THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

DEPARTMENT OF SUPPLY CHAIN & INFORMATION SYSTEMS

The Dangers of Malicious Hacking in Modern Day Automobiles

Michael Dulan
Spring 2015

A thesis
submitted in partial fulfillment
of the requirements
for a baccalaureate degree
in Management Information Systems
with honors in Supply Chain & Information Systems

Reviewed and approved* by the following:

Dr. Robert A. Novack
Associate Professor of Supply Chain Management
Thesis Supervisor

Dr. John C. Spychalski
Professor Emeritus of Supply Chain Management
Honors Adviser

* Signatures are on file in the Schreyer Honors College.

ABSTRACT

The purpose of this study is to understand how automobiles are vulnerable to cyber-attack, to identify the level of control outside parties can secure over a vehicle, and to determine the feasibility of an individual with malicious intent to put those using our transportation system in danger. The question of whether outside parties can access the network inside an automobile has already been answered. IEEE has a very stunning paper about their research into controlling a car remotely and proved that it was possible to enter the network a number of different ways (Checkoway, 2014). It is this study and the research performed by Dr. Charlie Miller and Chris Valasek, outlined in their paper "Adventures in Automotive Networks and Control Units," that has largely inspired this thesis (Miller, 2013). This paper will explain in simpler terms how networks are run in modern vehicles, where they are most vulnerable, what an attacker with access to a vehicle network is capable of, and how these networks can be improved to ensure safer road transportation. No cyber-attack has ever been recorded so far by vehicle owners or manufacturers. However, there is no detection system currently to record an attack should one occur, and likely if an attacker caused an incident it would be thought of as a malfunction. Through analysis, it is clear that the addition of detection systems and improved security is necessary to ensure customer safety. It is up to car manufacturers to determine what level of improvement customers will expect and be willing to pay for moving forward.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Background

Vehicle networks have expanded rapidly in the past few decades. As more and more electronic systems presented themselves in vehicles the number of wired connections became a problem. This is not only because of the sheer number of connections, but because the weight of the metal from wiring throughout the vehicle was negatively impacting performance. To combat this problem a type of network was implemented called the Controller Area Network, run on a single twisted wire pair connection between all of the Electronic Control Units (ECUs) of an automobile in which all system traffic is sent and received. The wires and connections to these ECUs together are called the CAN-bus. Many machine networks operate with CAN besides automobiles, such as elevators, boats, airplanes, and industrial equipment.

Research into the basic frameworks and set-up of CAN and automobile networking and communication is largely available to the public. Specifically, diagnostic information is standard in all vehicles and is needed for mechanics to understand what is broken on a vehicle in need of repair. A number of tools are available to interact with vehicles through their diagnostics port located under the steering wheel of all vehicles. Diagnostic messages themselves are simply for the purpose of mechanics; therefore these messages, when sent to a vehicle, will have no actual effect on an automobile. This diagnostics port, however, is an access point to the overall network of the vehicle as well, which contains many other important messages transmitted between different controllers in the vehicle.

These non-diagnostic messages provide important information to the ECUs of a car so they may perform their tasks. Modern automobiles can have up to forty separate ECUs that control everything from seat-belt tightening in the event of a crash to activating the anti-lock brake system. Thankfully, the messages sent to ECUs which cause a vehicle to alter behavior are proprietary and kept a closely guarded secret by vehicle and systems manufacturers to prevent people from altering their systems. Therefore, the only means to discover the meaning of data that would cause vehicle behavior as altering information on the dash, turning off anti-lock brakes, or shutting down the engine due to a critical failure is through analyzing traffic on the network as the vehicle is running, or by reverse engineering an ECU.

Few vehicle owners have the skills or time necessary to reverse engineer their systems, but the select few have made a number of modifications, such as integrated turbo additions to engines and improved sound systems. The unintended ability to make additions to the CAN-bus has some customer value, but the ease of entering the network and being able to make upgrades in this manner also leaves a door open to intruders. Research has proven that these doors are numerous and extend wirelessly beyond physical access points like the On Board Diagnostics (OBD-II) port under the steering wheel. These doors include wireless transmitting from the Tire Pressure Management System (TPMS), the On-Star system, Bluetooth amenities, telematics, and MP3 and other media amenities (Barry, 2011). Research in this paper will largely consist of analysis conducted from a direct connection. However, wireless intrusion can have the same capabilities once a connection to the bus is established.

Watching network traffic directly from an exterior device, such as a laptop, requires a means to transform live data to a format understood by a computer. Special connectors that attach to the OBD-II port and perform this task are only moderately expensive. However, the ability to receive data means nothing without knowing how to organize it. Interestingly, this is rather straight forward as instructions on a CAN-bus are transmitted in a very uniform way. One would only have to write a simple script to organize the data into the fields specific to CAN packets and print the fields in real time to be able to match instructions with actual actions of the vehicle. Software is also available for this purpose, but can be expensive, and is not necessary to anyone with network programming experience.

Every manufacturer has different identifiers for types of messages sent across the bus, and communication can differ drastically even between yearly models of the same type of vehicle. This is great news for the security of our own vehicles as discovering which instructions perform which actions can be a time consuming process for just one automobile. To map the traffic on a large number of vehicles in this way would be a daunting task and one that would require access to each specific make, model, and year of vehicle. However, an entire understanding of the information on a CAN-bus is not necessary to cause havoc in an automobile. In fact, problems can be caused without knowing any of this information at all.

The ability to read data is not dangerous in itself, but CAN in automobiles has a fatal flaw that leaves vehicles vulnerable to attack. Traffic across the network is accepted from all sources, which could include a laptop or other device that is attached to the network, either directly or wirelessly. If an identifier for an important system is known, a message with false information could be fabricated and sent over the network to alter the vehicle's behavior. With the appropriate set of commands an individual could even rewrite the behavior of an ECU.

Simple methods of protection would provide the automobile CAN-bus with a much-needed added level of security. For instance, encryption of traffic would add yet another obstacle to a hacker. The need to register a device with the network with a specific command before accepting information from it would be another avenue. Protection of this network has largely gone without research as vehicles were never thought to be in danger from cyber criminals until recently. It is important for vehicle manufacturers to take security into account for the modern CAN-bus, especially since wireless access points are increasing in vehicles each year.

The remainder of this paper will go into the breakdown of CAN-bus communication in detail, specifically the network traffic of a 2007 Lexus RX350. From this analysis current security measures will be evaluated and recommendations for safer communication will be provided.

Chapter 2

Analysis

Introduction

“Car Hacking” is a popular topic among media outlets today, usually highlighting what people can do to cars remotely through various channels described in the introduction. However, a large topic that is left out of these reports is how these results were achieved and most segments leave the public with the idea that controlling any car could be done quickly with an iPhone with no prior exposure or identification of the vehicle. Reality is slightly less frightening. Despite the lack of coverage on the topic, numerous security measures exist on automobile CAN buses. Bypassing them is not as simple as one would think by listening to NBC news and requires a detailed knowledge of the subject.

Most of the research for this thesis has been performed on a 2007 Toyota Lexus RX350, which has been a rather popular car in the past several years and will suffice as a good control. This section will discuss how the interface was prepared with the vehicle from a personal computer and also discuss the various levels of security found on the network. Then the discussion will relate this network in its similarity to other vehicles. Thus, a common degree of security can be confirmed across several manufacturers.

CAN in the Automobile

Though CAN was designed to require only one wired connection throughout an entire network, modern automobiles do not simply run on one bus alone. A picture of the basic structure of a CAN network can be found in Figure 1.

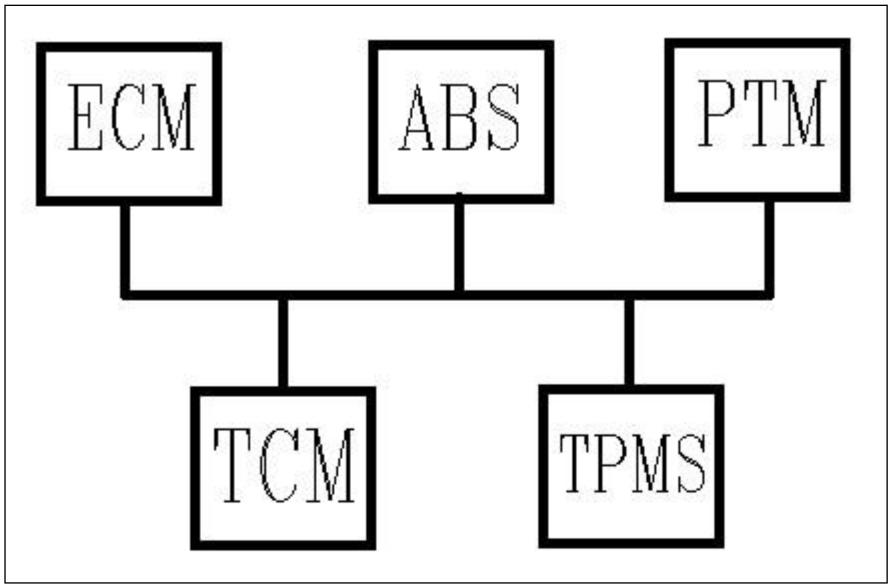
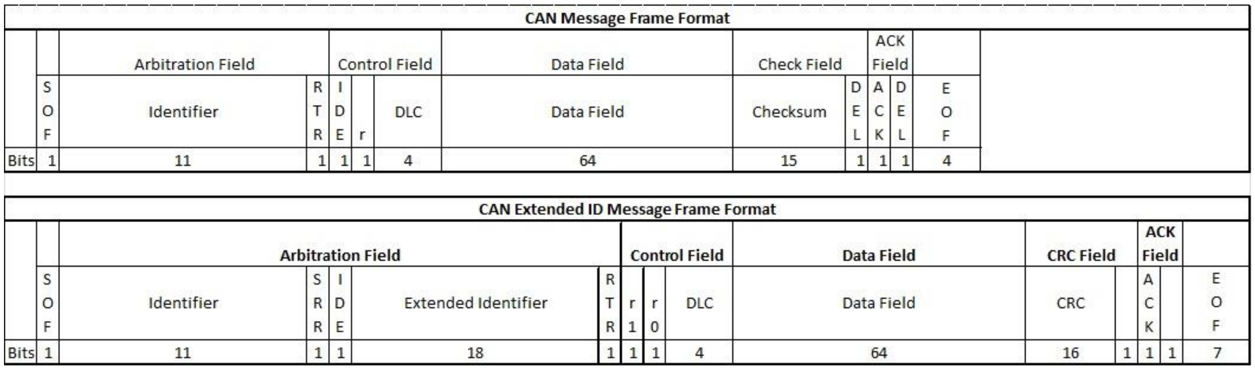


Figure 1. Vehicle CAN bus

As can be seen, only one pair of wires connects all devices and transmits all information to and from every ECU. Therefore all communication can be read from any access point to this wire and all messages are transmitted, though not necessarily accepted, by every ECU. Usually there are at least two different speed CAN-buses in a vehicle that run communication between modules separately. The reason behind this is partly to separate critical functions of an automobile with the non-critical and prevent errors in non-critical aspects to affect critical modules. A high-speed network always exists which runs communication from the engine and other key functions as the Anti-lock Braking System (ABS), Transmission Control Module (TCM), Powertrain Module (PTM), and other important control units that need real time information down to the millisecond. A low-speed bus would control aspects such as power windows, entertainment systems, and door locks. These systems are usually bridged by connections to a singular ECU allowing signals to be sent from one network to the next, but this is more extensive than the experiments conducted in this thesis.

CAN Messages

There are many types of CAN messages on a network, but all have a similar structure that can be used to easily identify one type from the next. The most important aspect in understanding CAN is to know the two main parts that make up a message: the Identifier, and the Data field. The packet structure that makes up a typical message can be seen in Figure 2.



CAN Message Frame Format

		Arbitration Field		Control Field			Data Field	Check Field		ACK Field		
	SOF	Identifier	RTR	IDE	r	DLC	Data Field	Checksum	DEL	ACK	DEL	EOF
Bits	1	11	1	1	1	4	64	15	1	1	1	4

CAN Extended ID Message Frame Format

		Arbitration Field					Control Field			Data Field	CRC Field		ACK Field		
	SOF	Identifier	SRR	IDE	Extended Identifier	RTR	r1	r0	DLC	Data Field	CRC		ACK		EOF
Bits	1	11	1	1	18	1	1	1	4	64	16	1	1	1	7

Figure 2. CAN Message Frame Formats

Both Message Frame Formats above can be expected from passenger automobiles, though the typical small number of proprietary information message types usually only requires the 11-bit identifier format, as is used in our control vehicle. The only two fields that will be manipulated in this paper are the Identifier section of the Arbitration Field and the Data Field, as the construction of the other parts of the CAN packet will be handled by a third party device discussed in the next section.

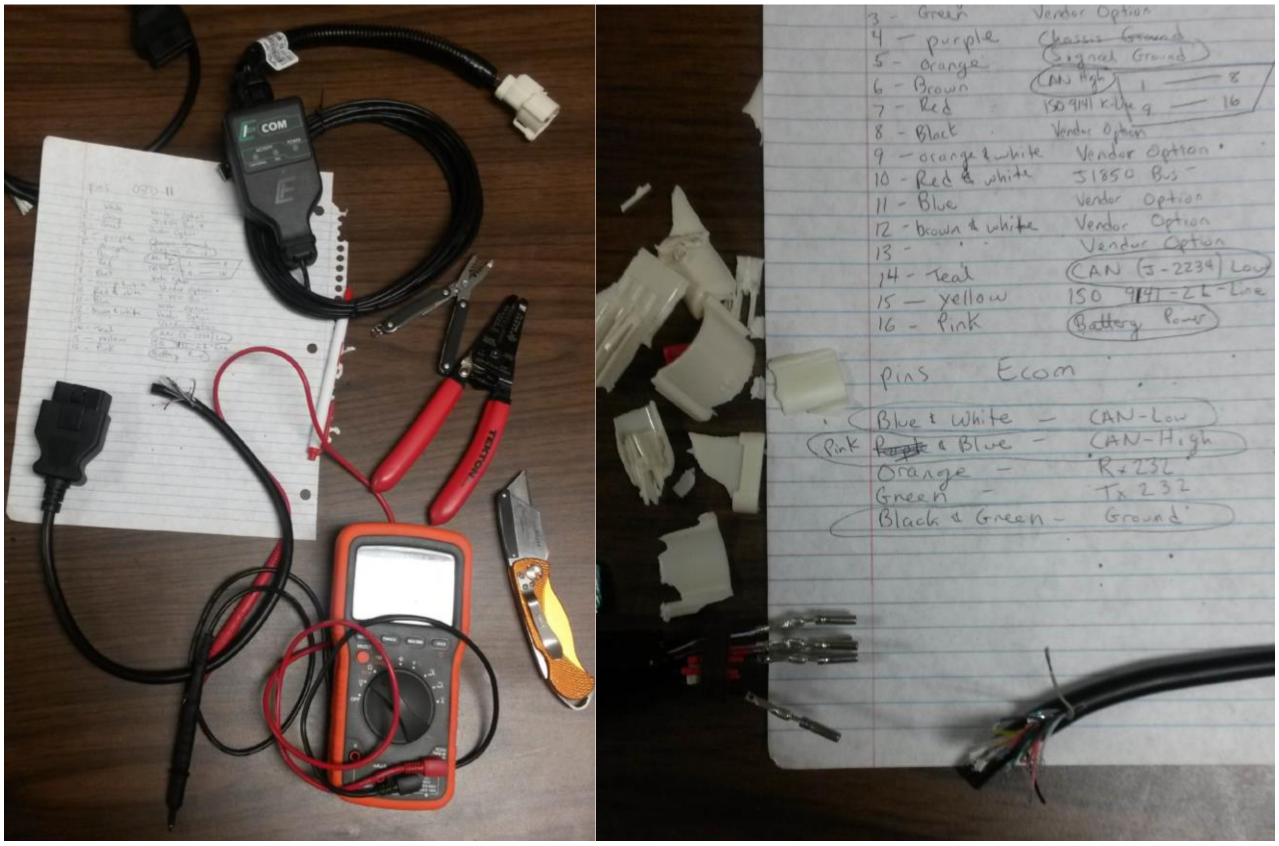
Identifiers in CAN packets hold a significant role in message flow as ECUs accept messages with a priority queue. Therefore, messages with more recessive priority identifiers are accepted first as they hold more important information. Identifiers also determine the message type and tell an ECU if it should and how to interpret information in the data field. Information in the data field will be presented uniquely for each message type (ISO 15765 CAN Diagnostics, 2012).

Experimentation Set-up

To begin gathering data a device called an Ecom Cable, developed by EControls Inc., that was also used by Miller and Valasek was bought for approximately $200. This cable allows for CAN messages to be interpreted by a laptop and vice versa through a USB connection. The Ecom Cable is created with an EPC connector that is designed to attach to the engine harness. However, in order to attach this device to the OBD-II diagnostics port under the dash a unique connector was created. As the vehicles used in this experiment for testing are owned by various third parties this was the easiest access point to the network without having to remove parts of the dash or under the hood.

An OBD-II cable was acquired and cut open to attach to the EPC end of the Ecom Cable. The materials used are shown in Figure 3. Wires can be easily matched to pins on the OBD-II cable by using a multi-meter and high gauge wire strippers since OBD-II wires are very small.

Only connections between four OBD-II pins and three Ecom wires are necessary to enable



3 - Green	Vendor Option
4 — purple	~~Chassis Ground~~
5 - Orange	Signal Ground
6 - Brown	CAN High
7 - Red	ISO 9141 K-Line
8 - Black	Vendor Option
9 — orange & white	Vendor Option
10 - Red & white	J1850 Bus
11 - Blue	Vendor Option
12 - brown & white	Vendor Option
13 -	Vendor Option
14 - Teal	CAN (J-2234) Low
15 — yellow	ISO 9141-2 L-Line
16 - Pink	Battery Power

1 — 8
9 — 16

Pins	Ecom
Blue & White —	CAN-Low
Pink ~~Purple~~ & Blue —	CAN-High
Orange —	Rx 232
Green —	Tx 232
Black & Green —	Ground

Figure 3. Materials and Identifying Pinouts

communication. Steps for the creation of this connector are shown in Figures 4-6.

After creating the connector the windows drivers for the Ecom Cable can be downloaded off of the cancapture.com website.

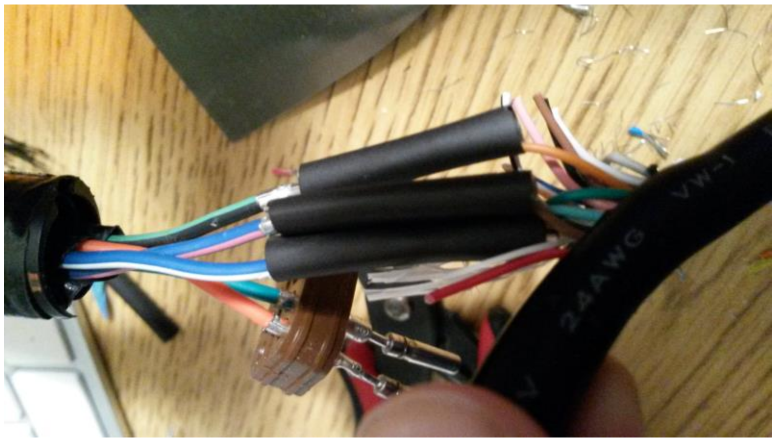


Figure 4. Attaching the Ecom to OBD-II

The drivers are known to be successfully installed if the light on the Ecom turns from red to green when plugged into the computer. A picture of the connector attaching the Lexus to a laptop can be found in Figure 7. Once connected a simple written script allows easy access to the traffic flow present on the CAN-bus of the vehicle. Although the Ecom device can only transmit or receive data at one time, a second device could be connected to perform both functions at the same time. For this experiment that was not necessary.

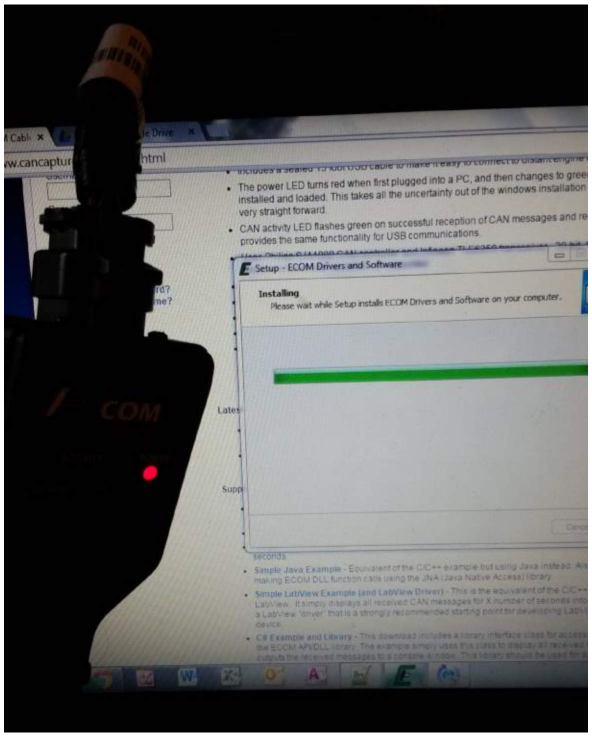

Setup - ECOM Drivers and Software
Installing
Please wait while Setup installs ECOM Drivers and Software on your computer.
COM

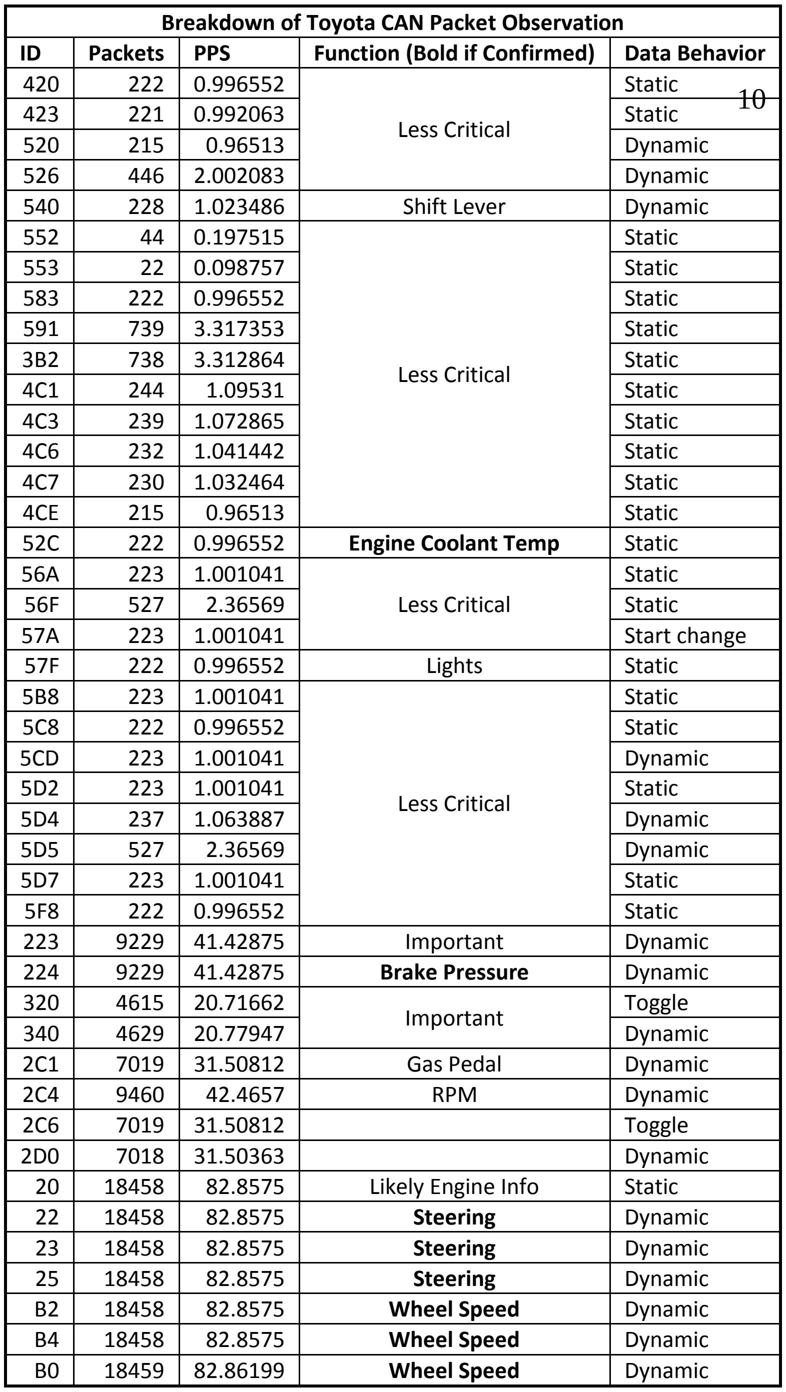
Figure 5. Installing Ecom Drivers

Data Analysis

It is important to note that nearly all messages on the CAN-bus are not commands. The messages describe the state of the car and do not tell the car to go faster or to turn the steering wheel to a specific position. However, these messages are necessary for ECUs to enact their internal functions. An example would be if the car is slipping on ice, determined by information on the bus pertaining to each tire’s wheel rotation and speed, to perform anti-lock braking.

Intercepted messages, structured as 11-bit identifier CAN packets, were carefully analyzed to determine their meaning. The difficultly of identifying the meaning of unique message types is due to the complexity of cars today. For instance a change in one message when you press the brake does not necessarily mean that that message has anything to do with the braking module. In fact, it could be intended for the cruise control module which would turn off, since the brake was pressed. Single messages can also hold multiple segments of information about the car beyond a single aspect. Theoretically every single bit in the data field of a CAN message could tell the status of a different state of the vehicle and there are 128 separate data bits per message type. Many different buses also exist in a vehicle of various message speeds and not all are connected. Therefore, it cannot be expected to see a difference in a message seen in the high speed bus when the seat warmers are turned on, the door is ajar, or the lights are turned on. It is more likely that those non-crucial functions are controlled by a low-speed bus.

Through several 20 minute trips out in the Lexus, 43 unique types of messages were identified from our access point. Specific confirmed messages identified included wheel speed, steering position, brake pressure, and engine coolant temperature, though the confirmed meaning of other messages proved elusive. The data was interesting in that messages of low, moderate, and high importance were easily identifiable since they are broadcast at vastly different rates. The breakdown of this data can be found in Table 1.



Breakdown of Toyota CAN Packet Observation				
ID	**Packets**	**PPS**	**Function (Bold if Confirmed)**	**Data Behavior**
420	222	0.996552	Less Critical	Static
423	221	0.992063		Static
520	215	0.96513		Dynamic
526	446	2.002083		Dynamic
540	228	1.023486	Shift Lever	Dynamic
552	44	0.197515	Less Critical	Static
553	22	0.098757		Static
583	222	0.996552		Static
591	739	3.317353		Static
3B2	738	3.312864		Static
4C1	244	1.09531		Static
4C3	239	1.072865		Static
4C6	232	1.041442		Static
4C7	230	1.032464		Static
4CE	215	0.96513		Static
52C	222	0.996552	**Engine Coolant Temp**	Static
56A	223	1.001041	Less Critical	Static
56F	527	2.36569		Static
57A	223	1.001041		Start change
57F	222	0.996552	Lights	Static
5B8	223	1.001041	Less Critical	Static
5C8	222	0.996552		Static
5CD	223	1.001041		Dynamic
5D2	223	1.001041		Static
5D4	237	1.063887		Dynamic
5D5	527	2.36569		Dynamic
5D7	223	1.001041		Static
5F8	222	0.996552		Static
223	9229	41.42875	Important	Dynamic
224	9229	41.42875	**Brake Pressure**	Dynamic
320	4615	20.71662	Important	Toggle
340	4629	20.77947		Dynamic
2C1	7019	31.50812	Gas Pedal	Dynamic
2C4	9460	42.4657	RPM	Dynamic
2C6	7019	31.50812		Toggle
2D0	7018	31.50363		Dynamic
20	18458	82.8575	Likely Engine Info	Static
22	18458	82.8575	**Steering**	Dynamic
23	18458	82.8575	**Steering**	Dynamic
25	18458	82.8575	**Steering**	Dynamic
B2	18458	82.8575	**Wheel Speed**	Dynamic
B4	18458	82.8575	**Wheel Speed**	Dynamic
B0	18459	82.86199	**Wheel Speed**	Dynamic

Table 1. Lexus Data Breakdown

Network Manipulation

After confirming a few types of messages, manipulated packets of the confirmed types were created and sent over the network with rapid speed to fill the ECUs priority queues with our data and less of the correct data. As sending the wrong messages over the network could have an adverse effect on the tested vehicles only packets confirmed to do no damage were sent.

The first attempt was to simply alter a gauge on the dashboard to see if the ECUs would accept messages from an outside source. The engine coolant gauge was chosen as a relatively harmless metric to alter artificially for a small period. A script called Send.c located in Appendix A, was created to send messages over the network that the engine coolant temperature was as low as possible. It was determined that the first byte of the data field contained the actual temperature information and thus the test message contained a byte value of 00. The range of the temperature in the first data byte of this message is unknown. A value of 00 could mean 0 degrees Fahrenheit or -23 degrees. However, this is the lowest value possible to send and thus should show a decrease in temperature on the gauge visibly to prove the message worked. After connecting to the OBD-II port, however, and sending the message at a rapid rate of 200 packets per second there was no response on the dash.

Looking at the history of this packet it was interesting to see that the second byte of the data field increased by a single digit with each message. The script was altered to incorporate the second byte of the data field to then be a rolling counter and the test was run again. This time the gauge showed a rapid decrease barely pumping up every so often when a true packet was recognized. The system was fooled to thinking the oil temperature was much lower than it was. Thus, it is proven that messages can be created and accepted by ECUs on the network as valid information through the diagnostics port. Figure 8 shows a laptop communicating with the vehicle.

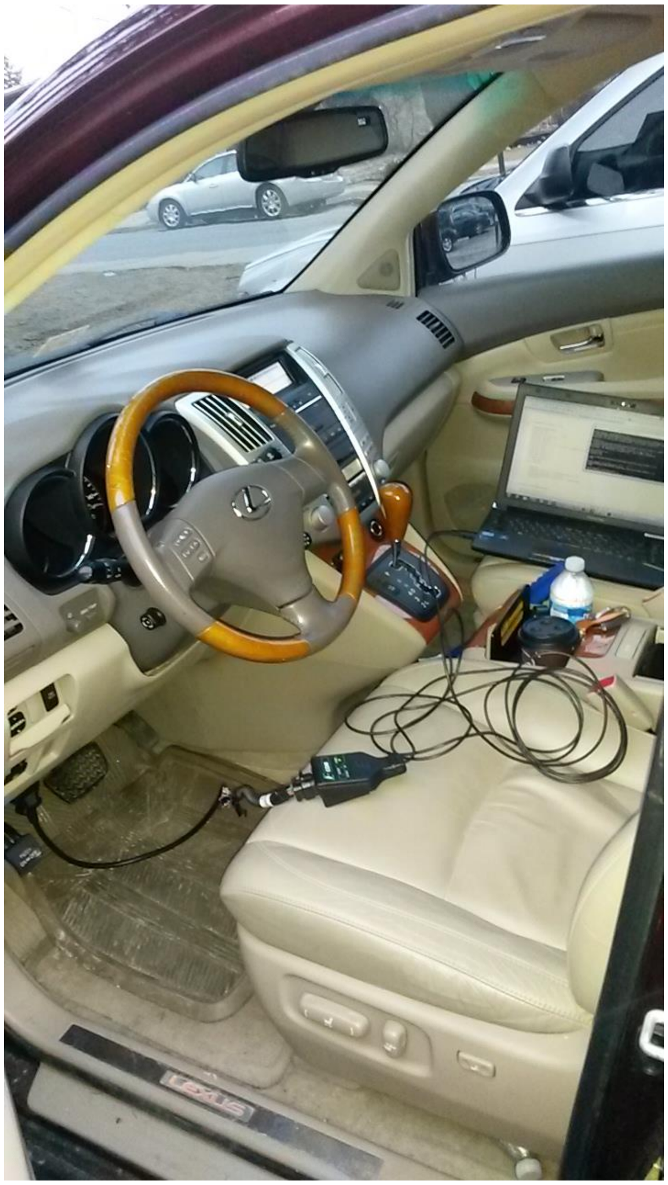


Figure 6. Laptop to Vehicle communication

The counter byte was interesting, however, in that it had no purpose other than to verify message flow. This byte, though not necessarily intentionally, acts as a security measure inside the engine coolant CAN packet against bad data as the ECU will only accept continuous information if the counter is present. After further analysis into this concept multiple other message flow bytes were evident beyond simple counters. Without these bytes in the correct format information is not recognized as valid by an ECU. An evident control flow byte is found in the last byte of a wheel speed message as well. This type of byte must equal the rolling addition of all other bytes in the message, and the packet will not be accepted as valid information unless this is the case.

With this discovery, it was necessary to determine if these control flow bytes were present in other vehicles and if the location and function of each was constant, and thus could be predetermined by attackers. Traffic from several other vehicles was observed, including a Honda, Hyundai, and a Subaru.

Through comparison of wheel speed messages in each of these vehicles, control flow bytes were present in each of them of the same addition type. However, the location of this byte varied among vehicles. As records of information such as this are not existent without being self-created, having access to many years, makes, and models of vehicles, these bytes could prove to be substantial roadblocks in sending accepted messages to ECUs.

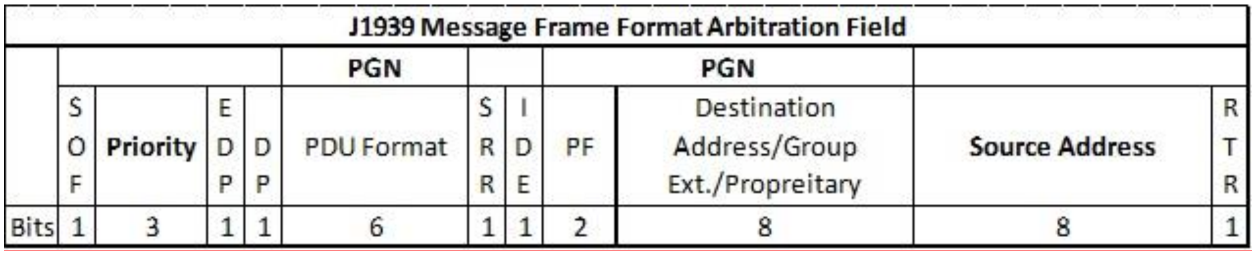
Network Message Scrambling

Experimentation discussed thus far has been focused on attempting to perform specific actions to a vehicle. However, a faster and more realistic attempt to cause a car to malfunction would be through sending random data at the network with the intent to send a message that would cause a failure in a critical component of the vehicle. On-Star is known to be able to send a specific message that would safely shut down an engine, but malformed information on a network could cause ECU and engine failures that could result in significant damage to the vehicle (Klimas, 2014).

Attempting to break a device through malformed or malicious data is a common practice that programmers use to test durability and security in their software. Thus, there are numerous free applications that are easily accessible to those who would attempt to use them in an illegal manner. Experimentation was needed to determine how vulnerable ECUs are to malformed data.

An ABS ECU from a truck was acquired to measure the robustness of the controller to attacks with malformed information. An open-source fuzzer, called Peach was selected as a good method to inject malformed packets into a home-made CAN-bus hooked up to the ECU. Peach was decided to be one of the free, popular, easiest to customize, and fastest methods of creating and sending test cases to the ECU. Peach also has built-in ability to recognize faults that occur with a device and will log each event since it is difficult to see changes at the speed of data sent over the network. A custom Peach pitfile that fits the data length of CAN messages was created to send mutated CAN data and discover any faults that may occur when an ECU is presented with rapid amounts of random information. The ECU was monitored to identify if it would send data different than the two standard Electronic Brake Controller 1(F001) and Wheel Speed Information (FEBF) messages that it communicates when monitoring normal traffic. The Peach pitfile can be found in Appendix A.

As this ECU was from a truck it utilized a specific CAN-Extended ID message frame standard developed by the Society of Automotive Engineers called J1939. This format is well documented online, and is also used on boats, airplanes, and many other large and industrial vehicles. The format of a J1939 packet is very similar to the standard CAN extended message frame format; however, the identifier is more specific. A detailed look at the beginning of this packet standard can be found in Figure 9.



J1939 Message Frame Format Arbitration Field

					PGN			PGN			
	SOF	**Priority**	EDP	DP	PDU Format	SRR	IDE	PF	Destination Address/Group Ext./Propreitary	**Source Address**	RTR
Bits	1	3	1	1	6	1	1	2	8	8	1

Figure 7. J1939 Identifier Format

J1939 information packets are distinguished by their 29-bit identifiers that are divided into four parts that this report is concerned with. The beginning 3 bits after the Start of Frame bit, as custom with CAN packets, contains the priority of the message. The interior 16 bits, divided by the substitute remote request and identifier extension flags that are not part of the identifier, represent the message type called the Protocol Group Number (PGN). The remaining 8 bit section contains a source address from the ECU or device of which the message originated (Understanding SAE J1939, n.d.). Other types of messages exist in the J1939 standard, such as error messages and information request packets, however, this is the identifier format that contains normal information read by vehicle ECUs, and will be the easiest message type to be manipulated to provoke vulnerabilities.

While manipulating this format in each round, testing over several days was conducted with all of the data mutators that are provided with Peach. No faults, responses, or alteration of data were detected from the ABS unit. It was determined that only a recognizable CAN packet would be accepted by the ABS to trigger a possible reaction. Thus, a Peach custom mutator and several strategies were created that would send packet structured data that is familiar with the ABS. These files determine how Peach creates test cases to send to a device and are easily included into the community Peach-nightly-build freely available online. A custom Peach mutator and strategy are included in Appendix A. With these mutators, identifier and data bytes could be configured to remain static or increment dynamically to narrow the test cases that are sent at the ABS unit. As J1939 CAN packet message frames consist of 12 hexadecimal pairs of data, about 256^{12} packet combinations are possible. With peach running test cases at a rate of 200 packets per second, it is easily understood why narrowing the information to include static hex bytes and thus minimizing the exponent value is necessary. Cycling every combination of three byte pairs would take nearly one day. Though this rate of data entry seems exceedingly slow, part of the restriction is due to inefficiencies in the code and the rate our Ecom device can send data onto the network.

As the PGN in a J1939 packet is only two hex-bytes long all possible message types were cycled with different data that was thought to possibly provoke a response from the ECU. It was thought that this would be the most likely way to trigger a fault in the unit. Numerous data values and some mutated data bytes were tested for several weeks. Fuzzing, in this timeframe, proved unsuccessful to affect this singular device and it was decided to attempt specific packet injection known to J1939 that should be successful in this regard. Fuzzing was not attempted on an actual vehicle network as it was determined too dangerous by vehicle volunteers.

Packet Injection

Though sending malformed and large amounts of random data to find specific harmful messages is not difficult, the ability to input packets at will to a network has other effects more likely to cause harm. By sending accepted but incorrect high priority data quickly, an ECU queue can fill up with incorrect information. Intrusion similar to a denial of service attack on a website is possible and could result in malfunction of a vehicle. For instance, if a message that measures ignition timing is being sent over the network to the ECM (Engine Control Module) and is required to wait in a long message queue by many false messages to the ECM, the ECM may have an inability to detect knock, a potentially destructive condition for an engine.

An accepted message by an ECU must still meet variable checksums built into the packets as described above and nearly all messages to important units like the ECM require them. However, by simply copying a pasting message on the bus, important information can be delayed to a unit. This type of attack is the easiest to perform as it requires little knowledge of a CAN network. Simply sending static data in a message of a type with regularly variable data can delay key functions for a vehicle such as valve control, air/fuel ratio, and shifting.

Though car manufacturers are installing many helpful automatic features into their new automobiles, these features could put drivers at risk from this type of attack. By listening to packet traffic when a vehicle uses a backup collision intervention system, as in the 2015 Infinity QX60 commercial, it would be possible to replicate this traffic and cause braking when a vehicle should not be (Infinity, 2015). The dangers of these features can be further described in the Forbes report with Charlie Miller and Chris Valasek when they convinced a vehicle to perform park assist functionality when driving down the road (Greenberg, 2013).

ECU Re-flashing

The most dangerous ability a hacker could obtain on a vehicle system would be accessing writing privileges to an ECU. Previous discussion has been focused on misinforming ECUs by providing false information and causing them to behave as if the vehicle were in a different state than it actually is. However, with a successful attack of this nature an ECU would perform as an attacker would program it to when provided normal information.

With the ability to re-flash, or re-program, an ECU an attacker could alter the functions that the ECU does to perform physical actions in a vehicle. For instance, a function could be input into the ECU that controls braking to apply maximum pressure to brakes once a message has been received that the vehicle has reached a speed of 70mph. Alternatively, an attacker with access to this privilege could simply erase all of the instructions on an ECU. This ability clearly would be extremely dangerous for an outside party, or even a vehicle owner, to be able to access, therefore, this capability has very secure security measures around it.

Car manufacturers utilize a seed/key security system for this purpose in that after a request for access an ECU will send a message containing a seed (bit sequence). A key can then be determined from using this seed in a proprietary algorithm and transmitting the key (algorithm output) within a certain amount of time. There are multiple levels of security that manufacturers utilize to keep others from beating this security. The lowest level would be using a simple math algorithm that causes a key to be the seed plus or minus some secret number. Some ECUs change the seed or even prevent access entirely until the vehicle is restarted after a certain amount of incorrect attempts, making discovery of the algorithm more difficult. The highest level of security that has been observed is an altering of the algorithm itself as well as the seed after a certain number of attempts have failed. Without utilizing a manufacturer’s tool to attempt to re-engineer and discover the algorithms involved in this sense it would be very difficult to access vehicle privileges in this manner (Rouf, 2010). It would be recommended that all car manufacturers extend their security to the highest level currently available for writing access.

The time and resources it would take to determine the correct algorithm and find the key to unlock an ECU would make it extremely improbable for an individual to access the internal functions of a random passing automobile. Therefore, everyday drivers should be reassured that there is little need to worry about outsiders re-programming their vehicles on the road. However, previously discussed data manipulation is still a valid risk.

Chapter 3

New Proposed Security Measures

Intrusion detection

A simple addition to security in the vehicle CAN bus would be the addition of an intrusion detection system on the network. Not only would this provide a warning to drivers that their car could be behaving strangely due to message traffic intrusion, but also there would also be a way to actually tell if automotive cyber intrusion is occurring today in society.

The installation would be relatively cheap and simple for this addition as all would be needed is a function that could be input to an already existing ECU, small amounts of dedicated memory to capture network traffic during a detected time of intrusion, and a new light on the driver dashboard. Thus, a vehicle manufacturer could analyze the traffic to determine if abnormal network behavior was the result of an intrusion or a malfunctioning ECU or sensor.

Detecting unusual behavior on the bus would be rather simple. As shown in Table 1 only a small number of identification numbers are ever present on the bus relative to the thousands of combinations possible. Also these IDs are transmitted at a specific rate of speed. If a number of CAN messages that contain unused IDs or the packet traffic of specific used IDs begins to exceed normal transmission rates this can be evidence that the network is being manipulated. When conducting experimentation it was determined that only by sending messages faster than normal data could the ECU be fooled enough that false instructions would affect the car. Therefore, intrusion that would be detrimental to consumer safety would be easy to spot by this means.

Encryption of network traffic

Encryption may be the oldest tool used in network security and it is curious why automotive networks do not already use it in their systems. Encryption, even in the simplest form with a singular proprietary key would be another obstacle for a would-be hacker to circumvent. Though, automobile manufacturers would need to work together with ECU manufacturers, custom encryption with public keys for each vehicle would be the safest option. Encryption keys could be included in the user manual of a car or even be proprietary numbers unknown to anyone but the manufacturer. Encryption in this sense would take long periods to crack without knowledge of the key and prevent outside signals from providing misleading information on the bus.

Encryption of CAN messages is not a new concept. In fact in some systems it is present, as in the CANlink GSM (ProEmion, 2014). Patents also exist for CAN communication encryption devices (Vasicheck, 2009). The only step left is actually including these controllers and security measures in vehicles. Investment in this existing technology will have backlash from those who customize their own vehicles by adding their own devices, such as a radar detector light to a dashboard (Szczys, 2013). However, this security could be crucial to preventing attacks.

Consumer Awareness

It is important to note that a car owner can take steps to safeguard their vehicle from the threat of hackers by utilizing protection already built into the vehicle that they may be unaware of. Bluetooth in particular requires a four digit password for connection to pair with an outside device. This password is usually default to 0000 or 1234 (Kumar, 2013). Simply changing this password to a different number could prevent easy access by another individual. Additionally, illegally downloaded music attached to an MP3 player could be malicious, and it would be wise not to connect a device with such material to your vehicle (Koscher, 2014). Consumers should be instructed about safety measures that require their input upon receiving a vehicle.

Chapter 4

Conclusions

Through analysis of available vehicles this thesis has explored automobile cyber vulnerabilities for malfunction of ECUs due to malformed packet injection, specific targeted packet injection on a vehicle CAN bus, network dilution to prevent reception of important vehicle information, and security to unlock ECU developer permissions. It has been determined through this research that malicious hacking to create a safety threat for drivers of modern automobiles is feasible for someone with enough drive to try to cause harm in this way. However, even though this threat is feasible the chance of causing physical harm to a driver or passenger seems relatively low, even if a failure causes damage to the vehicle.

This paper has outlined the security measures present in several automobiles that an average driver would own. It is a relief to know that security measures do exist to make cyber-attacks more difficult by a perpetuator, whether they were designed for that purpose or not. It is also important to understand what can be done to improve security in automobile CAN through means that are not yet utilized, such as network encryption.

No attacks have ever been recorded of a cyber-attack on a passenger automobile at this point in history. However, the increase in interest and research of the subject and the increase in cyber connectivity of vehicles could leave consumers at risk. Without intrusion detection systems in place it would be unlikely that even if an attack of this nature had caused a vehicle to behave irrationally the driver would attribute the malfunction to network intrusion, rather than a malfunctioning internal unit. Collecting data to see if cyber-attacks actually occur on the road is an important first step to determine if further security is necessary.

Car manufacturers will need to consider carefully how to improve their network security as vehicles evolve. Increase in security will cause increase in cost for consumers, make scrapping of cars easier as more expensive electronics are damaged in collisions, and require more expensive equipment for vehicle maintenance. Though car manufacturers have a duty to keep their customers safe it is possible that drivers would not accept the added cost that is attached to a more cyber-secure vehicle. Consumer research on this subject would need to be explored.

New vehicles that are equipped with Wi-Fi and electric and self-driving cars will have entirely new security holes and require even more consideration towards security. Manufacturers should consider network security as they are developing these new vehicles rather than as an afterthought.

This report is not exhaustive for this subject. Access to further materials and testing on various other automotive devices, ECUs, and vehicles is needed to further enlighten this topic. Limitations on funding and access to equipment were certainly evident in this study. Recognition of network security is an important factor to consider as vehicles are becoming more and more connected electronically, internally, and externally through wireless internet and other airwave communication. Continuous study and testing is required to maintain a safe transportation environment for passengers and materials.

Specific limitations that existed when conducting research were the inability to experiment with causing a critical or dangerous fault in an automobile, access to other ECUs that control critical functions as an Engine Control Module or a Power-train Module, testing through wireless or Bluetooth communication, access to a complete vehicle bus through means other than the OBD-ii port, and successful testing to re-flash an ECU.

Appendix A

Scripts Used in Data Collection and Analysis

Send.c

```
//Created by Michael Dulan

#include <stdio.h>
#include <time.h>
#include "ecommlib.h"
#include <conio.h>


HANDLE ECOMHandle; //This will be the HANDLE to our ECOM device used for all function calls      35921
HANDLE deviceHandle[5];
// Number of ECOM devices
int device_count;
// Flag used to stop the sending of packets to client
int stop_flag;
unsigned long number;
HANDLE h;

void SendTruckInBox()
{
        EFFMessage newMessage;
        newMessage.ID = 12;
        newMessage.DataLength = 8;
        newMessage.options = 0;
        newMessage.data[0] = 0;
        newMessage.data[1] = 0;
        newMessage.data[2] = 0;
        newMessage.data[3] = 0;
        newMessage.data[4] = 0;
        newMessage.data[5] = 8;   //First digit      Speed = (Firstdigit SecondDigit /100) kmph
        newMessage.data[6] = 8;   //Second digit
        newMessage.data[7] = 8;
        int timestamp = 12;
        EFFMessage newMessage2;
        while(!kbhit())
        {
                Sleep(20);
                newMessage.TimeStamp = timestamp + 12;
                CANTransmitMessageEx(deviceHandle[0], &newMessage);
                timestamp++;
        }
}
```

```
void Test()
{
	EFFMessage newMessage;
	newMessage.ID = 12;
	newMessage.DataLength = 8;
	newMessage.options = 0;
	newMessage.data[0] = 0;
	newMessage.data[1] = 0;
	newMessage.data[2] = 0;
	newMessage.data[3] = 0;
	newMessage.data[4] = 0;
	newMessage.data[5] = 8;     //First digit        Speed = (Firstdigit SecondDigit /100) kmph
	newMessage.data[6] = 8;     //Second digit
	newMessage.data[7] = 8;   //Checksum
	int timestamp = 12;
	while(!kbhit())
	{
		Sleep(20);
		newMessage.TimeStamp = timestamp + 12;
		CANTransmitMessageEx(deviceHandle[0], &newMessage);
		timestamp++;
	}
}

void SpeedTest()  //0B4
{
	SFFMessage newMessage;
	newMessage.IDH = 0;
	newMessage.IDL = 180;      //B4
	newMessage.DataLength = 8;
	newMessage.options = 0;
	newMessage.data[0] = 0;
	newMessage.data[1] = 0;
	newMessage.data[2] = 0;
	newMessage.data[3] = 0;
	newMessage.data[4] = 0;
	newMessage.data[5] = 4;     //First digit        Speed = (Firstdigit SecondDigit /100) kmph
	newMessage.data[6] = 28;    //Second digit
	newMessage.data[7] = (180 + 8 + 5 + 28) % 256; //Checksum
	int timestamp = 12;
	while(!kbhit())
	{
		Sleep(2);
		newMessage.TimeStamp = timestamp + 12;
		CANTransmitMessage(deviceHandle[0], &newMessage);
		timestamp++;
	}
}
```

```
void UnknownTest()  //320
{
	SFFMessage newMessage;
	newMessage.IDH = 3;		//3
	newMessage.IDL = 32;		//20
	newMessage.DataLength = 3;
	newMessage.options = 0;
	newMessage.data[0] = 0;
	newMessage.data[1] = 40;   //20 or 28
	newMessage.data[2] = (180 + 8 + 5 + 28) % 256;  //Checksum
	newMessage.data[3] = 0;
	newMessage.data[4] = 0;
	newMessage.data[5] = 0;
	newMessage.data[6] = 0;
	newMessage.data[7] = 0;
	int timestamp = 12;
	while(!kbhit())
	{
		Sleep(20);
		newMessage.TimeStamp = timestamp + 12;
		CANTransmitMessage(deviceHandle[0], &newMessage);
		timestamp++;
	}
}


void LockTest()  //750
{
	int count;
	SFFMessage newMessage;
	newMessage.IDH = 7;		//7
	newMessage.IDL = 80;		//50
	newMessage.DataLength = 8;
	newMessage.options = 0;
	newMessage.data[0] = 64;  //40
	newMessage.data[1] = 5;
	newMessage.data[2] = 48;  //30
	newMessage.data[3] = 17;  //11
	newMessage.data[4] = 0;
	newMessage.data[5] = 64;  //40 unlock, 80 lock all, 00 unlock trunk
	newMessage.data[6] = 0;   //80 unlock trunk, 0 else
	newMessage.data[7] = 0;
	int timestamp = 12;
	while(!kbhit())
	{
		Sleep(10);
		newMessage.TimeStamp = timestamp + 12;
		CANTransmitMessage(deviceHandle[0], &newMessage);
		timestamp++;
	}
}
```

```
void FuelTest()  //7C0
{
        SFFMessage newMessage;
        newMessage.IDH = 3;        //7
        newMessage.IDL = 192;      //C0
        newMessage.DataLength = 8;
        newMessage.options = 0;
        newMessage.data[0] = 4;
        newMessage.data[1] = 48;   //30
        newMessage.data[2] = 3;
        newMessage.data[3] = 0;
        newMessage.data[4] = 32;   //20 == 3/4 full
        newMessage.data[5] = 0;
        newMessage.data[6] = 0;
        newMessage.data[7] = 0;
        int timestamp = 12;
        while(!kbhit())
        {
                Sleep(5);
                newMessage.TimeStamp = timestamp + 12;
                CANTransmitMessage(deviceHandle[0], &newMessage);
                timestamp++;
        }
}


void LightsTest()  //57F
{
        SFFMessage newMessage;
        newMessage.IDH = 5;        //5
        newMessage.IDL = 127;      //7F
        newMessage.DataLength = 8;
        newMessage.options = 0;
        newMessage.data[0] = 232;  //E8
        newMessage.data[1] = 56;   //30 = On, 38 = High, 0 = Off, 10 = Parking
        newMessage.data[2] = 10;
        newMessage.data[3] = 0;
        newMessage.data[4] = 0;
        newMessage.data[5] = 0;
        newMessage.data[6] = 0;
        newMessage.data[7] = 0;    //80 if off
        int timestamp = 12;
        while(!kbhit())
        {
                Sleep(5);
                newMessage.TimeStamp = timestamp + 12;
                CANTransmitMessage(deviceHandle[0], &newMessage);
                timestamp++;
        }
}
```

```
void EngineCoolantTest()  //52C
{
	SFFMessage newMessage;
	newMessage.IDH = 5;	//5
	newMessage.IDL = 44;	//2C
	newMessage.DataLength = 2;
	newMessage.options = 0;
	newMessage.data[0] = 2;
	newMessage.data[1] = 20;   //temp 0.5 Celcius per bit
	newMessage.data[2] = 0;
	newMessage.data[3] = 0;
	newMessage.data[4] = 0;
	newMessage.data[5] = 0;
	newMessage.data[6] = 0;
	newMessage.data[7] = 0;
	int timestamp = 12;
	while(!kbhit())
	{
		Sleep(10);
		newMessage.data[1] = ((newMessage.data[1]+1))%256;
		newMessage.TimeStamp = timestamp + 12;
		CANTransmitMessage(deviceHandle[0], &newMessage);
		timestamp++;
	}
}

DeviceInfo *GetECOMDevicesEx()
{
	BYTE baud, ReturnError;
	DeviceInfo deviceInfoStruct;
	DEV_SEARCH_HANDLE searchHandle = StartDeviceSearch(FIND_ALL);
	int deviceCount = 0;
	DeviceInfo *toret = (DeviceInfo *) malloc(sizeof(DeviceInfo) * 10);
	memset(toret, 0, sizeof(DeviceInfo)*10);
	if (searchHandle == NULL){
		printf("Unexpected error allocating memory for device search\n");
	}

   while(FindNextDevice(searchHandle, &deviceInfoStruct) == ECI_NO_ERROR){
		memcpy((void *) &toret[deviceCount++], &deviceInfoStruct, sizeof(DeviceInfo));
   }
	printf("%lu\n",toret->SerialNumber);
	printf("%lu\n",toret->DeviceHandle);
	CloseDeviceSearch(searchHandle);
	device_count = 0;
	if (toret->SerialNumber == 0){
		printf("Problem getting ECOM devices\n");
	}
	while ( (toret + device_count)->SerialNumber != 0) {
		ULONG serial = (toret + device_count)->SerialNumber;
		baud = CAN_BAUD_250K;
		deviceHandle[device_count] = CANOpen(serial, baud, &ReturnError);
		(toret + device_count)->DeviceHandle = deviceHandle[device_count];
		printf("Return Error: %s",ReturnError);
		if (!deviceHandle || ReturnError != 0){
```

```
                char ErrMsg[400];
                printf("Error Message");
                GetFriendlyErrorMessage(ReturnError, ErrMsg, 400);
                printf("CANOpen failed with error message: ");
            }
            ReturnError = CANSetupDevice(deviceHandle[device_count], CAN_CMD_TRANSMIT,
                                         CAN_PROPERTY_ASYNC);
            device_count++;
        }
        printf("Devices: %d",device_count);
        return toret;
}

DeviceInfo *GetECOMDevices()
{
        BYTE baud, ReturnError;
        DeviceInfo deviceInfoStruct;
        DEV_SEARCH_HANDLE searchHandle = StartDeviceSearch(FIND_ALL);
        int deviceCount = 0;
        DeviceInfo *toret = (DeviceInfo *) malloc(sizeof(DeviceInfo) * 10);
        memset(toret, 0, sizeof(DeviceInfo)*10);
        if (searchHandle == NULL){
            printf("Unexpected error allocating memory for device search\n");
        }

        while(FindNextDevice(searchHandle, &deviceInfoStruct) == ECI_NO_ERROR){
            memcpy((void *) &toret[deviceCount++], &deviceInfoStruct, sizeof(DeviceInfo));
        }
        printf("SN: %lu\n",toret->SerialNumber);
        printf("DH: %lu\n",toret->DeviceHandle);
        CloseDeviceSearch(searchHandle);
        device_count = 0;
        if (toret->SerialNumber == 0){
            printf("Problem getting ECOM devices\n");
        }
        while ( (toret + device_count)->SerialNumber != 0) {
            ULONG serial = (toret + device_count)->SerialNumber;
            baud = CAN_BAUD_500K;
            deviceHandle[device_count] = CANOpen(serial, baud, &ReturnError);
            (toret + device_count)->DeviceHandle = deviceHandle[device_count];
            printf("Return Error: %s\n",ReturnError);
            if (!deviceHandle || ReturnError != 0){
                char ErrMsg[400];
                GetFriendlyErrorMessage(ReturnError, ErrMsg, 400);
                printf("CANOpen failed with error message: ");
            }
            ReturnError = CANSetupDevice(deviceHandle[device_count], CAN_CMD_TRANSMIT,
                                         CAN_PROPERTY_ASYNC);
            device_count++;
        }
        printf("Devices: %d",device_count);
        return toret;
}

int main()
```

```
{
        DeviceInfo * lolz;
        int i;
        BYTE ReturnError = 0;  //This will be used for all error return status codes
        lolz = GetECOMDevicesEx();
        //lolz = GetECOMDevices();

        //This executes until keyboard is hit.
        //SendTruckInBox();
        //Test();     //J1939
        //SpeedTest();
        //UnknownTest();
        //LockTest();
        //FuelTest();
        //LightsTest();
        EngineCoolantTest();
        //Now we are all done, so clean up by closing the ECOM
        while ((lolz+i)->DeviceHandle){
                CloseDevice((lolz+i)->DeviceHandle);
                i++;
        }
        return 0;
}
```

CAN Peach Pitfile

```
<Peach xmlns="http://peachfuzzer.com/2012/Peach" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://peachfuzzer.com/2012/Peach ../peach.xsd">

  <DataModel name="ID1">

    <Number mutable="true" name="ID11" signed="true" size="8" value="00" valueType="hex" />
    <Number mutable="true" name="ID12" signed="true" size="8" value="00" valueType="hex" />
    <Number mutable="true" name="ID13" signed="true" size="8" value="00" valueType="hex" />
    <Number mutable="true" name="ID14" signed="true" size="8" value="00" valueType="hex" />
    <Number mutable="false" name="IDOPT" signed="true" size="8" value="00" valueType="hex" />
    <Number mutable="true" name="IDTIME" size="32" value="00 00 00 00" valueType="hex" />
    <Number mutable="false" name="ID1LEN" signed="true" size="8" value="11" valueType="hex" />
    <Number mutable="false" name="ID1DATA1" signed="true" size="8" value="08" valueType="hex" />
    <Number mutable="true" name="ID1DATA2" signed="true" size="8" value="00" valueType="hex" />
    <Number mutable="true" name="ID1DATA3" signed="true" size="8" value="00" valueType="hex" />
    <Number mutable="true" name="ID1DATA4" signed="true" size="8" value="00" valueType="hex" />
    <Number mutable="true" name="ID1DATA5" signed="true" size="8" value="00" valueType="hex" />
    <Number mutable="true" name="ID1DATA6" signed="true" size="8" value="00" valueType="hex" />
    <Number mutable="true" name="ID1DATA7" signed="true" size="8" value="00" valueType="hex" />
    <Number mutable="true" name="ID1DATA8" signed="true" size="8" value="00" valueType="hex" />
  </DataModel>

  <DataModel name="DataResponse">
    <String name="response" token="true" />
  </DataModel>

  <StateModel initialState="Initial" name="StateID1">
    <State name="Initial">
      <Action type="connect" />
      <Action type="output"><DataModel ref="ID1" /></Action>
      <Action type="input"><DataModel ref="DataResponse" /></Action>
      <Action type="close" />
    </State>
  </StateModel>

<Agent name="LocalAgent">

        <Monitor class="Socket">
                <Param name="Timeout" value="3000" />
                <Param name="Host" value="127.0.0.1" />
                <Param name="Port" value="15765" />
        </Monitor>

</Agent>

<Agent location="tcp://127.0.0.1:15765" name="RemoteAgent">
        <Monitor class="WindowsDebugger">
                <Param name="CommandLine" value="C:\Users\Albany\Desktop\ecom final.exe" />
```

```
                        <Param name="WinDbgPath" value="C:\Windows\System32\debug.exe" />
                </Monitor>
</Agent>

  <Test name="Default">
                <Mutators mode="include">
                        <Mutator class="HexMutator" />
                </Mutators>

    <StateModel ref="StateID1" />
    <Publisher class="TcpClient">
      <Param name="Host" value="127.0.0.1" />
      <Param name="Port" value="15765" />
    </Publisher>
                <Strategy class="Sequential" />

    <Logger class="Filesystem">
      <Param name="Path" value="logs" />
    </Logger>
  </Test>

</Peach>
```

Peach Custom Mutator

```
//
// Copyright (c) Michael Eddington
//
// Permission is hereby granted, free of charge, to any person obtaining a copy
// of this software and associated documentation files (the "Software"), to deal
// in the Software without restriction, including without limitation the rights
// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
// copies of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included in
// all copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
// SOFTWARE.
//

// Authors:
//   Michael Eddington (mike@dejavusecurity.com)
//   Michael Dulan (michael.dulan@gmail.com)

// $Id$

using System;
using System.Collections.Generic;
using System.Text;
using Peach.Core.Dom;

namespace Peach.Core.Mutators
{
    [Mutator("HexMutator")]
    [Description("Increment each <Number> element")]
    public class HexMutator : Mutator
    {
        // members
        //
        int n;
        int size;
        int _value;
        bool signed;
        uint currentCount;
```

```
long minValue;
ulong maxValue;
int counter;

// CTOR
//
public HexMutator(DataElement obj)
{
   name = "HexMutator";
   currentCount = 0;
   n = getN(obj, 0);
   counter = 0;

   if (obj is Number)
   {
      signed = ((Number)obj).Signed;
      size = (int)((Number)obj).lengthAsBits;
      minValue = ((Number)obj).MinValue;
      maxValue = ((Number)obj).MaxValue;
      _value = (int)(((Number)obj).Value.ReadByte());
   }
}

// GET N
//
public int getN(DataElement obj, int n)
{
        n = (int)(n + ((Number)obj).lengthAsBits);       //calculate total length of all elements in bits
        n = (int)(n / 8);                                //convert bits to bytes
        n = (int)(Math.Pow(16,n));                       //create total iterations
        n = 256*256*256;
        return n;
}

// MUTATION
//
public override uint mutation
{
   get { return currentCount; }
   set { currentCount = value; }
}

// COUNT
//
public override int count
{
   get { return n; }
}

// SUPPORTED
//
public new static bool supportedDataElement(DataElement obj)
{
   // Ignore numbers != 8 bits
```

```
        if (obj is Number && obj.isMutable)
          if (((Number)obj).lengthAsBits%8 == 0)
         {
             return true;
         }
        return false;
    }

    // SEQUENTIAL_MUTATION
    //
    public override void sequentialMutation(DataElement obj)
    {
          dynamic val;
          counter++;
          val = counter;
          if (val > 255) //FF
          {
                // Restart mutation count
                val = 0;
                counter = 0;
          }

/*          if (signed)
        {
          if (size < 32)
              val = context.Random.Next((int)minValue, (int)maxValue);
          else if (size == 32)
              val = context.Random.NextInt32();
          else if (size < 64)
              val = context.Random.Next((long)minValue, (long)maxValue);
          else
              val = context.Random.NextInt64();
        }
        else
        {
          if (size < 32)
              val = context.Random.Next((uint)minValue, (uint)maxValue);
          else if (size == 32)
              val = context.Random.NextUInt32();
          else if (size < 64)
              val = context.Random.Next((ulong)minValue, (ulong)maxValue);
          else
              val = context.Random.NextUInt64();
        }*/

        obj.MutatedValue = new Variant(val);
        obj.DefaultValue = new Variant(val);
        obj.mutationFlags = MutateOverride.Default;
        obj.mutationFlags = MutateOverride.Transformer;
    }

    // RANDOM_MUTAION
    //
    public override void randomMutation(DataElement obj)
    {
```

```
             // Only called via the Sequential mutation strategy, which should always have a consistent seed
            //randomMutation(obj);
            dynamic val;
            val = counter;

           //counter = counter;

            obj.MutatedValue = new Variant(val);
            obj.DefaultValue = new Variant(val);
            //obj.fixup = new Variant(val);
            //obj.InternalValue = new Variant(val);
            //obj.GenerateValue();
            //obj.mutationFlags = MutateOverride.Fixup;
            obj.mutationFlags = MutateOverride.Default;
            //obj.mutationFlags = MutateOverride.Transformer;
        }
    }
}

// end
```

Custom Peach Strategy

```
//
// Copyright (c) Michael Eddington
//
// Permission is hereby granted, free of charge, to any person obtaining a copy
// of this software and associated documentation files (the "Software"), to deal
// in the Software without restriction, including without limitation the rights
// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
// copies of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included in
// all copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
// SOFTWARE.
//

// Authors:
//   Michael Eddington (mike@dejavusecurity.com)
//   Michael Dulan (michael.dulan@gmail.com)
// $Id$

using System;
using System.Collections.Generic;
using System.Text;
using Peach.Core.Dom;
using System.Reflection;
using System.Linq;

using NLog;

namespace Peach.Core.MutationStrategies
{
	[MutationStrategy("CAN", true)]
	[Serializable]
	public class CAN : MutationStrategy
	{
		//int[] counter = new int[18];
		int placeholder;
		int lastByteCurrentlyFuzzing = 0;
```

```
protected class Iterations : List<Tuple<string, Mutator, string>> { }

[NonSerialized]
protected static NLog.Logger logger = LogManager.GetCurrentClassLogger();

[NonSerialized]
protected IEnumerator<Tuple<string, Mutator, string>> _enumerator;

[NonSerialized]
protected Iterations _iterations = new Iterations();

private List<Type> _mutators = null;
private uint _count = 1;
private uint _iteration = 1;

public CAN(Dictionary<string, Variant> args)
        : base(args)
{
}

public override void Initialize(RunContext context, Engine engine)
{
        base.Initialize(context, engine);

        // Force seed to always be the same
        context.config.randomSeed = 33333;

        Core.Dom.Action.Starting += new ActionStartingEventHandler(Action_Starting);
        Core.Dom.State.Starting += new StateStartingEventHandler(State_Starting);
        context.engine.IterationFinished += new
        Engine.IterationFinishedEventHandler(engine_IterationFinished);
        context.engine.IterationStarting += new
        Engine.IterationStartingEventHandler(engine_IterationStarting);
        _mutators = new List<Type>();
        _mutators.AddRange(EnumerateValidMutators());
}

void engine_IterationStarting(RunContext context, uint currentIteration, uint? totalIterations)
{
        if (context.controlIteration && context.controlRecordingIteration)
        {
                // Starting to record
                _iterations = new Iterations();
                _count = 0;
        }
}

void engine_IterationFinished(RunContext context, uint currentIteration)
{
        // If we were recording, end of iteration is end of recording
        if(context.controlIteration && context.controlRecordingIteration)
                OnDataModelRecorded();
}

public override void Finalize(RunContext context, Engine engine)
```

```
{
	base.Finalize(context, engine);

	Core.Dom.Action.Starting -= Action_Starting;
	Core.Dom.State.Starting -= State_Starting;
	context.engine.IterationStarting -= engine_IterationStarting;
	context.engine.IterationFinished -= engine_IterationFinished;
}

protected virtual void OnDataModelRecorded()
{
}

public override bool UsesRandomSeed
{
	get
	{
		return false;
	}
}

public override bool IsDeterministic
{
	get
	{
		return true;
	}
}

public override uint Iteration
{
	get
	{
		return _iteration;
	}
	set
	{
		SetIteration(value);
		SeedRandom();
	}
}

private void SetIteration(uint value)
{
	System.Diagnostics.Debug.Assert(value > 0);

	if (_context.controlIteration && _context.controlRecordingIteration)
	{
		return;
	}

	if (_iteration == 1 || value < _iteration)
	{
		_iteration = 1;
		_enumerator = _iterations.GetEnumerator();
```

```
                _enumerator.MoveNext();
                _enumerator.Current.Item2.mutation = 0;
        }

        uint needed = value - _iteration;

        if (needed == 0)
                return;

        while (true)
        {
                var mutator = _enumerator.Current.Item2;
                uint remain = (uint)mutator.count - mutator.mutation;

                if (remain > needed)
                {
                        mutator.mutation += needed;
                        _iteration = value;
                        return;
                }

                needed -= remain;
                _enumerator.MoveNext();
                _enumerator.Current.Item2.mutation = 0;
        }
}

private void Action_Starting(Core.Dom.Action action)
{
        // Is this a supported action?
        if (!action.outputData.Any())
                return;

        if (!_context.controlIteration)
                MutateDataModel(action);

        else if(_context.controlIteration && _context.controlRecordingIteration)
                RecordDataModel(action);
}

void State_Starting(State state)
{
        if (_context.controlIteration && _context.controlRecordingIteration)
        {
                foreach (Type t in _mutators)
                {
                        // can add specific mutators here
                        if (SupportedState(t, state))
                        {
                                var mutator = GetMutatorInstance(t, state);
                                var key = "Run_{0}.{1}".Fmt(state.runCount, state.name);
                                _iterations.Add(new Tuple<string, Mutator, string>(key,
                                                mutator, null));
                                _count += (uint)mutator.count;
                        }
                }
        }
}

// Recursively walk all DataElements in a container.
// Add the element and accumulate any supported mutators.
private void GatherMutators(string instanceName, DataElementContainer cont)
{
        List<DataElement> allElements = new List<DataElement>();
        RecursevlyGetElements(cont, allElements);
        int length = allElements.Count();

        //for (int i = 0; i < length; i++)
        //      counter[i] = 0;

        foreach (DataElement elem in allElements)
        {
                var elementName = elem.fullName;

                foreach (Type t in _mutators)
                {
                        if (SupportedDataElement(t, elem))
                        {
                                var mutator = GetMutatorInstance(t, elem);
                                _iterations.Add(new Tuple<string, Mutator,
                                string>(elementName, mutator, instanceName));
                                _count += (uint)mutator.count;
                        }
                }
        }
}

private void RecordDataModel(Core.Dom.Action action)
{
        // ParseDataModel should only be called during iteration 0
        System.Diagnostics.Debug.Assert(_context.controlIteration &&
                                        _context.controlRecordingIteration);

        foreach (var item in action.outputData)
        {
                var instanceName = item.instanceName;
                GatherMutators(instanceName, item.dataModel);
        }
}

/// <summary>
/// Allows mutation strategy to affect state change.
/// </summary>
/// <param name="state"></param>
/// <returns></returns>
public override State MutateChangingState(State state)
{
        if (_context.controlIteration)
                return state;
```

```
		var key = "Run_{0}.{1}".Fmt(state.runCount, state.name);
		if (key == _enumerator.Current.Item1)
		{
			OnStateMutating(state, _enumerator.Current.Item2);
			logger.Debug("MutateChangingState: Fuzzing state change: " + state.name);
			logger.Debug("MutateChangingState: Mutator: " +
					_enumerator.Current.Item2.name);
			return _enumerator.Current.Item2.changeState(state);
		}

		return state;
	}

	private void ApplyMutation(ActionData data)
	{
		// Ensure we are on the right model
		bool check;
		if (_enumerator.Current.Item3 != data.instanceName)
			return;
		var fullName = _enumerator.Current.Item1;
		var dataElement = data.dataModel.find(fullName);


		if (dataElement != null)
		{
			// Apply increment mutation to first byte
			var mutator = _enumerator.Current.Item2;
			OnDataMutating(data, dataElement, mutator);
			logger.Debug("ApplyMutation: Fuzzing: JHJHJHJJHHJ" + fullName);
			logger.Debug("ApplyMutation: Mutator: " + mutator.name);
			mutator.sequentialMutation(dataElement);
			placeholder = 0;

			// Increment the element(s) after FF's
			while ((int)(dataElement.DefaultValue) == 0 && lastByteCurrentlyFuzzing > 0)
			{
				placeholder++;
				check = _enumerator.MoveNext();
				if (!check)
				{
					lastByteCurrentlyFuzzing = 0;
				}
				else
				{
					fullName = _enumerator.Current.Item1;
					dataElement = data.dataModel.find(fullName);
					mutator = _enumerator.Current.Item2;
					OnDataMutating(data, dataElement, mutator);
					logger.Debug("ApplyMutation: Fuzzing: " + fullName);
					logger.Debug("ApplyMutation: Mutator: " + mutator.name);
					mutator.sequentialMutation(dataElement);
				}
			}

			// Increment the pointer to the last element currently being incremented
```

```
                        if ((placeholder > lastByteCurrentlyFuzzing) || ((int)(dataElement.DefaultValue)
                            == 255 && lastByteCurrentlyFuzzing == 0))
                        {
                                lastByteCurrentlyFuzzing++;
                        }

                        // Keep all other bytes the same by "mutating" them
                        for (int i=placeholder; i<lastByteCurrentlyFuzzing; i++)
                        {
                                check = _enumerator.MoveNext();
                                if (check)
                                {
                                        fullName = _enumerator.Current.Item1;
                                        dataElement = data.dataModel.find(fullName);
                                        mutator = _enumerator.Current.Item2;
                                        OnDataMutating(data, dataElement, mutator);
                                        logger.Debug("ApplyMutation: Fuzzing: " + fullName);
                                        logger.Debug("ApplyMutation: Mutator: " + mutator.name);
                                        mutator.randomMutation(dataElement);
                                }
                        }

                        _enumerator.Reset(); //set to element before first
                        check = _enumerator.MoveNext(); //first element
                }
        }

        private void MutateDataModel(Core.Dom.Action action)
        {
                // MutateDataModel should only be called after ParseDataModel
                System.Diagnostics.Debug.Assert(_count >= 1);
                System.Diagnostics.Debug.Assert(_iteration > 0);
                System.Diagnostics.Debug.Assert(!_context.controlIteration);

                foreach (var item in action.outputData)
                {
                        ApplyMutation(item);
                }
        }

        public override uint Count
        {
                get
                {
                        return _count;
                }
        }
    }
}

// end
```

BIBLIOGRAPHY

Checkoway, Stephen, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czesksi, Franziska Roesner, and Tadayoshi Kohno. "Comprehensive Experimental Analyses of Automotive Attack Surfaces." Thesis. University of California, San Diego and University of Washington, 2011. Center for Automotive Embedded Systems Security. USENIX Security, 10-12 Aug. 2011. Web. 3 Jan. 2014.

Koscher, Karl, Alexei Czesksi, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Stefan Savage. "Experimental Security Analysis of a Modern Automobile." Thesis. University of California, San Diego and University of Washington, 2010. Center for Automotive Embedded Systems Security. IEEE Symposium on Security and Privacy, 16-19 May 2010. Web. 3 Jan. 2014.

Miller, Dr. Charlie, and Chris Valasek. "Adventures in Automotive Networks and Control Uni." Thesis. 2010. Web. 3 Aug. 2013

Rouf, Ishtiaq, Rob Miller, Hossen Mustafa, Travis Taylor, Sangho Oh, Wenyuan Xu, Marco Gruteser, Wade Trappe, and Ivan Seskar. "Security and Privacy Vulnerabilities of In-Car Wireless Networks: A Tire Pressure Monitoring System Case Study." Thesis. University of South Carolina and Rutgers University, n.d. Web. 3 Jan. 2014.

Kumar, Praveen, and B. Bhavani. "Controller Area Network for Monitoring and Controlling the Industrial Parameters Using Bluetooth Communication." Thesis. Osmania University, 2013. International Journal of Science and Research (IJSR), Aug. 2013. Web. 3 Jan. 2014.

Barry, Keith. "Can Your Car Be Hacked?" Car and Driver. Car and Driver, July 2011. Web. 3 Jan. 2014.

Klimas, Liz. "Big Brother? OnStar Shuts Down Stolen Car During Police Chase." The Blaze. N.p., 18 July 2012. Web. 3 Jan. 2014.

"ISO 15765 CAN Diagnostics." ISO 15765 Network Layer Diagnostics. Society of Automotive Engineers, 11 Dec. 2012. Web. 11 Apr. 2014.

"Understanding SAE J1939." Simmasoftware.com. Simma Software, Inc., n.d. Web. 3 Aug. 2013.

ProEmion Telematics Systems. "Global CAN Communication." CANlink GSM / UMTS (n.d.): n. pag. Web. 2 Dec. 2014.

Greenberg, Andy. "Hackers Reveal Nasty New Car Attacks--With Me Behind The Wheel (Video)." Forbes. Forbes Magazine, 12 Aug. 2013. Web. 08 Apr. 2014.

Infiniti QX60 Commercial – Reverse Worries. YouTube. N.p., n.d. Web. 4 Mar. 2015.

Szczys, Mike. "Radar Detector Integrated with Dashboard Display Screens and Steering Wheel Controls." Hackaday. N.p., 31 Aug. 2013. Web. 08 Apr. 2014.

Vasicheck, Shawn R. Control Area Network Data Encryption System and Method. Clark Equipment Company, assignee. Patent US20090169007 A1. 2 July 2009. Print.

ACADEMIC VITA

CURRENT ADDRESS:
328 E Foster Avenue
University Park, PA 16801

Michael Dulan
Michael.dulan@gmail.com
(571) 271 – 3738

PERMANENT ADDRESS:
6120 Green Cap Place
Fairfax, Virginia 22030

EDUCATION

The Pennsylvania State University: Smeal College of Business
University Park, PA
B.S. in Management Information Systems *Expected Graduation: May 2015*

- Schreyer Honors College
- Dean's List 5/5 semesters

RELEVANT WORK EXPERIENCE

BIT Systems – *Software Engineer Intern* 2013-2015
Sterling, VA

- Developed applications to analyze traffic, inject instructions and communicate with automobiles.
- Developed numerous applications in Windows and Linux using python, C, XML, Tshark, bash files, and make files for installation.
- Researched and developed a variety of fuzzers, mainly Sulley and Peach.
- Engineered hardware to communicate with vehicle ECUs, including ECOM and Uart devices.
- Utilized the Glade GUI creator to make an easy to use user interface for a vehicle communication program.
- Created a custom version of WireShark for collection of CAN packet data.

PROGRAMMING EXPERIENCE

Programming Languages:

- C++: Implemented a variety of applications leveraging various data structures, such as a resource management application for a small business with both a Customer and Employee UI to keep track of Payroll, Timecards, Inventory, the Cash Register, Orders, Menus, and other options. Also, created an application and detailed report comparing the running time of six different sorting algorithms during average, best, and worst case scenarios for different data sizes.
- Python: Developed applications to inject, parse, receive, and analyze packets from a vehicle.
- C: Redesigned drivers to better suit project needs, and develop data dictionaries for various programs.
- MS Visual Basic: Developed custom macros to improve MS Excel and integrate MS Excel and MS Word for financial record keeping and modeling.
- XML: Used with different web and fuzzing applications to test systems for vulnerabilities.
- Java: Designed and developed class-based applications
- SQL: Leveraged SQL to create, update, delete, and search databases and have created numerous databases
- Tshark: Used to integrate WireShark with other programs using the command line.
- ISO 15765 and SAE J1939 CAN: Can read and write code to communicate with modern vehicles and large trucks