without greatly impacting performance or adding large costs to a material budget. Challenges faced on this project created delays and sometimes changed the behavior of expected software, but each challenge was managed, and individual errors were addressed both with the hardware and software. While the results of the project were surprising in some cases most of the goals were fulfilled by the reported results. The effective use of SHA on a CAN system was seen to mitigate both timing-based attacks and ID spoofing attacks without adding additional hardware or reducing the speed at which the CAN system operated, showing that use of SHA can secure a CAN network.

# CHAPTER 6: CONCLUSION AND FUTURE DEVELOPMENT

This chapter finalizes and concludes all information about this project and briefly discusses the project and its results. It summarizes the importance of the project, methodology for implementation and key points from the results. Time constraints and resource setbacks are also acknowledged and recommendations for future work are included.

Through the development of this project knowledge was gained on the intricacies of security on a CAN bus. The CAN system is complicated and consists of many different function ECUs and implementation of transmission line hardware. The development of a secure CAN system was even more complicated with the addition of using SHA algorithms and logical processes to ensure message verification. The complete implementation of this project involved using the research provided for justification of the project, the development of multiple hardware CAN systems as well as the effective use of SHA algorithms on programmed ECUs to emulate attacks and the potential mitigation processes.

The development of the ECU and SHA algorithms for the use on the CAN system for message verification showed to be a very effective process. This process had minimal impact on the performance of the CAN system while allowing it to be secured. Since this implementation of message verification has no additional hardware components it is possible for the method to be implemented at a low cost and without impacting the production lines of modern vehicles.

The CAN harness used in this system along with the GUI provided valuable insight into the performance of the CAN system. Allowing the results of each test to be observed in real time the results were more easily determined to have an effective impact on the system. The

development of the CAN system being put onto a physical CAN harness also allowed the test benches to provide more accurate feedback since the system then represented a more realistic automotive CAN system.

The process of completing this project provided valuable insights to the security of CAN systems. The implemented use of message verification allowed of specific researched attacks to be mitigated. While challenges were faced on this project the success of the using light-weight encryption to verify messages proved to be an effective way of adding additional security to the system. The results of this project strongly support that uses of encryption on CAN systems, specifically SHA, allows for a cheap and robust solution to the growing problem of automotive security. The project should be continued through additional research and discover of new types of attacks and the continued exploration of what type of encryption will be the most efficient solution. Overall this project has highlighted the importance of explorative research in CAN security and provides a template for how continued development could provide additional insights into the security of CAN systems.

## 6.1 Future Work

Due to time constraints of the project there are still developments in the project that can be made to further the use of encryption to secure CAN systems. These developments include developing more vehicle functionality on the CAN, updating ECUs with additional encryption methods, continuing to work on mapping the pins on the CAN bus and continuing to predict types of attacks that can happen on the CAN bus.

One important item for future development is determining the pinout for more nodes on the CAN bus. Additional development can also be made in the way that the Harness and ECUs

are held. The harness is currently mounted in a configuration that makes it difficult to follow the communications path because it circles back in on itself. The shelfs for the ECUs can also use another round of development.

Another goal for future development is to update the ECUs on the CAN bus to more accurately represent parts of the motor and the electrical components. The best step to develop this is to either reference old reference tables for CAN codes or by attaching an ECU to an active CAN bus and recording the messages. The goal for this future work is to create the most robust test bed possible to get the best results on the encryption implementations.

Future work should also include more investigation into possible encryption methods. Continuing to explore encryption methods will not only reinforce the choice of encryption to use by providing supporting evidence but will also allow results from multiple methods to be explored, ensuring one of the most effective encryption method is used. Ideally both light weight encryption and encryption will be explored to widen the range of possible solutions.

# References

[1] Number of vehicles registered in the United States from 1990 to 2016. (2018). *Number of cars in U.s | Statista*. [online] Statista. Available: https://www.statista.com/statistics/183505/number-of-vehicles-in-the-united-states-since-1990/ [Accessed: March 22, 2018].

[2] R. Sharma, "In-Vehicular Communication Networking Protocol," Indiana University, Purdue University, Indianapolis, IN. [online] Available: https://pdfs.semanticscholar.org/f489/7797423eaa6c5a0a64891573b67d715d8a35.pdf [Accessed 1 March 2018].

[3] N. Gupta, V. Naik and S. Sengupta, "A firewall for Internet of Things," *2017 9th International Conference on Communication Systems and Networks (COMSNETS)*, Bengaluru, 2017, pp. 411-412.

[4] Texas Instruments. (2016). *Introduction to the Controller Area Network*. [online] Available: http://www.ti.com/lit/an/sloa101b/sloa101b.pdf [Accessed 1 March 2018]

[5] W. Voss, *Controller area network prototyping with Arduino*. Greenfield, MA: Copperhill Media, 2014.

[6] F. Sagstetter *et al.*, "Security challenges in automotive hardware/software architecture design," *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Grenoble, France, 2013, pp. 458-463. doi: 10.7873/DATE.2013.102 keywords: {Automotive

engineering;Batteries;Computer architecture;Plugs;Security;Vehicles;Wireless communication}, URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6513548&isnumber=6513446

[7] J. A. Bruton, "Securing CAN Bus Communication: An Analysis of Cryptographic Approaches," National University of Ireland, Galway, Ireland. Thesis. Aug. 2014. [online] Available: http://www.osna-solutions.com/wp-content/uploads/Securing-CAN-Bus-Communication-Thesis-Extract.pdf [Accessed 1 March 2018].

[8] J. Pagilery, "Your car is a giant computer - ant it can be hacked," *CNN Tech,* Jun. 2, 2014.

[9] P. Campbell, "Hackers have self-driving cars in their headlights," *Financial Times,* Mar. 15, 2018.

[10] J. Glassberg, "Cyber expert: Investors should get serious about smart cars being hackable," *Yahoo! Finance,* Mar. 25, 2018.

[11] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, T. Kohno et al., "Comprehensive experimental analyses of automotive attack surfaces." in USENIX Security Symposium. San Francisco, 2011.

[12] Github. (2018). *teachop/FlexCAN_Library*. [online] Available: https://github.com/teachop/FlexCAN_Library [Accessed: Jan 6, 2018].

[13] M. Moniruzzaman, Asaduzzaman, and M. F. Mridha, "Optimizing Controller Area Network System for Vehicular Automation," in *5th International Conference on Infomatics, Electronics and Vision*, pp. 586-591.

[14] Bosch, R. “CAN Specification,” Bosch, September, 1991. [online]. Available: https://www.kvaser.com/software/7330130980914/V1/can2spec.pdf [Accessed: Sep. 8, 2017].

[15] Dhananjaya Singh, Parma Nand, Rani Astya and Payal Dixit, “Improved DSA Cryptographic protocol and its comparative study with RSA protocol,” *2015 International Conference on Computing, Communication and Automation (ICCCA2015),* pp. 755-759, 2015.

[16] Rafael Alvarez, Candido Caballero-Gil, Juan Santonja and Antonio Zamora, “Algorithms for Lightweight Key Exchange,” *10th International Conference on Ubiquitous Computing and Ambient Intelligence*, pp. 536-543, 2016.

[17] A. Ali, "Comparison and Evaluation of Digital Signature Schemes Employed in NDN Network," *International Journal of Embedded systems and Applications (IJESA)* Vol.5, No.2, June, pp. 15, 2015

[18] C. Manifavas, G. Hatzivasilis, K. Fysarakis, K. Rantos, "Lightweight cryptography for embedded systems – a comparative analysis", in *SETOP (2013), LNCS, vol. 8247*, Springer: Egham, UK, 2013.

[19] M. Usman , I. Ahmed, M. Aslam, S. Khan and U. Sha ,"SIT: A Lightweight Encryption Algorithm for Secure Internet of Things", in International Journal of Advanced Computer Science and Applications (IJACSA), vol. 8, No. 1, 2017.

[20] B. Larry, M. Kerry, M. Nicky and T. Meltem, "Report On Lightweight Cryptography," National Institute of Standards and Technology, March, 2017, Pg 3. [online]. Available: https://doi.org/10.6028/NIST.IR.8114 [Accessed: Sep. 7, 2017]

[21] K. Masanobu and M. Shiho, “Lightweight Cryptography for the Internet of Things”, Sony Corporation, pp 7-10, 2008.

[22] 3rd, D. and Jones, P. (2018). *US Secure Hash Algorithm 1 (SHA1).* [online] Rfc-editor.org. Available: https://www.rfc-editor.org/info/rfc3174 [Accessed: Oct. 25, 2017].

[23] Google Online Security Blog. 2018. *Announcing the first SHA-1 Collision.* [online] Available at: https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html [Accessed: Dec. 2, 2017]

[24] “Laser Cutter User Manual” *WPI Manufacturing Labs*, 2015. [Online]. Available: https://drive.google.com/file/d/0B-WJgnYkeQC8aGJyNE5vOHhxWUU/view [Accessed: Feb. 26, 2018].

[25] Arduino.cc. (2018). *Arduino Reference.* [Online]. Available: https://www.arduino.cc/reference/en/language/functions/interrupts/interrupts/ [Accessed Oct. 16, 2017].

[26] Cantanko.com. 2018. *Reverse Engineering the RX-8’s instrument cluster, part one – Cantanko.* [Online]. Available: https://www.cantanko.com/rx-8/reverse-engineering-the-rx-8s-instrument-cluster-part-one/ [Accessed Jan. 20, 2018].

[27] Csrc.nist.gov. (2018). [online] Available at: https://csrc.nist.gov/csrc/media/publications/fips/180/2/archive/2002-08-01/documents/fips180-2withchangenotice.pdf [Accessed: Sep. 22, 2017]

[28] Wiki.python.org. (2018). *Tkinter – Python Wiki.* [online] Available: https://wiki.python.org/moin/TkInter [Accessed: Jan 15, 2018]

[29] H. Giannopoulos. “Re: CAN Harness” Personal Email (Dec. 19 2017).

[30] *2014 Chevrolet Impala Owner Manual*, General Motors LLC. 2014.

[31] Factory Repair Manuals, “2014 CHEVY IMPALA FACTORY SERVICE MANUAL SET ORIGINAL SHOP REPAIR,” *Factory Repair Manual*,[Online]. Available: https://www.factoryrepairmanuals.com/2014-chevy-impala-factory-service-manual-set-original-shop-repair/ [Accessed: Feb. 1 2018].

[32] MASTECH, “Series Digital Multimeter Instruction for MAS830 MAS830B MAS830L MAS838” [online] Available: https://www.jameco.com/Jameco/Products/ProdDS/220741.pdf [Accessed: Jan. 27, 2018]

[33] K. Linek and A. Hernandez, “Encoding and Physical Study of the CANBUS Sensor Network,” Worcester Polytechnic Institute, Worcester MA, Tech. Report. Apr. 2017. [online] Available: https://web.wpi.edu/Pubs/E-project/Available/E-project-042717-153728/unrestricted/ENCODING_AND_PHYSICAL_STUDY_OF_THE_CANBUS_SENSOR_NETWORK.pdf [Accessed: Jan. 10 2018]

[34] Gerardine Immaculate Mary, Z. C. Alex and Lawrence Jenkins, “Modeling and Analysis of Wireless Controller Area Network – A Review,” *Communications and Network*, 2013, pp. 126-133.

[35] Github. (2018). B-Con/crypto-algorithms. [online] Available at: https://github.com/B-Con/crypto-algorithms [Accessed: Sep. 22, 2017]

[36] Github. 2018. *FastLED/FastLED.* [Online] Available: https://github.com/FastLED/FastLED [Accessed Oct. 9, 2017].

[37] "Laser Cutter," in *Rho Beta Epsilon WPI Robotics wiki!* [online document], 2016. Available: http://wiki.wpi.edu/robotics/Laser_Cutter [Accessed: Feb. 26, 2018].

[38] Omar G. Abood, Mahmoud A. Elsadd and Shawkat K. Guirguis, "Investigation of Cryptography Algorithms used for Security and Privacy Protection in Smart Grid," *2017 Nineteenth International Middle East Power Systems Conference (MEPCON)*, Menoufia University, Egypt, pp. 644-649, 2017.

[39] Ieeexplore.ieee.org. (2018) *Optimizing controller area network system for vehicular automation – IEEE Conference Publication.* [online] Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=7760070 [Accessed: Dec 25, 2017]

[40] Infohost.nmmt.edu. (2018). *Tkinter 8.5 reference: a GUI for python.* [online] Available: http://infohost.nmt.edu/tcc/help/pubs/tkinter/web/index.html [Accessed: Jan. 15, 2018]

[41] Pjrc.com. (2018) *Teensyduino Tutorial #1: Software Setup for Teensy on Arduino IDE.* [online] Available: https://www.pjrc.com/teensy/tutorial.html [Accessed: Oct. 16, 2017]

[42] Intel. (2018). *Datasheet for the Intel Galileo Board.* [Online] Available at: https://www.intel.com/content/www/us/en/support/articles/000005735/boards-and-kits/intel-galileo-boards.html [Accessed 22 Sep. 2017]

[43] Ww1.microchip.com. (2018) Atmega328 Datasheet. [online] Available at: http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P_Datasheet.pdf [Accessed 22 Sep. 2017]

[44] Raspberrypi.org (2018) *Raspberry Pi 3*. [online] Available at: https://www.raspberrypi.org/documentation/hardware/computemodule/RPI-CM-DATASHEET-V1_0.pdf [Accessed 22 Sep. 2017]

[45] Sparkfun.com. 2018. [online] Available: https://www.sparkfun.com/datasheets/DevTools/Arduino/MCP2551.pdf [Accessed: Oct. 23, 2017]