Independent Study: Summary Report

Abstract

A Controller Area Network (CAN bus) is a protocol responsible for the communication between various embedded systems within an automotive vehicle. As automobiles advance into the digital age, the CAN bus has become a conduit for cyber-physical attacks that can potentially put a vehicle's passengers at great risk [1]. 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 that discusses 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 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.

Introduction

One paragraph on vulnerabilities of CAN using reference above. How that spurred on this research.

To understand the work done for the Independent Study, a certain level of background knowledge is required involving 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, a brief overview of aspects of their work pertaining to the study is covered. Finally, 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.

- Controller Area Network

The Controller Area Network (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 various Electronic Control Units (ECUs) in a vehicle. Here, ECU refers to an embedded system within a vehicle that sends and receives signals on the CAN bus and uses these signals to control one or more subsystems within the vehicle [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).

While the actual specifics of the CAN communication protocol are beyond the scope of this report, the architecture and message format of the protocol will be introduced at a top-level.

Modern day automotive vehicles will generally employ one or two CAN buses, commonly referred to as high-speed (HS) CAN and mid-speed (MS) CAN, where each bus consists of two wires that act as a differential pair and are terminated with a 120Ω resistor at each end.

The CAN is a shared bus, and all ECUs within a vehicle will typically have access to either the HS CAN, the MS CAN or both. The bus also runs to the on-board diagnostics (OBD-II) connector, present in most modern day automobiles and usually located somewhere under the steering wheel. This port allows for direct access to all CAN messages generated by the various ECUs within the vehicle.

The formatting of a CAN message has many facets, but two portions, the arbitration ID and the data, are most crucial to understanding a CAN message at a basic level. The arbitration ID is a unique 11-bit ID (for standard CAN) manually assigned to an ECU that, like the name suggests, 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 [4]. The data portion of the message consists of at most 8 bytes of ECU data.

Each ECU's arbitration ID and data formatting are proprietary to the maker of the vehicle and may be unique to that vehicle model or to that vehicle year. Recently, however, some auto manufacturers have begun to standardize CAN messages across all their vehicles' ECUs. Still, this information is not made public and, generally, closely guarded for security purposes. This does not mean that they are unknowable. Reverse engineering techniques (some of which will be discussed later) can be used to discover the arbitration ID mapping and the nature of the data formatting for each ECU.

It is important to realize that all ECUs connected to the CAN bus receive all CAN messages transmitted on that bus, where each ECU will filter messages pertinent to its functionality. There is no validation that a CAN message with a certain arbitration ID was actually generated by the ECU holding that ID.

- Hardware-In-the-Loop Simulation

The idea of a hardware-in-the-loop (HIL) simulation is 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. The goal here 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 CAN bus HIL simulation was developed by a team of faculty, undergraduate and graduate students from the University of Maryland throughout the summer of 2015. 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.

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 to see the action for 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. For example,

- 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].

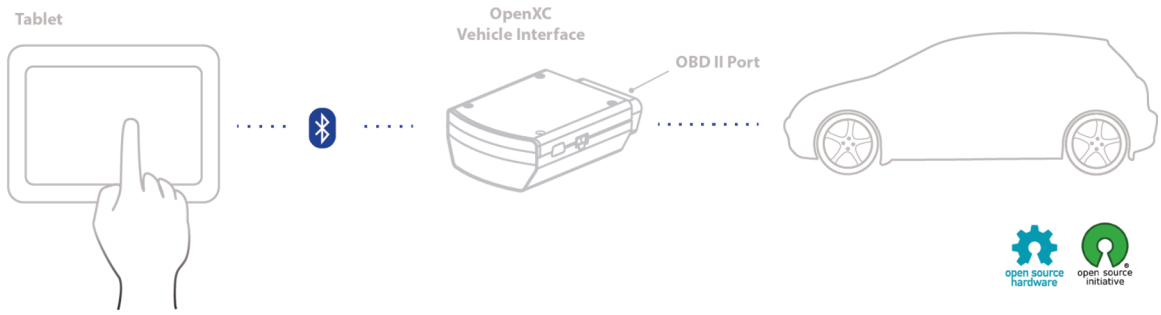

Figure 1: A top-level diagram depicting a connection scheme between tablet, VI and vehicle [6].

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 handle 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.

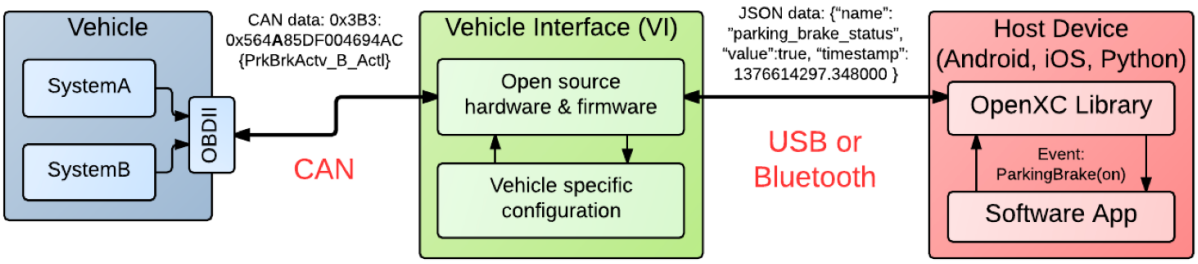

Figure 2: An architecture diagram depicting the transmission of the vehicle's parking brake status as a CAN message via OBD-II port to the VI, and then as a JSON formatted message to a host device via USB or Bluetooth [8].

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 and write 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.

CAN messages could also be issued 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. More on CAN lack of CAN validation? Using the HIL test bench, it was demonstrated that these messages effected ECUs just as any other CAN message would.

Method and Discussion

An Independent Study was started in the Fall of 2016 to further the

- Remote Access to the Vehicle

Building off of the work already done, in the Fall of 2016, a new project was undertaken that leveraged various OpenXC tools to log CAN data from a Ford vehicle and mirrored this data on the HIL simulation in the 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 raw CAN write functionality enabled.

Figure 3: A top-level architecture diagram depicting the transmission of a CAN message from the vehicle to the HIL test bench to allow for 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, he web app was live and ready to receive data. The web configuration file is included in the appendix.

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 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.

Using the same passthrough configuration file for the original VI (discussed in the OpenXC section of the Introduction), the C5 Cellular VI firmware was recompiled. 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.

Vehicle data is transmitted from the C5 Cellular VI in blocks of CAN messages, 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 means it has 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 as the hexadecimal representation of the CAN data, always starting with “0x”.

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

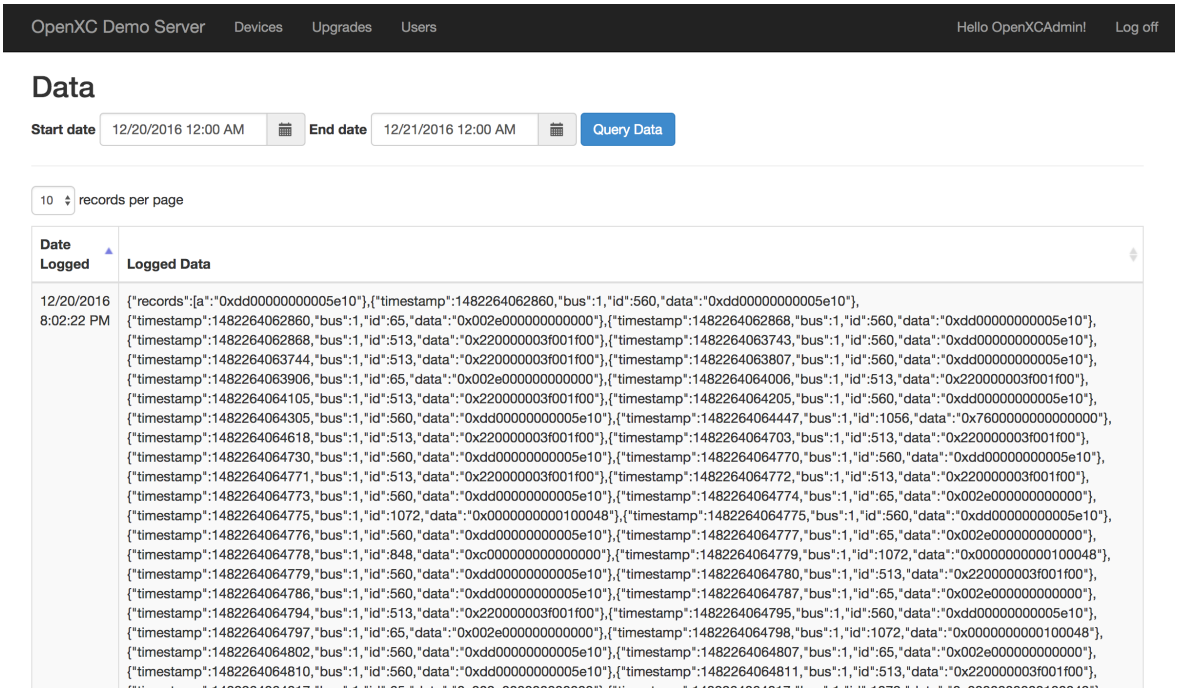

Figure 4: Raw CAN data stored on the server and displayed through the web app. The date selection can be seen at the top of the page and a timestamp for each block transmission is along the left side. Each individual CAN message has JSON fields for its own timestamp, bus, arbitration id and data.

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. From here, the user selects the date and start time on that date from which to start scraping data from the server. It arranges the data to show up to 100 transmission blocks per page with the newest block at the top of each page. The program then begins reading in block data from the bottom to the top of the page, checking the timestamp of each message against the start time indicated by the user. Before this occurs, however, the program must reformat the data in each block to comply with JSON format. As the data is displayed in each block, messages are often cut off at the beginning and end of the block leaving partial data. These portions are removed and then scanning resumes.

Once the program finds a message with a start time later than the one indicated by the user, it parses the four JSON fields of that message into a 4-dimensional array. It then continues scanning 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.

Discuss live test.

- Playback of an Assist Feature

Automated driver assist features are electronic systems in a automobile that use input data from one or more sensors on the vehicle to help the driver in some aspect of the driving process. Typically, each assist feature is designed to either automate some process or to enhance the vehicle's safety.

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 the assist features.

As part of this study, the 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, either parallel or perpendicular, and then notifies the driver once a spot has been found. The driver then puts the vehicle into reverse and removes their hands from the steering wheel. The car maneuvers into the spot while the driver controls the speed via the accelerator and brake pedals.

- Reverse Engineering OpenXC

Conclusions

- Too much work for one person. Needed at least one other person to examine assist feature for safety reasons.
- Lack of experience with Visual Studio and web development turned out to be a big problem. One should have been a quick deployment turned into a 3 week debugging session all due to a simple syntax error in the connection string.
- Have become very familiar with many aspects of OpenXC. Contributed to open source FW development by discovering bugs and issuing pull requests.
- Have become much more familiar with Python and the useful Selenium WebDriver.

- Got a taste of IDA Pro. Here too, one more person would have been crucial. No experience with this software. Made decent progress for relatively little time that was put into this part of the study.

References

1. http://www.autosec.org/pubs/cars-oakland2010.pdf
2. http://www.ti.com/lit/an/sloa101a/sloa101a.pdf
3. 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