

Data Logger Device for UDS Based Attacks in Automotive Vehicles

A Major Qualifying Project Report
Submitted to the Faculty
of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the
Degree of Bachelor of Science

By

Mark Bentson

Nicholas Kalamvokis

Santiago Rojas

Brian St. Germain

March 21, 2016

Advisors: Profs. Alexander Wyglinski & Hugh C. Lauer

This report represents work of WPI undergraduate students submitted to the faculty as evidence of a degree requirement. WPI routinely publishes these reports on its web site without editorial or peer review. For more information about the projects program at WPI, see <http://www.wpi.edu/Academics/Projects>.

Abstract

Vehicles are being manufactured with increasing numbers of programmable and interconnected electronic control units (ECUs), exposing them to hacking by outside agents. The Unified Diagnostics Services (UDS) have become an essential tool for manufacturers and technicians for servicing and updating vehicles [1]. However, they are also the path by which hackers can penetrate the electronics of a vehicle, gain access to its ECUs, corrupt their software, and gain ultimate control of the vehicle itself. In this project, a *data logger* was developed to keep a running history of communication within the vehicle to help a forensic team determine what was happening to the electronics before, during, and after an attack.

A prototype Data Logger was implemented using off-the-shelf microcontrollers and parts, and it successfully demonstrated the concept in a test bench that simulates an actual vehicle. However, the microprocessor and software libraries were discovered to be too slow to capture a perfect record of information *after* an attack. This report concludes with guidance for future projects.

Acknowledgments

Our group had great help and support throughout the completion of our project, and therefore we would like to thank:

- Our project advisors, Professor Alexander Wyglinski and Professor Hugh C. Lauer, for their timely feedback and assistance throughout the project.
- Hristos Giannopoulos for his help in the creation of our Data Logger prototype.
- William Appleyard for his help in gathering the necessary parts for our project.

Executive Summary

Vehicular electronics are becoming increasingly susceptible to hackers and the automotive sector is failing to catch up.

Introduction - The CAN Bus has exclusive control over the communication that occurs among the various electronic components within the vehicle. The CAN Bus features a set of diagnostic services call the *Unified Diagnostic Services* (UDS). These services are used to assist vehicle manufacturers, technicians, and owners in accessing critical parts of the vehicle such as diagnostic information and ECU specific functionality. If a malicious user were able to gain access to the vehicle's CAN Bus, he could potentially gain control over critical parts of the vehicle's infrastructure using UDS. After a vehicle hack and/or accident has occurred, it is vital that a forensics analyst can analyze a log of vehicle CAN data, conveying the state of the vehicle before, during, and after the attack has occurred.

The goal of this project was to develop a Data Logger that records sufficient information for a forensics analyst to determine a vehicle's condition before, during, and after a hack attempt and, if possible, discover the source of the hack attempt. We did this by developing a Data Logger that monitors a recent history of all CAN Bus messages and also captures and saves that history whenever a possible security threat is detected. The forensics analyst would use a UDS Guide to determine the extent of the attack, what ECU(s) were targeted, and if there was any damage to the vehicle. In an instance where an accident had occurred, the forensics analyst would be able to determine who is liable for the accident. The device developed in this project is a prototype of what we hope could become a common place tool for automotive experts to better protect their electronic subsystems from hacking.

Design Requirements - The data recorded must be read from a vehicle's OBD-II port and stored in a readable form, preferably in the format of a text file. This requires a programmable microcontroller for flexibility, portability, power efficiency, and the ability to adapt to differing attack strategies. The forensics analyst needs timestamps that indicate when packets were read from the bus, the packet's arbitration ID in HEX, and the packet's payload in HEX in order to properly determine the state of the vehicle and its peripherals. Table 1 shows the design requirements for the Data Logger.

Table 1: Design Requirements

| | |
|-------------------------------------|------------------|
| CAN Read Speed | 1500 packets/sec |
| External Storage Write Speed | 30 KB/sec |
| Random Access Memory (RAM) | > 64 KB |
| External Storage Space | > 1 GB |

Implementation - The hardware and storage analysis determined the selection of the Teensy 3.2 as the microcontroller and the SD card as the best storage option for the device. The Teensy 3.2 requires an SD card adapter with supporting C libraries to interface with SD cards. The BeagleBone Black was selected as the most desirable microcontroller to use for testing our Data Logger device. Figure 1 shows the final Data Logger setup also connected to the BeagleBone Black Test Bench.

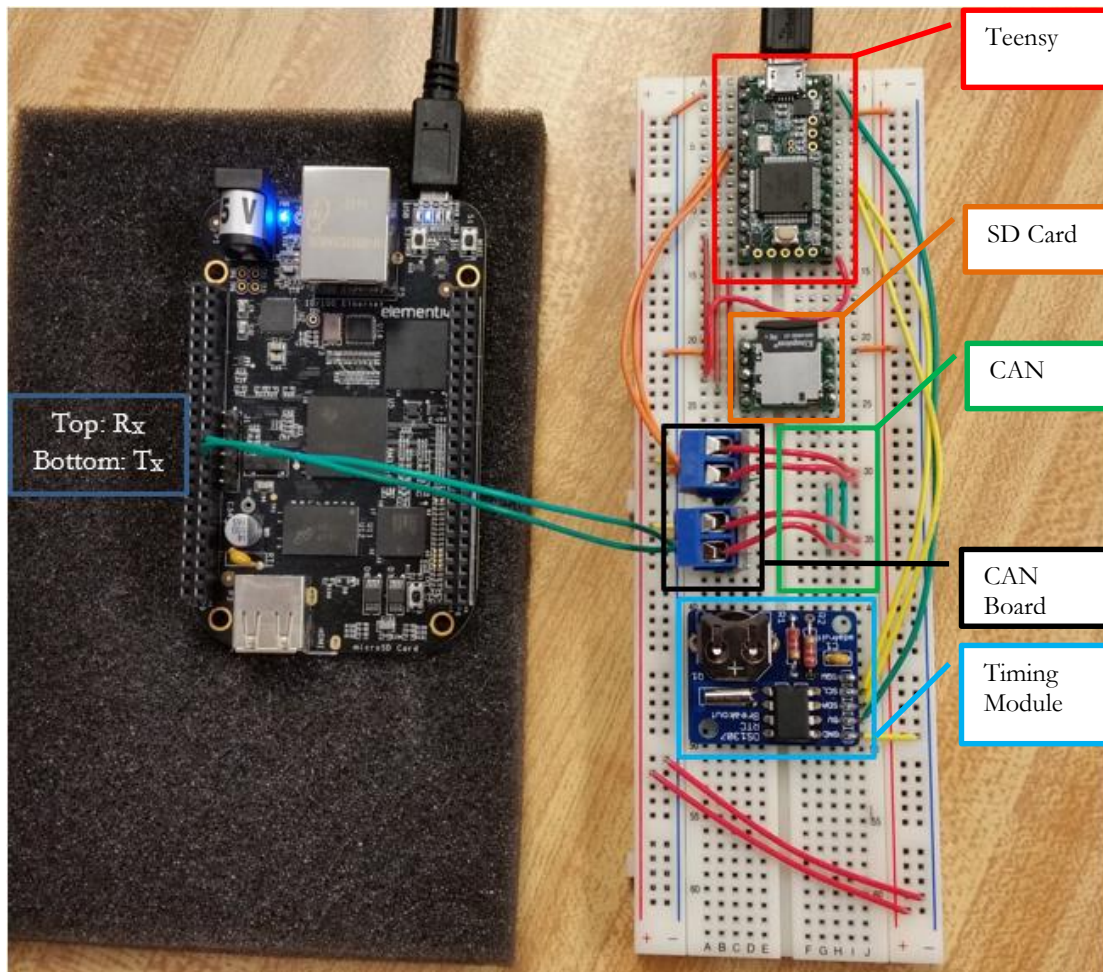


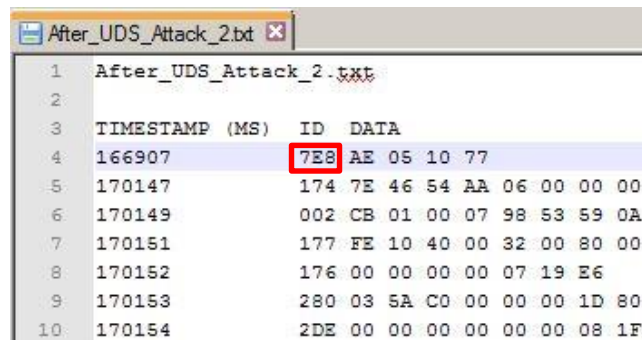
Figure 1: Picture showing the final Data Logger setup.

On the left is the BeagleBone Black sends the recorded Nissan Altima CAN packets to the SN65HVD230 CAN Board (Black box). This creates a CAN network (Green Box) which is then sent to the Teensy (Red Box). The information is saved to an SD card reader (Orange Box), using time information from the DS1307 RTC Timing Module (Light Blue Box).

The Data Logger device needs a timing module in order to get the current time. This is needed for recording the timestamps of the messages because the messages sent on the CAN network do not include timestamps in them. The timestamp is critical for this project because the forensics analyst needs to know when possible attacks occurred in the network and when any effects from the attack occurred.

The risk of damaging an actual vehicle was considered too great. So we created a Test Bench to simulate test data from an actual vehicle. The Test Bench was based on a BeagleBone Black microcontroller and the data was recorded from a Nissan Altima during normal driving in the Worcester area which is explained in Chapter 2.6. The Test Bench sends these test vectors to the Data Logger device by using the SN65HVD230 chip. The Data Logger reads all messages and runs an algorithm to select and log relevant messages when a UDS message is detected on the network. The Test Bench can also send modified test vectors that have more or fewer UDS messages.

Results - After executing multiple tests with the BeagleBone Black Test Bench and Data Logger, we examined the files created for each test. Figure 2 is a screenshot of the *After_UDS_Attack* text file, where the UDS packet AID of 7E8 is the first packet recorded. This screenshot displays the Data Logger is working properly and recording the CAN packets after the UDS packet is injected into the test bench.



| | TIMESTAMP (MS) | ID | DATA |
|----|----------------|-----|-------------------------|
| 1 | | | After_UDS_Attack_2.txt |
| 2 | | | |
| 3 | | | |
| 4 | 166907 | 7E8 | AE 05 10 77 |
| 5 | 170147 | 174 | 7E 46 54 AA 06 00 00 00 |
| 6 | 170149 | 002 | CB 01 00 07 98 53 59 0A |
| 7 | 170151 | 177 | FE 10 40 00 32 00 80 00 |
| 8 | 170152 | 176 | 00 00 00 00 07 19 E6 |
| 9 | 170153 | 280 | 03 5A C0 00 00 00 1D 80 |
| 10 | 170154 | 2DE | 00 00 00 00 00 00 08 1F |

Figure 2: Screenshot of the *After_UDS_Attack_2.txt* file.
The UDS arbitration ID is in the red square.

It became apparent during our analysis that about 28% of the packets were missed by the Data Logger when faced with a DoS attack. Since the Data Logger prototype is incapable of logging all packets that are monitored on the bus during and after a UDS attack has

occurred, further design and optimization is necessary. This issue can be attributed to the extended write latency when storing large amounts of CAN traffic to the SD card with a single threaded CPU. A possible solution to this problem is to use a microcontroller that is capable of running multiple threads on one or more CPU cores. One thread can be used to read the traffic on the CAN network while the other thread writes previously recorded traffic to the SD card. This design shift would entail further algorithm optimization in which a second circular buffer would be used to allow both threads to access the corrupt traffic at once. This would mitigate the need for the linear buffer currently used to assist in the stream to the SD card and make CAN data reading and writing parallel tasks.

Conclusion - Vehicle technology is continually advancing, in order to assist the driver and passengers by making their ride more safe and enjoyable with advancements such as assisted parallel parking, assisted braking, and automated highway driving. It is easy to see how this presents a security risk; if a hacker were to gain access to certain ECUs, he could control an automated or assisted vehicle service. Vehicle manufacturers are working to develop proper vehicle security, but have not been able to keep up with the technological advancements, in large part due to the long vehicle development time. This prototype Data Logger would be beneficial to both vehicle operators and manufacturers, as it would help protect the operators, while giving manufacturers important information about how hackers are using the UDS services to compromise vehicles.

Once final testing was completed, the Data Logger was concluded to perform appropriately, but some CAN packets were dropped. This can be attributed to two reasons: (1) the CPU on the Teensy is a single core, single threaded CPU and (2) the CAN libraries are not fully optimized. If the selected microcontroller had a CPU capable of multi-threading, one thread could read CAN data from the vehicle, while the other recorded this CAN data to external storage. A new library could be optimized for this new CPU in order to take advantage of onboard caches available to the developer.

Table of Contents

| | |
|--|-----|
| Abstract..... | i |
| Acknowledgments..... | ii |
| Executive Summary..... | iii |
| Table of Contents..... | vii |
| List of Figures..... | x |
| List of Tables..... | xi |
| 1.0 Introduction..... | 1 |
| 1.1 MQP Goals and Deliverables..... | 4 |
| 1.2 Report Organization..... | 5 |
| 2.0 Fundamentals of Vehicle Hacking..... | 7 |
| 2.1 Understanding Controller Area Network (CAN) Bus..... | 7 |
| 2.2 Unified Diagnostic Services (UDS)..... | 10 |
| 2.3 Attack Countermeasures..... | 12 |
| 2.3.1 Cyber Attacks on Automobile In-Vehicle Networks..... | 12 |
| 2.3.2 Securing Vehicles against Cyber Attacks..... | 14 |
| 2.4 History of Vehicle Hacking..... | 15 |
| 2.4.1 Ford Escape Hack..... | 15 |
| 2.4.2 Jeep Cherokee..... | 16 |
| 2.4.3 Telematics Unit Hack (UW and UCSD)..... | 18 |
| 2.4.4 LiDAR System and Autonomous Vehicle Hack..... | 20 |
| 2.4.5 Tesla Model S Hack..... | 21 |
| 2.5 Test Vehicle: 2014 Nissan Altima..... | 22 |
| 2.6 Initial CAN Bus Testing and Analysis..... | 23 |
| 2.6.3 CAN Bus Recording Analysis..... | 25 |
| 2.6.4 Linux Controller Area Networks Tools..... | 27 |
| 2.7 Chapter Summary..... | 27 |
| 3.0 Design Requirements..... | 29 |
| 3.1 Embedded Hardware Options..... | 34 |
| 3.1.1 Peripheral Buses..... | 34 |
| 3.1.2 Arduino..... | 34 |
| 3.1.3 Raspberry Pi B+..... | 35 |
| 3.1.4 Teensy 3.2..... | 36 |
| 3.1.5 BeagleBone Black Microcontroller..... | 37 |
| 3.1.6 Microcontroller Summary..... | 38 |
| 3.1.7 Arduino CAN Bus Shield..... | 39 |

| | |
|--|-----|
| 3.1.8 Raspberry Pi PICAN Module | 39 |
| 3.1.9 MCP2561 | 40 |
| 3.1.10 SN65HVD230 CAN board..... | 41 |
| 3.1.11 Adafruit DS1307 RTC Timing Module | 42 |
| 3.2 Hardware Analysis..... | 43 |
| 3.2.1 Data Logger Analysis..... | 43 |
| 3.2.2 Test Bench Analysis..... | 45 |
| 3.3 Storage Analysis | 47 |
| 3.3.1 Peripheral Buses | 47 |
| 3.3.2 Internal Memory (EEPROM)..... | 48 |
| 3.3.3 Flash Memory | 49 |
| 3.3.4 Secure Digital (SD) Card | 50 |
| 3.3.5 Selected Storage Device | 51 |
| 3.4 Project Logistics | 52 |
| 4.0 Implementation..... | 53 |
| 4.1 System Design..... | 53 |
| 4.2 Data Logging Algorithm | 57 |
| 4.3 Attack Scenarios | 59 |
| 4.4 Project Cost..... | 63 |
| 4.5 Chapter Summary..... | 65 |
| 5.0 Experimental Results | 66 |
| 5.1 Python Implementation Results | 66 |
| 5.2 Data Logger Implementation Results..... | 69 |
| 5.2.1 SD Card Writing Optimization Tests..... | 71 |
| 5.2.2 No UDS Attack Test..... | 72 |
| 5.2.3 Single UDS Attack | 73 |
| 5.2.4 Multiple UDS Attack..... | 74 |
| 5.2.5 Denial-Of-Service (DoS) Attack | 75 |
| 5.3 Chapter Summary..... | 76 |
| 6.0 Conclusion | 78 |
| 7.0 Recommendations..... | 79 |
| 8.0 Bibliography | 82 |
| Glossary Definitions..... | 88 |
| Appendices..... | 90 |
| Appendix A: Bibliography Annotations | 90 |
| Appendix B: Zip File Inventory | 101 |

Appendix C: Normal Packet Transfer (no UDS) Timing Data by Arbitration ID (AID)102
Appendix D: Packet Transfer (with UDS) Timing Data by Arbitration ID (AID).....104
Appendix E: Linux and Python Commands.....106
Appendix F: BeagleBone Black CAN Software Setup.....108

List of Figures

| | |
|---|----|
| Figure 1: Picture showing the final Data Logger setup..... | iv |
| Figure 2: Screenshot of the After_UDS_Attack_2.txt file..... | v |
| Figure 3: The number of ECUs in car brands such as a Ford Fusion and Toyota Prius..... | 1 |
| Figure 4: Example of general computer placement in a vehicle..... | 2 |
| Figure 5: CAN Bus Layer Architecture..... | 8 |
| Figure 6: Concept Diagram for Miller and Valasek 2011 Ford Escape Hack..... | 15 |
| Figure 7: A Jeep Cherokee's connections between the two CAN Buses and head unit..... | 16 |
| Figure 8: Black Hat security conference describing how the Jeep's Wi-Fi password was hacked..... | 17 |
| Figure 9: How GPS tracking was used to determine which Jeep Cherokee was their target vehicle..... | 18 |
| Figure 10: The hack performed by the University of Washington and San Diego students..... | 19 |
| Figure 11: Jonathan Petit imposed virtual objects with his \$60 device..... | 20 |
| Figure 12: Connecting an Ethernet cord to the dashboard and gain access to the vehicle..... | 22 |
| Figure 13: Sketch of BeagleBone Black CAN recording setup..... | 24 |
| Figure 14: Arduino Uno microcontroller..... | 35 |
| Figure 15: Raspberry Pi Schematics..... | 36 |
| Figure 16: Teensy 3.2 Schematic..... | 37 |
| Figure 17: BeagleBone Black Schematic..... | 38 |
| Figure 18: Arduino CAN Bus Shield..... | 39 |
| Figure 19: Raspberry Pi PICAN module..... | 40 |
| Figure 20: Pin outs for MCP2561 Chip..... | 40 |
| Figure 21: Schematic of the SN65HVD230 CAN Board..... | 41 |
| Figure 22: Schematic of the DS1307..... | 42 |
| Figure 23: Picture of the timing module on a breadboard..... | 43 |
| Figure 24: Image of the SD Card device for Teensy..... | 50 |
| Figure 25: Final Data Logger Setup..... | 54 |
| Figure 26: Final Test Bench Setup..... | 56 |
| Figure 27: Picture showing the final Data Logger setup..... | 57 |
| Figure 28: Algorithm flow chart describing both normal messages and UDS messages..... | 59 |
| Figure 29: Sketch of Case 1: No Attack..... | 60 |
| Figure 30: Sketch of the Case 2: Single UDS Attack..... | 61 |
| Figure 31: Sketch of Case 3: Multiple UDS Attacks..... | 62 |
| Figure 32: Screenshot of the two text files, <i>afterUDS_1.txt</i> & <i>beforeUDS_1.txt</i> | 68 |
| Figure 33: Screenshot of packets being received on the virtual CAN network..... | 68 |
| Figure 34: Screenshot is the after UDS file which consists of the packets after the UDS attack..... | 69 |
| Figure 35: Oscilloscope reading from the Data Logger with the BeagleBone Black..... | 71 |
| Figure 36: Screenshot of packets being received on the virtual CAN network..... | 71 |
| Figure 37: Screenshot of the Data Logger folder when there is no UDS attack..... | 73 |
| Figure 38: Screenshot of the Data Logger folder when there is a single UDS attack..... | 74 |
| Figure 39: Screenshot of the After_UDS_Attack_1.txt file..... | 74 |
| Figure 40: Screenshot of the Data Logger folder when there is multiple UDS attacks..... | 75 |
| Figure 41: Screenshot of the Data Logger folder when there is a DoS UDS attack..... | 76 |

List of Tables

| | |
|--|----|
| Table 1: Design Requirments | iv |
| Table 2: Portrays the different ‘attack surfaces’ that are available to a potential hacker. | 3 |
| Table 3: Normal Packet Transfer (no UDS) Timing Data by Arbitration ID (AID)..... | 25 |
| Table 4: Packet Transfer (with UDS) Timing Data by Arbitration ID (AID). | 26 |
| Table 5: Analysis of the Tick Representation. | 26 |
| Table 6: Packet format table which describes the fields of a CAN Bus packet. | 30 |
| Table 7: Circular Buffer (Normal Traffic)..... | 31 |
| Table 8: Linear Buffer after UDS packet detected (Corrupt Traffic). | 32 |
| Table 9: Microcontroller Specifications Summary. | 38 |
| Table 10: Mode specifications for the MCP2651..... | 41 |
| Table 11: Microcontroller Value Analysis. | 45 |
| Table 12: Test Bench Value Analysis. | 46 |
| Table 13: SPI Bus Signals and Respective Purpose. | 47 |
| Table 14: Teensy FlexRAM as EEPROM Specifications. | 49 |
| Table 15: Microchip SPI Flash Module Specifications..... | 50 |
| Table 16: SD Card Library Benchmarks..... | 51 |
| Table 17: Gantt Chart..... | 52 |
| Table 18: Hardware Specifications. | 55 |
| Table 19: Total MQP Cost Breakdown. | 64 |
| Table 20: Cost Breakdown for BeagleBone Black Test Bench..... | 64 |
| Table 21: Cost Breakdown for Data Logger Device. | 65 |
| Table 22: Linear buffer size C++ library results. | 72 |
| Table 23: Linear buffer size C library results. | 72 |

1.0 Introduction

Transportation has evolved since the introduction of the automobile in the late 19th and early 20th century. Primitive vehicles were essentially metal chassis coupled with gas powered engines. During this time period, there were approximately 8,000 vehicles in the US and only about 25,000 vehicles in the entire world [2]. As technologies became more advanced so did the number of automobiles that housed them. In the year 2002, there were about 130 million vehicles in the US, and about 530 million vehicles worldwide [2]. Starting in 1978, vehicle manufacturers began to place microcontrollers, sensors, and network interfaces into automobiles [3]. These microcontrollers are known as Electronic Control Units (ECUs). An *ECU* is one of many microcontrollers placed in vehicles in order to control the various operational, safety, and emissions functions of the vehicles. Before long, an automotive networking standard called *CAN Bus* (*i.e.*, *Controller Area Network*) emerged as the universal way for such devices to communicate and work together during the operation of the vehicle [4]. Figure 3 displays a few models of cars and their increasing implementation of microcontrollers in the recent years [5].

A growing number of computers in cars

The number of these computers has grown dramatically as manufacturers add more technology to improve car safety and the user experience. Often, new features are patched into existing systems, making the system even more interconnected and vulnerable to attack.

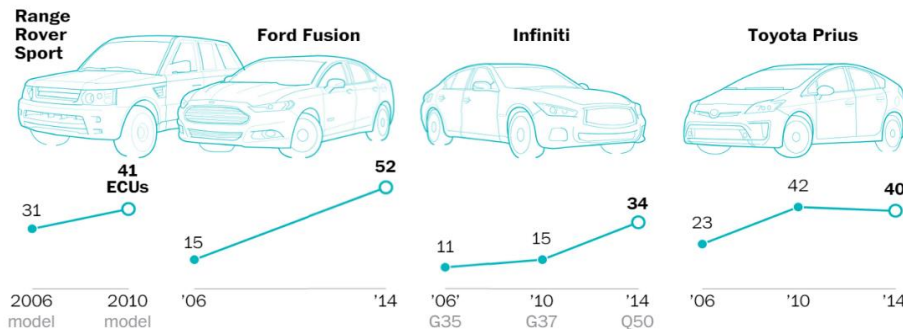


Figure 3: The number of ECUs in car brands such as a Ford Fusion and Toyota Prius.

This information shows how in only about a 10 year span, the number of computers in cars has grown a significant amount.

Some of these microcontrollers are used for processes such as fuel emissions, which are controlled by the vehicle's main ECU (*i.e.* the Central Processing Unit or CPU), in order to ensure safe levels of CO₂ are emitted into the atmosphere. Currently, vehicle manufactur-

UDS Based Attack Data Logger

ers are designing vehicles with additional electronic components to provide advanced capabilities such as built-in wireless networks. A general overview of the location of microcontrollers in a vehicle is shown in Figure 4 [6].

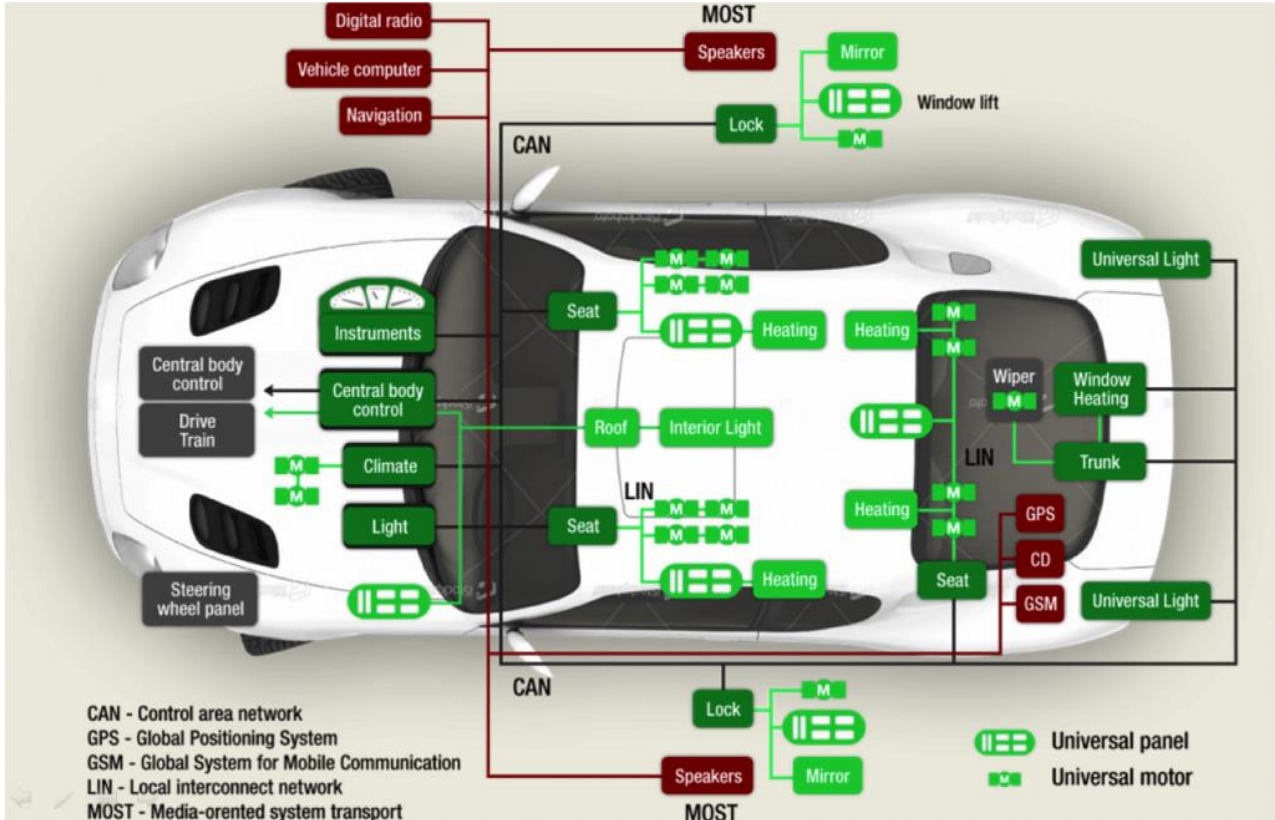


Figure 4: Example of general computer placement in a vehicle.

This is a big step for the future of vehicle manufacturing but comes with significant risks and vulnerabilities. Almost every computer has inherent security flaws, but the addition of a networking system creates new vulnerabilities. Some vehicle manufacturers have also begun to implement an increasing number of networks in new automobiles to accommodate other technologies such as cellphones, web-based services, and diagnostic tools. The reason for multiple CAN Bus networks is to divide the vehicle, as one network usually controls just the engine components, while the other controls the rest of the vehicle. An overview of some of the potential vulnerabilities in vehicles can be seen in Table 2 [5]. Consequently, this has led to an increasing number of attack surfaces¹ in vehicles; thus, increasing the number of vulnerabilities in a vehicle an attacker could exploit [7]. Even though vehicles today are

¹ An attack surface is any part of a vehicle that is vulnerable to an attack.

more advanced than those designed over 100 years ago, they present risks that the public never imagined and that the automobile industry cannot afford to neglect [7].

Table 2: Portrays the different ‘attack surfaces’ that are available to a potential hacker.

| Finding Ways into a Vehicle's Network | |
|--|---|
| Gaining Access/Attack Surfaces | Explanation |
| Bluetooth | Phones paired with the vehicle |
| Radio System | Processing data such as the name of the song |
| Cellular/WIFI | Connection to the Internet |
| Taking Control | Explanation |
| Adaptive Cruise Control | Computer controls speed of the car based on road conditions |
| Collision Prevention | Engages brakes to prevent a crash |
| Lane Assist | Uses vehicle's sensors to stay in the lane by sending signals to the steering wheel |

In order to accommodate all the technologies being added to automobiles today, manufacturers have designed a standardized vehicle network called the Controller Area Network (CAN) Bus [4]. The CAN Bus has exclusive control over the communication that occurs between the various electronic components within the vehicle. Devices connected to the CAN Bus communicate with each other in real-time. Security within the CAN Bus is a serious issue because the vehicles’ computers are connected to it. If a malicious user were able to gain access to the vehicle’s CAN Bus, he or she could potentially gain control over critical parts of the vehicle’s infrastructure. This malicious user could perform a range of attacks. The following are examples of such attacks performed in laboratory settings:

- Turned the steering wheel, controlled the light, and turned off the engine [8];
- Controlled the vehicle wirelessly (i.e. lights, brakes, and steering wheel) [9] [10];
- Gained access to a vehicle via the OnStar unit [11];
- Deceived an autonomous vehicle’s LiDAR system to make the sensors believe that there were objects nearby [12] [13];
- Left a virus in a Tesla S model that could turn off the power at any time [14].

Vehicle manufacturers are beginning to acknowledge the potential dangers of these attacks, and are working to find solutions to potential security flaws. For example, the Miller and Valasek network attack described above demonstrated that it is possible to hack into an

UDS Based Attack Data Logger

entire series of Chrysler vehicles through the Sprint cellular network featured in the dashboard infotainment system [15]. This demonstration forced Chrysler to recall over 1 million vehicles in order to fix this security flaw [15]. National news for these hacks demonstrate that there is a need for a device that can detect when a vehicle is being hacked and that logs relevant information about the attack.

1.1 MQP Goals and Deliverables

At the beginning of this project, the goal was to research into how vehicles communicate over the CAN Bus, and then reverse engineer a CAN Bus packet to force a test vehicle to perform a specified action. To accomplish this goal, we read CAN Bus packets from our test vehicle (a 2014 Nissan Altima) and attempted to determine which messages corresponded to which actions in the vehicle, such as door locks and door windows.

Once testing was completed, we began to send reverse engineered packets back into the test vehicle attempting to provoke changes. After many attempts, we were unable to force the vehicle to perform tasks such as locking/unlocking the doors or opening the windows. This led to more research about using UDS messages to hack the car. UDS messages are used to enter the diagnostic mode of the vehicle, which allows easier access to controlling the vehicle by directly communicating with the ECUs in the car. We discovered from our research that one problem with this approach is that UDS messages can seriously damage the ECUs of the vehicle if the wrong commands are used or if the ECUs are overloaded with them [1]. With this new knowledge in mind, we decided to change our scope to developing a Data Logger device, especially after the team discovered that safety articles described in Chapter 2.3.

The goal of this project is to develop a Data Logger that records sufficient information for an expert or forensics analyst to determine a vehicle's condition before, during, and after a hack attempt and, if possible, discover the source of the hack attempt. We will do this by developing a Data Logger that monitors a recent history of all CAN Bus messages and also captures and saves that history whenever a possible security threat is detected.

Our device will be a prototype of what we hope will become a common place tool for automotive experts to better protect their electronic subsystems from hacking. Given the previously mentioned goals, the team developed the following deliverables:

- A data logging algorithm hosted on a Linux virtual machine that serves as a proof of concept, used to simulate the microcontroller and the vehicle;
- A data logging device that uses an algorithm similar to the one developed on the virtual machine to capture CAN data in the case of a vehicle attack;
- A demonstration of this device working on a test bench that replicates a vehicle's CAN Bus;
- Ground work for future efforts and MQP in this area:
 - Documentation of a vehicle attack demonstration;
 - Tutorial explaining how to interact with the CAN Bus;
 - Information on how to interpret CAN Bus messages and what tools to use for reading;
 - A project report that details the background, methodology, results, analysis, and conclusion.

1.2 Report Organization

This report starts by providing a brief background in vehicle hacking in Chapter 2. This includes:

- Case studies highlighting recent hacks that have been conducted by students and professionals in order to give the reader background about current vehicle security vulnerabilities;
- Information about the internal communication system used in most vehicles;
- Diagnostic protocols used in the vehicles, as well as its potential dangers;
- Published articles displaying the need for improved vehicle security.

Chapter 3 analyzes the hardware options available, presents the hardware alternatives selected for the project, presents relevant information about the target vehicle, and explains the proposed attack model and research model. Chapter 4 explains design implementation

UDS Based Attack Data Logger

and testing, which includes the chosen hardware and software that fulfills the Data Logger's requirements. Chapter 5 discusses the final results from the Data Logger's testing. Chapter 6 and Chapter 7 concludes the report and provides suggestions for future projects.

2.0 Fundamentals of Vehicle Hacking

This section describes network interfaces, hack attempt case studies, vehicle safety articles, and potential hardware options. Information about network interfaces provides an understanding of networks currently used in vehicles. There are many potential hardware options to consider when developing a vehicle hack. A few of these options are discussed in this section. The case studies are presented to convey methods of vehicle hacking and outcomes of successful hacks.

2.1 Understanding Controller Area Network (CAN) Bus

The *CAN Bus* is a transfer medium by which Electronic Control Units (ECUs) in a vehicle can communicate, signal errors, and recover from faults. An *ECU* is one of many microcontrollers in a vehicle that operates specific actuators and components such as the windows, steering wheel, engine, transmission, etc. The CAN Bus features a versatile multicast, multi-master protocol, meaning that every ECU (node) on the CAN Bus can send data to and receive any data packet from any other node on the bus. The system supports multiple transfer rates (most manufacturers use a data rate of 500 Mbps) and no theoretical limit as to the number of nodes on the bus, making this a flexible solution for vehicle manufacturers. The CAN network implementation details in this section were obtained from the Bosch CAN specification documents [4] [16].

The CAN Bus protocol is partitioned into four layers; the Application Layer, Logical Link Layer (LLC), Medium Access Control (MAC) Layer, and Physical Layer. The detailed layer architecture of the CAN Bus is shown in Figure 5 [4].

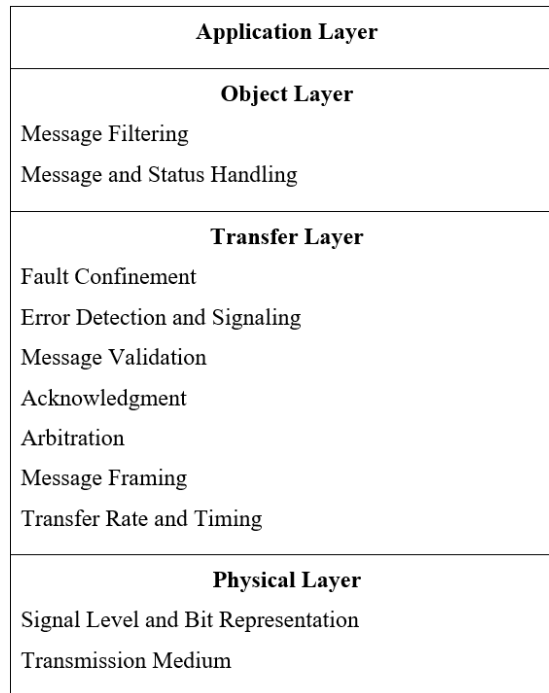


Figure 5: CAN Bus Layer Architecture

The *Application Layer* contains all the peripheral devices and utilities in the system. Some examples of peripherals include the odometer, infotainment system, windows, mirrors, etc. The *Logical Link Layer*, more widely known as the *Object Layer*, provides an interface for the Application Layer hardware. The Object Layer is responsible for determining what messages are to be transmitted. It is also the responsibility of this layer to decide which messages received at the transfer level will actually be used by a node. The layer below the Object Layer, the *Medium Access Control (MAC) Layer* (usually referred to as the *Transfer Layer*), is responsible for interacting with the hardware in order to handle timing, error checking/signaling, fault handling, and bus arbitration. The Transfer Layer is standardized and cannot be modified by the developers of the system. The *Physical Layer* is the physical medium in which the bits are transferred. This layer must be the same for all the nodes on a single network, but the developers of the system have the freedom to choose what wiring to use.

Several key features of the CAN Bus make it reliable in a real-time environment. The most prominent of these features is the CAN's arbitration technique. Arbitration is a method by which the CAN network gives access to a node attempting to transmit a message based on its *arbitration ID* (similar to a priority number). On the CAN Bus, any node can transmit a message as long as the bus is idle, but when more than one node wants to transfer data at

the same time, arbitration must be used to decide which node will send data first. When there is a dispute on the bus, the node with the highest priority (lowest arbitration ID) wins arbitration and the rest of the nodes listen, waiting for the bus to become idle again. Each node must check that the bus is either idle or transmitting at a lower priority than itself, as there is no arbitrator on the bus. For example, if both the transmission and the infotainment system would each like to send messages on the bus, the transmission gains access first because it holds a higher priority than the infotainment system. This makes the CAN more reliable than traditional forms of data transfer that work based on a FIFO queue, especially for mission critical applications within a vehicle.

Another important feature that the CAN Bus offers is the ability for one node to request information directly from another node. The node requesting data sends a *Remote Frame*. This frame contains the arbitration ID of the target node. When a node receives a remote frame with its arbitration ID, it returns a *Data Frame* with the data block that was requested. This makes it simple for a node to access data from other nodes within the CAN network. Data Frames and Remote Frames contain the same fields, with the exception of the Data Length Code and Data Field, which are only necessary in a Data Frame. The Data Field simply contains the data to be transferred. This field can contain 0 to 8 bytes, each containing 8 bits that are transferred most significant bit first. The number of bytes in the Data Field is indicated by the Data Length Code. The Data Length Code can be extended to fit larger data sizes in the flexible data-rate implementation of CAN, but for our purposes, this field will limit the Data Field to a maximum 8 bytes.

The CAN Bus relies on its network of nodes to detect and signal any data transmission errors. The types of errors experienced on the CAN network are defined in the Glossary. A node can detect an error while transmitting or receiving a data packet. For this reason, each node keeps record of errors using transmit and receive error counters. When a node receives a packet, it uses a CRC check to ensure the packet's integrity and will simply not reply with an acknowledgement packet if a mismatch is detected. In this case, the node that transmitted this packet will not receive an acknowledgement within a specified timeout and will resend the packet when the bus becomes idle. Nodes detect errors while transmitting packets by monitoring the bus as they send each bit. If the transmitting node detects that the data on the bus is not what was sent, it will retransmit the entire packet as soon as

UDS Based Attack Data Logger

the bus becomes idle again. This is an added check that makes the response time of the CAN faster than most data transfer protocols.

When an error is signaled, either an Active or Passive Error Flag is transmitted on the bus. An *Active Error Flag* consists of six bits of value zero, and a *Passive Error Flag* consists of six bits of value one. The type of error flag is decided by the state of the node sending the flag. A node self-polices and may set to an Error Active, Error Passive, or Bus Off state. A node in the *Error Active* state can take part in bus communication and sends an Active Error Flag when an error is detected. When a node in the Error Active state causes 128 or more errors, the node is set to the *Error Passive* state. In this state, a node can take part in bus communication but must suspend data transmission after every successful packet transfer. A node in the Error Passive state may only send Error Passive Flags, which have less authority than Error Active Flags. If a node in the Error Passive state proves itself to be trustworthy, meaning it does not produce any more errors, it can be set to the Error Active state again. If a node in the Error Passive state continues to produce errors up to a count of 256, the node enters the *Bus Off* state. This means that the node may no longer influence any communication on the bus. An outside source such as a vehicle restart must be used to reset this penalty.

2.2 Unified Diagnostic Services (UDS)

The UDS details in this section were obtained from an online article describing UDS [1]. Since the introduction of networks in automobiles, manufacturers have tried to push the bounds of what vehicle technologies can achieve. With these advancements came the Unified Diagnostic Services (UDS) described in the 14229-1:2013 International Organization for Standardization (ISO) specification [17]. The UDS standard is implemented to be transferred over the CAN Bus and is primarily accessed via the OBD-II port. The *OBD-II port* is a required standardized means of communication with the vehicle, usually located under the steering wheel. It is a node on the CAN network and, therefore, monitors all data packets. UDS serves as a means by which manufacturers can load firmware and run initial tests on vehicles. UDS capabilities also expand to the service market where technicians and vehicle owners can clear vehicle-warning codes and download diagnostic information from the vehicle's OBD-II port with external hardware. This is usually done to troubleshoot problems

with vehicles. These applications make UDS a very important tool for manufacturers, service technicians, and vehicle owners.

Vehicle technicians and owners can utilize UDS through a *UDS Diagnostic Reader*, which is a transceiver that connects to the OBD-II port of the vehicle. It sends UDS messages targeting certain ECUs and their status registers in order to query the diagnostic data. It then records the responses from the vehicle's ECUs and presents the data to the user in a readable format. These devices are typically sold in hardware stores and are fairly useful for determining which component(s) of a vehicle is not working correctly.

UDS messages are sent via the CAN Bus with a specific arbitration ID and are monitored on all ECUs as well as the OBD-II port. ECUs can choose whether or not to filter these messages based on their UDS Service IDs. The *UDS Service ID* is the first data byte of a UDS message and is used to communicate which ECU on the network the message is targeting. Since a UDS message is always directed at a particular ECU, the targeted ECU must respond to the request with a Reply Service ID. The *Reply Service ID* is typically the value of the UDS Service ID with a single bit flipped to create a small offset in value. All other ECUs on the network ignore the UDS message as it was not directed to them. The targeted ECU reads the data within the UDS message, which will indicate a command and/or data that must be acted upon. An example of this is the case where the manufacturer loads firmware onto ECUs in the factory. A UDS message is sent to command the targeted ECU to start its bootloader. After the target ECU has started its bootloader, UDS messages will continue to be transferred, providing parts of the new firmware until the entire image is loaded. This is just one of the many useful commands UDS provides manufacturers.

Although UDS has been proven to be an essential tool for manufactures, service technicians, and vehicle owners, it is also an avenue by which hackers gain control of the CAN Bus.

Hackers have been able to use the diagnostic capabilities of UDS to access ECU boot loaders for the purpose of injecting malicious software into the ECUs, which can have detrimental effects on the vehicle and its passengers, especially if the vehicle is in motion when the hackers exploit the system. Situations where hackers exploit the CAN Bus via UDS

UDS Based Attack Data Logger

are commonly known as *UDS Attacks*. What makes UDS attacks more dangerous than normal vehicle hacks is the array of functionality the hacker obtains once on the bus.

Normally, UDS messages used by UDS Diagnostic Readers only execute read operations on ECUs, making them safe for the vehicle. Hackers can gain access to the UDS write functionality either with the standard UDS guide for a particular line of vehicle or by testing commands on a test vehicle. By testing commands, the hacker can determine what UDS Service IDs correspond to ECUs and what message payload data can cause harm. Once a hacker compromises the CAN Bus of a vehicle, he is free to run whichever UDS commands he chooses. This means the hacker can compromise critical components on the vehicle's CAN network while the vehicle is in motion. Depending on the vehicle and manufacturer, this could mean that the brakes, transmission, and engine are capable of being compromised, making the vehicle dangerous for the all passengers.

2.3 Attack Countermeasures

In the past, there was little need to worry about wireless hacking into vehicles because not enough of the technology was in place. However, researchers and hackers have demonstrated that this is no longer the case. All modern vehicles manufactured after the 2008 standardization of CAN Bus are vulnerable to hacking. This section describes some of the methods by which hacking can be analyzed and potential ways by which vehicles can be secured. Much of this information is taken from articles written by Dennis Nilsson and Ulf Larson [18] [19].

2.3.1 Cyber Attacks on Automobile In-Vehicle Networks

In theory, a hacker could use the new wireless technology to gain remote access to the in-vehicle network. This type of attack becomes more dangerous as the in-vehicle wireless technology could be used to create a remote connection between two vehicles that can lead to a multi-vehicle attack. Since this development of new vehicle technology, the need for a new forensics investigation approach must be established. Nilsson and Larson make their point with the following example [18]:

“As an illustration, consider the case of a speeding vehicle that hits the face of a rock. This incident is either caused by the driver itself, or by vehicle malfunction or physical

tampering. If the brake wire is found to be cut, the cause of the accident is most certainly an act of physical tampering, and a criminal investigation needs to be initiated to bring the responsible party to a court of law. Consider instead the possibility that the brakes were disabled by a piece of malicious code. If there is no digital evidence available, the criminal would walk free, and the cause of the accident would wrongly be determined as malfunction.”

In the past, vehicle forensics were mainly focused on physical accident reconstruction; determining the physical condition of the car and checking the status of brakes, wipers, lights and other vehicle systems. This information, while useful, does not help during the investigation of a vehicle hacking attack. A solution for this was created in the early 80’s called the event data recorder (EDR). This device records critical event data such as speed, braking, rpm, seat belt status. These devices are now being implemented into modern vehicles for insurance companies to determine the driver at fault in an accident. An EDR helps for a vehicle crash, but will not provide sufficient information if a cyber-attack were to occur. The EDR only records when a crash occurs, and still only records for at most 5 seconds before the crash. A hacker could either wait to perform the hack or hack the EDR itself to give the insurance companies false information [20].

Nilsson and Larson suggested requirements and conditions to support a digital forensic investigation:

- A method to detect events in the vehicle must be present. To perform a digital forensics investigation, an alert about a security violating event must have been triggered to provide reason to initiate the forensics investigation.
- Data to answer the questions who, what, where, when, and why must be produced in the vehicle. During the forensics investigation, this data must be available in the ECUs for an investigator to extract the necessary information when needed.
- Information about the current state (e.g., firmware versions) in a vehicle must be available and stored in a secure location. To detect whether the vehicle has been tampered with, the extracted data must be compared to the original data.

An appropriate tool would be a Data Logger device to capture the states of all of the devices on the CAN bus. This tool would have to produce answers to the questions who,

UDS Based Attack Data Logger

what, where, when and why of the attack and also information about the current state of the vehicle.

2.3.2 Securing Vehicles against Cyber Attacks

In a different article, Nilsson and Larson [19] discuss the advancement in vehicles and how newer vehicles are being manufactured with wireless technology. This technology presents an increased potential for a cyber-attack. The major concern is that manufacturers are equipping new vehicles with these wireless capabilities, yet not putting in place significant security features in order to protect operators from an attack. The article goes into detail about the five specific layers of defense that should be put in place. The five layers are:

- Prevention – Defense such as a firewall that prevents an attack from happening.
- Detection – Defense that detects when an attack is happening and also logs the proper data in order for analysts to detect what exactly the hacker was trying to do
- Deflection – Defense that will lead the hacker to some insignificant part of the vehicle and deter him away from crucial parts of the vehicle.
- Countermeasures – Defense that if a hacker gains access to the vehicle, works in order to attack the hacker and prevent harm coming to the vehicle. It must also be built to not interfere with normal vehicle functions.
- Recovery – Defense that allows the vehicle to recover from a potential hacker attack. This involves logging data of the attack and being able to determine what has been touched and hacked into. This data must also help bring the vehicle back to its normal steady working state.

Our MQP would work to help with the Detection and Recovery layers. Our device would be placed or installed in a car, connected to the CAN bus, and would detect UDS attacks. Once this is detected, data will be logged so that after the vehicle has been secured, experts can read the data to determine what the hacker was able to gain access to. This also will help with the Recovery layer because it will enable the experts to determine the next steps in order to fix any part of the vehicle that may have been harmed. This is a very important first step in designing and implementing vehicle safety features that prevent potential cyber-attacks. It is important to note that our algorithms will not be able to diagnose the

problems directly, but instead will provide the needed information to a forensics expert to do so.

2.4 History of Vehicle Hacking

Everyday objects such as TVs and refrigerators are being installed with programmable microprocessors and Internet capabilities [21]. This technological expansion has also reached the automotive industry. Automobiles are essentially computers on wheels; they are not as mechanically controlled as they were, even in the recent past. Vehicle manufacturers are implementing new technologies such as built-in cellular network capabilities. Although these new technologies grant the driver more functionality, they create a significant safety risk due to hacking and malicious software. Since the design and production cycles of modern vehicles are normally between 2-5 years, vehicle manufacturers have not been able to keep up with the software security challenges [22]. Most of the work has been published by independent researchers working with recent model cars. This section describes some of that work.

2.4.1 Ford Escape Hack

Some of the early attempts to hack a vehicle were performed by researchers Miller and Valasek [8], who began their studies using a 2010 Ford Escape. Their first hack involved connecting a cable from the vehicle's On-Board Diagnostics (OBD-II) port to a laptop. The physical setup for this hack, with the laptop directly connected to the OBD-II port of the Ford Escape can be seen in Figure 6.

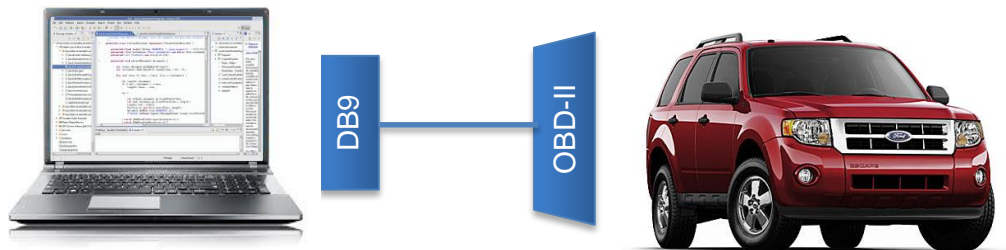


Figure 6: Concept Diagram for Miller and Valasek 2011 Ford Escape Hack.

Through this connection, they were able to gain direct access to the CAN Bus. The researchers found that the security system in this vehicle was not able to determine the origin

UDS Based Attack Data Logger

of the information packets. Exploiting this flaw, they were able to inject packets to the CAN bus and impersonate a certain ECU, such as the vehicle's engine. This enabled them to send commands to turn the steering wheel, turn on the lights, and even shut off the engine [8]. The hacking of the Ford Escape was the first step for these researchers in developing increasingly more sophisticated hacks.

However, this attack required the hacker to be in the vehicle with a laptop, which makes it highly likely the user of the vehicle would notice his or her presence. This hack still proved to be useful to Miller and Valasek as they continued to research and were able to develop a more impressive and serious hack.

2.4.2 Jeep Cherokee

Miller and Valasek's next hack involved a Jeep Cherokee and its *UConnect* system [9]. This vehicle has a direct connection between the radio and the vehicle's CAN Bus. The Jeep's CAN Bus has two sections; one connected to the engine components and the other connected to the rest of the vehicle. Both of the CAN Buses integrated into the Jeep Cherokee are wirelessly connected to the radio. This connection can be seen in Figure 7 [9].

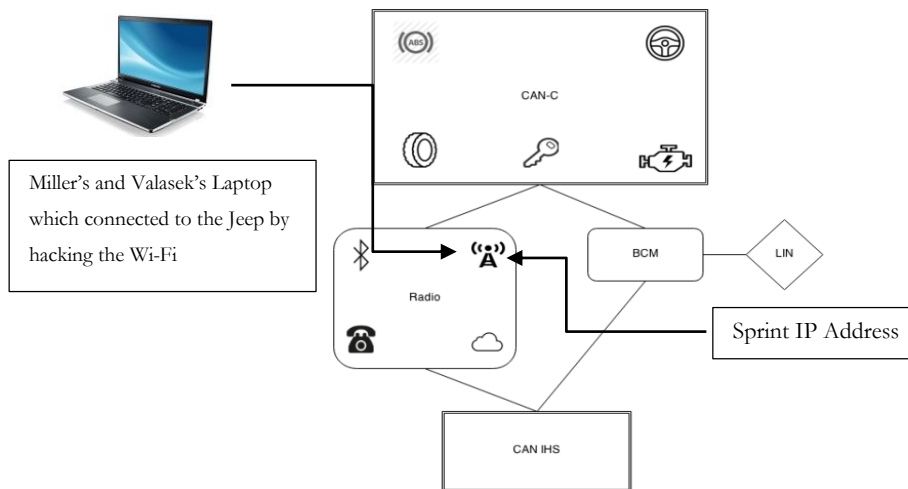


Figure 7: A Jeep Cherokee's connections between the two CAN Buses and head unit. This connection is what inspired Miller and Valasek to attempt the Jeep Cherokee hack. They connected using the Sprint IP address from the Jeep to connect to the vehicle from their laptop.

They first began by attempting to gain access to the vehicle using the Wi-Fi connection that customers can purchase. They discovered that the password for the Wi-Fi corresponds to the date and time of the hack, which can be seen here in Figure 8 [9].

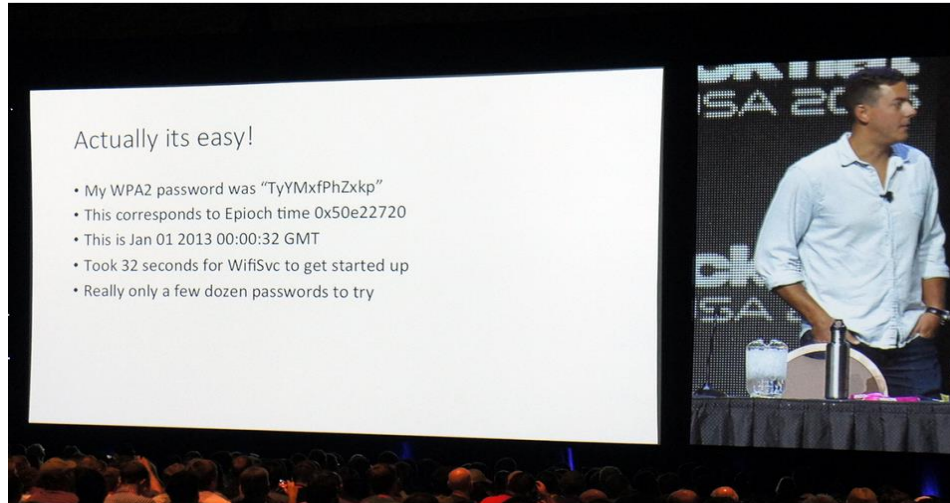


Figure 8: Black Hat security conference describing how the Jeep's Wi-Fi password was hacked.

Once connected to the Wi-Fi, they were able to manipulate the Linux software in the multimedia system, and then they had complete access to the Jeep. The only problem with this attack method is that not every Jeep Cherokee owner has purchased the Wi-Fi capabilities.

Miller and Valasek then began to look at another way to exploit the Jeep's CAN Bus. They discovered that even if the customer does not pay for the cellular network service, it is still standard in the Jeep Cherokee as a locked option [9]. This meant that every Jeep Cherokee is connected to the Sprint cellular network. They were able to scan for IPs on the Sprint network. They accomplished this by scanning the Sprint network for the IP addresses from 21.0.0.0/8 and 25.0.0.0/8, as these are the addresses used for the Jeep's Uconnect system [10]. In order to attack the specific Jeep they wanted, they used the GPS tracking to determine the location of the target vehicle, which can be seen in Figure 9 [9].



Figure 9: How GPS tracking was used to determine which Jeep Cherokee was their target vehicle.

The next step was going from the Jeep's multimedia system to the CAN Bus. Since there is no direct connection between the CAN Bus and the multimedia system, vehicles were generally believed to be secure. Miller and Valasek were able to refute this belief by changing the software in the V850 controller [23], which controls the interior high speed CAN and the primary CAN-C shown in Figure 7. The V850 is standard in automotive CAN set ups [10]. The V850 controller can only listen to CAN bus commands, but since it is a microcontroller, there is always a possibility for it to be hacked and have the firmware change [9]. Once they updated the software on the V850 controller, Miller and Valasek were now connected to the CAN bus and had control of the automobile.

All of Miller's and Valasek's testing has displayed that automobile manufacturers' stance on vehicle security is incorrect. Vehicles can be hacked, which puts users in potentially great danger. Even though this research took years to be completed, it encouraged others to study new ways to hack into vehicles in order to push vehicle manufacturers to be more diligent in their development process.

2.4.3 Telematics Unit Hack (UW and UCSD)

Students at both University of Washington and University of California San Diego were able to hack into a vehicle remotely and control all of its components [11]. The first step to hacking the vehicle was to dial the vehicle's emergency communication system. These

systems are known as the telematics units, or more commonly known as *OnStar* systems [24]. OnStar systems connect the vehicle's user to a worker on stand-by who can provide support and assistance if needed. General Motors created the OnStar system in order to provide the drivers of their vehicles 24/7 support for incidents such as crashes, lost keys, and stolen vehicle [24]. This team of research students discovered that by transmitting malicious signals to the telematics unit's phone number, they were able to confuse the vehicle. Figure 10 shows the research students dialing the vehicles telematics unit, then sending the malicious code in order to confuse the vehicle [11].

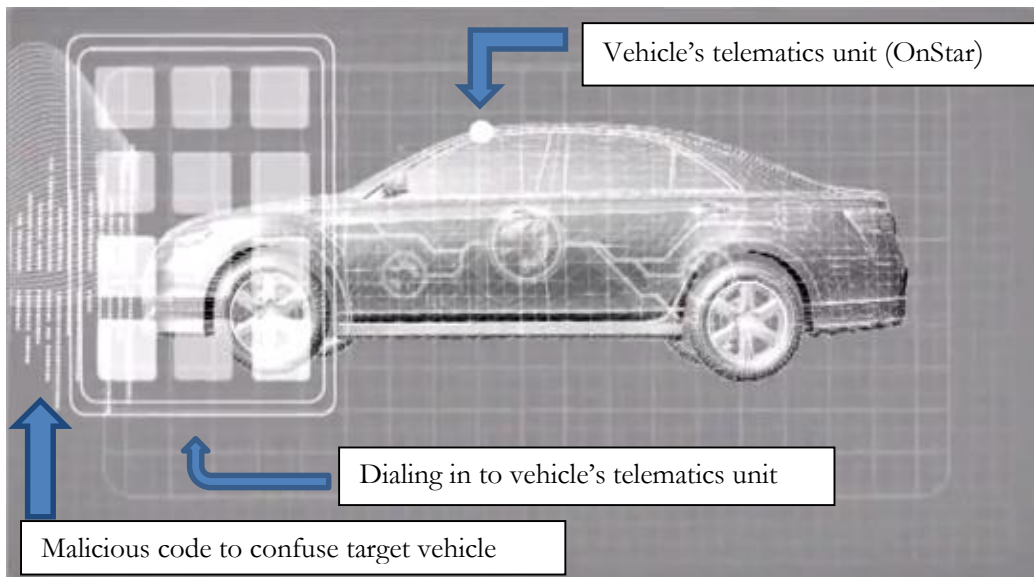


Figure 10: The hack performed by the University of Washington and San Diego students. They dialed into the vehicle's telematics unit and sent malicious code in order to confuse the system and grant them complete access.

While the vehicle's computer was attempting to figure out what was causing this flood of signals, the hackers were able to send their code into the vehicle. This code updated the firmware of the CAN Bus protocol, which gave them control of the vehicle. Using a laptop, they were able to control the dashboard, windshield wipers, horn, and even the vehicle's brakes while someone was operating the vehicle [11]. This shows yet another attack method where a hacker gained complete control of a certain target vehicle. All of these hacks are demonstrating to the public and vehicle manufacturers the unsafe security systems in vehicles.

UDS Based Attack Data Logger

Another dangerous aspect of this is that new technologies are being developed to allow vehicles to drive themselves. The potential danger is greatly increased in these vehicles because a user of a self-driving vehicle will have even less control if a hacker is able to compromise the security system.

2.4.4 LiDAR System and Autonomous Vehicle Hack

The evolution of vehicles is leading to increasingly autonomous vehicles in the near future. Semi-autonomous vehicles are on the road now, incorporating adaptive cruise control and automatic braking for obstacles. These types of vehicles present an even greater risk of being hacked because they include more computers and inject more automated control between the human and the wheel. Furthermore, automated vehicles are being designed to communicate with each other, so if one vehicle is compromised, all others in the area could be subjected to the same attack.

One researcher who wanted to address these safety concerns was Jonathan Petit, Principal Scientist at Security Innovation [12]. Most of the autonomous vehicles in development use the LiDAR systems in order to locate objects around the vehicle, such as Google's self-driving vehicle [13]. Petit used an inexpensive and easily obtainable equipment setup to confuse the LiDAR system into thinking there were objects such as other vehicles or walls all around the target vehicle. Petit said, 'I can take echoes of a fake car and put them at any location I want...And I can do the same with a pedestrian or a wall' [13]. This concept of placing objects around the autonomous vehicle can be seen in Figure 11 [12].

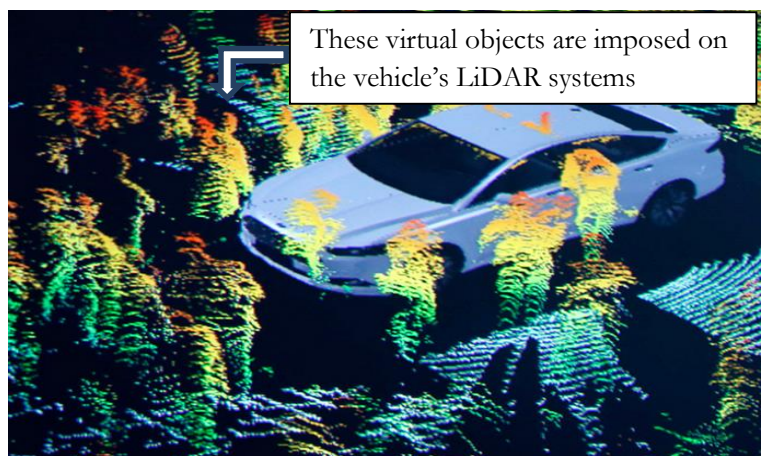


Figure 11: Jonathan Petit imposed virtual objects with his \$60 device. He was able to confuse the vehicle into believing there were objects such as walls, other cars, and humans located around it.

In order to accomplish this attack, Petit recorded typical laser pulses reflected by a commercial LiDAR system, then proceed to mimic these pulses and directed the laser at the vehicles navigation system [13]. During his testing, he was able to affect the LiDAR system in a car from about 300 feet away. Although considering he is using laser technology, accuracy is not overly important so he believes this attack could work anywhere from 50 to 1000 feet away from the target vehicle [13]. The whole setup consists of a low-power laser and a pulse generator, all together costing \$60. The most interesting part about this hack was that the vehicle itself cannot detect this happening; therefore, the vehicle cannot warn the driver about the threat [12].

This research is vital in the future development of autonomous vehicles. According to this report, in order for these types of vehicles to ever make it to market, there must be a 100 percent success rate to ensure the safety of the vehicles occupants [13]. Considering the technological advancements in automobiles, an increased number of researchers are developing ways to hack into vehicles in order to inform the manufacturers of their security flaws.

2.4.5 Tesla Model S Hack

One of the most technologically advanced vehicles on the market today is the Tesla. Since this vehicle is so advanced, it could potentially be one of the most vulnerable. Two researchers took on this challenge and were able to find a way to gain access to the Tesla Model S. Both Kevin Mahaffey, co-founder and CTO of mobile security firm *Lookout*, and Marc Rogers, principal security researcher for *CloudFlare*, were able to plug a network cable behind the driver's side dashboard, as shown in Figure 12 [14]. This allowed them to issue a software command to start the vehicle. They were also able to inject a virus that would allow the hacker to cut the Tesla's power remotely. After two years of research, they were able to discover that repeatedly building vulnerabilities upon one another increased their access to the target vehicle.



Figure 12: Connecting an Ethernet cord to the dashboard and gain access to the vehicle.

Tesla is actively working to solve these problems. Instead of the vehicle owners having to bring their vehicles in for service, Tesla can send software patches via cellular networks [14]. Tesla also has installed physical solutions, such as if the power of the vehicle is shut off, the vehicle will automatically stop or will allow the user to control the steering wheel while also having the airbags working [14].

This hack is closely related to our project as the potential hackers would have to gain physical access to the vehicle and plant a device that allows them remote access later. Tesla proved that they are actively working to solve the problem of vehicle security. This also gives a purpose to our project because if we are able to properly hack into a vehicle, this research could be used to help that specific vehicle company develop a security solution.

The capabilities of hacking a vehicle have advanced from the first attempted vehicle hacks. Now, researchers are able to gain control of a vehicle and run commands remotely. These hacks require years of experience and research into the CAN Bus and ECUs. Our plan is to use information from all of these hacks in order to build our own defense platform.

2.5 Test Vehicle: 2014 Nissan Altima

A test vehicle with an OBD-II port was required for this project to collect CAN data used for analysis and testing. A 2014 Nissan Altima was chosen because it was available to

the team on a weekly basis and has a modern CAN network with up to 100 ECUs [25]. The CAN network in the vehicle runs at a data rate of 500 kbps, which was determined through testing multiple data rates. Although the project would support other vehicles running at a data rate of 500 kbps, it is assured to support the 335,644 2014 Nissan Altimas sold in the United States [26].

2.6 Initial CAN Bus Testing and Analysis

Initial recording of the Nissan Altima CAN Bus was conducted in order to gather information about the vehicle's CAN Bus and obtain packet streams from different vehicle scenarios. CAN packet streams were recorded during five different scenarios in order to better understand how UDS messages affect the CAN network. The test cases and their respective pass conditions are described below.

No UDS attack

Test Files:

- 1) NO_UDS_IDLE.txt – Vehicle was started and idled
 - a. Test starts with the vehicle startup
 - b. Vehicle idles for a few minutes
 - c. Test ends with the vehicle still idling
 - d. No UDS messages sent on the CAN network in this test

- 2) NO_UDS_CITY_DRIVING.txt
 - a. Test starts with the vehicle startup
 - b. Vehicle is driven for a few minutes
 - c. Test end with the vehicle stopping
 - d. No UDS messages sent on the CAN network in this test

UDS Attack

Test Files:

- 1) UDS_IDLE.txt
 - a. Test starts with the vehicle already running
 - b. Vehicle idles for a few minutes
 - c. UDS device is plugged in and queries faults with the “Show Faults” UDS command
 - d. Test ends with the vehicle still idling

- 2) UDS_DURING_START_IDLE.txt
 - a. UDS device is plugged in prior to vehicle startup
 - b. Test starts with vehicle startup

UDS Based Attack Data Logger

- c. Vehicle idles for a few minutes with UDS messages being transferred over the CAN network
 - d. Test ends with the vehicle still idling
- 3) UDS_CITY_DRIVING.txt
- a. Test starts with the vehicle already running
 - b. Vehicle is driven for a few minutes and UDS device is plugged in while the vehicle is in motion
 - c. Vehicle is driven for a few more minutes with UDS messages being transferred over the CAN network
 - d. Test ends with vehicle still in motion

To obtain these recording scenarios, an OBD-II splitter was used to connect both a UDS Diagnostic tool and a recording device. The UDS Diagnostic tool simulated a hacker transmitting UDS messages to the vehicle. When UDS messages are sent to the vehicle, the ECUs targeted by the UDS message send a response. The diagnostic tool was controlled by a phone via Bluetooth. Using an app on the phone, UDS requests such as “Show all Faults” were sent to the vehicle. These UDS messages are considered to be “safe” as they do not write to the ECUs, instead only read status messages from the vehicle. The second OBD-II port on the splitter was connected to a high speed CAN data recording device. This recording device was built by Hristos Giannopoulos, a WPI graduate student. Details about the recording device setup can be found in [27]. The recording setup can be seen in Figure 13.

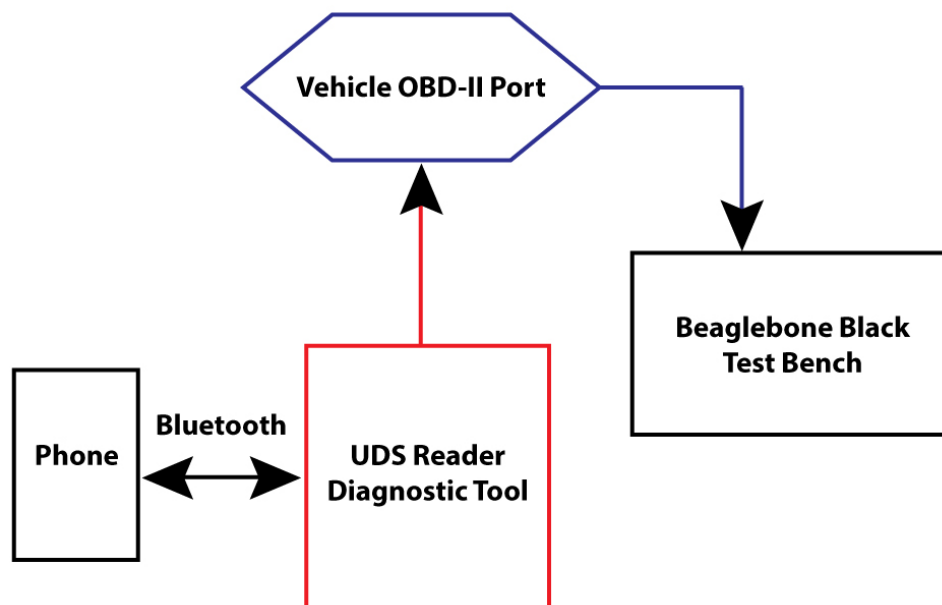


Figure 13: Sketch of BeagleBone Black CAN recording setup.

This set of recorded messages was used to determine the number of packets that should be recorded before and after the UDS attack. The library contains combinations of normal CAN traffic and UDS messages. In order to compare how the CAN Bus reacts when UDS packets are present, two scenarios were compared. The first involved CAN packets when the UDS Diagnostic tool was not connected to the OBD-II port, and the other was when the tool was connected. The resulting data log consists of the packet’s timestamp, the packet’s arbitration (process) ID, and the packet’s payload.

2.6.3 CAN Bus Recording Analysis

For this test, we were not focused on the actual payload of the packets, but instead the time interval at which they were being transmitted. When designing the Data Logger device, processing the payload of the packets is not necessary. This device will record all of the necessary information so that a forensics expert can decipher the CAN Bus packets and determine the vehicle’s condition during, before and after a potential attack. The maximum time interval is 105 ms when there is no UDS packets injected into the vehicle.

Table 3 shown provides a snap shot of results from the recording scenario with no UDS packets on the CAN Bus (Full table in Appendix C). The maximum time interval is 105 ms when there is no UDS packets injected into the vehicle.

Table 3: Normal Packet Transfer (no UDS) Timing Data by Arbitration ID (AID).

Bold values represent the longest packet transfer interval. AID means the Arbitration ID of the packet. The Max Interval is the maximum time between two packets of the same AID, while the Min Interval is the minimum time between two packets of the same AID. The Average Interval is the average time between two packets of the same AID and the Data Count is the number of the packets with the same AID.

| AID | Max Interval (ms) | Min Interval (ms) | Average Interval (ms) | Data Count |
|------------|-------------------|-------------------|-----------------------|------------|
| 560 | 102 | 98 | 100 | 165 |
| 580 | 104 | 101 | 102 | 161 |
| 6E2 | 105 | 100 | 102 | 161 |
| 625 | 103 | 97 | 100 | 164 |
| 5E4 | 102 | 98 | 100 | 164 |

UDS Based Attack Data Logger

The next scenario was recorded when UDS packets were present on the vehicle’s CAN Bus. Table 4 shown provides a snap shot of results from the recording scenario with UDS packets on the CAN Bus (full table in Appendix D). The maximum interval is 614 ms when there are UDS packets injected into the vehicle.

Table 4: Packet Transfer (with UDS) Timing Data by Arbitration ID (AID).

Bold values represent the longest packet transfer interval, while red represents the UDS message. AID means the Arbitration ID of the packet.

| AID | Max In- terval (ms) | Min In- terval (ms) | Avg In- terval (ms) | Data Count |
|------------|------------------------------------|------------------------------------|------------------------------------|-----------------------|
| 551 | 614 | 100 | 103 | 3027 |
| 580 | 614 | 100 | 103 | 3026 |
| 6E2 | 614 | 100 | 103 | 3026 |
| 7E8 | 1911 | 49 | 75 | 4100 |
| 7E9 | 2330 | 58 | 127 | 2428 |

The time interval information was used in order to determine the size of a log “tick”. A tick is the amount of logging time that will ensure that at least one packet of each arbitration ID will be logged. In order to determine the length of time for the tick, we developed an Excel macro to extract timing information from the log file captured from the test vehicle. The timing data is shown in Table 3 and Table 4, with an AID (Arbitration ID), maximum time, minimum time and average time interval between each packet transfer, and the data count for each packet ID. Table 5 shows the tick periods and their arbitration IDs which resulted from analyzing the data.

Table 5: Analysis of the Tick Representation.

| | Tick Period | Arbitration ID(s) |
|----------------------------|--------------------|--------------------------|
| Without UDS traffic | 105 milliseconds | 6E2 |
| With UDS traffic | 614 milliseconds | 551, 580 and 6E2 |

We expect all packets with other Arbitration IDs to be transmitted in this interval because they all have a shorter packet transfer period. From Table 5, it is seen that there are two types of ticks, one for normal traffic and one for corrupt traffic. For normal traffic, the tick time will be at least 105 milliseconds, and for corrupt traffic, the tick time will be at least

614 milliseconds. This will ensure that each tick will contain at least one packet with each Arbitration ID. For our implementation, we decided to increase these numbers to 120 and 750 milliseconds for non UDS and UDS traffic respectively in order to ensure at least every packet AID is in each tick.

The data collected was compared with a set of normal CAN traffic data previously recorded without the use of a UDS diagnostic tool. With this comparison, we were able to determine that the Arbitration IDs 7E8 and 7E9 (marked in red) referred to the diagnostic tool transmitting and receiving information respectively. According to the limited information available to us about the meaning of different UDS messages, the 7E Arbitration IDs refer to when there is a tester present in the vehicle. All of this information confirms that the vehicle was able to detect the diagnostic tool we connected to the OBD-II port.

2.6.4 Linux Controller Area Networks Tools

For purposes of researching, developing, and analyzing CAN Bus devices, Volkswagen Research created an open-source utility for simulating a CAN Bus on conventional Linux platforms such as Ubuntu. This utility [28] is known as the *SocketCAN* tools. These tools are used to simulate both physical and virtual CAN Bus networks. They are particularly useful for conducting potentially harmful testing as they can simulate a real CAN Bus, without the risks of damaging a vehicle. For example, Linux devices with these tools installed can record actual CAN Bus packets from a vehicle, and also playback these packets in order to simulate the vehicle's CAN Bus. The packets sent on the Linux CAN networks can be processed by Python programs using the SocketCAN Python libraries. Processing messages can be useful because it can allow users to store, modify or filter the data traveling on the bus or run algorithms to analyze the data. For details about the commands used to implement the virtual and physical CAN networks, refer to Appendix E.

2.7 Chapter Summary

Understanding the implementation and architecture of Controller Area Networks (CAN) in vehicles is essential to this project. The CAN network is the basis for the UDS protocol that is so widely used today in automobiles. Although UDS provides a wide array of

UDS Based Attack Data Logger

functionality for manufacturers, service technicians, and vehicle owners, it has been the foundation for dangerous vehicle hacks.

The national news about vehicle hacking has led professionals and researchers to publish articles describing the vulnerabilities in vehicles. These articles describe the safety steps that vehicle manufacturers should take in order to properly secure their vehicles. The EDR device currently records vehicle data during an accident, but does not provide sufficient information if a hack attempt were to occur. One step these professionals agree upon is a safety procedure that records a vehicle's CAN traffic if an attack has occurred.

Lately, there have been multiple hacks performed on modern vehicles as shown in recent publications. These hacks have shown there are multiple vulnerabilities in vehicle systems such as the OBD-II port, wireless networks, telematics unit, and LiDAR system. While the details of these hacks are not directly related to this project, they demonstrate the urgency of building a safety device that can detect a vehicle hacks taking place.

After an attack and/or accident has occurred, it is vital that a forensics analyst would be able to analyze a log of vehicle CAN data, conveying the state of the vehicle before, during, and after the attack has occurred. This would allow for the forensics analyst to diagnose the issue with the vehicle and determine who is liable for the accident, if any occurred. If an accident occurred and the vehicle acted erratically, a court would be responsible for determining whether the vehicle manufacturer, driver, or a hacker were liable for the accident.

The objective of this project is to devise a CAN Bus Data Logger that would allow a forensics analyst to analyze a recorded CAN Bus packet stream and determine if and how an attack occurred. The forensics analyst would use a UDS Guide to determine the extent of the attack, what ECU(s) was targeted, and if there was any damage to the vehicle. In an instance where an accident had occurred, the forensics analyst would be able to determine who is liable for the accident. Further research could be done by the vehicle's manufacturer to determine how the bug that allowed the attack through could be patched and what other actions need to be taken to assure the attack will not happen again.

3.0 Design Requirements

This section presents the design specifications involved in the development of a vehicle Data Logger, including an explanation of the data that should be recorded, potential hardware options, and a plan for integrating the hardware.

The objective of this project is to design and build a prototype data logging device that can log enough CAN Bus data for forensics analyst experts to be able to understand what the state of the vehicle was before, during, and after a UDS attack. The data logged must be useful to a forensics analyst, but not so extensive that it is impossible to locate particular occurrences that may give insight into the nature of the incident that occurred.

The data recorded must be read from a vehicle's OBD-II port and stored in a readable form, preferably in the format of a text file. This requires a programmable microcontroller for flexibility, portability, power efficiency, and the ability to adapt to differing attack strategies. The forensics analyst needs the timestamp that indicates when the packet was read from the bus, the packet's arbitration ID in HEX, and the packet's payload in hex in order to properly determine the state of the vehicle and its peripherals. Each packet is of size 20 bytes, including padding. The breakdown of the packet size and stored packet format is shown in Table 6.

UDS Based Attack Data Logger

Table 6: Packet format table which describes the fields of a CAN Bus packet.

| Data | Data Type | Size (Bytes) | Description |
|-----------------------|------------------------------|--------------|--|
| Timestamp | 32 Bit Unsigned Long | 4 | Milliseconds since the program started. This is implemented as a 32 bit unsigned long to handle long stretches of usage. For instance, a 16 bit unsigned integer would only be able to count to 65,536 milliseconds (slightly over a minute) whereas a 32 bit unsigned long would be able to count to 2,147,483,647 milliseconds (24.8 days - over 3 weeks). |
| Arbitration ID | 32 Bit Unsigned Long | 4 | Arbitration ID of the logged packet. |
| Packet Length | 8 Bit Unsigned Integer | 1 | Length of data in packet payload. |
| Packet Payload | 8 Bit Unsigned Integer Array | 0 to 8 | Data contained in the logged packet. This can range from 0 to 8 Bytes of data. |
| Padding | N/A | 3 | Padding aligns struct members in memory. |
| Stored Packet | Struct | 20 | Packet Format: <[Timestamp] [Arbitration ID] [Packet Payload]> Example: Timestamp – 8876 Arbitration ID – 2ED Packet Payload – AFE843D1683 8876 2ED AFE843D1683 |

The forensics analyst will need to make use of a “safe” or “steady” state of the vehicle using the normal CAN traffic previous to the vehicle attack. This will give the forensics analyst background information such as the speed of the vehicle, the peripherals that were in use, and the normal behavior of the vehicle as a whole. This data must be a complete record of the CAN traffic within a short period of time immediately before the incident. In order to record the state of the vehicle before an attack occurs, the Data Logger must store a fixed number of packets previous to the attack. Using a *circular buffer* to store these messages provides the functionality of a normal buffer, but after it writes to its last entry, it simply wraps around to the start of the buffer. This makes it a convenient data structure for capturing the recent history of CAN traffic immediately prior to the UDS attack.

Ten normal ticks of the bus is a sufficient logging time for a forensics analyst to interpret the state of the vehicle before an attack occurs. This provides at least 10 messages from each device on the network. Normal CAN traffic on the 2014 Nissan Altima has a tick time of 0.12 seconds. Therefore, the device is required to store the past 1.2 seconds of the packet stream. The circular buffer needs to be approximately 36 KB in size, having capacity to log 1800 packets previous to an attack. Table 7 shows a summary of this data, calculations, and the resulting circular buffer size.

Table 7: Circular Buffer (Normal Traffic).

| Parameter | Value | Calculation |
|---|-------|--|
| Normal ticks Stored | 10 | Given |
| Time per Tick (Seconds) | 0.12 | Given |
| Log Time (Seconds) | 1.2 | (Ticks Stored x Time per Tick) |
| Packet Size - including padding (Bytes) | 20 | Given |
| Packets Stored per Second | 1500 | Given |
| Packets Stored | 1800 | (Packets Stored per Second x Log Time) |
| Size of Buffer (KB) | 36 | (Packets Stored x Packet Size) x (1000 Bytes/1 KB) |

The circular buffer will be stored in the Data Logger’s RAM, and therefore it is required that the device have at least 36KB of RAM available for the circular buffer, with more space for program data and additional data structures used in the implementation. The Data Logger is also required to have a form of external storage in order to store the circular buffer after an attack occurs. Some of the available memory options are Flash and EEPROM. It is important to also consider the speed of the libraries that interface with these memory options. Additionally, this will ensure the logged data remains in storage across power cycles and can be read by the forensics analyst at a later time.

After an attack has occurred, CAN traffic data must be recorded to help the forensics analyst determine exactly what components were affected by the attack and how the system’s behavior changed as a result. A forensics analyst’s goal is to determine what ECUs were targeted based on the UDS messages recorded and how their behavior caused any incidents to occur. The state of the vehicle after a UDS attack may take a longer period of time to manifest into erratic behavior. Therefore, the method used to record the vehicle’s corrupted traffic (CAN traffic after a UDS attack) must provide enough information to make the aforementioned goal achievable. A UDS attack does not occur in a specific amount of

UDS Based Attack Data Logger

time, but varies in length depending on the number of UDS messages in the attack. Through our CAN Bus analysis, it was determined that 10 normal ticks before the first UDS message is detected and 80 corrupt ticks after the UDS message is detected would be sufficient for this task.

The method used to record corrupt traffic must accommodate more data than the circular buffer being used to store the normal CAN traffic. The size of the corrupted traffic to be recorded would exceed the size of any microcontroller's RAM. For this reason, the stream of packets after the UDS attack occurs should be stored to an external storage device. This data stream will prevent excessive RAM usage and will ensure that the data is periodically saved to the external storage device before power loss occurs. This is particularly important in the case that a UDS attack causes an accident or any other event that would cause loss of power on the OBD-II port.

The log file containing corrupt traffic should store 80 corrupt ticks of CAN data after the last UDS message monitored. If only one UDS message is detected in the attack, the file will only contain 80 corrupt ticks of packets. However, if multiple attacks occur within the 80 tick period of time, the file will be extended to contain 80 corrupt ticks after the last UDS message in the attack. The file size of this data will be roughly a minimum of 1.8 MB, but would potentially have a maximum of the full external storage capacity. Table 8 shows a summary of this data, calculations, and the resulting file size on the external storage device.

Table 8: Linear Buffer after UDS packet detected (Corrupt Traffic).

| Parameter | Value | Calculation |
|---|-------|--|
| Corrupt ticks Stored | 80 | Given |
| Time per Tick (Seconds) | 0.75 | Given |
| Log Time (Seconds) | 60 | (Ticks Stored x Time per Tick) |
| Packet Size - including padding (Bytes) | 20 | Given |
| Packets Stored per Second | 1500 | Given |
| Packets Stored | 90000 | (Packets Stored per Second x Log Time) |
| Size of File on External Storage (KB) | 1800 | (Packets Stored x Packet Size) x (1000 Bytes/KB) |

This Data Logger implementation requires that the external storage device can store both the circular buffer's data dump (roughly 36 KB) and the corrupt traffic data stream (at least 1.8 MB). The amount of data required by the external storage could quickly surpass

these values as more attacks occur and more UDS messages are sent over the bus. In conclusion, the device chosen for this implementation must have at least 64 KB of RAM and support for an external storage device that can hold over one GB of data. These values are chosen as resource constraints to ensure that the device chosen can support situations with multiple UDS attacks and have enough RAM to support the circular buffer, linear buffer, and program data.

Other constraints that are relevant are the speed of the CPU (for both the Test Bench and the Data Logger), the speed of the SD card library, the power consumption of the device, the cost of the parts, and the programmability of the devices. The CPU speed is relevant for the Data Logger because most microcontrollers only have a single CPU core. The Data Logger must use this single core to read and write each message monitored on the bus. This also applies to the Test Bench; it must be fast enough to send messages at the same speed that the vehicle recorded them, and to do any additional tasks such as processing acknowledgement messages for each sent packet. The SD card library must be fast enough to write messages and avoid missing any packets on the network. The power consumption is relevant for the Data Logger device because it will be connected to the OBD-II port, and it will use the vehicle's energy to get power. The cost of the prototype is important because any future project groups may need to reproduce this set up with limited budgets, and in case it would get to the market. Finally, size matters because it must be able to fit in the vehicle without being obstructive and for privacy concerns.

This design is meant to serve as a proof of concept, and therefore, should not be connected to a vehicle. Testing UDS attacks on an actual vehicle could cause unwanted damage to the vehicle and require service from the manufacturer before it could be driven again. A test bench would completely mitigate this issue and act as a safe alternative. The test bench is required to have packet stream play back functionality, meaning the Data Logger could connect to the test bench and read CAN data as if it was connected to the OBD-II port of a vehicle. The test bench is also required to play back this CAN data at the same rate as the test vehicle, the 2014 Nissan Altima.

3.1 Embedded Hardware Options

This section presents the different hardware devices that were studied as possible options for the proposed Data Logger device and for the Test Bench. This section also includes some accessory hardware such as CAN Shields for the microcontrollers, a high-speed CAN transceiver, and a timing module.

3.1.1 Peripheral Buses

The microcontrollers discussed in this section use two fairly common data transfer protocols to communicate with peripheral devices, I2C and SPI. These two protocols are described in detail below.

Inter Integrated Circuit Communications (I2C) - I2C is a data transfer protocol found in every TV, monitor, and computer motherboard today. It uses two wires to transfer data between two nodes, a master and a slave. The two lines are called SDA (data line) and SCL (synchronization clock). When data is sent on the SDA line, pulses are sent on the SCL line to keep the two nodes in sync. I2C has a maximum data rate of 50 KB/sec and is supported on most microcontrollers today [29].

Serial-Peripheral-interface (SPI) - SPI is a general purpose data transfer protocol with many applications in devices. It often uses three wires to communicate between a master and slave device. The three wires are MISO (Master in, Slave out), MOSI (Master out, Slave in), and M-CLK (synchronization clock). After eight clock pulses on the M-CLK line, a full byte of data has been transmitted to each node. SPI offers a higher transfer rate than I2C and is also supported on most microcontrollers [29].

3.1.2 Arduino

Arduino is a simple microcontroller board that can interface with an array of expansion devices [30]. It has 14 GPIO pins; these pins are used to communicate with expansion boards that can either be soldered to a breadboard or attached directly to the Arduino. The board also has a USB connection, power jack and a reset button. The Arduino board connects to a computer via USB and interfaces with the Arduino integrated development environment (IDE). Code can be transferred from the computer onto the Arduino board. The

Arduino environment has a programming language with syntax similar to C or C++. An image of the Arduino UNO R3 is shown in Figure 14 [31].

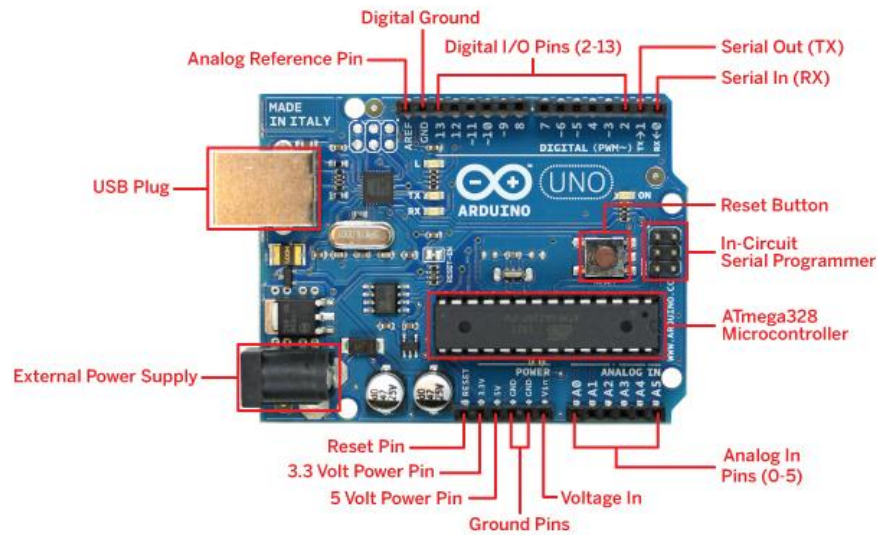


Figure 14: Arduino Uno microcontroller.

The Arduino can be programmed through the Arduino IDE, which makes it easier to interface with available libraries and provides some tutorials on how to access Arduino basic functions. The libraries allow the Arduino to use the SD card reader and to communicate through SPI with other devices.

3.1.3 Raspberry Pi B+

Raspberry Pi is a microcontroller that runs the Linux operating system and can be used for an array of applications [32]. It has the ability to interface with many peripherals, such as keyboards, monitors, and a mouse in order to allow the user to code the board using different languages. The main programming language for the Raspberry Pi is Python. The Raspberry Pi has the ability to connect to other devices through Bluetooth, Ethernet, Wi-Fi, and other network interfaces. An image of a Raspberry Pi is shown in Figure 15 [33].

UDS Based Attack Data Logger

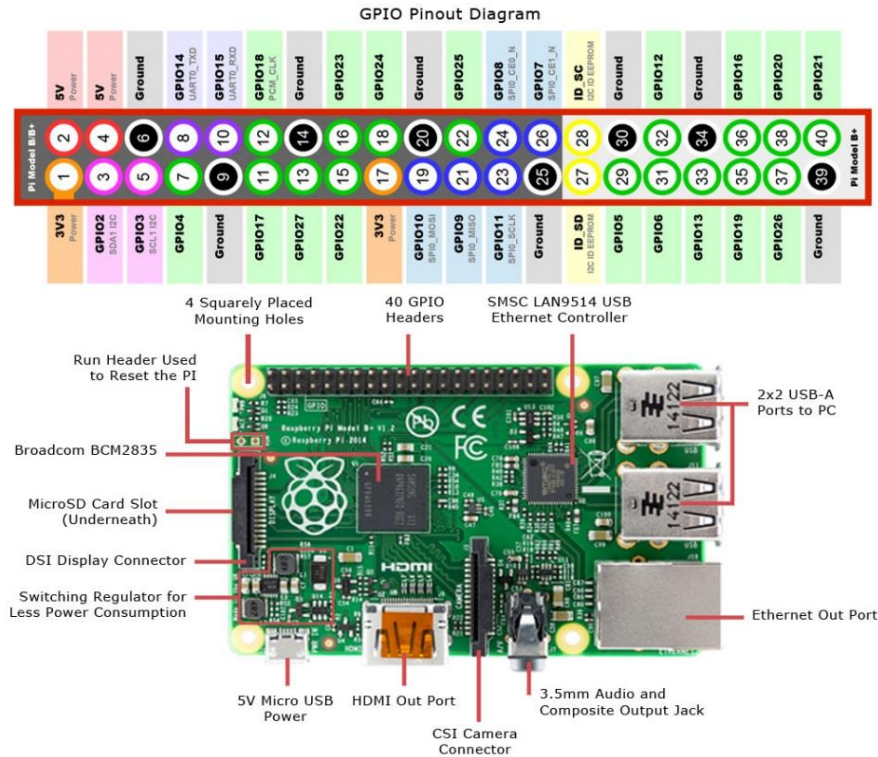


Figure 15: Raspberry Pi Schematics.

3.1.4 Teensy 3.2

The Teensy 3.2 is a Cortex-M4 based microcontroller, which offers multiple communication buses and expansion to an array of hardware expansions [34]. These hardware expansions comprise memory, Bluetooth, Wi-Fi, cellular adapters, timing modules, etc. The Teensy is smaller and simpler than most microcontrollers in its class, which makes it a more power efficient option for an embedded project. Since the device is so simple, it does not have certain built-in modules such as a timer and SD card slot. Although there are libraries to support such expansions, the hardware must be either soldered to the device or a breadboard setup in order to interface with the Teensy. A picture of the Teensy 3.2 can be found in Figure 16 [35].

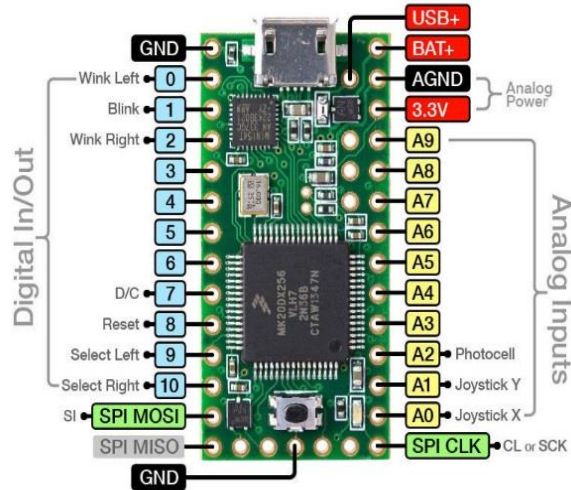


Figure 16: Teensy 3.2 Schematic.

The Teensy uses an IDE similar to that of the Arduino, with software extensions that allow for communications with the Teensy, hence the IDE name, *Teensyduino*. Teensyduino allows the user to develop code for the Teensy as well as perform other actions such as overclocking. It also supports a C library that allow interfacing with the CAN bus via the OBD-II port. This library was optimized by Hristos Giannopoulos to be able to use FIFO queues to record packets more efficiently [36]. C/C++ programs can utilize these FIFO queues, which allows for reading packets in the order they were recorded by the CAN hardware.

3.1.5 BeagleBone Black Microcontroller

The BeagleBone Black is a microcontroller based on the Am335x 1GHz ARM Cortex-A8 processor [37]. It has 4GB eMMC (similar to SD) on-board flash storage and a micro SD card adapter for external storage. It can be flashed with almost any version of Linux, but the stock operating system is Ångström Linux. The fact that this microcontroller runs a Linux operating system makes it versatile in the sense that it can run open source tools and has the capabilities of a full computer. With these capabilities comes an increase in power consumption, which makes this board difficult to power for extended periods of time with a battery. The BeagleBone Black’s schematic can be seen in Figure 17 [38].

UDS Based Attack Data Logger

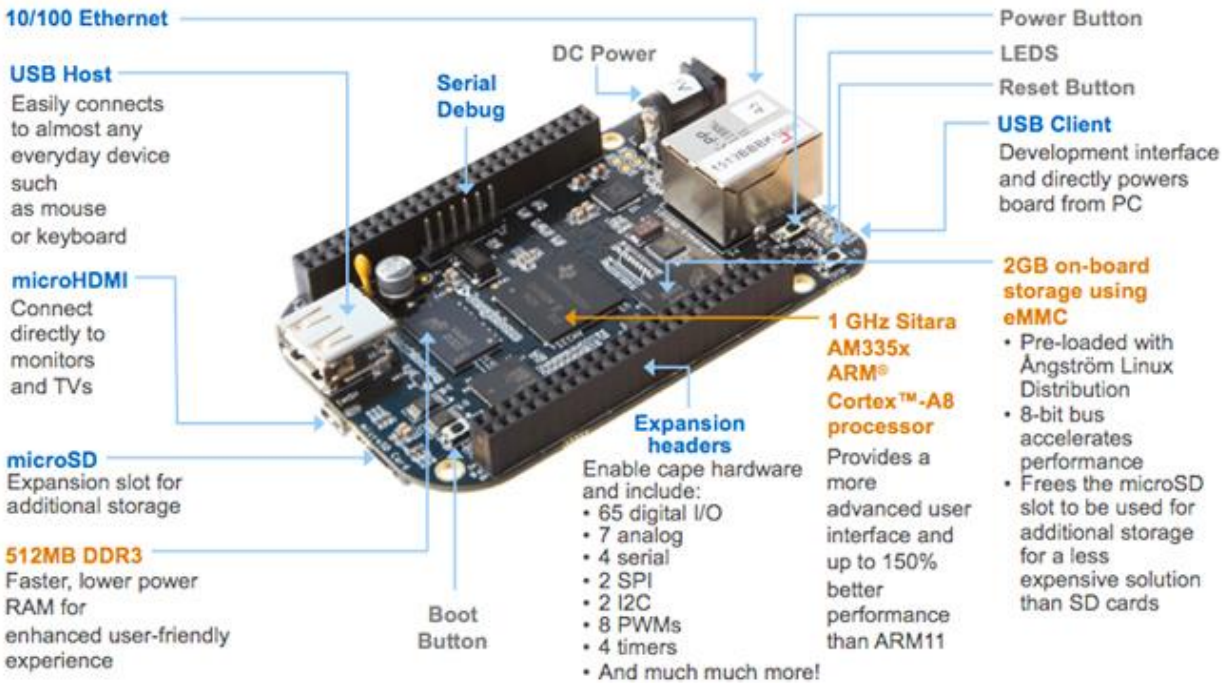


Figure 17: BeagleBone Black Schematic.

3.1.6 Microcontroller Summary

A summary of all the microcontroller options is shown in Table 9 [39] [40] [41] [34] [42].

Table 9: Microcontroller Specifications Summary.

| | Arduino Uno | Raspberry Pi B+ | Teensy 3.2 | BeagleBone Black |
|--------------------------|-------------------|---------------------------|---------------------------------|--|
| CPU | ATmega328P | Low Power ARM1176JZFS ARM | MK20DX256VLH7 Cortex-M4 ARM | AM335x 1GHz ARM Cortex-A8 |
| Frequency | 20 MHz | 700 MHz | 72MHz (up to 96MHz overclocked) | 1GHz |
| Number of Cores | 1 | 1 | 1 | 1 |
| Operating Voltage | 5V | 5V | 5V | 5V |
| Flash Memory | 32 KB | N/A | 256 KB | 4GB eMMC |
| RAM | N/A | 512MB | 64 KB | 512MB DDR3 |
| EEPROM | 1 KB | N/A | 2 KB | N/A |
| GPIO Pins | 14 | 40 | 21 | 2 x 46 |
| Software | Arduino IDE, Java | Python, Raspbian, Noobs | Arduino IDE, Teensyduino | Debian, Android, Ubuntu, Clode9 IDE on Node.js |
| Price | \$24.95 | \$29.95 | \$19.80 | \$54.95 |

There are many different types of microcontrollers available but we focused on these 4 because they are the most widely used. We needed a microcontroller for the Data Logger device and for the test bench setup. Just looking into the microcontrollers briefly, we were able to narrow it down to the ones we mainly wanted to focus on. Since the Arduino and the

Teensy 3.2 were both inexpensive and simple to use, we wanted either of those to be the microcontroller for the Data Logger. Since the Raspberry Pi and the BeagleBone Black were both more expensive and primarily used as Linux machines, these two were the primary options for the test bench microcontroller.

3.1.7 Arduino CAN Bus Shield

The Arduino CAN Bus Shield is a module that provides a direct link from an Arduino board to the CAN Bus. The module connects to the Arduino pin outs, allowing for high speed transfer to the CAN Bus via OBD-II port. It is compatible with both standard and extended frames for CAN v2.0B and transfers at speeds up to 1 mbps (highest speed supported by the CAN Bus). An image of the CAN Bus Shield is shown in Figure 18 [43]. The Arduino uses CAN libraries that allow for the microcontroller to communicate with other devices connected to a CAN network by using the CAN Bus Shield. This shield must be purchased separately and costs approximately \$25 at the Sparkfun website [43].

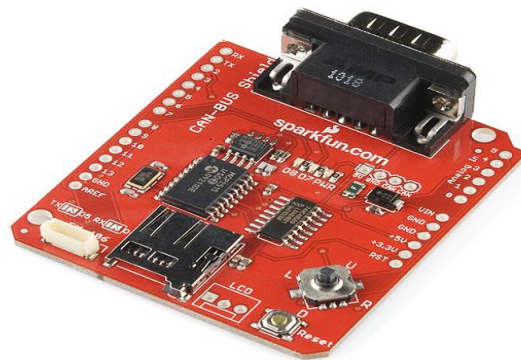


Figure 18: Arduino CAN Bus Shield.

3.1.8 Raspberry Pi PICAN Module

The PICAN module functions similar to the Arduino CAN Bus Shield, but is made for the Raspberry Pi interface. It provides a direct connection to the CAN Bus via OBD-II port and supports both standard and extended frames. An image of the PICAN module is shown in Figure 19 [44].

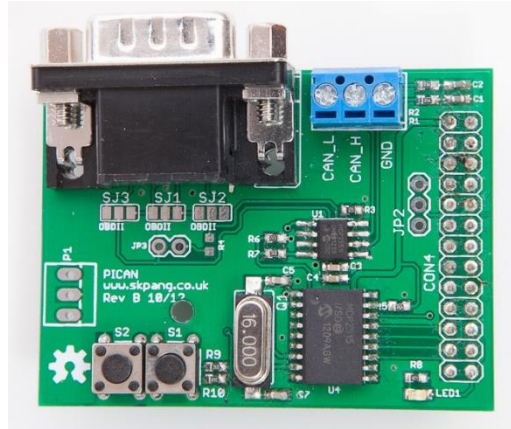


Figure 19: Raspberry Pi PICAN module.

3.1.9 MCP2561

The MCP2561 chip is a high-speed CAN transceiver developed by Microchip Technology Inc. [45]. Many of the microcontrollers previously mentioned require external hardware to interface with CAN networks. This chip can be used with these microcontrollers, allowing them to interface with the physical two-wire CAN bus available in modern vehicle. It has a data transfer rate of 1 Mbps. This chip supports CAN data monitoring of up to 112 nodes connected to the network. The pin outs for this chip can be seen in Figure 20 [45].

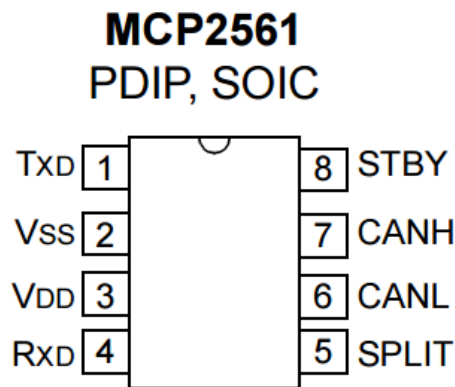


Figure 20: Pin outs for MCP2561 Chip.

The CANH and CANL pins are to be directly connected to the OBD-II port of the vehicle. The chip then connects with microcontrollers through the TXD and RXD pins. The VSS and VDD pins are used to power the chip. It is able to interface with CAN Controllers and Microcontroller devices with voltages that range from 1.8V to 5.5V I/O. Since the CAN bus has a characteristic line impedance of 120 Ohms, in order to avoid reflection, a resistor

of 120 Ohms is needed to be added to the end of each bus to match with the characteristic impedance of the transmission line [46].

The chip has two modes of operation: the normal mode and the standby mode. The specification of how to use these modes is shown in the Table 10 [45]:

Table 10: Mode specifications for the MCP2651.

| Mode | STBY Pin | RXD Pin | |
|---------|----------|-----------------------------|-----------------------------|
| | | Low | High |
| Normal | Low | Bus is dominant | Bus is recessive |
| Standby | High | Wake-up request is detected | No wake-up request detected |

3.1.10 SN65HVD230 CAN board

The SN65HVD230 CAN board is a high-speed CAN transceiver manufactured by Waveshare. The SN65HVD230 CAN board is made up of three parts. The main part is the SN65HVD230 CAN chip. The other two parts are a 120 Ohm resistor and a 10K Ohm resistor. The 120 Ohm resistor is soldered between the CANH and CANL pins to avoid reflection. The 10K Ohm resistor is soldered between the Rs pin and ground. The Rs pin on the SN65HVD230 CAN chip has 3 different modes: high speed mode, slope control mode and low power mode. The high speed mode is selected by connecting Rs to ground and it has no limitation on the rise and fall slopes. During low power mode the driver is switched off until a high logic level is applied to the Rs pin [47]. Slope Control mode is selected when a resistor is added in series between the Rs pin and ground. Since the 10K Ohm resistor is soldered between the Rs pin and ground, the SN65HVD230 CAN board is in slope control mode. Slope control helps avoid reflection as well; signal reflections can develop in a stub since stub lines are unterminated. To minimize reflection, the stub-line length should not exceed one third of the line’s critical length [48]. A schematic of the SN65HVD230 CAN board including pin outs and resistors can be found in Figure 21.

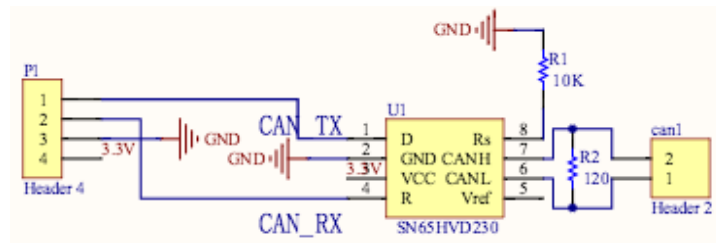


Figure 21: Schematic of the SN65HVD230 CAN Board.

3.1.11 Adafruit DS1307 RTC Timing Module

Low level microcontrollers may not include operating systems or hardware that provides them with a Real Time Clock (RTC). Therefore, specialized hardware that will provide the microcontroller with the time and will keep track of time even when the power is off is required. The Adafruit DS1307 RTC timing module requires 5V to power and also a connection to ground. In order to keep track of time when the power is off, it requires a 12mm 3V lithium coin cell [49].

The picture below shows the schematic of the Adafruit DS1307 RTC timing module. Note that it is not necessary to use a coin battery, any battery with voltage range from 2.0V to 2.5V is sufficient, but the hardware module has dedicated space that fits the coin module in a convenient fashion. It is also important to note than any supply voltage higher than 5.5V will burn the chip [50]. A schematic of the DS1307 can be found in Figure 22 [50].

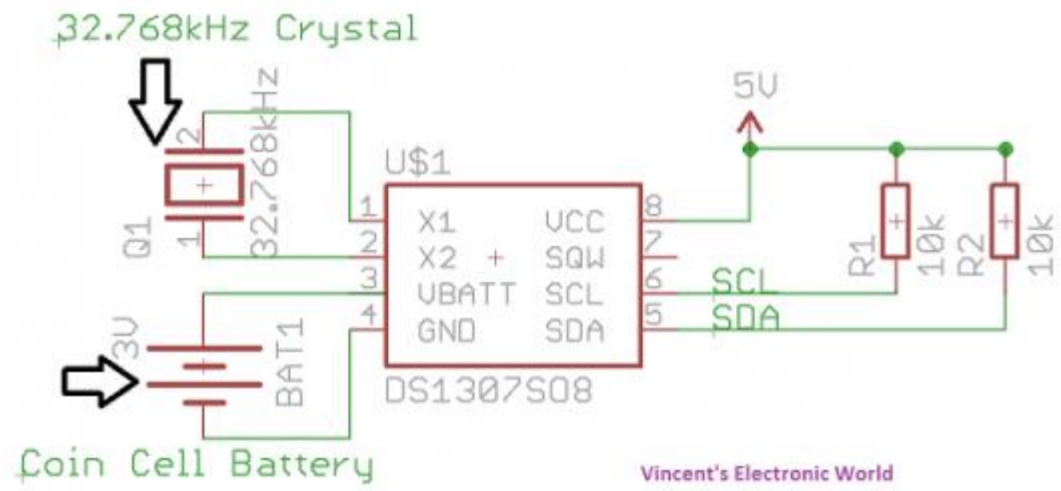


Figure 22: Schematic of the DS1307.

A picture of the Adafruit DS1307 with all of its components is shown in Figure 23 [49]. After the board is assembled the pins must be soldered as referenced in the Adafruit tutorial [49].

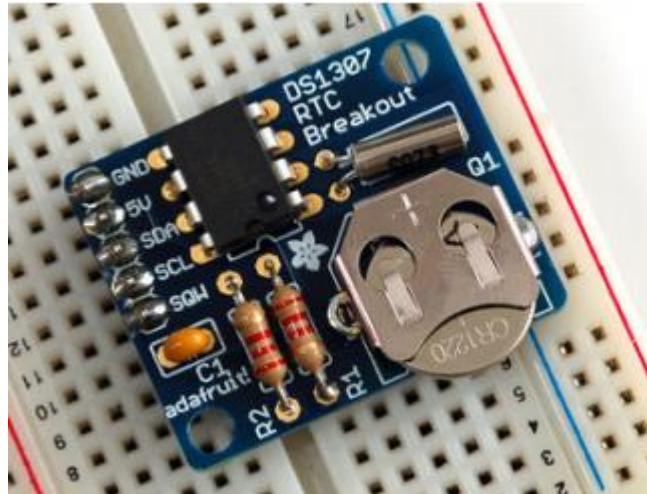


Figure 23: Picture of the timing module on a breadboard.

3.2 Hardware Analysis

In this section the microcontrollers considered for both the Data Logger and test bench are discussed and compared to one another. The microcontrollers are compared in two separate value analyses to narrow down which would be best suited to each application.

3.2.1 Data Logger Analysis

A value analysis table was used to decide which microcontroller is best suited for this Data Logger implementation. A numerical value is assigned to the value analysis specifications for each microcontroller considered in order to indicate how well they fit the design constraints. The relative value, or weight, given to each value analysis are the following:

- Processor Speed (3)
- Power (2.5)
- Cost (2)
- Programmability (1.5)
- Size (1)

These values show that processor speed is the most important variable (this is shown by having the largest weight) and size is the least important (this is shown by having the smallest weight). Since these variables are very important for the overall design, it was necessary to assess each variable quantitatively. The values for the cutoffs between rankings (ex. between 50MHz and 800MHz) were assigned based on the average of all the microcontrollers' specs.

UDS Based Attack Data Logger

The processor speed is the most important variable as it directly relates to how fast the Data Logger can read and write the packets to the SD card adapter. The slower the CPU, the more packets the Data Logger will fail to capture when monitoring the bus. The minimum requirement for a CPU is that it can read 1500 packets per second, as this is the transfer rate of the 2014 Nissan Altima CAN traffic. Although there is no way to determine whether a CPU can handle writing at this speed, the frequency of the CPU can be used to make a conjecture. We assigned a value of 1 (lowest respective value) to a CPU that has a frequency of less than 50MHz, a value of 2 to a CPU with frequency between 50MHz and 800MHz, and a value of 3 (highest respective value) to a CPU with frequency above 800MHz.

The next variable is power; this is how much power the microcontroller consumes on average under normal conditions. We assigned a value of 1 (lowest respective value) to a microcontroller that uses more than 1W, a value of 2 to microcontroller that uses between 0.2W and 1W, and a value of 3 (highest respective value) to a microcontroller that uses less than 0.2W.

The next variable is cost; this is how much the microcontroller costs. We assigned a value of 1 (lowest respective value) to a microcontroller that costs more than \$50, a value of 2 to microcontroller that costs between \$20 and \$50, and a value of 3 (highest respective value) to a microcontroller that costs less than \$20.

The next variable is programmability; this is how easy the board is to program and how many libraries are available for CAN Bus communication. We assigned a value of 1 (lowest respective value) to a microcontroller that is difficult to program and has almost no libraries available, a value of 2 to microcontroller that is fairly easy to program and has between 1 and 3 libraries, and a value of 3 (highest respective value) to a microcontroller that is easy to program and has more than 3 libraries.

The last variable is size; this is the physical size of the microcontroller. We assigned a value of 1 (lowest respective value) to a microcontroller that is larger than 70mm x 50mm, a value of 2 to microcontroller that is between 40mm x 20mm and 70mm x 50mm, and a value of 3 (highest respective value) to a microcontroller that is smaller than 40mm x 20mm. A value analysis of the four microcontrollers is shown in Table 11.

Table 11: Microcontroller Value Analysis.

| Micro-controllers | Relative Value | Arduino Uno | | Raspberry Pi B+ | | Teensy 3.2 | | BeagleBone Black | |
|-------------------|----------------|-------------|-------------|-----------------|-----------|-------------|-------------|------------------|-----------|
| | | Value Point | Total | Value Point | Total | Value Point | Total | Value Point | Total |
| Speed | 3 | 1 | 3 | 2 | 6 | 2 | 6 | 3 | 9 |
| Power | 2.5 | 2 | 5 | 1 | 2.5 | 3 | 7.5 | 2 | 5 |
| Cost | 2 | 2 | 4 | 2 | 4 | 3 | 6 | 1 | 2 |
| Program-mability | 1.5 | 3 | 4.5 | 1 | 1.5 | 2 | 3 | 2 | 3 |
| Size | 1 | 2 | 2 | 1 | 1 | 3 | 3 | 1 | 1 |
| Total | | | 18.5 | | 15 | | 25.5 | | 20 |

Based on this value analysis, the best microcontroller for the Data Logger design was the Teensy 3.2. The Teensy 3.2 is second in terms of speed, but its speed is sufficient for the data logging application as it runs at 72MHz and is documented to be capable of safely achieving 96MHz with overclocking. The Teensy 3.2 is one of the most power efficient options, and it is the cheapest option. The Teensy 3.2 will be the microcontroller selected for the Data Logger prototype for the aforementioned reasons.

3.2.2 Test Bench Analysis

A value analysis table was created to determine which microcontroller was best suited to be used as the test bench device. The test bench must be able to simulate a CAN network and send CAN messages to the Data Logger device at the same rate as the Nissan Altima. A numerical value is assigned to the value analysis specifications for each microcontroller considered in order to indicate how well they fit the test bench constraints. The relative value given to each value analysis are as follows:

- OS Tools (3)
- CAN Interface (2.5)
- Speed (2)
- Cost (1.5)

These values show that OS Tools is the most important variable (this is shown by having the largest weight) and cost is the least important (this is shown by having the smallest weight). Since these variables are very important for the overall design of the test bench, it was necessary to assess each variable quantitatively. The values for the cutoffs between rankings (ex. between 50MHz and 800MHz) were assigned based on the average of all the microcontrollers' specs.

UDS Based Attack Data Logger

The OS Tools variable is assessed first since it is most important. We assigned a value of 1 (lowest respective value) to a microcontroller that has no CAN tools, a value of 2 to a microcontroller that only has reading tools, and a value of 3 (highest respective value) to a microcontroller that has both reading and writing tools.

The next variable is CAN interface; this is the available hardware that the microcontroller has to interact with a CAN network. We assigned a value of 1 (lowest respective value) to a microcontroller that has no hardware available to interact with a CAN network, a value of 2 to a microcontroller that has external hardware that interfaces with the CAN network, and a value of 3 (highest respective value) to a microcontroller that has a built-in CAN network interface.

The next variable is processor speed; this is the frequency at which the CPU operates. We assigned a value of 1 (lowest respective value) to a CPU that has a frequency of less than 50MHz, a value of 2 to a CPU with frequency between 50MHz and 800MHz, and a value of 3 (highest respective value) to a CPU with frequency above 800MHz.

The last variable is cost; this is how much the microcontroller costs. We assigned a value of 1 (lowest respective value) to a microcontroller that costs more than \$50, a value of 2 to microcontroller that costs between \$20 and \$50, and a value of 3 (highest respective value) to a microcontroller that costs less than \$20. A value analysis of the three microcontrollers is shown in Table 12.

Table 12: Test Bench Value Analysis.

| Micro-controllers | Relative Value | Arduino Uno | | Raspberry Pi B+ | | Teensy 3.2 | | BeagleBone Black | |
|-------------------|----------------|-------------|-----------|-----------------|-------------|-------------|-----------|------------------|-----------|
| | | Value Point | Total | Value Point | Total | Value Point | Total | Value Point | Total |
| OS Tools | 3 | 1 | 3 | 3 | 9 | 1 | 3 | 3 | 9 |
| CAN Interface | 2.5 | 2 | 5 | 2 | 5 | 3 | 7.5 | 3 | 7.5 |
| Speed | 2 | 1 | 2 | 2 | 4 | 2 | 4 | 3 | 6 |
| Cost | 1.5 | 2 | 3 | 1 | 1.5 | 3 | 4.5 | 1 | 1.5 |
| Total | | | 13 | | 19.5 | | 19 | | 24 |

Based on this value analysis, the best microcontroller to be used as a test bench is the BeagleBone Black. Neither the Arduino nor Teensy 3.2 possess the software or hardware required to run CAN tools that can produce the volume of messages at the rate required to

test the Data Logger, so they were not chosen to be used in the test bench. The Raspberry Pi and BeagleBone Black are both Linux machines that are capable in terms of hardware and software, fitting the constraints of the test bench. The BeagleBone Black does however, have the capability to mimic CAN hardware, mitigating the need to buy CAN hardware, which is the case with the Raspberry Pi. For these reasons, the BeagleBone Black was chosen to be used as the test bench for the Data Logger.

3.3 Storage Analysis

This section covers the storage option analysis for the selected microcontroller, the Teensy 3.2. The goal of this section is to provide insight into what storage options were available for storing CAN data, explain why certain storage devices were selected, and to indicate tradeoffs between these options.

3.3.1 Peripheral Buses

The Teensy has two serial communications buses, SPI and I2C. For this Data Logger implementation, SPI is favorable over I2C for the following reasons [51]:

- SPI has a higher transfer rate than I2C.
- SPI draws less power than I2C.
- I2C is favorable for applications with a higher data transfer distance, but this comes with slower transfer rates. Since the travel distance of the CAN data is within a breadboard, the main advantage of the I2C bus would not be utilized.

Memory peripherals that use SPI instead of I2C were selected for the aforementioned reasons. Table 13 outlines the pins on the Teensy used to communicate over the SPI Bus.

Table 13: SPI Bus Signals and Respective Purpose.

| Name | Pin Number | Purpose |
|----------------------------------|------------|--|
| Slave Select/Chip Select (SS/CS) | 10 | Selects one or more chips to communicate with |
| Clock Signal (SCK) | 13 | Clock Signal |
| Master In Slave Out (MOSI/DOUT) | 11 | Sends data from the SPI master to slave(s) |
| Master In Slave Out (MISO/DIN) | 12 | Sends data from the slave(s) to the SPI master |

3.3.2 Internal Memory (EEPROM)

The Teensy EEPROM specifications in this section are from the K20 Sub-Family Reference Manual [52]. The Teensy has Electrically Erasable Programmable Read-Only Memory (EEPROM). This memory is non-volatile and is used to store relatively small amounts of data that must remain saved across power cycles. On the Teensy, EEPROM is overwritten every time a new program is loaded onto the board. EEPROM is most effective when saving settings and other program specific data as the storage space is fairly small compared to other forms of storage. The Teensy 3.2 includes a model MK20DX256VLH7 memory chip from Freescale Semiconductor, containing 2 KB of FlexRAM memory. This type of memory can be configured to store data like typical RAM or EEPROM. Typical RAM does not retain stored data when power to the microcontroller is cycled, but EEPROM does. When FlexRAM is configured as EEPROM, it supports 1, 2, and 4 byte read and writes. This makes EEPROM a simple way to restore program settings and parameters across power cycles.

As stated above, EEPROM has the ability to read and write from a single memory address rather than pages, which is typical for other storage devices such as flash memory. When loading program data, such as a single character or integer, it is advantageous and highly efficient to just read that one integer instead of a full page of data. EEPROM is also more reliable than flash as it can retain data for more read/write cycles (rated for over 100,000 read/write cycles) than typical flash memory.

A drawback of EEPROM on the Teensy is that it has a significantly smaller storage space compared to other memory types. This greatly limits its usability for certain applications. For instance, it would not be feasible to store all data logged during program execution to EEPROM. Although it may seem convenient to do so in terms of read and write speeds, the small amount of space to work with limits the program and any expansions that may be added in the future. The specifications for the Teensy EEPROM can be seen in Table 14 [52].

Table 14: Teensy FlexRAM as EEPROM Specifications.

| | |
|---|--|
| Storage Space | 2 KB (2048 Bytes) |
| Byte Write Latency (8 Bit) | Typical: 175 μ s, Max: 260 μ s |
| Word Write Latency (16 Bit) | Typical: 175 μ s, Max: 260 μ s |
| Long Work Write Latency (32 Bit) | Typical: 360 μ s, Max: 540 μ s |
| Data Retention | 100,000 Read/Write cycles (minimum) |

For this implementation, using FlexRAM allocated as EEPROM is out of scope as this project is focused on logging CAN data in real-time, not device recovery after power cycle. With that said, FlexRAM is a notable storage option for future expansions. As stated above, EEPROM is a useful storage type for program data and other data that needs to be restored across power cycles.

3.3.3 Flash Memory

The information in this section about flash memory can be found in the 1 Mbit SPI Serial SRAM with SDI and SQI Interface [53]. Flash memory is a type of nonvolatile memory, which is typically used for long-term storage. Flash erases data in blocks or pages, which are sections of memory. Before data can be stored on a flash memory block, the data on that block must be erased.

The 512 KB and 1024 KB flash memory modules made by Microchip are compatible with the Teensy SPI Bus. SPI Flash offers high bandwidth and a fairly low storage capacity, therefore the concern with this type of memory is that it does not offer a large enough storage space to contain all the packets recorded by the Data Logger. 1024 KB of memory is not enough space to store all CAN data required to diagnose one or more vehicle attacks. This is discussed further later in this section. Table 15 shows the specifications for Microchip's SPI Flash memory module [53].

Table 15: Microchip SPI Flash Module Specifications.

| | |
|---------------------------|--|
| Storage Space | 512 KB or 1024 KB |
| Byte Write Speed | Typical: 2000 Bytes/s , Min: 1428.57 Bytes/s |
| Byte Write Latency | Typical: 50 μ s, Max: 70 μ s |
| Data Retention | 100,000 Read/Write cycles (minimum), 100 years |

3.3.4 Secure Digital (SD) Card

The last storage option for this Data Logger implementation is SD Card technology. SD is the only storage option that can easily be removed from the Data Logger. It contains the largest storage space, but the slowest read and write speeds via SPI Bus. Teensy-compatible SD cards are available in sizes ranging from 2 – 8 GB. The speed at which data can be read and written to the SD Card is determined by the speed of the Teensy, the SD library implementation, and the SD Card itself. Figure 24 shows an image of the SD card reader for the Teensy microcontroller [54].

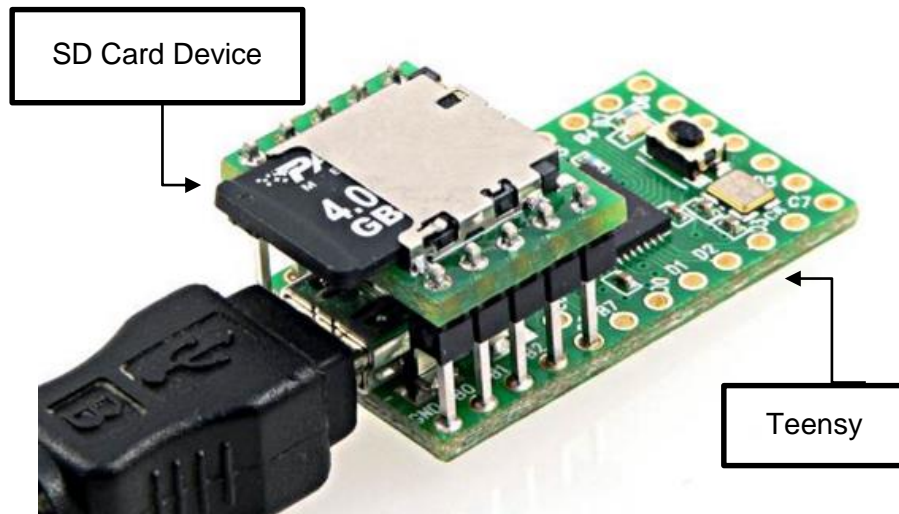


Figure 24: Image of the SD Card device for Teensy.

There are many SD Card libraries that support the Teensy, but the one that seems to be most commonly used is the *SdFat* libraries developed by Bill Greiman. According to his test benchmark, he was able to get the following speeds by using a Teensy 3.0 running at 96MHz, an SPI clock speed of 24 MHz, and the SdFat Beta library, which can be seen in Table 16 [55].

Table 16: SD Card Library Benchmarks.

| Command | File Size | Buffer Size | Speed (KB/sec) | Max Latency | Average Latency |
|---------|-----------|-------------|----------------|---------------|-----------------|
| write | 5MB | 4096 Bytes | 1776.44 | 65790 μ s | 2300 μ s |
| read | 5MB | 4096 Bytes | 2037.15 | 2356 μ s | 2008 μ s |
| write | 10MB | 8192 Bytes | 2002.05 | 6777 μ s | 4098 μ s |
| read | 10MB | 8192 Bytes | 2121.47 | 4231 μ s | 3860 μ s |

Bill Greiman, the creator of the SdFat library, indicated that the data size should be a multiple of 512 Bytes (SD Card page size) to avoid the need of copying data to the cache. Using a data size that is a power of two increases the performance slightly for this reason.

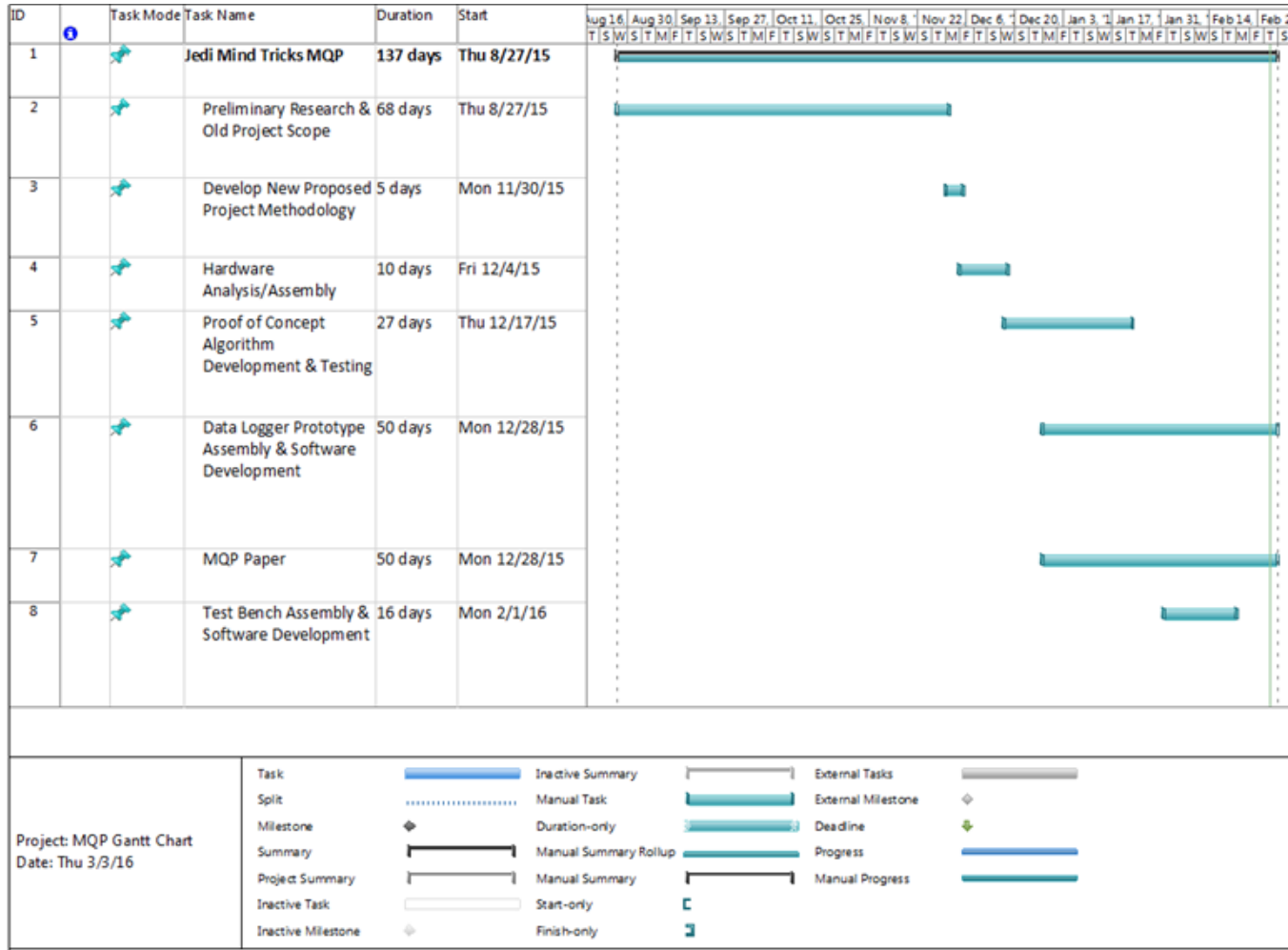
3.3.5 Selected Storage Device

The Data Logger requires a large storage space for the multiple logs that will be stored in the case of one or more UDS attacks. The SPI Flash modules do not fit the needs of the Data Logger as they cannot store enough data in the case that one or more attacks occur. The CAN data files necessary to diagnose the attack would not fit on a single flash module. Furthermore, removing an SPI Flash module from the device and copying its contents to a computer would be a complicated task for the user. Using an SD Card, the Data Logger can store all the CAN data a forensics analyst would deem fit to diagnose a vehicle attack and make transferring the data simple for the user.

The SD Card option was chosen as a storage device for this application because it has the highest storage capacity and can be easily transferred to a laptop, phone, or other device. The SD Card is capable of storing large log files at a rate fast enough to store most, if not all CAN packets monitored on the bus, which is of most importance for this implementation. Note that while the speed of the SD card is fast enough to process the incoming rate of the messages from the Nissan Altima, processing the messages could take more time. Therefore, running benchmarks is needed in order to know the exact throughput for our Data Logging algorithm.

3.4 Project Logistics

Table 17: Gantt Chart



4.0 Implementation

This section discusses the Data Logger implementation details including the system design, algorithm used to record UDS attacks including safeguards necessary to ensure data integrity, and attack scenarios supported.

4.1 System Design

The objective of this project was to design and build a prototype Data Logger device that can log enough CAN bus messages for a forensics analyst to be able to understand what the state of the vehicle was before, during and after a UDS attack. The proposed design has two main components, the logger device and a CAN simulator test bench. The Data Logger device is meant to be connected to the CAN Bus of a vehicle or included in the vehicle as a node by manufacturers so it can monitor the messages sent on the CAN Bus. The test device is needed for simulating a CAN Bus because it allows for testing the Data Logger device without risking damaging an actual vehicle.

The hardware and storage analysis determined the selection of the Teensy 3.2 as the microcontroller and the SD card as the optimal storage option for the device. The Teensy 3.2 requires an SD card adapter with supporting C libraries to interface with SD cards. The BeagleBone Black was selected as the most desirable microcontroller to use for testing our Data Logger device.

For testing and simulation purposes, the Data Logger device must communicate with the Test Bench device. Both the Data Logger device and the Test Bench require external CAN hardware to communicate with each other. There are many different CAN microchips that interface with microcontrollers, but we only focused on testing three in particular. The CAN hardware acquired consisted of models MCP2561 -E/P, MCP2551 -I/P, and the SN65HVD230. Each was tested with an oscilloscope when connected to the Teensy and Test Bench to check whether or not the input signal from one microcontroller was transmitted across the chip to the other microcontroller. Both the MCP microchips were unsuccessful when tested in this fashion, but the SN65HVD230 was capable of transmitting and receiving a signal.

UDS Based Attack Data Logger

The Data Logger device needs a timing module in order to get the current time. The current time is needed for recording the timestamps of the messages because the messages sent on the CAN network do not include the timestamp in them. The timestamp is critical for this project because the forensics analyst needs to know when possible attacks occurred in the network and when any effects from the attack occurred.

The BeagleBone Black microcontroller in the Test Bench used the data vectors recorded from the Nissan Altima. These data vectors were recorded while driving the Nissan Altima around Worcester and include the different scenarios explained in Chapter 2.6. The Test Bench sends these test vectors to the Data Logger device by using the SN65HVD230 chip. The Data Logger reads all messages and runs an algorithm to select and log relevant messages when a UDS message is detected on the network. The Test Bench can also send modified test vectors that have more or fewer UDS messages. Figure 25 shows a diagram for the Data Logger device.

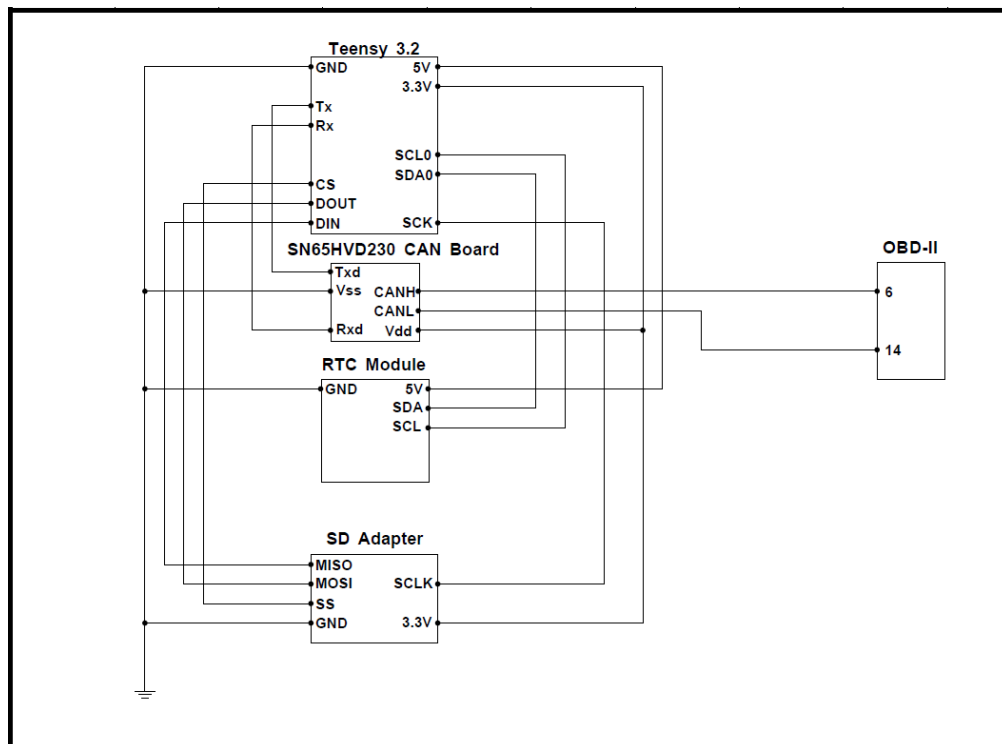


Figure 25: Final Data Logger Setup.

The main component is the Teensy 3.2 microcontroller; this microcontroller connects to the Micro SD card reader and to the SN65HVD230 microchip. The SN65HVD230

microchip is the CAN transceiver. It connects to the CAN transmitter (Tx) and receiver (Rx) pins of the Teensy 3.2 microcontroller as well 3.3V and ground. The other side of the CAN chip is CAN High and Low which connect to the OBD-II port on the vehicle. The CANH is pin 6, and the CANL is pin 14, as shown in the diagram. The Micro SD card adapter is connected on the other side of the Teensy 3.2 microcontroller. It has 3.3V, ground and different SPI port pin outs such as SCK, CS, DOUT and DIN. The last component of this device is the RTC clock module, which connects to the teensy using the SDA and SCL pin outs; it also requires 5V and ground. A chart all pin outs and voltages is shown in Table 18.

Table 18: Hardware Specifications.

| | Input Voltage | Pins Used | Function |
|------------------------|---------------|--|--|
| Teensy 3.2 | 5V | Tx, Rx (SN65HVD230 CAN Board) CS, DOUT, DIN, SCK (SD Card Adapter) SCL0, SDA0 (RTC Module) | Controls all components of the Data Logger device. Connects over CAN, SPI and I2C ports. |
| RTC Module | 5V | SDA, SCL | Used to synchronize Teensy 3.2 time on startup. Connects over I2C ports. |
| SD Card Adapter | 3.3V | MISO, MOSI, SCLK, SS | Stores data from Teensy 3.2 onto micro SD card. Connects through SPI ports |
| SN65HVD230 | 3.3V | TxD, RxD, CANH, CANL | These chips act as the interface between the CAN_BUS and the microcontroller. Connects over CAN ports |

Figure 26 shows a diagram for the test bench setup. The main component of the test bench setup is the BeagleBone Black microcontroller. CAN messages are simulated by playing back previously recorded messages by connecting to the SN65HVD230 chip through Tx and Rx; the purpose of this is to simulate a vehicle. In order to create the virtual CAN network, the two SN65HVD230 chips are connected together through CANH and CANL. Once the packets are sent from the BeagleBone Black, our Data Logger reads these packets and identifies the harmful UDS messages occurring if there are any. The test bench setup is used as a tool to simulate packets over a CAN network in order for us to see if our Data Logger is working correctly.

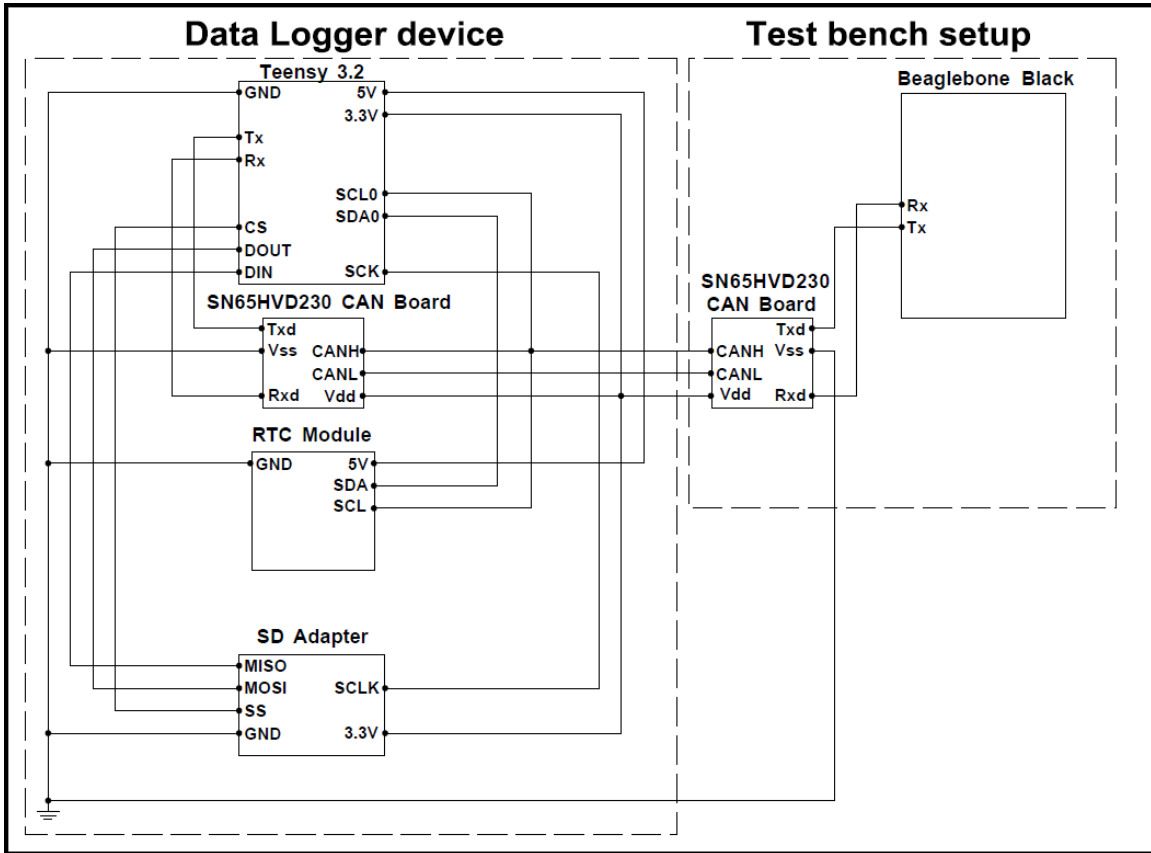


Figure 26: Final Test Bench Setup.

Figure 27 shows the final Data Logger setup with the BeagleBone Black connected to it. The BeagleBone Black acts as a playback device and sends the recorded Nissan Altima CAN packets to the Tx and Rx ports of one of the SN65HVD230 CAN Board. This chip connects to another SN65HVD230 CAN Board through the CAN High and CAN Low ports to create a CAN network. The second SN65HVD230 CAN Board sends the CAN packet through its Tx and Rx ports to the Tx and Rx of the Teensy microcontroller. The Teensy then analyzes the CAN packets to determine the proper course of action.

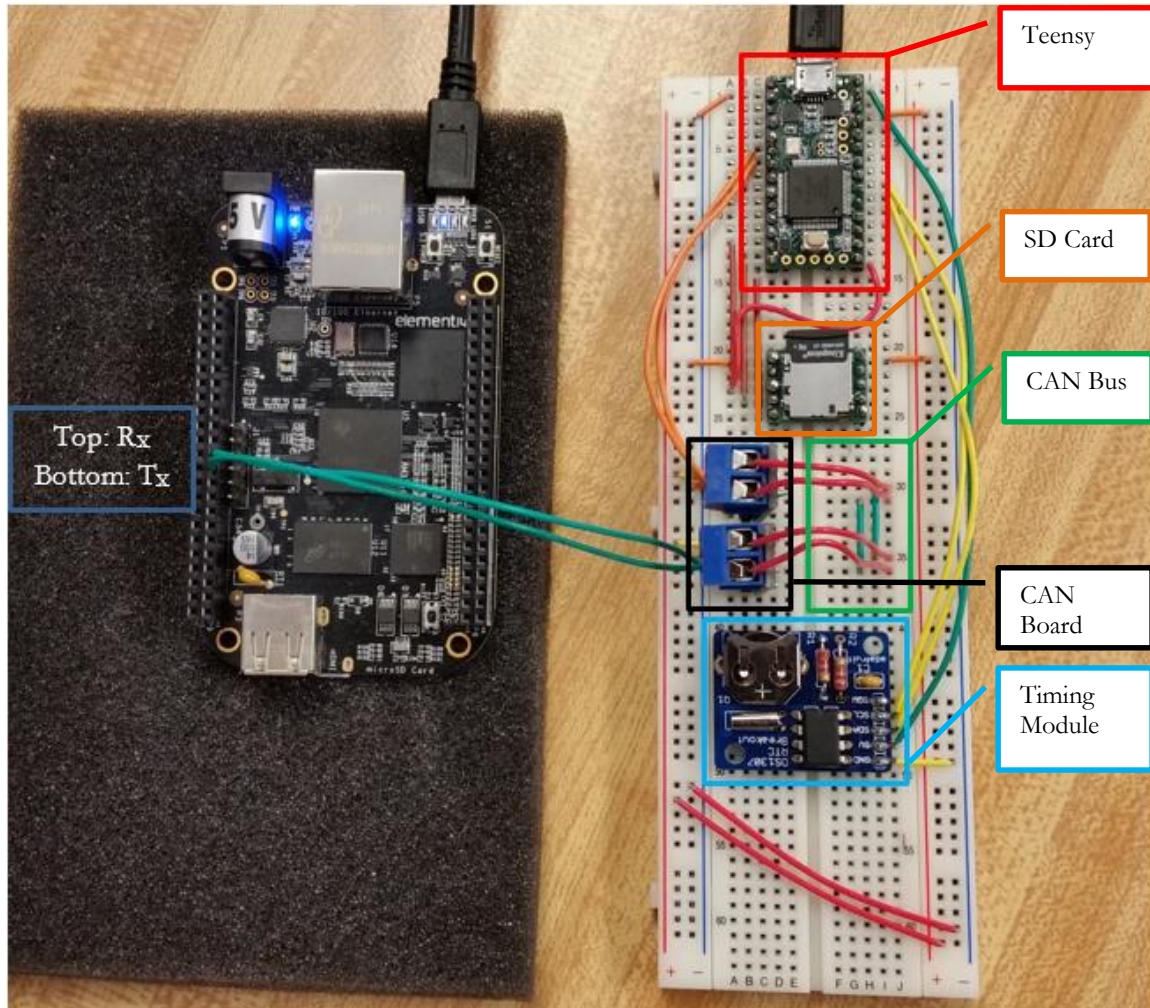


Figure 27: Picture showing the final Data Logger setup.

On the left is the BeagleBone Black sends the recorded Nissan Altima CAN packets to the SN65HVD230 CAN Board (Black box). This creates a CAN network (Green Box) which is then sent to the Teensy (Red Box). The information is saved to an SD card reader (Orange Box), using time information from the DS1307 RTC Timing Module (Light Blue Box).

4.2 Data Logging Algorithm

The algorithm used to detect and record UDS was first developed and tested on an Ubuntu Linux virtual machine in Python as a proof of concept. This Python code was then ported to C++ to execute on the Teensy based Data Logger prototype. The data logging algorithm processes each packet monitored on the bus. It first checks the packet's arbitration ID to determine whether or not it is a UDS message. This is done by simply checking if the arbitration ID is equal to 0x7E8, which is the identifier of UDS messages. The Data Logger

UDS Based Attack Data Logger

saves every message read to a circular buffer holding a maximum of 1800 packets (approximately 1.2 seconds of data) until a UDS message is processed – as referenced in Chapter 3.0.

There are two possible states for the bus. Normal state is when there has not been a UDS attack detected. During the normal state, the Data Logger device reads messages and puts them in a circular buffer. Once a UDS message is monitored on the bus, the Data Logger switches to the corrupted state and creates a new directory where all data is stored until the next power cycle occurs. Then, the Data Logger must write all contents of the circular buffer to a new file on the SD card. This file is called *Before_UDS_Attack_X.txt* indicating that this data was recorded previous to the first UDS attack monitored on the network. After the circular buffer is saved, the system changes state and it considers the incoming CAN traffic as corrupt. The corrupted state only becomes normal again when there is a power cycle.

The corrupt traffic is saved to a new file on the SD card marked as *After_UDS_Attack_1.txt* indicating that this is the first UDS attack to occur. A relatively short linear buffer (256 packets in length) is used to expedite writing to the SD card in a data stream. This is done by filling the buffer with packets, opening the data file, writing all the contents of the buffer to the file, and closing the file. The file is opened and closed to ensure that a power cycle will not result in data loss. If the buffer were very large and the device lost power, all of the contents of the buffer would be lost. The Teensy continues to save corrupt messages using this method until 80 corrupt ticks have elapsed after the last UDS message was found on the CAN Bus – as mentioned in Chapter 3.0. Finally, the Data Logger writes a string to the bottom of the file recording how many UDS messages were found in the attack.

After the Data Logger has recorded the messages for 80 corrupt ticks after the last UDS messages was detected, it returns to storing data to the circular buffer. This data is still considered to be corrupt traffic as the bus had been previously experiencing UDS messages. If another attack is monitored, the Data Logger repeats the actions previously discussed (saving the circular buffer and then 80 corrupt ticks of corrupt data after the last UDS attack). The files in subsequent attacks are named in the same format, except the attack number would be incremented. For example, the UDS attack data recorded during attack three

would be saved as *After_UDS_Attack_3.txt*. This algorithm is represented graphically in Figure 28.

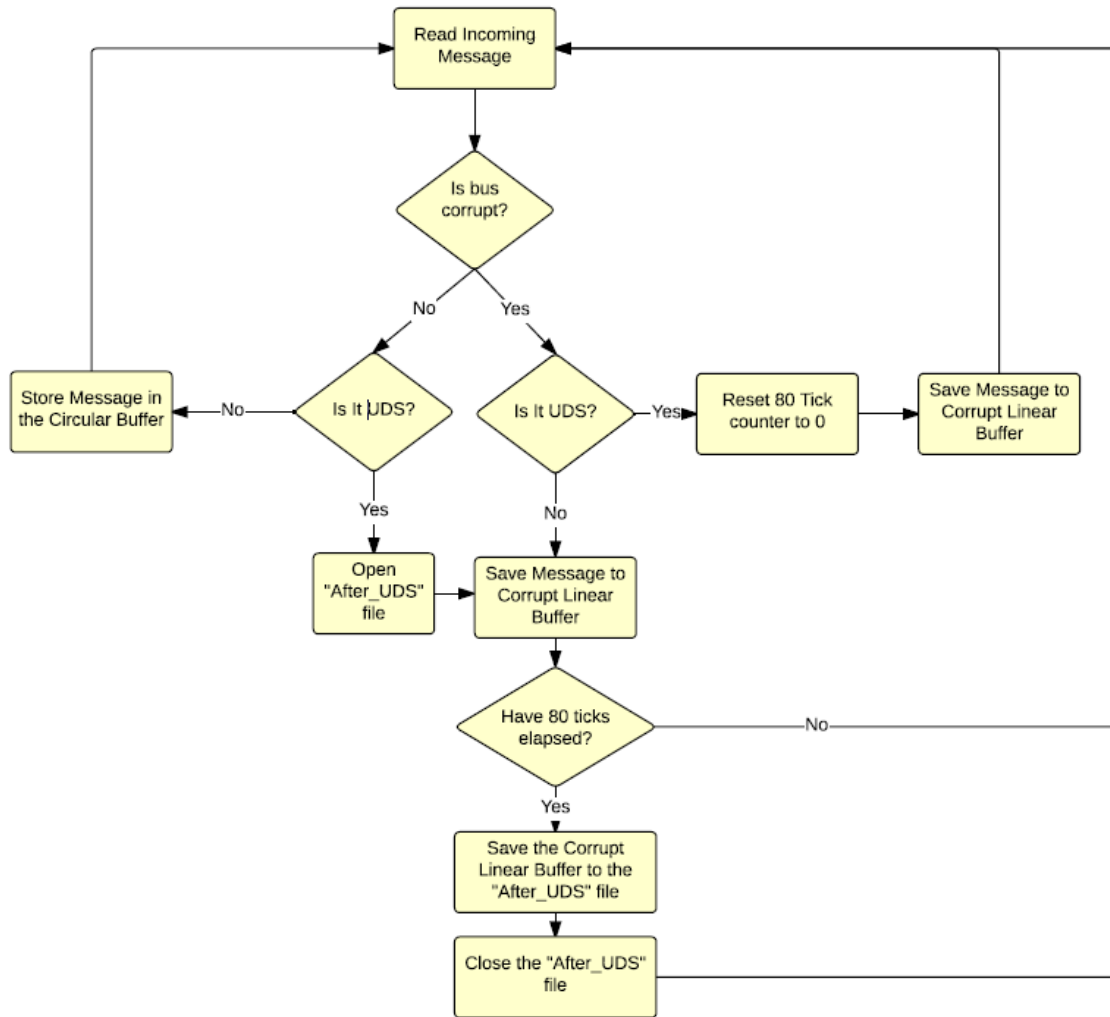


Figure 28: Algorithm flow chart describing both normal messages and UDS messages.

4.3 Attack Scenarios

The Data Logger was designed to handle multiple situations in which an attack could occur. It is not possible to anticipate the exact route a hacker will take to compromise a system, so the Data Logger was designed to take any UDS attack and break it down into its essential components. The Data Logger bases the importance of CAN traffic on the frequency of UDS messages and the spacing between attacks. The following attack scenarios show how the Data Logger responds to attacks and why it is important to do so.

UDS Based Attack Data Logger

Case 1: No Attack: This scenario occurs when the vehicle is either in motion or stationary and the CAN network has not been compromised. The vehicle is acting as it would under normal circumstances and no UDS messages are being transferred on the bus. Figure 29 shows the behavior of the system when the vehicle has not experienced a UDS attack.

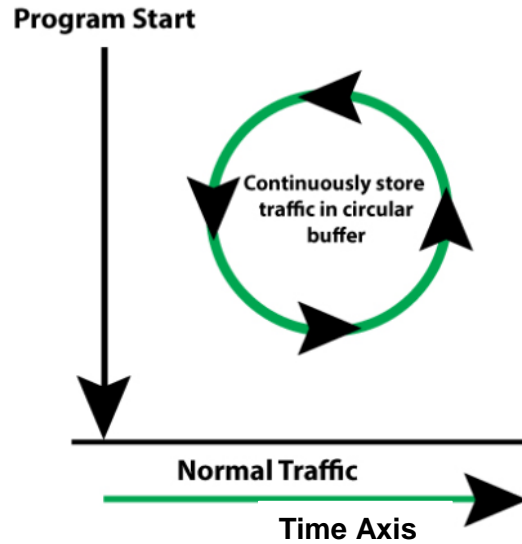


Figure 29: Sketch of Case 1: No Attack.

The Data Logger stores the past 1.2 seconds of normal CAN traffic to the circular buffer continuously, monitoring the bus for UDS messages.

Case 2 - Single UDS Attack: A Single UDS Attack occurs when all the UDS messages sent during the attack are time stamped within 80 corrupt ticks of the previous UDS message. An example of this scenario is when the attacker would like to load new firmware onto a target ECU. The bootloader for that ECU would consistently read UDS messages containing the malicious firmware. All the UDS messages in this attack would be sent in tight timing in order to quickly load the new firmware onto the targeted ECU. Since the attack should be read in order within a single file, the Data Logger will respond to such an attack in the manner shown in Figure 30.

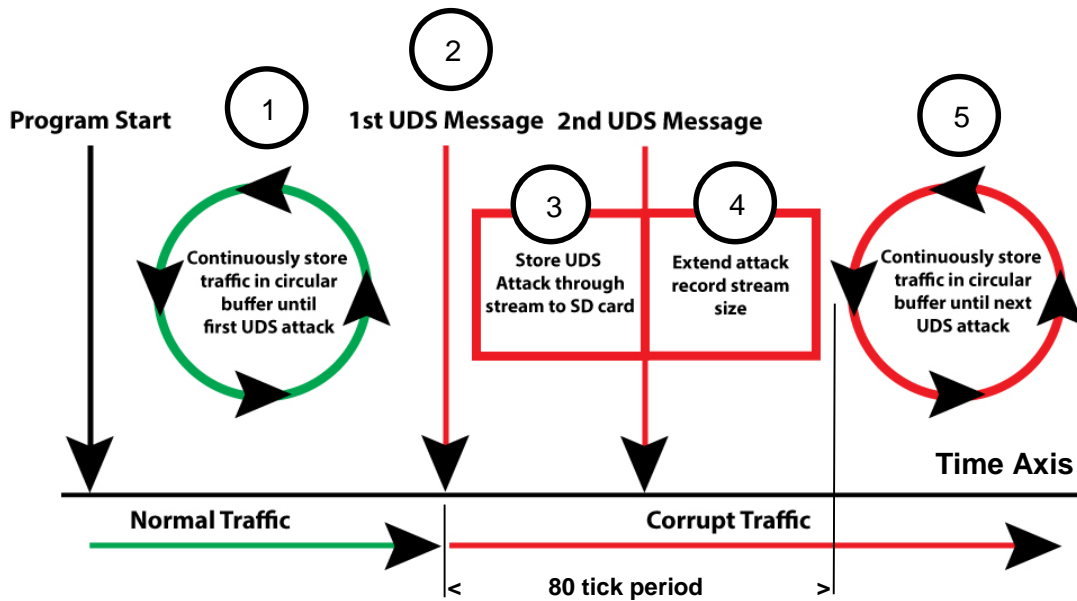


Figure 30: Sketch of the Case 2: Single UDS Attack.

The steps of the algorithm shown in Figure 30 are described in more detail:

1. Data Logger stores past 1.2 seconds of normal CAN traffic to the circular buffer continuously, monitoring the bus for UDS messages.
2. The first UDS message is monitored on the bus. Freeze the circular buffer, make a new directory for the attack, and dump all data to the SD card.
3. Record 80 corrupt ticks after the first UDS message was found through a data stream to the SD card.
4. In case another UDS message is monitored on the bus within the 80 corrupt tick period, record all data up to 80 corrupt ticks after the last UDS message is transferred on the bus through a data stream to the SD card.
5. Begin recording CAN traffic to circular buffer once again and wait for more UDS messages to come.

Case 3: Multiple UDS Attacks: Multiple UDS attacks occur when there is a time gap of more than 60 seconds between transfers of UDS messages. The two attacks are considered to be separate solely because of this time gap. An example of this occurrence is when the hacker would like to cover up his tracks after attempting a UDS attack. UDS messages could be used after the attack to make the vehicle seem as if it was functioning normally when an acci-

UDS Based Attack Data Logger

dent, if any, occurred. Without recording this data, a forensics analyst would assume the vehicle was not attacked and the hacker's effort to cover up the attack would pay off. The Data Logger responds to this attack situation in the manner shown in Figure 31.

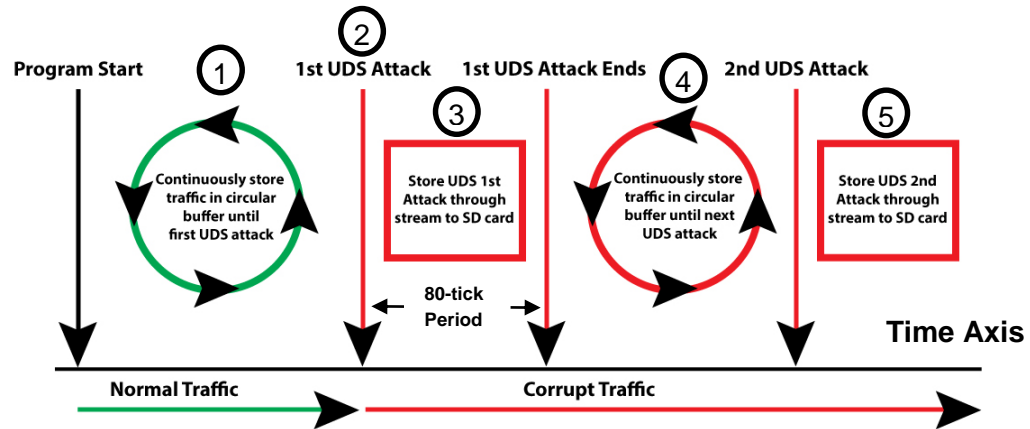


Figure 31: Sketch of Case 3: Multiple UDS Attacks.

The steps of the algorithm shown in Figure 31 are described in more detail:

1. Data Logger stores past 1.2 seconds of normal CAN traffic to the circular buffer continuously, monitoring the bus for UDS messages.
2. The first UDS attack is monitored on the bus. Freeze the circular buffer, make a new directory for the attack, and record all data to the SD card.
3. Record first UDS attack through a stream to the SD card until the attack has completed.
4. Begin recording CAN traffic to circular buffer once again. Wait for a new UDS attack to be monitored in the bus.
5. When another attack is monitored, repeat step 2 through 5.

Case 4 - Denial-Of-Service (DoS) Like Attack: In this scenario, the attacker would bombard the CAN network with UDS messages in order to gain control over the network. This would cause a “jamming” effect on the network, in the sense that no packets from ECUs on the network would be able to send messages. If this attack is used, there is not much the logger can do other than record what packets are monitored on the bus. Typically a node executing a DoS attack on the CAN network will be booted from the network and not trusted. Even though this seems like a self-healing event, in the case that the DoS attack removed a vital ECU from the network, the logger would have this recorded. Figure 31 shows a sketch of this type of attack.

1. Data Logger stores past 1.2 seconds of normal CAN traffic to the circular buffer continuously, monitoring the bus for UDS messages.
2. Attack occurs, freeze recording to the normal state circular buffer
3. Record Y seconds after the UDS attack of corrupted messages into a new linear buffer
4. When another UDS message is sent within the Y seconds, extend the recording time to include Y seconds after the last UDS message
5. Save circular buffer to a file indicating the number of the first UDS attack detected
6. Save UDS attack messages and corrupted traffic to a another file with the same UDS attack number
7. Begin recording CAN traffic to circular buffer once again - this traffic is still considered corrupted
 - a) Then record all UDS attacks of length Z into one *DOS ATTACK* file

4.4 Project Cost

This section covers the overall cost of this MQP project and the final deliverable. As previously stated, there was a shift in project scope for this project, so some components were not used in the final implementation of the Data Logger. The costs presented in this section are divided into three tables, the total cost of the MQP, the cost of the test bench, and the cost of the Data Logger. Table 19 outlines the total cost of this MQP project, totaling to \$309.08, which includes every component purchased throughout the project.

UDS Based Attack Data Logger

Table 19: Total MQP Cost Breakdown.

| Seller | Part Number | Part Name | Cost | Quantity | Total |
|----------|----------------------|---|-------|--------------|---------------|
| Adafruit | 264 | DS1307 Real Time Clock breakout board kit | 7.95 | 1 | 7.95 |
| Amazon | SN65HVD230 CAN Board | SN65HVD230 CAN Board | 9.99 | 2 | 19.98 |
| DigiKey | MCP2561 -E/P | MCP2561 microchip | 1.29 | 2 | 2.58 |
| DigiKey | MCP2551 -I/P | MCP2551 microchip | 1.29 | 2 | 2.58 |
| Groupon | Bluetooth OBD-II | OBD-II Diagnostic Tool | 12.99 | 1 | 12.99 |
| PJRC | SD_ADAPTER | Micro SD adapter | 8 | 2 | 16 |
| Seeed | 113030021 | Arduino SEEED Shield | 23.5 | 2 | 47 |
| SparkFun | DEV-11021 | Arduino Uno-R3 | 24.95 | 2 | 49.9 |
| SparkFun | DEV-10039 | Arduino Can-Bus shield | 39.95 | 1 | 39.95 |
| SparkFun | CAB-10087 | OBD-II to DB9 Cable | 9.95 | 1 | 9.95 |
| SparkFun | DEV-09911 | OBD-II connector | 3.95 | 1 | 3.95 |
| SparkFun | PRT-12794 | Jumper Wires (pack) | 1.95 | 1 | 1.95 |
| SparkFun | PRT-00116 | Break away headers | 1.5 | 1 | 1.5 |
| SparkFun | COM-11609 | Micro SD Kit | 13.95 | 1 | 13.95 |
| SparkFun | DEV-13736 | Teensy 3.2 board | 19.95 | 1 | 19.95 |
| SparkFun | PRT-12043 | Bread board | 3.95 | 1 | 3.95 |
| SparkFun | DEV-12857 ROHS | Beaglebone Black | 54.95 | 1 | 54.95 |
| | | | | Total | 309.08 |

A breakdown of the cost of the BeagleBone Black test bench is shown in Table 20. The test bench total was \$70.84.

Table 20: Cost Breakdown for BeagleBone Black Test Bench.

| Seller | Part Number | Part Name | Cost | Quantity | Total |
|----------|----------------------|----------------------|-------|--------------|--------------|
| SparkFun | PRT-12794 | Jumper Wires (pack) | 1.95 | 1 | 1.95 |
| SparkFun | DEV-12857 ROHS | Beaglebone Black | 54.95 | 1 | 54.95 |
| SparkFun | PRT-12043 | Bread board | 3.95 | 1 | 3.95 |
| Amazon | SN65HVD230 CAN Board | SN65HVD230 CAN Board | 9.99 | 1 | 9.99 |
| | | | | Total | 70.84 |

A breakdown of the cost involved in building a Data Logger is shown in Table 21. The cost per Data Logger is a total of \$71.19.

Table 21: Cost Breakdown for Data Logger Device.

| Seller | Part Number | Part Name | Cost | Quantity | Total |
|----------|----------------------|---|-------|--------------|--------------|
| Adafruit | 264 | DS1307 Real Time Clock breakout board kit | 7.95 | 1 | 7.95 |
| Amazon | SN65HVD230 CAN Board | SN65HVD230 CAN Board | 9.99 | 1 | 9.99 |
| PJRC | SD_ADAPTER | Micro SD adapter | 8 | 1 | 8 |
| SparkFun | DEV-09911 | OBD-II connector | 3.95 | 1 | 3.95 |
| SparkFun | PRT-12794 | Jumper Wires (pack) | 1.95 | 1 | 1.95 |
| SparkFun | PRT-00116 | Break away headers | 1.5 | 1 | 1.5 |
| SparkFun | COM-11609 | Micro SD Kit | 13.95 | 1 | 13.95 |
| SparkFun | DEV-13736 | Teensy 3.2 board | 19.95 | 1 | 19.95 |
| SparkFun | PRT-12043 | Bread board | 3.95 | 1 | 3.95 |
| | | | | Total | 71.19 |

4.5 Chapter Summary

The final Data Logger hardware consisted of the Teensy 3.2 microcontroller, the SD card as the optimal storage option for the device and the Adafruit DS130 RTC Timing Module. In order to properly test this prototype, the BeagleBone Black was selected as the most desirable microcontroller. The total price to build the Data Logger is about \$72. It also cost about \$71 in order to setup the BeagleBone Black Test Bench with the SN65HVD230 CAN boards.

The data logging algorithm processes each packet monitored on the bus and first checks the packet's arbitration ID to determine whether or not it is a UDS message (0x7E8). The Data Logger saves every message to a circular buffer which holds a maximum of 1800 packets (approximately 1.2 seconds of data) until a UDS message is processed – as referenced in Chapter 3.0. If a UDS message is detected, the Teensy saves the corrupt messages until 80 corrupt ticks have elapsed after the last UDS message was found on the CAN Bus. Once this is completed and the file saved, the circular buffer starts over. The device was tested using four attack scenarios: (1) No Attack, (2) Single UDS Attack, (3) Multiple UDS Attacks, and (4) Denial of Service Attack.

5.0 Experimental Results

This section covers the tests that were performed on both the Python virtual proof of concept and our prototype implementation of the Data Logger. As previously mentioned, the Data Logger is expected to record enough data that a forensics analyst could determine exactly what a UDS attack was attempting to accomplish and what actually occurred as a result. There are approximately 1500 packets transferred on the 2014 Nissan Altima CAN network per second. The Data Logger is not required to save all of this data, just the messages that happen before and after a UDS attack. As mentioned in Chapter 4.1 it was determined that recording 10 normal ticks before and 80 corrupt ticks after the UDS attack would be enough to pinpoint any changes in the system caused by a UDS attack.

Packet streams recorded from the OBD-II of a 2014 Nissan Altima in multiple scenarios were used to test the functionality of the Data Logger. In order to establish a controlled testing environment, we used the NO_UDS_CITY_DRIVING with no UDS messages present in the vehicle's CAN Bus. This allowed us to send in UDS packets at specific times in order to simulate the scenarios in Chapter 2.3. The four attack scenarios tested on the prototype were:

1. No UDS Attack
2. Single UDS Attack
3. Multiple UDS Attacks
4. Denial of Service (Dos) Attack

5.1 Python Implementation Results

The algorithm for the Data Logger device was first developed in Python. The Python programmed used the SocketCAN APIs (referenced in Appendix E) in order to read packets being sent on the Virtual CAN Bus networks and process them. The program processes them by checking whether the message is an UDS message (for which the ID = 0x7e), creating a message object, and storing the message either in the circular buffer or linear buffer depending upon its ID. A test vector and the packet stream previously recorded from the 2014 Nissan Altima were used in order to test the Python version of the algorithm.

The Python algorithm used a circular buffer big enough to capture 1800 packets as mentioned in Chapter 3.0. The linear buffer can contain up to 1024 packets and after it is full, the buffer is then written into the created file. The algorithm will only store 90000 messages after the last UDS packet, which is equivalent to 80 corrupt ticks. The Python algorithm also prints the messages on the terminal in order to facilitate debugging. It also prints statistics both to the terminal and the *after_UDS* file.

A test vector was created because we needed simpler message streams in order to run initial tests. These test vectors were made in the simple form of a Linux bash script files. Running these two scripts would send a series of messages using the *cansend* commands from the Linux CAN utilities. These simpler tests used arbitrary IDs and payloads. The Python algorithm had smaller page sizes and buffer sizes to simplify the number of messages in the folder and to easily identify any potential lost messages or possible errors in the data. This phase of testing was focused to test the logic of the program and the overall behavior of the algorithm. Only a single UDS attack was tested as the Python testing was just for proof of concept. From this test, a directory is created every time the program is run (and files are overwritten if they existed before as well).

The *canplayer* tool was then used to send test vectors recorded from the vehicle. The *canplayer* tool sends data at the rate recorded which makes it a slower process than the bash scripts. In order to play the data back in a Virtual CAN network, it is necessary to modify the file and change the network device from *can0* to *vcan0* (virtual can 0) so the tools can send it over the virtual CAN, and not a physical CAN network as it happened in the car. The screenshot shows the terminal displaying the messages received from another terminal using the *canplayer* tool to send messages from the *UDS_CITY_DRIVING* log file. When the message with ID *0x07e8* is received, the system records that a UDS message is detected. A screenshot of the packets on the virtual CAN network is shown in Figure 33.

UDS Based Attack Data Logger

```
1455919241.515986      01f9      000      8      68 00 16 8f 00 00 00 00
1455919241.517068      0002      000      8      88 01 00 07 ac 71 46 04
1455919241.519241      0285      000      8      00 00 00 00 00 00 9f 26
1455919241.520333      0174      000      8      7e 46 54 aa 08 00 00 00
1455919241.521426      0177      000      8      fe 10 40 00 32 00 80 00
1455919241.522504      07e8      000      8      03 41 04 45 00 00 00 00
UDS attack detected
```

Figure 33: Screenshot of packets being received on the virtual CAN network.

The format of the packets is printed as follows:

<timestamp><ArbitrationID> <ErrorFrame>

<Size of the Payload> <Payload>.

The files are created in the DataLoggerFiles Folder. In the *beforeUDS_1.txt* there are 1279 messages (1 per line). A screenshot of the before UDS file in Figure 32. In this case there were 1,977 UDS messages in the *afterUDS_1.txt* file as shown in Figure 34.

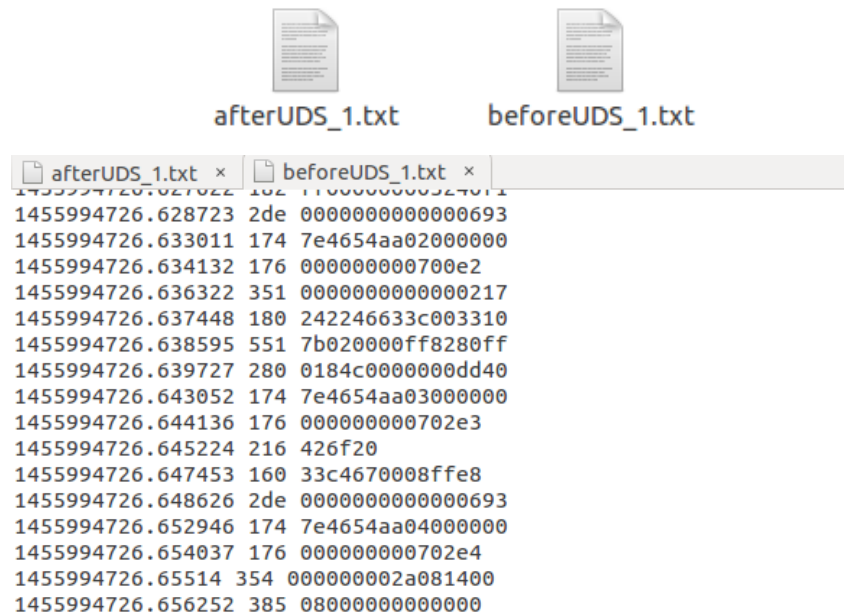


Figure 32: Screenshot of the two text files, *afterUDS_1.txt* & *beforeUDS_1.txt*. The first screenshot is the *beforeUDS* file which consists of the packets before the UDS attack.

```

afterUDS_1.txt x beforeUDS_1.txt x
1455994728.674422 180 206c3
1455994728.675601 245 fff00
1455994728.677806 02 1f000007f40700c00
1455994728.678894 280 0184c00000003880
1455994728.679979 355 0000000020006000
1455994728.683232 174 7e4654aa0f000000
1455994728.684349 160 3443200008ffe8
1455994728.685468 1f9 2000206c00000000
1455994728.686584 215 fff0ff00ffff
1455994728.687723 02 1f0000077e96bc08
1455994728.688856 5c5 400070a3000c0000
1455994728.693241 174 7e4654aa00000000
1455994728.69435 177 fe10400032008000
1455994728.695462 1f9 2000206c00000000
1455994728.69656 35d 8003000000000055
1455994728.69765 02 1f0000076f96bc08
1455994728.698713 280 0183c00000003880
1455994728.699803 2de 0000000000000693
1455994728.704132 174 7e4654aa01000000
1455994728.705258 180 2085320344003d10
1455994728.706379 215 fff0ff00ffff
1455994728.707513 02 200000075096bc08
1455994728.708643 2de 0000000000000693
1455994728.714047 174 7e4654aa02000000
1455994728.715147 180 2085321344003e10
1455994728.71627 358 0282000000000000
1455994728.717394 7e8 04410c1036000000
1455994728.719566 355 0000000020006000
1455994728.723897 174 7e4654aa03000000
  
```

Figure 34: Screenshot is the after UDS file which consists of the packets after the UDS attack.

The results from the Python testing show that the algorithm's logic is correct. A single UDS packet was sent to the virtual CAN network, and the algorithm detected this packet, and then began to record two files. One file stores the data before the UDS attack and the other after the UDS attack. These results confirmed the proof of concept for the algorithm that was used to build the code for the Data Logger.

5.2 Data Logger Implementation Results

A series of tests were conducted in order to confirm the Data Logger prototypes functionality. Initial tests consisted of generated packets on the Teensy Board and storing them in the buffers simulating that they were received from the CAN Bus Network. Subsequently, the Test Bench was set up and tested to ensure that messages were being sent and received. Finally, a data stream of packets from the Nissan Altima 2014 was used to test the algorithm.

UDS Based Attack Data Logger

Data Logger Development Testing

During development, tests were run internal to the Data Logger's code without connection to the test bench. This was done to increase the number of tests that could be executed during development and allowed for running pseudo-random UDS attacks without having to record each one from an actual vehicle. These tests were created by generating randomized normal and corrupt traffic test vectors during program execution. The Data Logger processed packets during attack simulations as it normally would, but no CAN hardware was used. The goal of simulating attacks was to help debug the code with simpler test vectors and in incremental stages, similar to in an Agile development environment.

Design Testing

The BeagleBone Black Test Bench was set up to simulate a physical CAN network by using the SocketCAN utilities. The SocketCAN physical network functions allow for playing back the packet streams at the exact timing in which they were recorded, making these tests capable of simulating an actual vehicle perfectly. Using a simulated CAN network not only assisted in saving test time, but also mitigated risking to damage the test vehicle, made potentially dangerous tests safe, and allowed us to modify packet streams or create new ones for testing the algorithm when needed. The next step was to use to communicate with the Data Logger device. This was done by interfacing both devices by using the SN65HVD230 high-speed CAN boards. Initially, some troubleshooting and testing was done to ensure that the SN65HVD230 CAN boards were sending and receiving CAN signals correctly.

Figure 35 shows an oscilloscope reading which was used to test the BeagleBone Black and the SN65HVD230 CAN board. The yellow input line is the CAN signal being transmitted from the BeagleBone Black to the CAN board. The green input line is the CAN signal being received by the Teensy microcontroller from the two CAN boards. The two signals are identical which proves that the CAN boards are working and creating a CAN network that receives and CAN packet and then transmits it to the microcontroller for analysis. This testing was necessary to complete before analyzing the final testing to ensure that the CAN boards were working properly.



Figure 35: Oscilloscope reading from the Data Logger with the BeagleBone Black. The yellow signal is the CAN packet from the BeagleBone Black and the green signal is the CAN packet at the Rx pin of the Teensy.

After the oscilloscope testing proved that a CAN packet transmitted successfully from the BeagleBone Black to the Teensy microcontroller, the NO_UDS_CITY_DRIVING text file was transmitted across the virtual CAN network. Figure 36 is a screenshot of the CAN packets being received by the Teensy.

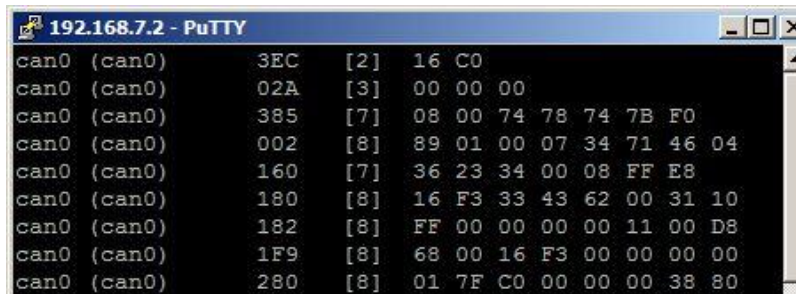


Figure 36: Screenshot of packets being received on the virtual CAN network. The format of the packets is printed as follows: <FromNetwork> <ToNetwork> <ArbitrationID> <Size of the Payload> <Payload>.

5.2.1 SD Card Writing Optimization Tests

A preliminary test was conducted to obtain the most effective linear buffer size used to write pages of CAN data to the SD card. In order to determine the size of the linear

UDS Based Attack Data Logger

buffer, multiple buffer sizes were tested with the same attack packet stream and two different CAN libraries, one written in C++ that uses polling and another written in C that uses interrupts. The packet stream chosen to run these tests was considered the most challenging for the Data Logger to record completely. This was done to obtain a better spread of the data, making the buffer size choice more obvious based on the results. The metric used to choose which linear buffer size to use was the percentage of the total number of attack packets recorded to the SD card. The results of these tests conclude that the ideal buffer size is 1024 packets for the C++ library and 512 packets for the C library as shown below in Table 22 and Table 23.

Table 22: Linear buffer size C++ library results.

| Linear Buffer Size (Packets) | Attack Packets Recorded |
|------------------------------|-------------------------|
| 32 | 58.1% |
| 64 | 66.3% |
| 128 | 70.0% |
| 256 | 71.0% |
| 512 | 6.75% |
| 1024 | 71.7% |

Table 23: Linear buffer size C library results.

| Linear Buffer Size (Packets) | Attack Packets Recorded |
|------------------------------|-------------------------|
| 32 | 63.5% |
| 64 | 68.0% |
| 128 | 70.3% |
| 256 | 63.0% |
| 512 | 71.6% |
| 1024 | 67.1% |

5.2.2 No UDS Attack Test

A packet stream with no UDS messages present on the vehicle's CAN Bus was used for this test. We did not send any UDS packets to the simulated CAN network because this test should not have any UDS packets present. The Data Logger should respond to this packet stream by not recording any packets as no attack occurred in this scenario.

Pass Conditions:

1. The Data Logger does not create any log files as no UDS attacks have occurred in this test case.
2. The Data Logger continuously records traffic into the circular buffer, checking that an attack has occurred after the entire packet stream has been sent to the device.
3. No errors are flagged by the Data Logger during the entirety of the test.

Figure 37 is a screenshot that shows the resulting root directory of the SD card after this test has been executed. This test has been passed as no files were recorded and no errors were signaled by the device during test execution.

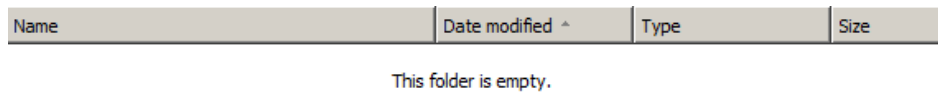


Figure 37: Screenshot of the Data Logger folder when there is no UDS attack.

5.2.3 Single UDS Attack

A packet stream with no UDS messages present on the vehicle's CAN Bus was also used for this test. In order to simulate a single UDS attack, we sent a UDS packet on the bus once during the execution of this test. This simulated a malicious UDS device that can communicate over the network via either wired or wireless connection. The resulting packet streams contain both normal and corrupt CAN traffic, which will test the Data Logger's ability to determine when the attack occurs and how to store the data properly. The Data Logger should respond to this packet stream by creating two log files, one before and one after the single UDS attack.

Pass Conditions:

1. The Data Logger creates a log file, containing the 10 normal ticks (1800 packets) immediately prior to the UDS attack that occurred.
2. The Data Logger creates a second log file, containing 80 corrupt ticks (90000 packets) after the last UDS message recorded in the attack.
3. The Data Logger continuously records corrupt traffic into the circular buffer, checking that an attack has occurred after the entire packet stream has been sent to the device.

UDS Based Attack Data Logger

4. No errors are flagged by the Data Logger during the entirety of the test.

The screenshot in Figure 38 is this test's resulting folder containing the single attack's packet stream data in which both the data before and after the attack have been stored. This test has been passed as the data before and after the attack was recorded to the SD card without any errors signaled by the Data Logger.

| Name | Date modified ^ | Type | Size |
|---------------------|-------------------|---------------|----------|
| Before_UDS_Attack_1 | 2/29/2016 3:56 PM | Text Document | 61 KB |
| After_UDS_Attack_1 | 2/29/2016 3:57 PM | Text Document | 3,097 KB |

Figure 38: Screenshot of the Data Logger folder when there is a single UDS attack.

Figure 39 is a screen shot of the *After_UDS_Attack_1* text file. The UDS packet is the first one recorded as this is when the linear buffer begins. The complete file only contains one UDS packet and 80 corrupt ticks of data after the UDS packet.

```
1 After_UDS_Attack_2.txt
2
3 TIMESTAMP (MS) ID DATA
4 166907 7E8 AE 05 10 77
5 170147 174 7E 46 54 AA 06 00 00 00
6 170149 002 CB 01 00 07 98 53 59 0A
7 170151 177 FE 10 40 00 32 00 80 00
8 170152 176 00 00 00 00 07 19 E6
9 170153 280 03 5A C0 00 00 00 1D 80
10 170154 2DE 00 00 00 00 00 00 08 1F
```

Figure 39: Screenshot of the *After_UDS_Attack_1.txt* file.
The UDS arbitration ID is in the red square.

This is the same format of the other attack scenarios files, except the DoS attack. The *After_UDS_Attack_1* text file for DoS attack will include however many UDS packets were sent during the attack.

5.2.4 Multiple UDS Attack

A packet stream with no UDS messages present on the vehicle's CAN Bus was used again during this test. In order to simulate specific UDS attack scenarios, we sent in UDS packets at certain times. The resulting packet streams contain both normal and corrupt CAN traffic, which will test the Data Logger's ability to determine when multiple attacks occur and

how to store the data properly. The Data Logger should respond to these packet streams by creating two log files for each UDS attack (one before and one after each attack).

Pass Conditions:

1. The Data Logger creates a log file for each UDS attack, containing the 10 normal ticks (1800 packets) immediately prior to the UDS attack that occurred.
2. The Data Logger creates a log file for each UDS attack, containing 80 corrupt ticks (90000 packets) after the last UDS message recorded in the attack.
3. The Data Logger continuously records corrupt traffic into the circular buffer, checking that an attack has occurred after the entire packet stream has been sent to the device.
4. No errors are flagged by the Data Logger during the entirety of the test.

The screenshot in Figure 40 shows the resulting folder containing the five attacks' data both before and after each attack occurs. This test has been passed as the data before and after each attack was recorded to the SD card without any errors signaled by the Data Logger.

| Name | Date modified ^ | Type | Size |
|---------------------|-------------------|---------------|----------|
| Before_UDS_Attack_1 | 2/29/2016 3:45 PM | Text Document | 62 KB |
| After_UDS_Attack_1 | 2/29/2016 3:47 PM | Text Document | 3,157 KB |
| Before_UDS_Attack_2 | 2/29/2016 3:47 PM | Text Document | 64 KB |
| After_UDS_Attack_2 | 2/29/2016 3:48 PM | Text Document | 3,185 KB |
| Before_UDS_Attack_3 | 2/29/2016 3:48 PM | Text Document | 64 KB |
| After_UDS_Attack_3 | 2/29/2016 3:50 PM | Text Document | 3,185 KB |
| Before_UDS_Attack_4 | 2/29/2016 3:52 PM | Text Document | 64 KB |
| After_UDS_Attack_4 | 2/29/2016 3:53 PM | Text Document | 3,767 KB |
| Before_UDS_Attack_5 | 2/29/2016 3:54 PM | Text Document | 64 KB |
| After_UDS_Attack_5 | 2/29/2016 3:55 PM | Text Document | 3,185 KB |

Figure 40: Screenshot of the Data Logger folder when there is multiple UDS attacks.

5.2.5 Denial-Of-Service (DoS) Attack

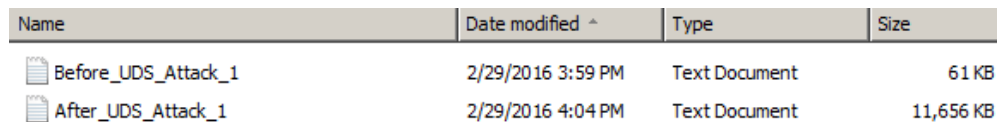
To test the Data Logger's ability to record a DoS attack, a packet stream containing multiple UDS messages in each corrupt tick was used. The Data Logger should respond to this packet stream by creating two log files, one before and one during the UDS attack.

UDS Based Attack Data Logger

Pass Conditions:

1. The Data Logger creates a log file, containing the 10 normal ticks (1800 packets) immediately prior to the DoS attack that occurred.
2. The Data Logger creates a log file, containing 80 corrupt ticks (90000 packets) after the last UDS message recorded in the DoS attack.
3. The Data Logger continuously records corrupt traffic into the circular buffer, checking that an attack has occurred after the entire packet stream has been sent to the device.
4. No errors are flagged by the Data Logger during the entirety of the test.

The screenshot in Figure 41 shows the resulting folder containing the DoS attack packet stream. This test was passed as the data before and during the DoS attack was stored to the SD card without any errors signaled by the Data Logger.



| Name | Date modified ^ | Type | Size |
|---------------------|-------------------|---------------|-----------|
| Before_UDS_Attack_1 | 2/29/2016 3:59 PM | Text Document | 61 KB |
| After_UDS_Attack_1 | 2/29/2016 4:04 PM | Text Document | 11,656 KB |

Figure 41: Screenshot of the Data Logger folder when there is a DoS UDS attack.

5.3 Chapter Summary

After executing multiple tests with the BeagleBone Black Test Bench and Data Logger, we examined the files created for each test. It became apparent during our analysis that 28.2 % of the packets were missed by the Data Logger when faced with a DoS attack. This happens when using the C++ library and a linear buffer size of 1024 packets, which is the most efficient of the options previously discussed. Other options including different linear buffer sizes and the C library have proven to be less effective than the current prototype implementation.

Since the Data Logger prototype is incapable of logging all packets that are monitored on the bus during and after a UDS attack has occurred, further design and optimization is necessary. This issue can be attributed to the extended write latency when storing large amounts of CAN traffic to the SD card with a single threaded CPU. A possible solution to this problem is to use a microcontroller that is capable of running multiple threads on one or more CPU cores. One thread can be used to read the traffic on the CAN network

while the other thread writes previously recorded traffic to the SD card. This design shift would entail further algorithm optimization in which a second circular buffer would be used to allow both threads to access the corrupt traffic at once. This would mitigate the need for the linear buffer currently used to assist in the stream to the SD card and make CAN data reading and writing parallel tasks.

6.0 Conclusion

Although detection and recovery are progress for vehicle security, there is still the issue of preventing vehicle attacks and forming countermeasures that can potentially save lives. A device that addresses all of these issues could prevent UDS attacks, as long as there is support from vehicle manufacturers. A data logger could be installed in all vehicles or instead the data logged from the vehicle could be used to patch bugs in source code. Regardless of how vehicle manufactures install and use a data logging device, there must be a way to prevent and/or counter UDS attacks from within the vehicle itself.

Vehicle technology is continually advancing, in order to assist the driver and passengers by making their ride more safe and enjoyable. Some of these advancements include assisted parallel parking, assisted braking, and automated highway driving. Even though these advancements provide better service to the consumers, they present security risks. Vehicle advancements usually involve automated or assisted services. This means the vehicle operator is not in control while the vehicle's microcontrollers and ECUs are actually in control. It is easy to see how this presents a security risk; if a hacker were to gain access to certain ECUs, he could control an automated or assisted vehicle service. Vehicle manufacturers are working to develop proper vehicle security, but have not been able to keep up with the technological advancements, in large part due to the long vehicle development time. This prototype Data Logger would be beneficial to both vehicle operators and manufacturers, as it would help protect the operators, while giving manufacturers important information about how hackers are using the UDS services to compromise vehicles.

Once final testing was completed, the Data Logger was concluded to perform appropriately, but some CAN packets were dropped. This can be attributed to two reasons; the CPU on the Teensy is a single core, single threaded CPU and the CAN libraries are not fully optimized. If the selected microcontroller had a CPU capable of multi-threading, one thread could read CAN data from the vehicle, while the other recorded this CAN data to external storage. A new library could be optimized for this new CPU in order to take advantage of onboard caches available to the developer.

7.0 Recommendations

There are many expansions that could be completed by future project groups using the Data Logger developed in this MQP as a starting point. Research completed in this MQP covers vehicle hacks, the CAN network, vehicle simulation with a test bench, and data logging hardware. A project team starting a project in this field would have a solid foundation of knowledge, hardware, and software to build upon in pursuing the following recommended expansions. One of these project expansions alone would not by itself merit a full MQP project, but using some of the expansions in combination would make the device a standalone product that is desperately needed in the vehicle market today.

Power via Vehicle - Although the Data Logger developed in this MQP can run via USB, it was out of project scope to use the device on a vehicle, and therefore, power the device via OBD-II port. The 12 volt rail on the OBD-II port can be dampened to 5V with a voltage regulator to power the Teensy and all other devices in the system. Additionally, a battery could be added to prevent the Teensy from losing memory if the vehicle's power is turned off.

Wireless Communication - The Teensy 3.2 board has the ability to interface with Bluetooth, Wi-Fi, and cellular peripherals. A future project group could use one of these peripherals to send data wirelessly to off-site storage. Using cellular capability may be the best option as data can be sent from anywhere. This expansion would also allow for email and text messages to be sent to the device owner, providing updates about the vehicle.

Vehicle Tests - The testing for this MQP was done using vehicle packets but on a separate Beagle Bone Black test bench. This is because sending UDS packets to a vehicle is dangerous and can damage the vehicle. A future MQP group should work to obtain an actual test vehicle, in order to perform in-vehicle testing.

Algorithm Optimization - One optimization for a future MQP group is to have the algorithm automatically determine the length of a normal and corrupt tick. This would be helpful because then the Data Logger could connect to any vehicle and determine the proper length of a normal and corrupt tick to gather sufficient information during an attack. Furthermore, the

UDS Based Attack Data Logger

optimizations described in Chapter 5.3 would assist in recording all packets monitored on the network.

Teensy CAN Library Optimization - The CAN libraries tested on this prototype have limited CAN Bus capabilities that allow them to interface with CAN and are not optimized to be as fast as possible. This C++ library uses polling to retrieve the incoming messages instead of more advanced mechanisms such as interrupts. The C library, developed by Hristos Giannopoulos, is more advanced and was optimized to use the Teensy FIFO queues. It uses interrupts to directly interface with the FIFO buffers. This library offered slightly better performance overall, but testing has proven the Teensy has speed limitations. Once a multicore or multithreaded microcontroller has been chosen for a new prototype Data Logger, a new optimized CAN library will be necessary to interface with the hardware efficiently.

Multithread Application - Since the Data Logger prototype is incapable of logging all packets that are monitored on the bus during and after a UDS attack has occurred, further design and optimization is necessary. This issue can be attributed to the extended write latency when storing large amounts of CAN traffic to the SD card with a single threaded CPU. A possible solution to this problem is to use a microcontroller that is capable of running multiple threads on one or more CPU cores. One thread can be used to read the traffic on the CAN network while the other thread writes previously recorded traffic to the SD card. This would entail choosing a multithreaded CPU and implementing the CAN reading algorithm to take advantage of the new CPU's multithreading capabilities.

Report on UDS Attack - Another recommendation is to purchase the UDS ISO guide. This guide will provide information about what UDS packet corresponds to each UDS service. This would be useful because now a future MQP would be able to determine exactly what the UDS packets mean. They could use this information to determine the proper form of action in order to protect the vehicle.

Expansion of Security - Currently, the Data Logger helps in the detection and recovery from a vehicle hack. A future MQP group could work to evolve the device to help with prevention, deflection and countermeasures to a vehicle hack. Prevention involves developing a defense that prevents a hacker from ever sending hazardous UDS message to the vehicle. Deflection

is a similar type of defense, but now the device would re-direct the hacker to a non-vital part of the vehicle. A countermeasure defense would involve the device attacking the hacker and preventing them from harming the vehicle.

8.0 Bibliography

- [1] R. R. Nayak, "Automotive Diagnostic Services," 11 December 2011. [Online]. Available: <http://unifieddiagnosticservices.blogspot.com/2011/12/automotive-diagnostics-services.html>. [Accessed 1 December 2015].
- [2] Carhistory4u.com, "Car History 4U," 2015. [Online]. Available: www.carhistory4u.com/the-last-100-years/car-production. [Accessed 12 October 2015].
- [3] J. Turley, "Motoring with Microprocessor," 11 August 2003. [Online]. Available: <http://www.embedded.com/electronics-blogs/significant-bits/4024611/Motoring-with-microprocessors>. [Accessed 10 February 2016].
- [4] Bosch, "CAN Specification, 1st ed. Bosch," 1991. [Online].
- [5] C. Timberg, "The definitive account of how hackers can gain access to our cars," 22 July 2015. [Online]. Available: <http://www.washingtonpost.com/sf/business/2015/07/22/hacks-on-the-highway/>. [Accessed 2015 October 2015].
- [6] A. Dharmawan, "Electric Cloud," 8 December 2014. [Online]. Available: <http://electric-cloud.com/blog/2014/12/continuous-delivery-puts-automotive-software-high-gear/>. [Accessed 3 January 2016].
- [7] B. World, "127 Years of Modern Automobile Evolution," 21 October 2013. [Online]. Available: <http://blog.world-mysteries.com/science/127-years-of-modern-automobile-evolution/>. [Accessed 6 January 2016].
- [8] C. Poladian, "Car Hacking White Paper Released By DARPA-Funded Researchers, Shows How To Hack A Ford And Toyota Prius," 2013. [Online]. Available: <http://www.ibtimes.com/car-hacking-white-paper-released-darpa-funded-researchers-shows-how-hack-ford-escape-toyota-prius>. [Accessed 5 October 2015].
- [9] A. Drozhzhin, "Black Hat USA 2015: The full story of how Jeep was hacked," 2015. [Online]. Available: <https://blog.kaspersky.com/blackhat-jeep-cherokee-hack-explained/9493/>. [Accessed 5 October 2015].
- [10] C. M. a. C. Valasek, Remote Exploitation of an Unaltered Passenger Vehicle 1st ed., 2015, pp. 3-88.
- [11] L. Stahl, "DARPA: Nobody's safe on the internet," 2015. [Online]. Available: <http://www.cbsnews.com/news/darpa-dan-kaufman-internet-security-60-minutes/>. [Accessed 5 October 2015].
- [12] M. Harris, "Researchers Hacks Self-driving CAR Sensors," 2015. [Online]. Available: http://spectrum.ieee.org/cars-that-think/transportation/self-driving/researcher-hacks-selfdriving-car-sensors?utm_campaign=Weekly%20Notification%20IEEE%20Spectrum%20Tech%20Alert&utm_source=boomtrain&utm_medium

- =email&utm_content=&bt_alias=eyJ1c2VySWQiOiI3YTc0Mzk0NC1mODFjLTQzMzItYTJiMCM0ZDdlODI0YzIxNjJlQ%3D%3D. [Accessed 5 October 2015].
- [13] D. Gershgorn, "Hackers Can Trick Driverless Cars With A Handheld Laser," 2015. [Online]. Available: <http://www.popsci.com/hackers-can-trick-lidar-used-in-autonomous-cars-with-laser-pointer>. [Accessed 12 October 2015].
- [14] K. Zetter, "Researchers Hacked a Model S, But Tesla's Already On Itq," 2015. [Online]. Available: <http://www.wired.com/2015/08/researchers-hacked-model-s-teslas-already/>. [Accessed 5 October 2015].
- [15] BBC, "Fiat Chrysler Recalls 1.4 Million Cars after Jeep Hack," p. 1, 24 July 2015.
- [16] Bosch, "CAN with Flexible Data-rate, 1st ed. Bosch," 2012. [Online].
- [17] ISO, "ISO 14229-1:2013," 15 March 2013. [Online]. Available: http://www.iso.org/iso/catalogue_detail.htm?csnumber=55283. [Accessed 1 December 2015].
- [18] D. K. Nilsson and U. E. Larson, "Conducting Forensic Investigations of Cyber Attacks on Automobile In-Vehicle Networks," Chalmers University of Technology, Goteborg, 2008.
- [19] U. E. Larson and D. K. Nilsson, "Securing Vehicles against Cyber Attacks," Chalmers University of Technology, Goteborg, 2008.
- [20] National Highway Traffic Safety Administration, "EDR Q&As," 11 August 2006. [Online]. Available: http://www.nhtsa.gov/DOT/NHTSA/Rulemaking/Rules/Associated%20Files/EDR_QAs_11Aug2006.pdf. [Accessed 11 February 2016].
- [21] D. Burrus, "The Internet of Things Is Far Bigger Than Anyone Realizes," 2015. [Online]. Available: www.wired.com/insights/2014/11/the-internet-of-things-bigger/. [Accessed 25 October 2015].
- [22] Wikipedia, "Automotive Design," 30 January 2016. [Online]. Available: https://en.wikipedia.org/wiki/Automotive_design. [Accessed 10 February 2016].
- [23] R. Electronics, "V850E2/FG4, 1st ed. Renesas Electronics," 2013. [Online].
- [24] G. Motors, "General Motors | GM Vehicles & onStar: A Powerful Combination | GM.com," 2015. [Online]. Available: http://www.gm.com/vision/design_technology/onstar_safe_connected.html. [Accessed 12 October 2015].
- [25] G. Pitcher, "Growing Number of ECUs Forces New Approach to Cars Electrical Architecture," NewElectronics, 2012.
- [26] T. Cain, "Nissan Altima Sales Figures - GoodCarBadCar," 1 January 2011. [Online]. Available: <http://www.goodcarbadcar.net/2011/01/nissan-altima-sales-figures.html>. [Accessed 1 December 2015].
- [27] H. Giannopoulos, "Private Communication," 2015.

UDS Based Attack Data Logger

- [28] A. Berezovsky, "GitHub," Volkswagen Research, September 2015. [Online]. Available: <https://github.com/linux-can/can-utils/>. [Accessed November 2015].
- [29] Quick2Wire, "I2C and SPI," 2016. [Online]. Available: <http://quick2wire.com/articles/i2c-and-spi/>. [Accessed 20 February 2016].
- [30] Arduino.cc, "ArduinoBoardUno," 2015. [Online]. Available: <https://www.arduino.cc/en/Main/ArduinoBoardUno>. [Accessed 5 October 2015].
- [31] B. Bindu, "Getting Started with Arduino," Student Companion SA, 18 December 2015. [Online]. Available: <http://www.studentcompanion.co.za/getting-started-with-arduino/>. [Accessed 2 February 2016].
- [32] Adafruit.com, "Raspberry Pi Model B+ 512MB RAM ID: 1914," 2015. [Online]. Available: <https://www.adafruit.com/products/1914>. [Accessed 5 October 2015].
- [33] S. Robillard, "Raspberry Pi," 31 December 2015. [Online]. Available: <http://raspberrypi.stackexchange.com/questions/40318/raspberry-pi-2-can-gpio-pins-29-40-be-used-gpio-gen-input-output-configurable-in>. [Accessed 2 February 2016].
- [34] Sparkfun, "Teensy 3.2 DEV-13736," [Online]. Available: <https://www.sparkfun.com/products/13736>. [Accessed 9 February 2016].
- [35] P. Burgess, "AdaFruit," 7 September 2015. [Online]. Available: <https://learn.adafruit.com/animated-electronic-eyes-using-teensy-3-1/wiring>. [Accessed 2 February 2016].
- [36] H. Giannopoulos, "GitHub," 20 November 2015. [Online]. Available: https://github.com/hngiannopoulos/FlexCAN_Library. [Accessed 1 December 2015].
- [37] Sparkfun, "Beaglebone Black - Dev 12857 - Rev C," [Online]. Available: <https://www.sparkfun.com/products/12857>. [Accessed 16 February 2016].
- [38] BeagleBoard, "BeagleBoard.org Digital Media," 2 December 2015. [Online]. Available: <http://beagleboard.org/media>. [Accessed 2 February 2016].
- [39] Atmel, "ATmega328P," [Online]. Available: <http://www.atmel.com/devices/atmega328p.aspx>. [Accessed 9 February 2016].
- [40] AdaFruit, "Raspberry Pi Datasheet Specs," [Online]. Available: <https://www.adafruit.com/datasheets/pi-specs.pdf>. [Accessed 9 February 2016].
- [41] ARM, "ARM1176," [Online]. Available: <https://www.arm.com/products/processors/classic/arm11/arm1176.php>. [Accessed 9 February 2016].
- [42] Texas Instruments, "AM3358," [Online]. Available: <https://www.ti.com/product/am3358>. [Accessed 9 February 2016].
- [43] Sparkfun.com, "CAN-BUS Shield-DEV-10039," 2015. [Online]. Available: <https://www.sparkfun.com/products/10039>. [Accessed 5 October 2015].

- [44] Skpang.co.uk, "PICAN CAN-BUS Board for Raspberry Pi [RSP-PICAN]," 2015. [Online]. Available: <http://skpang.co.uk/catalog/pican-canbus-board-for-raspberry-pi-p-1196.html>. [Accessed 5 October 2015].
- [45] Microchip, "MCP2561/2 High-Speed CAN Transceiver," [Online]. Available: <http://ww1.microchip.com/downloads/en/DeviceDoc/25167B.pdf>. [Accessed 9 February 2016].
- [46] M. Thompson, "Why does the CAN bus use a 120 Ohm resistor as the terminating resistor and not any other value," Electrical Engineering, 18 January 2013. [Online]. Available: <http://electronics.stackexchange.com/questions/55389/why-does-the-can-bus-use-a-120-ohm-resistor-as-the-terminating-resistor-and-not>. [Accessed 14 February 2016].
- [47] Texas Instruments, "SN65HVD230," [Online]. Available: <http://www.ti.com/product/SN65HVD230/description>. [Accessed 26 February 2016].
- [48] T. I. Steve Corrigan, "Controller Area Network Physical Layer Requirements," January 2008. [Online]. Available: <http://www.ti.com/lit/an/slla270/slla270.pdf>. [Accessed 25 February 2016].
- [49] T. Copper, "DS1307 Real Time Clock Breakout Board Kit," AdaFruit, 15 October 2015. [Online]. Available: <https://learn.adafruit.com/downloads/pdf/ds1307-real-time-clock-breakout-board-kit.pdf>. [Accessed 16 February 2016].
- [50] V. Kok, "Digital Clock with Arduino and DS1307," Electro Schematics, [Online]. Available: <http://www.electroschematics.com/8921/digital-clock-with-arduino-and-ds1307/>. [Accessed 16 February 2016].
- [51] M. Burris, "Selecting Between I2C and SPI," About Tech, [Online]. Available: <http://components.about.com/od/Theory/a/Selecting-Between-I2c-And-Spi.htm>. [Accessed 14 February 2016].
- [52] PJRC, "K20 Sub-Family Reference Manual," December 2012. [Online]. Available: <https://www.pjrc.com/teensy/K20P64M72SF1RM.pdf>. [Accessed 14 February 2016].
- [53] Microchip, "23A1024/23LC1024 1Mbit SPI Serial SRAM with SDI and SQI Interface," [Online]. Available: <https://www.pjrc.com/teensy/23LC1024.pdf>. [Accessed 14 February 2016].
- [54] PJRC, "Micro SD Card Adapter," [Online]. Available: http://www.pjrc.com/store/sd_adaptor.html. [Accessed 14 February 2016].
- [55] Arduino, "Arduino Forum," 12 October 2012. [Online]. Available: <http://forum.arduino.cc/index.php?topic=128335.0>. [Accessed 14 February 2016].
- [56] Yegorich, "StackOverflow," 6 July 2015. [Online]. Available: <http://stackoverflow.com/questions/21022749/how-to-create-virtual-can-port-on-linux-c>. [Accessed 1 December 2015].

UDS Based Attack Data Logger

- [57] Linux, "CAN Bus," 2 April 2015. [Online]. Available: http://elinux.org/CAN_Bus#SocketCAN_Tutorials. [Accessed 1 December 2015].
- [58] Armadeus Project, "CAN bus Linux driver," 18 March 2015. [Online]. Available: http://www.armadeus.com/wiki/index.php?title=CAN_bus_Linux_driver. [Accessed 1 December 2015].
- [59] A. Nunes, "StackOverflow," 7 October 2015. [Online]. Available: <http://stackoverflow.com/questions/31328302/canplayer-wont-replay-candump-files>. [Accessed 7 December 2015].
- [60] O. Hartkopp, "Python-CAN," 2013. [Online]. Available: http://python-can.readthedocs.org/en/latest/socketcan_native.html. [Accessed 1 December 2015].
- [61] Bluetooth.com, "Basics," 2015. [Online]. Available: www.bluetooth.com/Pages/Basics.aspx. [Accessed 5 October 2015].
- [62] Developer.bluetooth.org, "Baseband Architecture," 2015. [Online]. Available: <https://developer.bluetooth.org/TechnologyOverview/Pages/Baseband.aspx>. [Accessed 5 October 2015].
- [63] Instructables.com, "Car to Arduino Communication: CAN BUS Sniffing and Broadcasting with Arduino," 2015. [Online]. Available: <http://www.instructables.com/id/CAN-Bus-Sniffing-and-Broadcasting-with-Arduino/?ALLSTEPS>. [Accessed 7 October 2015].
- [64] Instructables.com, "Hack your vehicle CAN-BUS with Arduino and Seeed CAN-BUS Shield," 2015. [Online]. Available: <http://www.instructables.com/id/Hack-your-vehicle-CAN-BUS-with-Arduino-and-Seeed-C/?ALLSTEPS>. [Accessed 5 October 2015].
- [65] GitHub, "Seeed-Studio/CAN_BUS_Shield," 2015. [Online]. Available: https://github.com/Seeed-Studio/CAN_BUS_Shield. [Accessed 5 October 2015].
- [66] Driveaccord.net, "CAN-BUS- Drive Accord Honda Forums," 2015. [Online]. Available: <http://www.driveaccord.net/forums/15-7th-generation/54096-can-bus.html>. [Accessed 14 October 2015].
- [67] Goodcarbadcar.net, "Acura TL Sales Figures," 2015. [Online]. Available: <http://www.goodcarbadcar.net/2011/01/acura-tl-sales-figures.html>. [Accessed 14 October 2015].
- [68] C. V. a. C. Miller, "Adevntures in Automative Networks and Control Units," 2014. [Online]. Available: http://www.ioactive.com/pdfs/IOActive_Adventures_in_Automotive_Networks_and_Control_Units.pdf. [Accessed 5 October 2015].
- [69] C. Smith, "Car Hacker's Handbook," 2015. [Online]. Available: http://opengarages.org/handbook/2014_car_hackers_handbook_compressed.pdf. [Accessed 28 August 2015].

- [70] Wikipedia, "OBD-II PIDS," 2015. [Online]. Available: https://en.wikipedia.org/wiki/OBD-II_PIDS. [Accessed 20 August 2015].
- [71] C. Studios, "CANned Pi: (Part1)," 2014. [Online]. Available: <http://www.cowfishstudios.com/blog/canned-pi-part1>. [Accessed 20 September 2015].
- [72] T. Others, "PICAN CAN-BUS Board for Raspberry Pi," 2015. [Online]. Available: <http://skpang.co.uk/catalog/pican-canbus-board-for-raspberry-pi-p-1196.html>. [Accessed 10 September 2015].
- [73] Seeedstudio.com, "CAN-BUS Shield," 2014. [Online]. Available: http://www.seeedstudio.com/wiki/CAN-BUS_Shield. [Accessed 20 September 2015].
- [74] www3.epa.gov, "On-board Diagnostics (OBD)," 2015. [Online]. Available: <http://www3.epa.gov/obd/>. [Accessed 26 August 2015].
- [75] Skpang.googlecode.com, 2015. [Online]. Available: http://skpang.googlecode.com/files/Canbus_v4.zip. [Accessed 5 October 2015].
- [76] T. Others, "Arduino CAN_BUS Shield with SD Card Holder," 2015. [Online]. Available: <http://skpang.co.uk/catalog/arduino-canbus-shield-with-usd-card-holder-p-706.html>. [Accessed 5 October 2015].
- [77] Intel Corporation, "Automotive Security Best Practices," Santa Clara, 2015.
- [78] Wikipedia , "CAN Bus," 22 January 2016. [Online]. Available: https://en.wikipedia.org/wiki/CAN_bus. [Accessed 1 February 2016].

Glossary Definitions

CAN Bus

This is the current universal network standard within vehicles. This has been implemented in all production vehicles since 2008. The CAN Bus is a multicast system, meaning any node on the bus can send data packets. Other nodes choose whether or not they would like to filter packets monitored on the bus based on the packet's Arbitration ID.

Arbitration ID

This is the unique identifier for different types of packets. This allows each ECU to filter exactly which packets it wants to act upon within the CAN network. For instance, in a 2014 Nissan Altima, UDS packets have an Arbitration ID of 0x7E0 and other peripheral control packets could have Arbitration IDs such as 0x60D or 0x358.

OBD-II Port

This port is usually found under the steering wheel of the vehicle. It can be accessed by an OBD-II reader and used by service technicians to clear warning codes and download diagnostic information from the vehicle. The OBD-II port is directly tied into the CAN Bus meaning any device on the OBD-II port can monitor all packets on the CAN Bus.

Service ID

This is the unique identifier for the different types of UDS commands. These range from restarting to loading firmware onto ECUs. Some of these Service IDs can be especially harmful to the vehicle, even if they are sent in arbitrarily generated, meaningless packets. The Service ID is the first data byte within a UDS packet.

Types of CAN Errors:

Bit Error

A node that is sending data on the bus also monitors the bus. A Bit Error is detected when the value monitored differs from the value sent.

Stuff Error

This error is flagged when the 6th consecutive identical bit is found in a field where bit stuffing should occur.

CRC Error

This error is flagged by the receiver when the calculated CRC does not equal the CRC sent in the packet.

Form Error

This error is flagged when a fixed form field contains one or more illegal bits.

Acknowledgement Error

This error is flagged by a transmitter whenever it does not monitor a “dominant” bit in the ACK Slot.

Appendices

Appendix A: Bibliography Annotations

- 1) Wikipedia CAN
Has general information about the CAN Bus and its structure.
- 2) R.R Nayak Automotive Diagnostic Services
Explains general information about what UDS messages are and how they are sent on the CAN Bus. It also describes its applications such as: troubleshooting issues with the vehicle and installing firmware updates for the system. The article also mentions the replies received in the Bus when diagnosing messages are sent.
- 3) Carhistory4u.com, 'Car History 4U - History of Motor Car / Automobile Production 1900 - 2003', 2015. [Online]. Available: <http://www.carhistory4u.com/the-last-100-years/car-production>. [Accessed: 12- Oct- 2015].
This website gives data about the history of cars. More specifically, how many cars were in use in the 1900s compared to the early 2000's.
- 4) J. Turley, "Motoring with Microprocessor," 11 August 2003. [Online]. Available: <http://www.embedded.com/electronics-blogs/significant-bits/4024611/Motoring-with-microprocessors>. [Accessed 10 February 2016].
Includes facts about when microcontrollers were introduced to vehicles and how microcontrollers have become prevalent in vehicles.
- 5) CAN Specification, 1st ed. Bosch, 1991.
This is the 1991 CAN Bus specification released by Bosch. The specification explains functionalities of the CAN Bus including network communication methods, fault detection, error handling, and expandability. This was especially helpful when developing our plan of attack.
- 6) C. Timberg, "The definitive account of how hackers can gain access to our cars", Washington Post, 2015. [Online]. Available: <http://www.washingtonpost.com/sf/business/2015/07/22/hacks-on-the-highway/>. [Accessed: 12- Oct- 2015].
This newspaper article gives details about many of the recent car hacks. These hacks include the ones performed by Charlie Miller and Chris Valasek and also the hacks that were performed by students at the University of Washington and University of California San Diego.
- 7) A. Dharmawan, "Electric Cloud," 8 December 2014. [Online]. Available: <http://electric-cloud.com/blog/2014/12/continuous-delivery-puts-automotive-software-high-gear/>. [Accessed 3 January 2016].
Explains what an ECU is and has an image that shows some of the systems in the car that use microcontrollers (ECUs) in the cars.

8) B. World, "127 Years of Modern Automobile Evolution," 21 October 2013. [Online]. Available: <http://blog.world-mysteries.com/science/127-years-of-modern-automobile-evolution/>. [Accessed 6 January 2016].

Explains the vulnerabilities caused by an increasing number of microcontrollers in the vehicles. It describes possible exploitations that hackers can have.

9) C. Poladian, 'Car Hacking White Paper Released By DARPA-Funded Researchers, Shows How To Hack A Ford Escape And Toyota Prius', International Business Times, 2013. [Online]. Available: [9]. [Accessed: 05- Oct- 2015].

This article describes the steps that Miller and Valasek used in order complete their hacks. It briefly describes how they were able to hack into both the Ford Escape and the Toyota Prius.

10) A. Drozhzhin, 'Black Hat USA 2015: The full story of how that Jeep was hacked', Kaspersky Lab. The Power to Protect | Official Blog, 2015. [Online]. Available: [10] [24]. [Accessed: 05- Oct- 2015].

This article is a summary of the presentation that Miller and Valasek gave at the Black Hat convention. In this presentation, they described the steps they took in order to gain access to the Jeep Cherokee. This article also has pictures which show exactly what they did at each step.

11) C. M. a. C. Valasek, Remote Exploitation of an Unaltered Passenger Vehicle 1st ed., 2015, pp. 3-88.

This is the actual published paper by Miller and Valasek about the technical steps they took in order to hack the Jeep Cherokee.

12) L. Stahl, "DARPA: Nobody's safe on the internet," 2015. [Online]. Available: <http://www.cbsnews.com/news/darpa-dan-kaufman-internet-security-60-minutes/>. [Accessed 5 October 2015].

This links to the research done by University of Washington and University of California San Diego. Students in those schools were able to hack into the vehicle remotely and control all of its components. They hacked into the vehicle's emergency communication system (telematics unit).

13)M. Harris, "Researchers Hacks Self-driving CAR Sensors," 2015. [Online]. Available: http://spectrum.ieee.org/cars-that-think/transportation/self-driving/researcher-hacks-selfdriving-car-sensors?utm_campaign=Weekly%20Notification-%20IEEE%20Spectrum%20Tech%20Alert&utm_source=boomtrain&utm_medium=email&utm_content=&bt_alias=eyJ1c2VySWQiOiI3YTc0Mzk0NC1mODFjLTQzMzItYTJiMC04ZDd-IODI0YzIxNjIifQ%3D%3D. [Accessed 5 October 2015].

This article describes how Jonathan Petit, Principal Scientist at Security, was able to hack into an autonomous car and confuse the system which is used to detect objects such as walls and other cars. He was able to gain access and then trick the system into thinking there were cars and/or walls around the car in order to force it to stop suddenly.

UDS Based Attack Data Logger

14) D. Gershgorn, 'Hackers Can Trick Driverless Cars With A Handheld Laser', Popular Science, 2015. [Online]. Available: [14]. [Accessed: 12- Oct- 2015].

This article explains what Jonathan Petit did in order to accomplish his hack of the LiDAR system. It describes how much money his hardware setup cost and how he essentially replicated echoes from objects such as cars and tricked the LiDAR system into believing there was actually other cars around.

15) K. Zetter, 'Researchers Hacked a Model S, But Tesla's Already On It', WIRED, 2015. [Online]. Available: [15]. [Accessed: 05- Oct- 2015].

This article is about how Kevin Mahaffey and Marc Rogers were able to hack into a Tesla by plugging a laptop into the Ethernet port into the driver's side dashboard. They were able to start the car using a software command and also leave a Trojan horse on the system which would allow them remote access later on.

16) BBC, "Fiat Chrysler Recalls 1.4 Million Cars after Jeep Hack," p. 1, 24 July 2015.

Shows the 1 million vehicle recall done by Chrysler to fix a security flaw exposed by C. Miller and Valasek.

17) CAN with Flexible Data-Rate, 1st ed. Bosch, 2012.

This is the 2012 CAN Bus flexible data-rate specification. The specification explains functionalities of the CAN Bus flexible data-rate implementation, including network communication methods, fault detection, error handling, and expandability. This was especially helpful when developing our plan of attack.

18) ISO, "ISO 14229-1:2013," 15 March 2013. [Online]. Available:

http://www.iso.org/iso/catalogue_detail.htm?csnumber=55283. [Accessed 1 December 2015].

This is the ISO spec that specifies the CAN Standard and the UDS standard. There are multiple parts to this guide that specify different parts of the system.

19) D. K. Nilsson and U. E. Larson, "Conducting Forensic Investigations of Cyber Attacks on Automobile In-Vehicle Networks," Chalmers University of Technology, Goteborg, 2008.

Mentions the article about security and attack countermeasures written by Dennis Nilsson and Ulf Larson. These articles explain the different 5 layers of protection: Prevention, Detection, Deflection, Countermeasures, and Recovery.

20) U. E. Larson and D. K. Nilsson, "Securing Vehicles against Cyber Attacks," Chalmers University of Technology, Goteborg, 2008.

Mentions the article about security and attack countermeasures written by Dennis Nilsson and Ulf Larson. These articles explain the different 5 layers of protection: Prevention, Detection, Deflection, Countermeasures, and Recovery.

21) National Highway Traffic Safety Administration, "EDR Q&As," 11 August 2006.

[Online]. Available: http://www.nhtsa.gov/DOT/NHTSA/Rulemaking/Rules/Associated%20Files/EDR_QAs_11Aug2006.pdf. [Accessed 11 February 2016].

Explains details about the Event Data Recorder developed by the National Highway Traffic Safety Administration in their Q&A done in August 11th, 2006.

22) D. Burrus, 'The Internet of Things Is Far Bigger Than Anyone Realizes', WIRED, 2015. [Online]. Available: <http://www.wired.com/insights/2014/11/the-internet-of-things-bigger/>. [Accessed: 12- Oct- 2015].

This article gives a brief description of what the Internet of Things is and how it has been expanding in recent years. This article was written in WIRED magazine.

23) Wikipedia, "Automotive Design," 30 January 2016. [Online]. Available: https://en.wikipedia.org/wiki/Automotive_design. [Accessed 10 February 2016].

Talks about the history of vehicle hacking, and explains the length of the vehicle manufacturing cycle and why hackers can exploit cars more quickly than manufacturers can fix the issues.

24) R. Electronics, "V850E2/FG4, 1st ed. Renesas Electronics," 2013. [Online].

Explains what the V850 controller in the Jeeps is and gives perspective on how the hack performed by Miller and Valasek was done.

25) G. Motors, "General Motors | GM Vehicles & onStar: A Powerful Combination | GM.com," 2015. [Online]. Available: http://www.gm.com/vision/design_technology/onstar_safe_connected.html. [Accessed 12 October 2015].

Article describing the creation of OnStar system, how it has evolved over time, and the services that it provides.

26) G. Pitcher, "Growing Number of ECUs Forces New Approach to Cars Electrical Architecture," NewElectronics, 2012.

Provides details about the number of ECUs in the Nissan Altima 2014, and how the number of ECUs has increased over time.

27) T. Cain, "Nissan Altima Sales Figures - GoodCarBadCar," 1 January 2011. [Online]. Available: <http://www.goodcarbadcar.net/2011/01/nissan-altima-sales-figures.html>. [Accessed 1 December 2015].

This link includes information about the sale figures of the Nissan Altima, showing why it is a relevant car model for this project since it is decently represented in the car market.

28) Quick2Wire, "I2C and SPI," 2016. [Online]. Available: <http://quick2wire.com/articles/i2c-and-spi/>. [Accessed 20 February 2016].

Explains differences between I2C and SPI and compares them. Published by Quick2Wire.com

29) Arduino.cc, "ArduinoBoardUno," 2015. [Online]. Available: <https://www.arduino.cc/en/Main/ArduinoBoardUno>. [Accessed 5 October 2015].

UDS Based Attack Data Logger

Contains specs of the Arduino Uno board.

30) B. Bindu, "Getting Started with Arduino," Student Companion SA, 18 December 2015. [Online]. Available: <http://www.studentcompanion.co.za/getting-started-with-arduino/>. [Accessed 2 February 2016].

Contains diagrams of the Arduino UNO and specs about it. Used particularly for the image of the board which indicates what the ports and features are.

31) Adafruit.com, "Raspberry Pi Model B+ 512MB RAM ID: 1914," 2015. [Online]. Available: <https://www.adafruit.com/products/1914>. [Accessed 5 October 2015].

Include Raspberry Pi Model B+ specs in more details.

32) S. Robillard, "Raspberry Pi," 31 December 2015. [Online]. Available: <http://raspberrypi.stackexchange.com/questions/40318/raspberry-pi-2-can-gpio-pins-29-40-be-used-gpio-gen-input-output-configurable-in>. [Accessed 2 February 2016].

Includes more information about the pins of the Raspberry Pi and its possible configurations.

33) Sparkfun, "Teensy 3.2 DEV-13736," [Online]. Available: <https://www.sparkfun.com/products/13736>. [Accessed 9 February 2016].

Contains the specifications of the Teensy 3.2. It specifies its pins and its ports.

34) P. Burgess, "AdaFruit," 7 September 2015. [Online]. Available: <https://learn.adafruit.com/animated-electronic-eyes-using-teensy-3-1/wiring>. [Accessed 2 February 2016].

Contains diagrams for the Teensy 3.2. It includes an image of the board with its pins and ports.

35) H. Giannopoulos, "GitHub," 20 November 2015. [Online]. Available: https://github.com/hngiannopoulos/FlexCAN_Library. [Accessed 1 December 2015].

Contains H. Giannopoulos CAN library that is an alternative to the official CAN library provided in the Arduino/Teensy website.

36) Sparkfun, "Beaglebone Black - Dev 12857 - Rev C," [Online]. Available: <https://www.sparkfun.com/products/12857>. [Accessed 16 February 2016].

Contains the specifications for the BeagleBone Black board. Obtained from the sparkfun website.

37) BeagleBoard, "BeagleBoard.org Digital Media," 2 December 2015. [Online]. Available: <http://beagleboard.org/media>. [Accessed 2 February 2016].

Explains information about the operating system of the BeagleBone Black, and presents an schematic of the board.

38) Atmel, "ATmega328P," [Online]. Available: <http://www.atmel.com/devices/atmega328p.aspx>. [Accessed 9 February 2016].

Contains information of the ATmega238P processor used in the Arduino Uno

38) Atmel, "ATmega328P," [Online]. Available: <http://www.atmel.com/devices/atmega328p.aspx>. [Accessed 9 February 2016].

This article describes the CPU of the Arduino Uno board. The CPU in the Arduino Uno is 20MHz ATmega328P processor.

39) AdaFruit, "Raspberry Pi Datasheet Specs," [Online]. Available: <https://www.adafruit.com/datasheets/pi-specs.pdf>. [Accessed 9 February 2016].

This article describes the specs of the Raspberry Pi B+ model. Some specs include operating voltage, flash memory, ram and eeprom.

40) ARM, "ARM1176," [Online]. Available: <https://www.arm.com/products/processors/classic/arm11/arm1176.php>. [Accessed 9 February 2016].

This article is the datasheet for the CPU of the Raspberry Pi. The CPU is a low power 700MHz ARM1176JZFS ARM processor

41) Texas Instruments, "AM3358," [Online]. Available: <https://www.ti.com/product/am3358>. [Accessed 9 February 2016].

This is a datasheet for the Beaglebone Black processor. The Beaglebone Black has a AM3358 1GHz ARM Cortex-A8 processor

42) Sparkfun.com, "CAN-BUS Shield-DEV-10039," 2015. [Online]. Available: <https://www.sparkfun.com/products/10039>. [Accessed 5 October 2015].

This article is about the Arduino CAN Bus Shield. This source was used to obtain an image of the Arduino CAN Bus shield.

43) Skpang.co.uk, "PICAN CAN-BUS Board for Raspberry Pi [RSP-PICAN]," 2015. [Online]. Available: <http://skpang.co.uk/catalog/pican-canbus-board-for-raspberry-pi-p-1196.html>. [Accessed 5 October 2015].

This article is about the Raspberry Pi PICAN Module. This source was used to obtain an image of the Raspberry Pi PICAN module.

44) Microchip, "MCP2561/2 High-Speed CAN Transceiver," [Online]. Available: <http://ww1.microchip.com/downloads/en/DeviceDoc/25167B.pdf>. [Accessed 9 February 2016].

This article summarizes the MCP2561 high speed CAN transceiver chip. This chip is used to interact with a CAN network through a microcontroller. It has a data transfer rate of 1 Mbps. This source was also used to obtain an image of the MCP2651 Chip and spec sheet.

45) M. Thompson, "Why does the CAN bus use a 120 Ohm resistor as the terminating resistor and not any other value," Electrical Engineering, 18 January 2013. [Online]. Available:

UDS Based Attack Data Logger

<http://electronics.stackexchange.com/questions/55389/why-does-the-can-bus-use-a-120-ohm-resistor-as-the-terminating-resistor-and-not>. [Accessed 14 February 2016].

This article discusses why a 120 Ohm resistor is needed between the CANH and CANL buses on the CAN transceiver chip. A resistor of 120 Ohms is added to across the bus to avoid reflection by matching the characteristic line impedance of the CAN Bus.

46) Texas Instruments, "SN65HVD230," [Online]. Available: <http://www.ti.com/product/SN65HVD230/description>. [Accessed 26 February 2016].

This article summarizes the SN65HVD230 CAN board. The CAN board is used to interact with the CAN network. This board consists of a SN65HVD230 CAN chip, a 120 Ohm resistor and a 10K Ohm resistor. Both of these resistors are added to the chip to avoid reflecting. The 120 Ohm resistor is added across the CANH and CANL to match the characteristic line impedance of the CAN Bus. The 10K Ohm resistor is added between the Rs pin and ground to put the chip in slope control mode. Slope control mode also helps avoids reflection.

47) T. I. Steve Corrigan, "Controller Area Network Physical Layer Requirements," January 2008. [Online]. Available: <http://www.ti.com/lit/an/slla270/slla270.pdf>. [Accessed 25 February 2016].

This article talks about the physical layer requirements of the CAN. This source is used to understand how to minimize reflections. In order to minimize reflections, the stub-line length should not exceed one third of the lines critical length.

48) T. Copper, "DS1307 Real Time Clock Breakout Board Kit," AdaFruit, 15 October 2015. [Online]. Available: <https://learn.adafruit.com/downloads/pdf/ds1307-real-time-clock-breakout-board-kit.pdf>. [Accessed 16 February 2016].

This article is used to describe the the RTC timing module used in the Data Logger device. This module is a clock that keeps internal time for the Teensy 3.2 microcontroller. The timing module requires a 3V battery in order to make sure that the clock never runs out of power.

49) V. Kok, "Digital Clock with Arduino and DS1307," Electro Schematics, [Online]. Available: <http://www.electroschematics.com/8921/digital-clock-with-arduino-and-ds1307/>. [Accessed 16 February 2016].

This article describes the micro chip that is used for the RTC timing module. This micro chip is the DS1307 chip. This source was used to get an image of the RTC timing module and DS1307 chip.

50) M. Burris, "Selecting Between I2C and SPI," About Tech, [Online]. Available: <http://components.about.com/od/Theory/a/Selecting-Between-I2c-And-Spi.htm>. [Accessed 14 February 2016].

This article talks about the peripheral buses and why some are better than others. For our data logger, SPI is favorable over I2C since SPI has higher transfer rate and draws less power.

- 51) PJRC, "K20 Sub-Family Reference Manual," December 2012. [Online]. Available: <https://www.pjrc.com/teensy/K20P64M72SF1RM.pdf>. [Accessed 14 February 2016].
 This is the K20 sub-family reference manual. The Teensy 3.2 has EEPROM memory that is non-volatile and is used to store small amounts of data that remain saved across power cycles.
- 52) Microchip, "23A1024/23LC1024 1Mbit SPI Serial SRAM with SDI and SQI Interface," [Online]. Available: <https://www.pjrc.com/teensy/23LC1024.pdf>. [Accessed 14 February 2016].
 This article talks about flash memory. Flash memory is nonvolatile and before data can be stored on a flash memory block, the data on that block must be erased. This source was used to create a chart of the Microchip SPI Flash module specs for the Teensy 3.2 microcontroller.
- 53) PJRC, "Micro SD Card Adapter," [Online]. Available: http://www.pjrc.com/store/sd_adapter.html. [Accessed 14 February 2016].
 This article is about a micro sd card adapter. This adapter is used with the Teensy 3.2 microcontroller to store external data. This method of storage contains the largest storage space available but has the slowest read and write speeds.
- 54) Arduino, "Arduino Forum," 12 October 2012. [Online]. Available: <http://forum.arduino.cc/index.php?topic=128335.0>. [Accessed 14 February 2016].
 This article talks about read and write speeds for the micro sd card adapter. This information is used to create a table of specs which includes file size, buffer size, speed and latency.
- 55) Developer.bluetooth.org, "Baseband Architecture," 2015. [Online]. Available: <https://developer.bluetooth.org/TechnologyOverview/Pages/Baseband.aspx>. . [Accessed 5 October 2015].
 This article describes how packets are sent over Bluetooth. It shows pictures of the packet frames as well as the different channels that are associated with Bluetooth technology.
- 56) Instructables.com, "Car to Arduino Communication: CAN BUS Sniffing and Broadcasting with Arduino," 2015. [Online]. Available: <http://www.instructables.com/id/CAN-Bus-Sniffing-and-Broadcasting-with-Arduino/?ALLSTEPS>. [Accessed 7 October 2015].
 This article describes how to connect an Arduino and CAN Bus Shield to the OBD-II port of a vehicle in order to transmit and receive packets. The article features some useful code for reading and writing CAN Bus packets. This will be very useful when writing the code to sniff CAN Bus traffic.

UDS Based Attack Data Logger

57) Instructables.com, "Hack your vehicle CAN-BUS with Ardiuno and Seeed CAN-BUS Shield," 2015. [Online]. Available: <http://www.instructables.com/id/Hack-your-vehicle-CAN-BUS-with-Arduino-and-Seeed-C/?ALLSTEPS>. [Accessed 5 October 2015].

This article presents how to use the CAN library for the seeed shield works. It explains how to read, write to the can bus.

58) GitHub, "Seeed-Studio/CAN_BUS_Shield," 2015. [Online]. Available: https://github.com/Seeed-Studio/CAN_BUS_Shield. [Accessed 5 October 2015].

Open source tools for the Seeed shield. This link has the code repository that includes all of the functions.

59) Driveaccord.net, "CAN-BUS- Drive Accord Honda Forums," 2015. [Online]. Available: <http://www.driveaccord.net/forums/15-7th-generation/54096-can-bus.html>. [Accessed 14 October 2015].

This is a blog where other curious Acura/Honda owners were attempting to discover information about the vehicle's CAN Bus. No one the blog could provide any useful data.

60) Goodcarbadcar.net, "Acura TL Sales Figures," 2015. [Online]. Available: <http://www.goodcarbadcar.net/2011/01/acura-tl-sales-figures.html>. [Accessed 14 October 2015].

This website provided us with the data about the number of Acura 3.2TLs were sold in the United States.

61) C. V. a. C. Miller, "Adevntures in Automative Networks and Control Units," 2014. [Online]. Available: http://www.ioactive.com/pdfs/IOActive_Adventures_in_Automotive_Networks_and_Control_Units.pdf. [Accessed 5 October 2015].

Report on their hack and their hacking tools. It explains in detail how they hacked the cars. Shows screenshots of the messages that they sent to the can and the effect they had on the vehicle's system. It explains a series of tools they developed for it. A link to download the tools is presented, but they mostly work on Python.

62) C. Smith, "Car Hacker's Handbook," 2015. [Online]. Available: http://opengarages.org/handbook/2014_car_hackers_handbook_compressed.pdf. [Accessed 28 August 2015].

This handbook was used as a basis for our hardwire hack ideas. The handbook discusses connecting to the OBD-II port and how to interact with the diagnostics system. CAN Bus packet format is also discussed in detail, especially how arbitration works in terms of CAN packets.

63) Wikipedia, "OBD-II PIDS," 2015. [Online]. Available: https://en.wikipedia.org/wiki/OBD-II_PIDs. [Accessed 20 August 2015].

This article outlines the common Parameter IDs (PIDs) for requesting device specific data over the OBD-II port. This will prove extremely helpful when testing communication with the CAN Bus over OBD-II.

64) C. Studios, "CANned Pi: (Part1)," 2014. [Online]. Available: <http://www.cowfishstudios.com/blog/canned-pi-part1>. [Accessed 20 September 2015].

This article outlines key steps in interacting with the OBD-II port using a Raspberry Pi board and CAN Bus capable PICAN module. The article helped outline a plan of attack using a Raspberry Pi board and assisted in our decision in what microcontroller to use for the OBD-II based attack.

65) T. Others, "PICAN CAN-BUS Board for Raspberry Pi," 2015. [Online]. Available: <http://skpang.co.uk/catalog/pican-canbus-board-for-raspberry-pi-p-1196.html>. [Accessed 10 September 2015].

SK Pang develops microcontrollers and option boards that were considered for our OBD-II based attack. They sell a wide array of useful products and have many tutorials that were found to be useful when developing vehicle attack strategies. The company is based out of the UK, so their products were not considered for use in this project.

66) Seeedstudio.com, "CAN-BUS Shield," 2014. [Online]. Available: http://www.seeedstudio.com/wiki/CAN-BUS_Shield. [Accessed 20 September 2015].

This article describes how to interface with an Arduino CAN Bus option board built by Digital Inc. Although we did not consider this particular option board for our OBD-II based attack, the information and sample code in this article were very useful and will prove even more useful when it comes time to build attack code.

67) www3.epa.gov, "On-board Diagnostics (OBD)," 2015. [Online]. Available: <http://www3.epa.gov/obd/>. [Accessed 26 August 2015].

This article outlines the uses of the OBD-II port, particularly how it helps repair technicians maintain and inspect vehicles. The information in this article served as a starting place for building our OBD-II based attack strategy.

68) skpang.googlecode.com, 2015. [Online]. Available: http://skpang.googlecode.com/files/Canbus_v4.zip. [Accessed 5 October 2015].

This is the zip file with the code for the spark fun shield. This code allows the user to communicate with the CAN BUS.

69) T. Others, "Arduino CAN_BUS Shield with SD Card Holder," 2015. [Online]. Available: <http://skpang.co.uk/catalog/arduino-canbus-shield-with-usd-card-holder-p-706.html>. [Accessed 5 October 2015].

Examples on how to use the sparkfun shield code to read and write from the can bus.

UDS Based Attack Data Logger

70) Yegorich, "StackOverflow," 6 July 2015. [Online]. Available: <http://stackoverflow.com/questions/21022749/how-to-create-virtual-can-port-on-linux-c>. [Accessed 1 December 2015].

71) Linux, "CAN Bus," 2 April 2015. [Online]. Available: http://linux.org/CAN_Bus#SocketCAN_Tutorials. [Accessed 1 December 2015].

This article shows a list of linux commands. These commands are used to set up the CAN network. In order to send files on the network you can use the tools cansend command.

72) ArmadeuS Project, "CAN bus Linux driver," 18 March 2015. [Online]. Available: http://www.armadeus.com/wiki/index.php?title=CAN_bus_Linux_driver. [Accessed 1 December 2015].

This article shows a list of linux commands. These commands are used to set up the CAN network.

73) A. Nunes, "StackOverflow," 7 October 2015. [Online]. Available: <http://stackoverflow.com/questions/31328302/canplayer-wont-replay-candump-files>. [Accessed 7 December 2015].

This article shows a list of linux commands. These commands are used to set up the CAN network.

74) O. Hartkopp, "Python-CAN," 2013. [Online]. Available: http://python-can.readthedocs.org/en/latest/socketcan_native.html. [Accessed 1 December 2015].

This article is about the Pip linux client. In order to use the Socketcan library, one has to include the Python module by using the standard import statement.

75) Bluetooth.com, "Basics," 2015. [Online]. Available: www.bluetooth.com/Pages/Basics.aspx. [Accessed 5 October 2015].

77) A. Berezovskyi, "GitHub," Volkswagen Research, September 2015. [Online]. Available: <https://github.com/linux-can/can-utils/>. [Accessed November 2015].

This GitHub includes the Socketcan tools used in order to simulate a virtual CAN Bus on a Linux machine

Appendix B: Zip File Inventory

Beaglebone_Black_Setup - Test Bench setup files and script to be used with the Beaglebone Black tutorial described in the MQP Report Appendix

Data_Logger_Prototype_Code - Libraries and code used in the prototype Data Logger (options for both C and C++ CAN libraries)

Experimental_Results - Contains test data read from the 2014 Nissan Altima, tests conducted to optimize the linear buffer, and final Data Logger prototype results

Appendix C: Normal Packet Transfer (no UDS) Timing Data by Arbitration ID (AID)

| AID | Max Interval (ms) | Min Interval (ms) | Average Interval (ms) | Data Count |
|------------|--------------------------|--------------------------|------------------------------|-------------------|
| 002 | 11 | 9 | 10 | 1647 |
| 160 | 11 | 10 | 10 | 1609 |
| 174 | 11 | 9 | 10 | 1647 |
| 176 | 12 | 8 | 10 | 1647 |
| 177 | 12 | 7 | 10 | 1647 |
| 180 | 11 | 9 | 10 | 1609 |
| 182 | 11 | 9 | 10 | 1609 |
| 215 | 21 | 18 | 20 | 823 |
| 216 | 21 | 18 | 20 | 823 |
| 245 | 21 | 19 | 20 | 823 |
| 280 | 21 | 19 | 20 | 824 |
| 284 | 21 | 19 | 20 | 824 |
| 285 | 21 | 18 | 20 | 824 |
| 292 | 21 | 18 | 20 | 823 |
| 300 | 22 | 18 | 20 | 823 |
| 351 | 102 | 98 | 100 | 165 |
| 354 | 42 | 38 | 40 | 411 |
| 355 | 41 | 39 | 40 | 412 |
| 358 | 101 | 99 | 100 | 165 |
| 385 | 101 | 98 | 100 | 164 |
| 421 | 62 | 58 | 60 | 275 |
| 551 | 104 | 100 | 102 | 161 |
| 560 | 102 | 98 | 100 | 165 |
| 580 | 104 | 101 | 102 | 161 |

UDS Based Attack Data Logger

| | | | | |
|-----|-----|-----|-----|------|
| 6E2 | 105 | 100 | 102 | 161 |
| 625 | 103 | 97 | 100 | 164 |
| 5E4 | 102 | 98 | 100 | 164 |
| 02A | 101 | 99 | 100 | 165 |
| 1F9 | 11 | 9 | 10 | 1610 |
| 2DE | 13 | 7 | 10 | 1647 |
| 35D | 102 | 98 | 100 | 165 |
| 3EC | 23 | 17 | 20 | 823 |
| 54C | 101 | 99 | 100 | 164 |
| 5C5 | 102 | 99 | 100 | 165 |
| 60D | 102 | 97 | 100 | 165 |

Appendix D: Packet Transfer (with UDS) Timing Data by Arbitration ID (AID)

| AID | Max Inter- val (ms) | Min Inter- val (ms) | Avg Inter- val (ms) | Data Count |
|------------|------------------------------------|------------------------------------|--------------------------------|-----------------------|
| 002 | 500 | 9 | 10 | 30993 |
| 02A | 600 | 99 | 100 | 3099 |
| 160 | 501 | 9 | 10 | 30269 |
| 174 | 500 | 9 | 10 | 30987 |
| 176 | 501 | 7 | 10 | 30987 |
| 177 | 500 | 7 | 10 | 30987 |
| 180 | 502 | 9 | 10 | 30269 |
| 182 | 501 | 9 | 10 | 30269 |
| 1F9 | 501 | 9 | 10 | 30269 |
| 215 | 500 | 15 | 20 | 15635 |
| 216 | 500 | 4 | 20 | 15639 |
| 245 | 519 | 17 | 20 | 15497 |
| 280 | 500 | 17 | 20 | 15646 |
| 284 | 499 | 18 | 20 | 15499 |
| 285 | 500 | 18 | 20 | 15499 |
| 292 | 519 | 17 | 20 | 15497 |
| 2DE | 489 | 7 | 10 | 31292 |
| 300 | 500 | 17 | 20 | 15498 |
| 342 | 329 | 329 | 329 | 2 |
| 351 | 599 | 20 | 100 | 3137 |
| 354 | 519 | 37 | 40 | 7750 |
| 355 | 520 | 37 | 40 | 7823 |
| 358 | 599 | 20 | 100 | 3137 |

UDS Based Attack Data Logger

| | | | | |
|-----|------|-----|-----|-------|
| 35D | 600 | 10 | 99 | 3155 |
| 385 | 500 | 10 | 100 | 3135 |
| 3EC | 500 | 17 | 20 | 15494 |
| 421 | 540 | 10 | 60 | 5178 |
| 512 | 276 | 52 | 164 | 3 |
| 54C | 501 | 98 | 100 | 3100 |
| 551 | 614 | 100 | 103 | 3027 |
| 560 | 601 | 10 | 100 | 3103 |
| 580 | 614 | 100 | 103 | 3026 |
| 5C5 | 600 | 97 | 100 | 3130 |
| 5E4 | 500 | 97 | 100 | 3101 |
| 60D | 578 | 19 | 98 | 3204 |
| 625 | 599 | 6 | 100 | 3141 |
| 6E2 | 614 | 100 | 103 | 3026 |
| 71D | 71 | 49 | 51 | 23 |
| 71F | 530 | 471 | 501 | 3 |
| 72D | 70 | 49 | 51 | 23 |
| 7DF | 565 | 49 | 75 | 4120 |
| 7E8 | 1911 | 49 | 75 | 4100 |
| 7E9 | 2330 | 58 | 127 | 2428 |

Appendix E: Linux and Python Commands

The Virtual CAN networks are simulated within the Linux machine and use descriptors that allow virtual devices through the terminal to send and receive messages on them. Virtual networks simulate real networks and are useful for testing hardware or running simulations. The physical CAN networks allow Linux machines to communicate with physical devices through GPIO pins. *Vcan* stands for Virtual CAN, and *type can* should be used to initialize physical networks. Setting up the CAN networks is done by using the following commands [56] [57] [58] [59].

```
$sudo modprobe vcan
$sudo ip link add dev vcan0 type vcan
$sudo ip link set up vcan0
```

For physical devices you also have to use an additional command to indicate the bitrate in bits. For example, this would set the bitrate to 1MB:

```
$sudo ip link set can0 up type can bitrate 1000000
```

In order to send files on the network you can use the tools *cansend* command. Shell scripts can be used to send series of messages as well. Another useful command is the *canplayer* command that allows you to playback previously recorded commands on the network. Finally, the *candump* command prints all of the messages that are being sent on the specified network to the terminal. Some example commands are listed below

```
$candump vcanX
$canplayer -I file.txt
```

The *canplayer* tool has multiple options. The *-I* option indicates the input file. There is also a *-l* option to indicate how many times to play that file. The messages are played back on the network at the same speed there were recorded. Log files use the following format

```
(TIMESTAMP) <CAN_NAME> <ARBITRATION_ID#PACKET_PAYLOAD>
```

The following example uses the *cansend* command to send a message on the Virtual CAN network number 0. In this example, 123 is the arbitration ID in hexadecimal, and DEADBEEF is payload of the message in hexadecimal as well.


```
$cansend vcan0 123#DEADBEEF
```

. The Pip Linux client has to be installed to facilitate the installation of the Python Linux tools. In order to use the *Socketcan* library, one has to include the Python module by using the standard import statement [60]. This Python tool allows the program to receive CAN bus messages sent in CAN networks (virtual or physical). The Python API is used in the following way:

```
#initializes the CAN bus interface  
bus = can.interface.Bus('vcan0', bustype='socketcan_native')  
#retrieves message from the CAN into a message object  
message = bus.recv()
```

Appendix F: BeagleBone Black CAN Software Setup

This section covers the setup required to install and configure the SocketCAN Linux tools on a BeagleBone Black microcontroller. The following steps were completed in order to configure the device according to the constraints required of a CAN test bench.

Initial Device Setup

The initial device setup can be found at Beagleboard.org. This part of the setup includes configuring communications with the BeagleBone Black over USB and getting familiar with the features of the board.

Now that the initial configuration is complete, you can use an SSH client to connect to the board with the following parameters:

IP address: 192.168.7.2

Username: root

Password: No password until you have set it (this is not required)

WinSCP can also be used with the same parameters to move files between a PC and the BeagleBone Black.

Setup BeagleBone Black internet connection link over USB

In order to download Linux tools, the BeagleBone Black must be connected to the internet. Since the board does not have built in wireless hardware, it will have to share the resources of the computer it is connected to via USB. Using the link below, the internet connection on a windows machine can be setup to be shared with the BeagleBone Black.

<http://ofitselfso.com/BeagleNotes/HowToConnectBeagleBoneBlackToTheInternetViaUSB.php>

Install and Configure SocketCAN Utilities

Note: You must be logged in as root for this to work correctly.

Step 1: Device tree overlay to set pin multiplexing

This is done to setup the GPIO pins on the BeagleBone Black to receive and send CAN data on two pins (Tx and Rx lines).

<http://www.embedded-things.com/bbb/enable-canbus-on-the-BeagleBone-black/>

The device tree overlay file (found in the zipped appendix) must be copied to the BeagleBone Black. Make sure it is saved as “BB-DCAN1-00A0.dts”, the .dts refers to device tree source file.

Now compile this overlay, creating an overlay binary, “BB-DCAN1-00A0.dtbo”

```
$dtc -O dtb -o BB-DCAN1-00A0.dtbo -b 0 -@ BB-DCAN1-00A0.dts
```

To use the overlay, copy it to /lib/firmware:

```
$sudo cp BB-DCAN1-00A0.dtbo /lib/firmware
```

And execute the following to add the CAN hardware to the selection of available pinout pairs.

```
$echo BB-DCAN1 > /sys/devices/bone_capemgr.*/slots
```

Now all the correct pin outs are set for creating a physical CAN network. The next step is to install all the CAN Linux tools.

First, install GCC tools:

```
$sudo apt-get install git build-essential gcc make autoconf  
libtool
```

Next, install CAN utilities:

```
$mkdir can-dev && cd can-dev  
$git clone https://github.com/linux-can/can-utils.git  
$cd can-utils  
$./autogen.sh  
$./configure
```

UDS Based Attack Data Logger

```
$make  
$sudo make install
```

The following commands will help in setting up and turning off physical can networks.

Setup can tools:

```
$sudo modprobe can  
$sudo modprobe can-dev  
$sudo modprobe can-raw
```

Turn on can0:

```
$ifconfig can0 txqueuelen 10000  
$sudo ip link set can0 up type can bitrate 500000 loopback off  
listen-only off
```

Turn off can0:

```
$sudo ifconfig can0 down
```

Get can0 status:

```
$ip -details link show can0
```

Now, you can use all the tools available with the Linux SocketCAN Library.