Remote Virtual Access to a Vehicle Using Ford's OpenXC and Exploring Related Vehicle Security Vulnerabilities

Kyle Kelly

ENEE699: Independent Study, Fall 2016

The University of Maryland

Supervisor: Professor Gang Qu

Assistant Supervisor: Donald Woodbury

Assistant Supervisor: Professor Adam Porter

Abstract

A Controller Area Network (CAN bus) is the protocol responsible for the communication between various embedded systems within an automobile. Recently, the CAN bus has become a conduit for cyber-physical attacks that can potentially put a vehicle's passengers at great risk [1]. Therefore, as we transition in the era of autonomous vehicles, an understanding of the CAN bus and its vulnerabilities has never been more important. This report summarizes previous work done by students at the University of Maryland to create a hardware-in-the-loop (HIL) simulation of a vehicle's CAN bus. It also summarizes a continuation of that work done as part of a collaboration between the University of Maryland and Ford to investigate vulnerabilities of the CAN bus using an open sourced project developed by Ford, called OpenXC. The report focuses on an Independent Study involving a project that leveraged various OpenXC tools to log CAN data from a Ford vehicle and mirror that data on the a HIL simulation in a lab, in real-time. Finally, tangential projects in which an automated assist feature from a Ford Fusion 2017 was examined for playback vulnerability, and where a portion of OpenXC code was reverse engineered in an attempt to discover critical CAN information, are also discussed.

1. Introduction

Automobiles are no longer the mechanical devices that they were just a generation ago. With every new model year vehicle pushed from the factory floor, modern automobiles relinquish more operational control to internal digital networks. It was recently demonstrated that an attack on certain electronic control systems within a vehicle can lead to adverse effects on automotive functions including disabling the brakes, selectively braking individual wheels and stopping the engine [1].

Recognizing these vulnerabilities, it has never been more important to probe the control systems within a vehicle, embedded microprocessors known as Electronic Control Units (ECUs), and the digital network by which they communicate. By studying this network and its current vulnerabilities, we hope to advance vehicle security and help mitigate potential risks to passengers.

To fully appreciate the work done for this Independent Study, a certain level of background knowledge is required. The next three sections of this report aim to provide this information. To understand the control systems within a vehicle, the reader must first understand the digital network by which they communicate. Section 2 discusses the Controller Area Network (CAN bus) and digital communication between embedded systems within an automobile. Additionally, as this study builds on the work of students and faculty at the University of Maryland, Section 3 covers a brief overview of aspects of their work pertaining to the study. Finally, in Section 4, the Ford OpenXC open sourced hardware and software platform is introduced and some prior work by students at the University of Maryland involving progress in that field is discussed.

The remainder of the report covers, in detail, the work done directly as a result of the study. Section 5 is introduces the goals and methodology of each of the three main aspects of

the study: virtualizing access to the vehicle, playback of an assist feature to explore security vulnerabilities and reverse engineering Ford's OpenXC firmware. Section 6 is dedicated to the results of each of the goals. Section 7 holds conclusions from the study and, finally, Section 8 touches on opportunities for related future works. The appendices hold all relevant code necessary to reproduce this study.

2. Controller Area Network and its Vulnerabilities

The CAN bus was developed by BOSCH in the 1980s as a multi-master, message broadcast system [2] and is used in automotive applications for the communication between Electronic Control Units (ECU) within a vehicle. Starting in 2008, all cars in the U.S. were required to implement a CAN bus for diagnostics [1].

An ECU refers to an embedded system within a vehicle that controls one or more subsystems of that vehicle. An ECU may be connected to the CAN bus to send and receive signals to assist in its control operation [3]. Examples of such ECUs would be the Engine Control Module (ECM or ECU), the Powertrain Control Module (PCM) and the Anti-lock Breaking System (ABS).

2.1 CAN Frames, Arbitration and Buses

The standard CAN frame format has several bitfields, but two fields, arbitration ID and data, are most crucial to understanding a CAN message at a basic level.

The data portion of the message consists of at most 8 bytes of ECU data. The length of the data is designated by the 4-bit Data Length Code (DLC), which gives the number of bytes being transmitted [2]. Each byte in the data will contain information pertaining to the ECU transmitting on the bus. Figure 1 shows the basic format of a CAN frame.

The arbitration ID is a unique 11-bit ID (for standard CAN) manually assigned to an ECU that arbitrates for priority on the bus. One ECU may transmit from multiple IDs but no two ECUs will have the same ID. If two ECUs attempt to send a CAN message at the same time, the one transmitting with the lower arbitration ID will be successful [2]. Therefore, ECUs with with the most critical functions are generally assigned a lower arbitration ID.

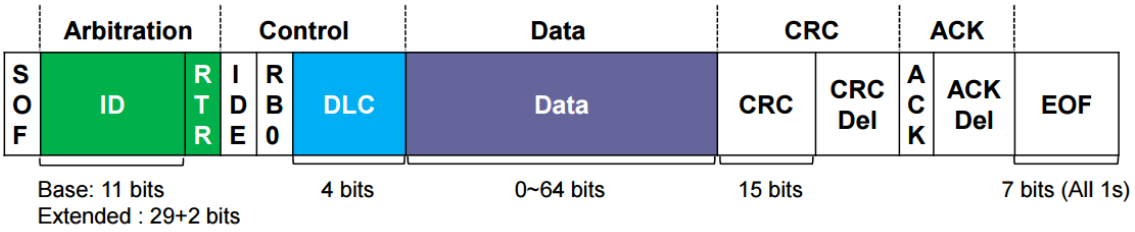

Figure 1: The basic format of a CAN frame [5]. From left to right, the bit fields are: start of frame (SOF), arbitration ID, remote transmission request (RTR), identifier extension (IDE), data length code (DLC), data, cyclic redundancy check (CRC), acknowledgment (ACK), and end of frame (EOF) [2].

Modern-day automobiles tend to employ two or more CAN buses, commonly referred to as high-speed (HS) CAN and mid-speed (MS) CAN. The HS CAN transmits at 500 kbps and is primarily used by the powertrain systems, while the MS CAN typically transmits at 125 kbps and connects less-demanding components [1]. Each bus consists of two wires that act as a differential pair and are terminated with a 120Ω resistor at each end [2].

The CAN is a shared bus, and most ECUs within a vehicle will have access to either the HS CAN, the MS CAN or both. The buses also run to the on-board diagnostics (OBD-II) connector which is present in all modern day automobiles.This port allows for direct access to all CAN messages generated by the various ECUs within the vehicle. A high-level representation of the CAN bus is shown in Figure 2, below.

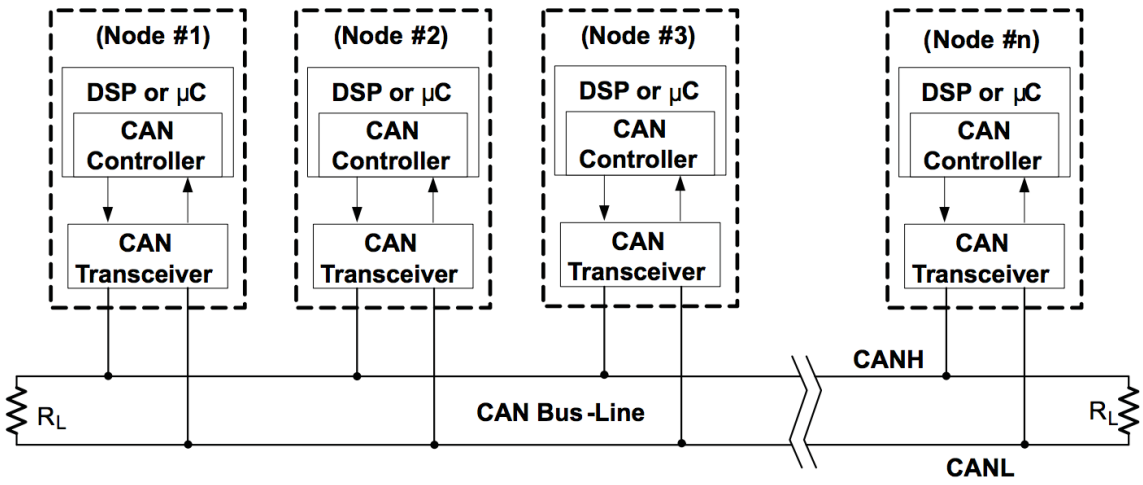

Figure 2: A high-level representation of the CAN bus [2]. Each node represents a complete ECU or part of an ECU. The differential pair can be seen connecting to each node, with the ends terminated with the specified impedance.

2.2 Inherent Security Weaknesses

Several inherent weaknesses in the CAN protocol make vehicle security a challenge. Chief among these are the broadcast nature, no authenticator fields [1] and the lack of encryption.

It is important to realize that all ECUs connected to a CAN bus receive all CAN messages broadcast on that bus. Each ECU will then filter messages pertinent to its functionality. However, this makes for easy network snooping, a fact that is leveraged by most vehicle analyzer tools [1].

Additionally, there is no authenticator field in the CAN frame to provide validation that a CAN message with a certain arbitration ID was actually generated by the ECU holding that ID. Therefore, any component connected to the bus can send a packet to any ECU without it being able to verify the validity of the message.

The CAN protocol currently does not support any security or encryption features. This makes the CAN network an inherent security vulnerability.

3. Foundational Work at the University of Maryland

A hardware-in-the-loop (HIL) simulation attempts to assess the performance of hardware in real-time based on virtual data generated by a computer [5]. This data is then used to determine the validity of a model.

A CAN bus HIL simulation was developed by a team of faculty, graduate and undergraduate students from the University of Maryland throughout the summer of 2015. The goal was to model a CAN network within an automobile using ECUs and then use this model to find potential cyber-physical vulnerabilities in an automobile. The specifics of their work have been discussed in detail in a report produced by the team, but an overview of the simulation will be included here.

ECUs were salvaged from a wrecked Ford Fusion 2011 and pulled into a lab on the university campus. The ECUs included in the model were the Instrument Panel Cluster (IPC), Front Controls Interface Module, Front Display Interface Module and the Audio Control Module.

Using schematics and data sheets acquired from online sources, wires were run from each ECU to a breadboard to deliver power and to establish a HS and MS CAN bus with proper 120Ω termination. Finally, both CAN buses were connected to an ODB-II port, thereby completing the basic structure of the HIL simulation setup. By simply providing power to each ECU, CAN messages would be pushed onto the bus and could be read via the OBD-II port. Their lab setup is show in Figure 3, below.

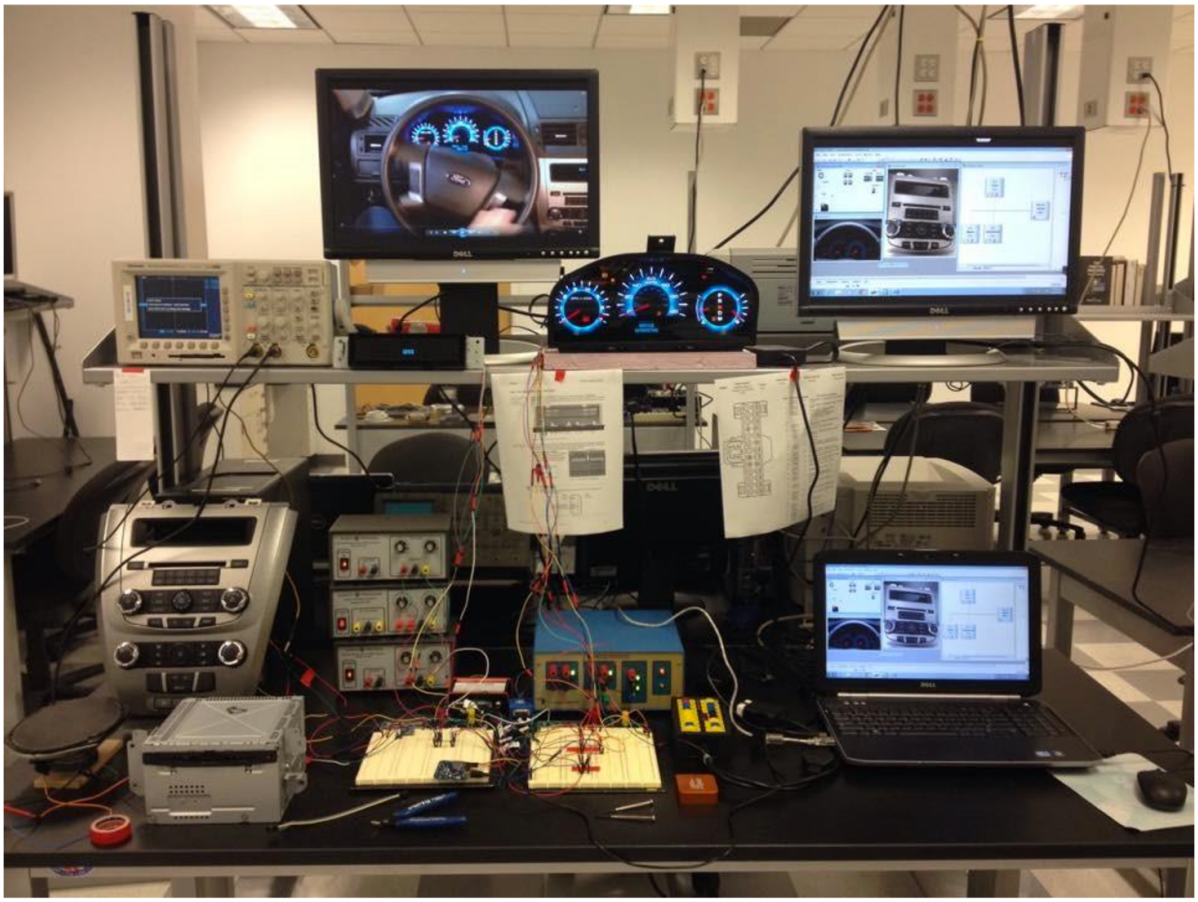

Figure 3: The lab setup of the University of Maryland team from the summer of 2015. Included in the picture are the following ECUs: Instrument Panel Cluster (IPC), Front Controls Interface Module, Front Display Interface Module and the Audio Control Module. Also shown is the simulation GUI (top right), the power supplies and the CAN bus on two breadboards (bottom center)

The simulation was validated using the Vector CANoe software which generated virtual CAN data for the HS and MS CAN buses. Using Vector CANoe, a digital replica of the Ford Fusion interface was created. Digital buttons, toggle switches and slider bars all generated CAN messages that controlled various aspects of the ECUs such as the RPM gauge, MPH gauge, speaker volume, radio station and more.

The CAN messages for the validation were discovered through an intensive reverse engineering process with the assistance of the Vehicle Spy software. Vehicle Spy allows for the

creation of custom CAN messages and for the user to monitor and record vehicle CAN traffic in real-time, to be replayed on the bus via the ODB-II port at a later time.

By performing some action within the vehicle, such as putting down the driver's window, and recording vehicle CAN traffic during that action, the exact CAN message to perform that action can be discovered. Using a binary search technique, the recorded CAN traffic during the initial action can be cut in half, where each half is then replayed in the vehicle in a search to determine the desired CAN message. If for example, the first half of the recorded traffic is replayed and produces no action, but the second half is replayed and causes the driver's window to go down, then the CAN message for that action must reside in the second half of the recorded traffic.

This search procedure is repeated until a single CAN message is identified. It is confirmed by crafting the message as a custom message in Vehicle Spy and then sending it to the bus where the action provides confirmation.

Once an arbitration ID is found for a particular ECU, it can be individually monitored in Vehicle Spy and by performing other actions on that ECU more information can be discovered regarding the arbitration ID's data formatting without the need for a binary search.

4. OpenXC

OpenXC, developed by Ford, is "an open source, data-focused API for your car" that "lets you extend your vehicle with custom applications and pluggable modules" [6]. By connecting the OpenXC Vehicle Interface (VI) hardware into the OBD-II port of a Ford vehicle and then connecting this interface to a smartphone, tablet, computer or web server, the user gains access to data from numerous sensors within the vehicle, in real-time. The data set varies depending on the model and year of the Ford vehicle, with newer vehicles having a larger data

set. A full set would include steering wheel angle, accelerator pedal position, fuel level, engine speed and more [7].

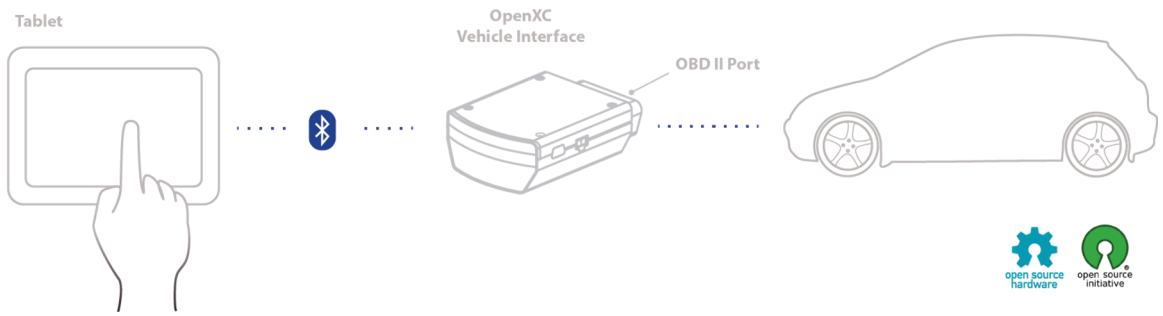

Figure 4: A top-level diagram depicting a connection scheme between tablet, VI and vehicle [6]. The Ford vehicle (right) produces CAN bus messages which are sent to its OBD-II port. The VI (center) is plugged into this port and can relay the CAN traffic to the user (left) via USB or Bluetooth.

CAN messages are sent from the vehicle to the VI via the OBD-II port where arbitration IDs and data are mapped to the data set and translated into human readable JSON data. This data is then sent to the host device via a USB or Bluetooth connection where the user can view or analyze the data as needed.

An Android application, OpenXC Enabler, available from Google Play is free and can be installed on any smartphone or tablet running the Android OS. It connects to the VI via Bluetooth and includes a GUI dashboard that displays all available data from the vehicle and can also send vehicle diagnostic requests. This provides for a simple plug-and-play option for the OpenXC platform.

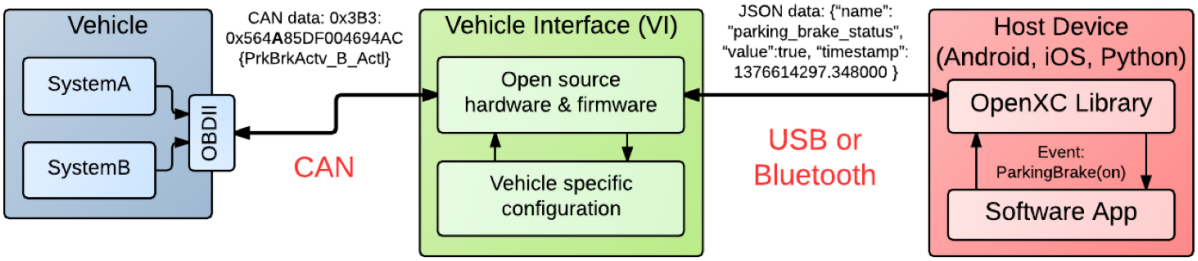

Figure 5: An architecture diagram depicting the transmission of the vehicle's parking brake status as a CAN message via OBD-II port to the VI, and then as a JSON formatted message to a host device via USB or Bluetooth [8].

For those users looking to do more with the data through software, the open source software includes Python and Android libraries that greatly simplify user generated code and ease access to data from the VI. Python programs and Android apps can be created to use the OpenXC data in any way the user finds suitable or interesting.

The framework of the VI firmware is also open source and comes with extensive documentation to assist the user in adding to or reconfiguring the firmware to meet specific needs. The code was written in C++ and has numerous configuration and makefile options that create swift but profound changes to the VI functionality.

In the summer of 2016, a new team of faculty, undergraduate and graduate students from the University of Maryland (myself included) partnered with Ford to leverage these tools to investigate related vulnerabilities of the CAN bus.

With the assistance of the documentation, a passthrough configuration file was created to change the VI firmware to read raw CAN messages. So, rather than mapping the messages to sensors and translating the messages to JSON, the VI just passed along the CAN message to the host device, via USB or Bluetooth, to be seen by the user. This passthrough configuration file is included in the Appendix A.1.

CAN messages could also be issued and written from the host device, including the OpenXC Enabler app, and sent to the VI. Amazingly, these messages would be passed along to the vehicle where they were placed on the CAN bus for all vehicle ECUs to see. Using the HIL test bench, it was demonstrated that these messages effected ECUs just as any other CAN message would.

5. Goals and Methodology

In the Fall of 2016 I began an Independent Study, under the supervision of faculty and staff at the University of Maryland, to continue investigating the capabilities of OpenXC and expand the research in new directions. The study established three goals: (1) remotely virtualize access to a vehicle so vehicle CAN traffic could be viewed in a lab in real-time from a vehicle on the road, (2) investigate the security vulnerabilities related to automated assisted features and (3) reverse engineer Ford's OpenXC pre-compiled firmware to discover the mapping of arbitration IDs to vehicle data.

5.1 Remote Virtual Access to a Vehicle

Building off of the work already done, a new project was undertaken to leverage various OpenXC tools to log CAN data from a Ford vehicle and mirror the data on the HIL simulation in a lab, in real-time.

The project relied on a new VI, the CrossChasm C5 Cellular, with a 3G connection capability if equipped with a SIM card by the user. This feature allowed the C5 Cellular VI to transmit data from the vehicle to the Internet in real-time from anywhere with a cellular signal. To compliment this feature, Ford also released a sample Microsoft Azure OpenXC web server and web app to receive data from the C5 Cellular device [9].

The idea, then, was to transmit the raw CAN traffic from the vehicle to an Azure server and then pull the data from the server with a computer and push it onto the HIL test bench using the original VI with its raw CAN write functionality enabled. A top-level diagram of this architecture is shown in Figure 6, below.

Figure 6: A top-level architecture diagram depicting the transmission of a CAN message from the data collection vehicle to the HIL test bench, to allow for virtual remote access to the vehicle.

First, the Azure web app and SQL server instances were created[1]. An appropriate URL was chosen for the both the web app and web server, and the two were linked via a database connection string. Next, using Visual Studio, the web application was configured according to the documentation, linked via the web app configuration string and launched. At this point, the web app was live and ready to receive data. The web configuration file is included in the Appendix A.2.

The next step was to configure the C5 Cellular VI to transmit data to the web server. The SIM card used by the C5 Cellular was acquired from AT&T, allowing for 300MB of data over 6 months [10]. The configuration C++ file was built to match this SIM via the Access Point Name (APN), and the server connection settings were also set to match the web server address [11]. The C5 Cellular VI configuration file is included in the Appendix A.3.

Using the same passthrough configuration file for the original VI (see Section 4), the C5 Cellular VI firmware was recompiled and loaded onto the C5 Cellular device. At this point, the C5 Cellular VI could be plugged into the OBD-II port of a vehicle and would transmit raw CAN traffic to the Azure server in real-time.

[1] This was set up for free using Microsoft Imagine for students, through DreamSpark.

Vehicle data is transmitted from the C5 Cellular VI in blocks of CAN messages, each with its own timestamp and each containing numerous individual CAN messages in JSON format. Each CAN message has 4 fields: “timestamp”, “bus”, “id” and “data”. The “timestamp” is a 13 second Unix timestamp which corresponds to millisecond granularity. The “bus” always takes values of either 1 or 2, with 1 signifying the message was transmitted on the HS CAN and 2 signifying it was transmitted on the MS CAN. The “id” is the arbitration ID of the CAN message in base 10, and the “data” is the hexadecimal representation of the CAN data, always starting with “0x”. A typical block of transmitted CAN data with its corresponding timestamp as displayed on the Azure web app is shown in Figure 7, below.

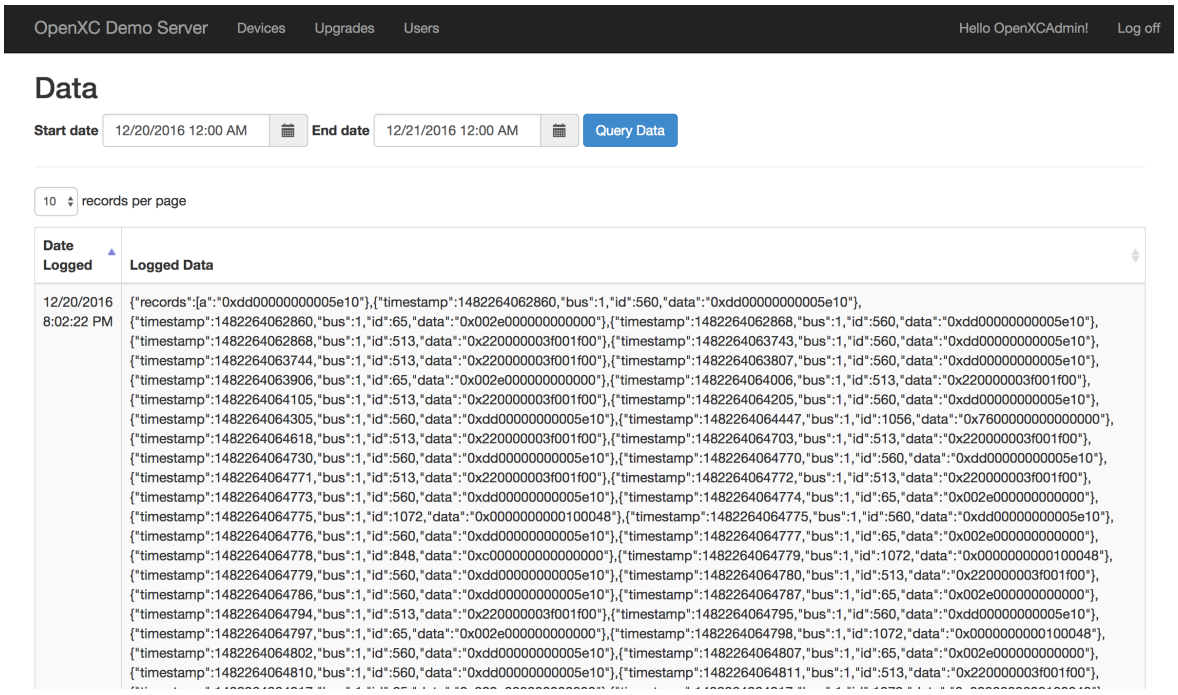

Figure 7: Raw CAN data stored on the server and displayed through the web app. The date selection can be seen at the top of the page and a timestamp for each block transmission is along the left side. Each individual CAN message has JSON fields for its own timestamp, bus, arbitration id and data.

With no backend experience for manipulating server data, it was determined that the most efficient way to pull the CAN data from the server was through a Python program. The benefit here was that tools from the OpenXC Python library could be incorporated so that data could be scraped from the server and sent to the VI, and ultimately onto the HIL simulation, within the same program.

To pull the CAN messages from the server, the Selenium API for Python was used to run the Selenium WebDriver. The web driver has the ability to navigate and interact with webpages as though it were a normal human user.

The program launches an instance of Google Chrome and then navigates through the HTML to log into the website. Next, the user is prompted to enter the start date and start time of the simulation. This input is used to query the data so that only valid transmission blocks, with a timestamp after the one entered by the user, appear.

The program arranges the data to display up to 100 transmission blocks per page with the newest block at the top of each page. As the data is transmitted in each block, messages are often cut off at the beginning and end of the block leaving partial data (seen in Figure 7). These portions are removed to reformat the data in each block to comply with JSON format.

The program then begins reading in block data from the bottom to the top of the page. The four JSON fields of each message are parsed into a 4-dimensional array. It then continues reading data blocks, only storing CAN messages with timestamps greater than the timestamp of the last message stored in the array. In this way, the array is only comprised of valid CAN messages, all arranged in chronological order.

The program then begins sending CAN messages to the VI to be passed to the HIL simulation. Beginning at the top of the array with the earliest CAN message, the timestamp is normalized and compared to a timer that was was started at the beginning of the program. If the timer is greater than the normalized timestamp then the message is ready to be sent.

Using the OpenXC Python library, a write command is sent to the VI containing the CAN message from the top of the array. Upon a successful write, the CAN message is removed from the array and the timestamp of the next message is then compared against the timer. In this way, the timestamp of each message is maintained.This process is continued until the all messages in the array have been written.

The program then refreshes the webpage and begins scanning for new data to scrape into the array. The C5 Cellular VI transmits data blocks every 3-6 seconds and the webpage can hold 100 data blocks per page. So, due to this relatively long transmit latency, the webpage should never should never be filled with more than a few new blocks per refresh. Therefore, new data should not be lost to a refresh as long at the original page had fewer than 100 entries.

The complete program for scraping and writing data is included in the Appendix A.4.

5.2 Playback of an Assist Feature

Automated driver assist features are electronic systems in an automobile that use input to one or more sensors on the vehicle to effect some output action. Typically, each assist feature is designed to either automate some process or to enhance the vehicle's safety.

However, CAN signals generated by these features inherently have a profound control over the vehicle. They directly effect on the operation of the vehicle in ways that are critical to the vehicle's occupants' safety. Additionally they are self generated. The driver plays no role in sensor output or how the vehicle reacts to that data. For these reasons, automated driver assist features pose an enticing, prima facie, security vulnerability.

As automobiles move into the digital age, automated driver assist features are becoming more commonplace in vehicles. While these features move towards the vehicle standard and

drivers become gradually less involved in controlling the vehicle, CAN bus vulnerabilities and CAN bus security are more important than ever.

In the Summer of 2016, as part of the collaboration between the University of Maryland and Ford, a 2017 Ford Fusion Energi Titanium was acquired for research. The car came equipped with many automated assist features, with the goal of investigating cyber-physical attacks via the CAN bus and CAN messages related to these features. As part of this study, Ford's Active Park Assist feature was examined using Vehicle Spy's record and playback functions.

On a button press, Active Park Assist searches for a parking spot on either the left or right side of the vehicle, informed by the driver via the turn signal. The vehicle will search for either a parallel or perpendicular spot and then notify the driver once a spot has been found. The driver removes their hands from the steering wheel as the car maneuvers into the spot while the driver switches gears (reverse, drive or park) when notified to do so by the vehicle's touchscreen display. The speed of the maneuver is at the driver's discretion and is controlled, per usual, via the accelerator and brake pedals. The vehicle notifies the driver once the car is appropriately positioned in the spot, and the car can be put into park.

To perform this operation the vehicle relies on proximity sensors on the front and rear bumpers of the vehicle. Typically, these are ultrasonic sensors that use the frequency shift in acoustic pulses to determine the proximity of an adjacent object. The Ford Fusion Active Park Assist also incorporates a sensor along the side of the vehicle to identify an available parking spot.

The goal of this portion of the study was to observe how the vehicle reacted when CAN data collected during an Active Park Assist were replayed on the vehicle's CAN network during a normal driving session. Here, it was presumed that all or most of this sensor data was put onto the CAN so that other ECUs could act appropriately to carry out the parking algorithm. For

example, the gear status may tell the parking assist when to begin and end, and the vehicle's speed and the proximity of adjacent vehicles may determine how the control the steering wheel angle during the maneuver.

In the study, a successful Active Park Assist was performed into a parallel parking space and the entire set of CAN data was recorded using Vehicle Spy. The data were then played back continuously into the vehicle while driving on a closed course at speeds between 5 and 10 miles per hour (mph) and any effect was observed.

5.3 Reverse Engineering the OpenXC Firmware

As previously mentioned (see Section 4), the data set available to an OpenXC user varies depending on the model and year of the user's Ford vehicle, with newer vehicles having a larger data set. Ford distributes a set of precompiled firmware binaries, with each binary in the set working best with a particular Ford vehicle. To maximize the data set, users can check their vehicle against a Ford maintained spreadsheet to determine which binary should be programmed onto their VI [17].

Each binary, before being compiled, contains a mapping from each member in the available data set to its corresponding CAN message arbitration ID, as well as a translation for the CAN message data. This mapping is unique to Ford vehicles. The mapping is not included in the OpenXC open source firmware or in the documentation for security purposes. As formerly discussed (see Section 2.2), having access to this map would completely expose the security flaw in the CAN protocol. Without any obfuscation of CAN messages and their format, a malicious attack could easily be performed against a Ford vehicle through a targeted injection of CAN messages onto the bus. However, if the mapping could somehow be recovered from the binary file then this would present a serious security flaw within the OpenXC project.

Therefore, the goal of this portion of the study was to extract this mapping from the binary, if possible, and determine the level of difficulty of the task. The strategy was to compile the open sourced firmware and then compare this to the precompiled firmware binary distributed by Ford. The theory was that the only major substantive difference between the two files was the mapping.

To analyze the binaries, Interactive Disassembler (IDA Pro) software was used to in an attempt to generate human readable assembly language source code from the binary machine code. The VI used an LPC17xx microcontroller with and ARM 7 architecture [13]. The Makefile and linker files in the source code called out the memory locations for the flash (ROM) and RAM as “0x10000” and “0x100000C8”, respectively [14]. Their respective lengths are also defined [14].

After compiling the open source firmware, the binary was loaded into IDA Pro, with the file and processor options selected accordingly. The memory organization for RAM and ROM was configured as well, as seen in Figure 8 below. The starting location in memory is called out in the documentation as well: “Start the user code 64KB into flash, as the USB bootloader expects” , which is equivalent to “0x10000” [14]. Once loaded, IDA Pro displays the memory locations and the data stored in each location. Moving to the starting address “0x10000”, I began to auto-analyze the undefined bytes and then create functions.

This same process was repeated for the precompiled firmware binary distributed by Ford.

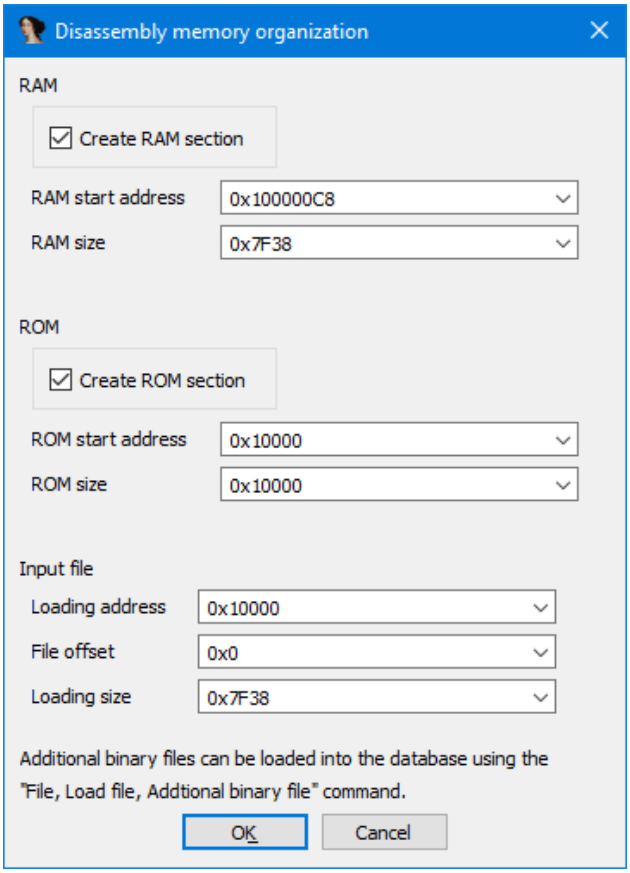

Figure 8: Configuration of the memory organization for the RAM and ROM (flash) in IDA Pro.

6. Results

6.1 Remote Virtual Access to a Vehicle

Initially, tests were run using CAN data from the 2017 Ford Fusion Energi Titanium with disappointing results. Data was flowing from the server to the HIL test bench in the lab, as confirmed by live monitoring with Vehicle Spy, but the ECUs were not responding. It was assumed that there would be some overlap between arbitration IDs in the 2017 Fusion and the arbitration IDs from the 2011 Fusion ECUs in the lab. This turned out not to be the case.

Data was collected again, this time using a 2011 Ford Fusion borrowed from the University of Maryland motor pool. Results were better and the program worked as expected,

pushing CAN messages from the server to the HIL test bench with ECUs responding accordingly. Dials on the IPC corresponding to MPH, RPM and fuel level, and the light signaling the vehicle's gear, all reacted to CAN messages scraped from the server.

However, it was observed that MS CAN messages from the vehicle never appeared on the server (which can be seen in Figure 7 with all "bus" values equal to 1) and therefore were never pushed to the the HIL test bench. It was discovered that the provided cabling for the C5 Cellular device only connects to the HS CAN and must be modified or replaced to connect to the MS CAN [15].

Additionally, the throughput for the CAN messages was too low for the dials and lights on the IPC in the lab to continuously track the IPC in the data collection vehicle. The bottleneck occurs when collecting the the CAN data in the vehicle and transmitting it to the server (see the left side of Figure 6). During collection, a buffer in the C5 Cellular is filled with CAN data and then sent across the 3G network to the server [16]. This buffer size needs to be increased or it needs to be filled and transmitted at a higher frequency.

6.2 Playback of an Assist Feature

Visually, it was immediately obvious that playing back the CAN data from the assist feature had an adverse effect on the vehicle. Numerous warning lights on the IPC became illuminated and the driver was warned that traction control was turned off. According to the user's manual for the vehicle, "If the stability control and traction control light is still illuminating steadily, have the system serviced by an authorized dealer immediately. Operating your vehicle with the traction control disabled could lead to an increased risk of loss of vehicle control, vehicle rollover, personal injury and death [12]." The touchscreen display also began to flash between screens: the standard user interface (UI), the rear view camera and a black screen.

More alarming than the warnings was the actual operation of the car. It felt as though the clutch was in continuous cycle of engaging and disengaging. Attempting to drive at speeds under 10 MPH caused the car the car to advance in a stop-and-go fashion. In fact, the lights signifying the gear of the vehicle moved between “park” and “reverse” at very high speeds. The car was essentially un-drivable.

6.3 Reverse Engineering the OpenXC Firmware

Through IDA Pro, I was able to make the several function graphs, such as the one in Figure 9, below. These were generated for both the open sourced firmware and the precompiled firmware binary distributed by Ford. Unfortunately, there was not enough time to do a full analysis on the results and no mapping of CAN arbitration IDs to vehicle signals was discovered.

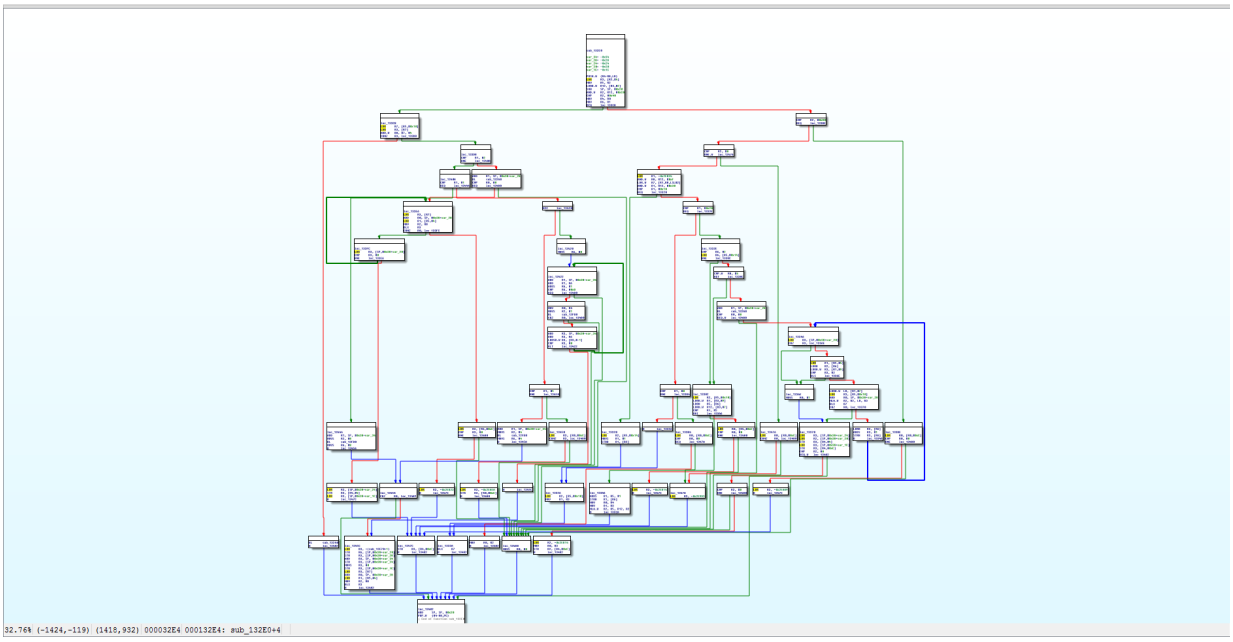

Figure 9: Function graph created in IDA Pro for the open sourced firmware.

7. Conclusions

The study pursued its three goals to varying levels of success. A remote virtual connection was successfully established between a vehicle and a server and that data could be pushed to the HIL test bench in a lab, in real-time. However, a bottleneck created by a buffer and the transmission of the data in that buffer prevented a smooth, continuous, digital representation of the vehicle. It was shown that the playback of one particular automated assist feature indeed had a detrimental effect on vehicle functionality and safety. Unfortunately, this playback was not analyzed at depth and no other automated assist features were examined. Function graphs were created in IDA Pro for both binaries, but a full analysis was never conducted and it is unclear if the signal mapping could be extracted.

In the end, the three goals were too ambitious for one graduate student and proved to be more work than I could handle. Lack of experience with Visual Studio and web development turned out to be a problem when trying to create the web app and Azure server. What should have been a quick deployment turned into a multi-week debugging session all due to a simple syntax error in the connection string. Also, to fully examine the automated assist feature required at least one other person for safety reasons. With more of a focus on the other two goals and a lack of experience with IDA Pro, reverse engineering the firmware was not investigated to its full potential.

Through this study I have become very familiar with OpenXC and was able to contribute positively to its open source development. I also became proficient with Python and learned about several useful APIs and gained some experience with IDA Pro. And, of course, I also became very knowledgeable about the CAN bus, security vulnerabilities within vehicles on the road today and the dangers posed to autonomous and semi-autonomous vehicles as we advance into the digital age.

8. Future Work

The work done as part of this independent study presents many opportunities for future research. Regarding remote virtual access to a vehicle, increasing the size of the transmit buffer and trying to increase throughput of CAN data. Also, modifying or purchasing a new cable to have access to MS CAN data. Another option would be to modify the passthrough configuration file in the C5 Cellular firmware so that only targeted arbitration IDs are transmitted. It would also be interesting to create a bi-directional connection between the vehicle and the lab.

For the playback of an assist feature, the recorded CAN traffic can be examined at depth to determine a single CAN message or a more precise sequence of CAN messages that would duplicate the peculiar behavior of the vehicle. There are several more assist features that could be examined as well.

Clearly, more can be done in IDA Pro to determine the mapping of vehicle signals to their arbitration IDs. After my study had concluded, this topic was investigated by a third party at the request of Ford. A webcast entitled “The Impacts of JSON on Reversing Your Firmware” was put on by Ben Gardiner of Irdeto, and can be found at [18]. This would be a powerful resource for future work in this area.

References

[1] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage. Experimental security analysis of a modern automobile. In IEEE S&P, May 2010.

[2] Corrigan, S. (2008). Introduction to the Controller Area Network (CAN) (Application Report No. SLOA101A). Retrieved from Texas Instruments website: http://www.ti.com/lit/an/sloa101a/sloa101a.pdf

[3] National Instrument Whitepaper: ECU Designing and Testing using National Instrument Products. Online: http://www.ni.com/white-paper/3312/en/

[4] http://opengarages.org/handbook/2014_car_hackers_handbook_compressed.pdf

[5] https://www.mathworks.com/help/physmod/simscape/ug/what-is-hardware-in-the-loop-simulation.html

[6] http://openxcplatform.com/

[7] http://openxcplatform.com/about/data-set.html

[8] http://openxcplatform.com/vehicle-interface/concepts.html

[9] https://github.com/openxc/openxc-azure-webserver

[10] https://starterkit.att.com/kits

[11] http://vi-firmware.openxcplatform.com/en/master/advanced/c5_cell_config.html

[12] http://cdn.dealereprocess.com/cdn/servicemanuals/ford/2017-fusionhybrid.pdf

[13] http://vi.openxcplatform.com/electrical/design/microcontroller.html

[14] https://github.com/openxc/vi-firmware/blob/45eeb2ac664b8f5969a928c31262de01626e0d3b/src/platform/lpc17xx/LPC17xx-bootloader.ld

[15] http://openxcplatform.com/vehicle-interface/hardware.html#crosschasm-c5-cellular

[16] https://github.com/openxc/vi-firmware/blob/45eeb2ac664b8f5969a928c31262de01626e0d3b/src/platform/pic32/telit_he910.h#L130

[17] https://docs.google.com/spreadsheets/d/1hOBi9-tFwR1KRFXfeaHTAddwJuSGx5Ir1ET4N2zWAiE/edit#gid=2

[18] https://www.sans.org/webcasts/impacts-json-reversing-firmware-104357

Appendices

The text from these files is included here for archiving purposes. To better view each file, copy into a text editor of your choosing (e.g., Sublime Text) and save with the file extension called out in parenthesis after each Appendix title.

A.1 OpenXC VI Passthrough Configuration File (.json)


```
{       "name": "passthrough",
   "buses" : {
      "hs": {
         "controller": 1,
         "raw_can_mode": "unfiltered",
         "raw_writable": true,
         "speed": 500000
      },
      "ms": {
         "controller": 2,
         "raw_can_mode": "unfiltered",
         "raw_writable": true,
         "speed": 125000
      }
   }
}
```

A.2 Azure Web.config File (.xml)

```
<?xml version="1.0" encoding="utf-8"?>
<!--
  For more information on how to configure your ASP.NET application, please visit
  http://go.microsoft.com/fwlink/?LinkId=301880
  -->
<configuration>
  <configSections>
    <!-- For more information on Entity Framework configuration, visit http://go.microsoft.com/
fwlink/?LinkID=237468 -->
    <section name="entityFramework"
type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection, EntityFramework,
Version=6.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
requirePermission="false" />
  </configSections>
  <connectionStrings>
    <add name="OpenXCDbEntities"
         connectionString="Server=tcp:crosschasm-c5.database.windows.net,1433;Initial
Catalog=openxc-data;Persist Security Info=False;User ID=Kyle;Password=Password
$;MultipleActiveResultSets=False;Encrypt=True;TrustServerCertificate=False;Connection
Timeout=30;"
         providerName="System.Data.SqlClient" />
  </connectionStrings>
```

```
  <appSettings>
    <add key="webpages:Version" value="3.0.0.0" />
    <add key="webpages:Enabled" value="false" />
    <add key="ClientValidationEnabled" value="true" />
    <add key="UnobtrusiveJavaScriptEnabled" value="true" />
  </appSettings>
  <system.web>
    <authentication mode="None" />
    <compilation debug="true" targetFramework="4.5" />
    <httpRuntime targetFramework="4.5" />
  </system.web>
  <system.webServer>
    <modules>
      <remove name="FormsAuthentication" />
    </modules>
  <handlers>
      <remove name="ExtensionlessUrlHandler-Integrated-4.0" />
      <remove name="OPTIONSVerbHandler" />
      <remove name="TRACEVerbHandler" />
      <add name="ExtensionlessUrlHandler-Integrated-4.0" path="*." verb="*"
type="System.Web.Handlers.TransferRequestHandler"
preCondition="integratedMode,runtimeVersionv4.0" />
    </handlers></system.webServer>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
```

```
      <dependentAssembly>
        <assemblyIdentity name="Newtonsoft.Json" culture="neutral"
publicKeyToken="30ad4fe6b2a6aeed" />
        <bindingRedirect oldVersion="0.0.0.0-6.0.0.0" newVersion="6.0.0.0" />
      </dependentAssembly>
      <dependentAssembly>
        <assemblyIdentity name="System.Web.Optimization"
publicKeyToken="31bf3856ad364e35" />
        <bindingRedirect oldVersion="1.0.0.0-1.1.0.0" newVersion="1.1.0.0" />
      </dependentAssembly>
      <dependentAssembly>
        <assemblyIdentity name="WebGrease" publicKeyToken="31bf3856ad364e35" />
        <bindingRedirect oldVersion="0.0.0.0-1.6.5135.21930" newVersion="1.6.5135.21930" />
      </dependentAssembly>
      <dependentAssembly>
        <assemblyIdentity name="System.Web.Http" publicKeyToken="31bf3856ad364e35"
culture="neutral" />
        <bindingRedirect oldVersion="0.0.0.0-5.2.2.0" newVersion="5.2.2.0" />
      </dependentAssembly>
      <dependentAssembly>
        <assemblyIdentity name="Microsoft.Practices.Unity" publicKeyToken="31bf3856ad364e35"
culture="neutral" />
        <bindingRedirect oldVersion="0.0.0.0-3.5.0.0" newVersion="3.5.0.0" />
      </dependentAssembly>
      <dependentAssembly>
```

```
      <assemblyIdentity name="Antlr3.Runtime" publicKeyToken="eb42632606e9261f"
culture="neutral" />
      <bindingRedirect oldVersion="0.0.0.0-3.5.0.2" newVersion="3.5.0.2" />
    </dependentAssembly>
    <dependentAssembly>
      <assemblyIdentity name="Microsoft.Practices.ServiceLocation"
publicKeyToken="31bf3856ad364e35" culture="neutral" />
      <bindingRedirect oldVersion="0.0.0.0-1.3.0.0" newVersion="1.3.0.0" />
    </dependentAssembly>
    <dependentAssembly>
      <assemblyIdentity name="System.Web.Helpers" publicKeyToken="31bf3856ad364e35" />
      <bindingRedirect oldVersion="1.0.0.0-3.0.0.0" newVersion="3.0.0.0" />
    </dependentAssembly>
    <dependentAssembly>
      <assemblyIdentity name="System.Web.WebPages" publicKeyToken="31bf3856ad364e35"
/>
      <bindingRedirect oldVersion="1.0.0.0-3.0.0.0" newVersion="3.0.0.0" />
    </dependentAssembly>
    <dependentAssembly>
      <assemblyIdentity name="System.Web.Mvc" publicKeyToken="31bf3856ad364e35" />
      <bindingRedirect oldVersion="0.0.0.0-5.2.2.0" newVersion="5.2.2.0" />
    </dependentAssembly>
    <dependentAssembly>
      <assemblyIdentity name="System.Net.Http.Formatting"
publicKeyToken="31bf3856ad364e35" culture="neutral" />
```

```
      <bindingRedirect oldVersion="0.0.0.0-5.2.2.0" newVersion="5.2.2.0" />
    </dependentAssembly>
    <dependentAssembly>
      <assemblyIdentity name="Microsoft.Owin" publicKeyToken="31bf3856ad364e35"
culture="neutral" />
      <bindingRedirect oldVersion="0.0.0.0-3.0.0.0" newVersion="3.0.0.0" />
    </dependentAssembly>
    <dependentAssembly>
      <assemblyIdentity name="Microsoft.Owin.Security" publicKeyToken="31bf3856ad364e35"
culture="neutral" />
      <bindingRedirect oldVersion="0.0.0.0-3.0.0.0" newVersion="3.0.0.0" />
    </dependentAssembly>
    <dependentAssembly>
      <assemblyIdentity name="Microsoft.Owin.Security.Cookies"
publicKeyToken="31bf3856ad364e35" culture="neutral" />
      <bindingRedirect oldVersion="0.0.0.0-3.0.0.0" newVersion="3.0.0.0" />
    </dependentAssembly>
    <dependentAssembly>
      <assemblyIdentity name="Microsoft.Owin.Security.OAuth"
publicKeyToken="31bf3856ad364e35" culture="neutral" />
      <bindingRedirect oldVersion="0.0.0.0-3.0.0.0" newVersion="3.0.0.0" />
    </dependentAssembly>
  </assemblyBinding>
 </runtime>
 <entityFramework>
```

```
    <defaultConnectionFactory type="System.Data.Entity.Infrastructure.SqlConnectionFactory,
EntityFramework" />
    <providers>
      <provider invariantName="System.Data.SqlClient"
type="System.Data.Entity.SqlServer.SqlProviderServices, EntityFramework.SqlServer" />
    </providers>
  </entityFramework>
</configuration>
```

A.3 CrossChasm Config.cpp File (.cpp)

```
#include "config.h"
#include "signals.h"
#include "md5.h"

using openxc::pipeline::Pipeline;
using openxc::interface::uart::UartDevice;
using openxc::payload::PayloadFormat;

namespace usb = openxc::interface::usb;
namespace ble = openxc::interface::ble;
namespace fs =  openxc::interface::fs;
namespace telit = openxc::telitHE910;
namespace signals = openxc::signals;

#ifdef TELIT_HE910_SUPPORT
static void getFlashHash(openxc::config::Configuration* config);
#endif

static void initialize(openxc::config::Configuration* config) {
    config->pipeline = {
        &config->usb,
        &config->uart,
```

```
#ifdef BLE_SUPPORT
        config->ble,
#endif
#ifdef FS_SUPPORT
        config->fs,
#endif
#ifdef TELIT_HE910_SUPPORT
        config->telit,
#endif
#ifdef __USE_NETWORK__
        &config->network,
#endif // __USE_NETWORK__
    };
    #ifdef TELIT_HE910_SUPPORT
    // run flashHash
    getFlashHash(config);
    #endif
    #ifdef TELIT_HE910_SUPPORT
    config->telit->uart = &config->uart;
    #endif
    config->initialized = true;
}

#ifdef FS_SUPPORT
fs::FsDevice fsDevice;
```

```
#endif



#ifdef BLE_SUPPORT
ble::BleDevice bleDevice = {

    descriptor: {
        allowRawWrites: DEFAULT_ALLOW_RAW_WRITE_BLE
    },
    blesettings: {
        "OpenXC_C5_BTLE",
        adv_min_ms: 100,
        adv_max_ms: 100,
        slave_min_ms : 8, //range 0x0006 to 0x0C80
        slave_max_ms : 16,
    }
};
#endif


// if we're going to conditionally compile our "Device" config structs, we
 // will need to declare them here (conditionally) and assign the pointers
 // on the config init call (conditionally)
#ifdef TELIT_HE910_SUPPORT
openxc::telitHE910::TelitDevice telitDevice = {
    descriptor: {
```

```
    allowRawWrites: DEFAULT_ALLOW_RAW_WRITE_UART
},
config: {
    globalPositioningSettings: {
        gpsEnable: false,
        gpsInterval: 5000,
        gpsEnableSignal_gps_time: false,
        gpsEnableSignal_gps_latitude: true,
        gpsEnableSignal_gps_longitude: true,
        gpsEnableSignal_gps_hdop: false,
        gpsEnableSignal_gps_altitude: true,
        gpsEnableSignal_gps_fix: true,
        gpsEnableSignal_gps_course: false,
        gpsEnableSignal_gps_speed: true,
        gpsEnableSignal_gps_speed_knots: false,
        gpsEnableSignal_gps_date: false,
        gpsEnableSignal_gps_nsat: false
    },
    networkOperatorSettings: {
        allowDataRoaming: true,
        operatorSelectMode: telit::AUTOMATIC,
        networkDescriptor: {
            PLMN: 0,
            networkType: telit::UTRAN
        }
```

```
        },
        networkDataSettings: {
            "m2m.com.attz"
        },
        socketConnectSettings: {
            packetSize: 0,
            idleTimeout: 0,
            connectTimeout: 150,
            txFlushTimer: 50
        },
        serverConnectSettings: {
            "umd-openxc.azurewebsites.net",
            port: 80
        }
    }
};
#endif



openxc::config::Configuration* openxc::config::getConfiguration() {
    static openxc::config::Configuration CONFIG = {
        messageSetIndex: 0,
        version: "7.2.0",
        payloadFormat: PayloadFormat::DEFAULT_OUTPUT_FORMAT,
```

```
recurringObd2Requests: DEFAULT_RECURRING_OBD2_REQUESTS_STATUS,
obd2BusAddress: DEFAULT_OBD2_BUS,
powerManagement: PowerManagement::DEFAULT_POWER_MANAGEMENT,
sendCanAcks: DEFAULT_CAN_ACK_STATUS,
emulatedData: DEFAULT_EMULATED_DATA_STATUS,
loggingOutput: DEFAULT_LOGGING_OUTPUT,
calculateMetrics: DEFAULT_METRICS_STATUS,
desiredRunLevel: RunLevel::CAN_ONLY,
initialized: false,
runLevel: RunLevel::NOT_RUNNING,
uart: {
    descriptor: {
        allowRawWrites: DEFAULT_ALLOW_RAW_WRITE_UART
    },
    baudRate: UART_BAUD_RATE
},
network: {
    descriptor: {
        allowRawWrites: DEFAULT_ALLOW_RAW_WRITE_NETWORK
    }
},
usb: {
    descriptor: {
        allowRawWrites: DEFAULT_ALLOW_RAW_WRITE_USB
    },
```

```
    endpoints: {
        {IN_ENDPOINT_NUMBER, DATA_ENDPOINT_SIZE,
            usb::UsbEndpointDirection::USB_ENDPOINT_DIRECTION_IN},
        {OUT_ENDPOINT_NUMBER, DATA_ENDPOINT_SIZE,
            usb::UsbEndpointDirection::USB_ENDPOINT_DIRECTION_OUT},
        {LOG_ENDPOINT_NUMBER, DATA_ENDPOINT_SIZE,
            usb::UsbEndpointDirection::USB_ENDPOINT_DIRECTION_IN},
    }
},
#ifdef BLE_SUPPORT
ble: &bleDevice,
#else
ble: NULL,
#endif

#ifdef FS_SUPPORT
fs : &fsDevice,
#else
fs: NULL,
#endif

#ifdef TELIT_HE910_SUPPORT
telit: &telitDevice,
#else
telit: NULL,
```

```
        #endif



        diagnosticsManager: {},
        pipeline: {},
    };

    if(!CONFIG.initialized) {
        initialize(&CONFIG);
    }
    return &CONFIG;
}

void openxc::config::getFirmwareDescriptor(char* buffer, size_t length) {
    snprintf(buffer, length, "%s (%s)", getConfiguration()->version,
            signals::getActiveMessageSet() != NULL ?
                signals::getActiveMessageSet()->name : "default");
}

#ifdef TELIT_HE910_SUPPORT
static void getFlashHash(openxc::config::Configuration* config) {
    MD5_CTX md5Context;
    unsigned char result[16];
    MD5_Init(&md5Context);
    MD5_Update(&md5Context, (const void*)0x9D001000, (unsigned long)0x2FC);
```

```
    MD5_Update(&md5Context, (const void*)0x9D001348, (unsigned long)0x7DCB8);
    MD5_Final(result, &md5Context);
    sprintf(config->flashHash, "%02x%02x%02x%02x%02x%02x%02x%02x%02x%02x%02x
%02x%02x%02x%02x%02x",
    result[0], result[1], result[2], result[3],
    result[4], result[5], result[6], result[7],
    result[8], result[9], result[10], result[11],
    result[12], result[13], result[14], result[15]);
}
#endif
```

A.4 Scraping and Writing Data (.py)

This was organized into two separate files. The first file, “virtual_access.py”, calls on the second file, “openxc_script.py”.

---

virtual_access.py

```
#!/usr/local/bin/python
from selenium import webdriver
from selenium.webdriver.chrome.options import Options
from selenium.webdriver.common.keys import Keys
from openxc_script import write_CAN_message, init_VI
from datetime import datetime
import time
import urllib
import urllib2
import json
import calendar



def navigate_to_data(driver):
    """Opens browser and queries data on server"""

    #Chrome browser navigates to azure site
```

```
    driver.get("http://umd-openxc.azurewebsites.net/devices/352682050225977/data")

    username = driver.find_element_by_id("UserName")
    password = driver.find_element_by_id("Password")

    #Enters username and password
    username.send_keys("OpenXCAdmin")
    password.send_keys("VXdDaBvdDU29rofs4Bmg")

    #Clicks Login
    xpath = '//input[@type="submit"]'
    driver.find_element_by_xpath(xpath).click()

    return


def query_data_at_datetime(driver):
    """Searches data based on date and time input from user"""

    startdatetime = get_start_datetime()
    enddatetime = get_end_datetime(startdatetime)

    #Enters start datetime
    startdate = driver.find_element_by_id("StartDate")
    startdate.clear()
```

```
        startdate.send_keys(startdatetime)

        #Enters end datetime
        enddate = driver.find_element_by_id("EndDate")
        enddate.clear()
        enddate.send_keys(enddatetime)

        #Query data at that date
        xpath = '//button[@type="submit"]'
        driver.find_element_by_xpath(xpath).click()

        #Display 100 elements
        xpath = '//select[@name="loggedDataTable_length"]//option[@value="100"]'
        driver.find_element_by_xpath(xpath).click()

        #Sort newest first
        xpath = '//th[@class="sorting_asc"]'
        driver.find_element_by_xpath(xpath).click()

        return


def get_start_datetime():
        """Gets a valid start date and time from the user"""
```

```
"""Thanks StackOverflow"""

#Prompts user to enter date. Repeat until vaild entry.

while True:

        date = raw_input("Enter start date (mm/dd/yyyy): ")

        try:

          valid_date = time.strptime(date, '%m/%d/%Y')

          break

        except ValueError:

          print "Invalid date!"


#Prompts user to enter time. Repeat until vaild entry.

while True:

        tme = raw_input("Enter start time (hh:mm AM/PM): ")

        try:

          valid_time = time.strptime(tme, '%I:%M %p')

          break

        except ValueError:

          print "Invalid time!"


#Crafts datetime string

startdatetime = str(date + " " + tme)


return startdatetime
```

```
def get_end_datetime(startdatetime):
    """Parse entry date. Set end date to 12:00 AM of the nexy day"""

    #Recover start day
    month, day = startdatetime.split("/",1)
    day, year = day.split("/",1)
    year, time = year.split(" ", 1)

    #Increment the day
    day = int(day) + 1
    day = str(day)

    #Craft string
    enddate = month + "/" + day + "/" + year
    endtime = "12:00:00 AM"
    enddatetime = str(enddate + " " + endtime)

    return enddatetime

def get_messages(driver, CAN_data, base_timestamp, last_timestamp, message_count):
    """Read all CAN messages into the CAN_data array"""

    #Get message table rows
    xpath = '//table[@id="loggedDataTable"]/tbody/tr'
    rows = driver.find_elements_by_xpath(xpath)
```

```
rowCount = len(rows)


#Reading from bottom to top because new data loads at top
for row in reversed(range(rowCount)):
        header, data = rows[row].text.split("M",1)
        #This is the length of an empty record
        if len(data) > 15:
                #Reformat JSON to remove partial data at beginning and end of string
                records, data = data.split("[",1)
                trash, data = data.split("{",1)
                data = data[::-1]
                trash, data = data.split("]",1)
                trash,data = data.split("}",1)
                data = data[::-1]
                #Recreate poperly formatted JSON
                data = records + '[{' + data + '}]}'
                parsed = json.loads(data)
                #Reads timestamp for every record in the row
                for each in parsed['records']:
                        #Assigns base timestamp
                        if base_timestamp == 0:
                                base_timestamp = int(each['timestamp'])
                        #Reads
                        if int(each['timestamp']) > last_timestamp:
                                CAN_data[0].append(each['timestamp'])
```

```
                                        CAN_data[1].append(each['bus'])
                                        CAN_data[2].append(each['id'])
                                        CAN_data[3].append(each['data'])
                                        last_timestamp = int(each['timestamp'])
                                        message_count += 1

        return base_timestamp, last_timestamp, message_count


def start_program_timer():
        """Get current timestamp for comparison when writing CAN messages"""

        current_timestamp = datetime.utcnow()
        current_timestamp = calendar.timegm(current_timestamp.utctimetuple())

        return current_timestamp



def main():

        #Create array to hold CAN messages: timestamp / bus / id / data
        CAN_data = []
        for i in range(4):
                CAN_data.append([])

        #Initialize Chrome Driver with options to disable extensions/passwords
```

```
chrome_options = Options()
chrome_options.add_argument("--disable-extensions")
chrome_options.add_experimental_option('prefs', {
        'credentials_enable_service': False,
'profile': {
        'password_manager_enabled': False
}
})
driver = webdriver.Chrome(chrome_options=chrome_options)

#Initialize vars
base_CAN_timestamp = 0
last_CAN_timestamp = 0
message_count = 0
start_program_timestamp = 0

#set refreshrate if applicable
refreshrate = int(5)

navigate_to_data(driver)
query_data_at_datetime(driver)
vi = init_VI()

while True:
```

```
                base_CAN_timestamp, last_CAN_timestamp, message_count =
get_messages(driver, CAN_data, base_CAN_timestamp, last_CAN_timestamp,
message_count)
                print 'Message Count = ', message_count
                print 'Basetimestamp = ', base_CAN_timestamp
                print 'Lasttimestamp = ', last_CAN_timestamp
                start_program_timestamp = start_program_timer()
                print 'Start Program Timestamp = ', start_program_timestamp
                message_count = write_CAN_message(vi, CAN_data, base_CAN_timestamp,
start_program_timestamp, message_count)
                #Refresh
                #print 'Sleep'
                #time.sleep(refreshrate)
                print 'Refresh'
                driver.refresh()


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

---

openxc_script.py

```
from openxc.interface import UsbVehicleInterface
from datetime import datetime
import calendar
```

```
def init_VI():
        """Initializes VI with JSON formatting"""

        print 'Setting VI'
        vi = UsbVehicleInterface(payload_format="json")

        return vi


def write_CAN_message(vi, CAN_data, base_CAN_timestamp, start_program_timestamp,
message_count):
        """This writes a low-level CAN message to the bus"""

        while len(CAN_data[0]) != 0:

                #Get current timestamp for comparison
                current_timestamp = datetime.utcnow()
                current_timestamp = calendar.timegm(current_timestamp.utctimetuple())

                #Note: must divide OpenXC timestamp by 1000 because it includes ms
                #Note: May have to switch data from unicode to string
                if (current_timestamp - start_program_timestamp) > ((CAN_data[0][0] -
base_CAN_timestamp)/1000):
                        #TODO: Continue trying to write until successful
```

```
                vi.write(bus=CAN_data[1][0], id=CAN_data[2][0], data=CAN_data[3][0])
                CAN_data[0].pop(0)
                CAN_data[1].pop(0)
                CAN_data[2].pop(0)
                CAN_data[3].pop(0)
                message_count -= 1
return message_count
```