A thesis submitted in partial satisfaction of the requirements
for the degree of Master of Computer Science and Engineering
in the Graduate School of the University of Aizu

Developing a CAN Bus-based Secure System for Automotive Module Connectivity

by

William Hutchinson Putnam III

March 2018

The thesis titled

Developing a CAN Bus-based Secure System for Automotive Module Connectivity

by

William Hutchinson Putnam III

is reviewed and approved by:

Main referee

Professor, Computer Networks Laboratory

Zixue Cheng Zixue Cheng

Associate Professor, Computer Networks Laboratory

Lei Jing Lei Jing

Professor, Database Systems Laboratory

Subhash Bhalla Subhash Bhalla

THE UNIVERSITY OF AIZU

March 2018

Contents

List of Figures

List of Tables

List of Abbreviations

AES	Advanced Encryption Standard
AFV	Average Factor Value
ASCII	American Standard Code for Information Interchange
CAN	Controller Area Network
DDoS	Distributed Denial of Service
HMAC	Hash-based Message Authentication Code
ISO	International Organization for Standardization
IV	Initialization Vector
IVI	In-Vehicle Infotainment
MCU	Microcontroller Unit(s)
MitM	Man-in-the-Middle [attack]
OBD	On-Board Diagnostics
OSI	Open Systems Interconnection
SHA	Secure Hash Algorithm
SSL	Secure Sockets Layer
SSM	Signal Security Module
TCB	Transmission Control Block
TCP	Transmission Control Protocol
TCP/IP	Transmission Control Protocol over Internet Protocol
TLS	Transport Layer Security

List of Symbols

a	the size of a set in the awareness algorithm
ACK	Acknowledgement flag for acknowledging receipt of a packet
b	the number of size a sets in the SSM's awareness algorithm
FIN	Finish flag for gracefully terminating a connection
MCU1	the sender MCU in a given exchange between two MCUs
MCU2	the destination MCU in a given exchange between two MCUs
n	the number of most recent entries in the SSM's stack algorithm
PSH	Push flag for signifying the arrival of new data
RST	Reset flag for abruptly terminating a connection
SYN	Synchronize flag for starting a connection

Acknowledgment

I would like to thank the following persons and parties for their professional and technical advice:

- Professors Zixue Cheng (Shigaku Tei) and Lei Jing, advisors of the University of Aizu Computer Networks Laboratory, and Professor Subhash Bhalla of the Database Systems Laboratory
- Yilang Wu, Ph.D., post-doctorate of the Computer Networks Laboratory
- Michiko Hoshi, Kouichi Sato, and the other members of the University of Aizu Computer Networks Laboratory, for their generous assistance in the laboratory

I would also like to thank the following persons and parties for their personal and emotional support:

- Yuriko Nagashima, the University of Aizu Foreign Personnel Advisor who assisted in my adjustment to daily life in Japan
- Masayuki Hisada, my boss and the CEO of Aizu Laboratory, Inc.
- My mother, Joanne, who is responsible for raising the person that I am today

Finally, I dedicate this master's thesis to my grandfather, the late Joseph John Chojnowski, who helped finance my college tour trip to Japan and whose love of all things automotive lives on in his bloodline.

Abstract

Since the late 1980s, CAN bus protocol has been the de-facto and legal standard of an automobile network. With a simple design and very low fault rate, it allows all MCUs within a vehicle to stay interconnected and transmit messages at very low latencies and minimal error. However, according to research by Moore et al. and real-world analysis by Miller and Valasek, the CAN bus protocol contains no security measures, allowing any third party to send messages to various MCUs on the network or intercept and change messages already being sent.

The purpose of this research is to expand on the existing CAN bus protocol by updating the standard message contents and message exchange sequences to meet the specifications of the CIA security triage. It considers three issues to address:

- an onslaught of different messages to attempt an unusual reaction, also known as fuzzing. This process can have significant impact on real-time MCU operation. The research will propose a method to either eliminate or mitigate these effects.
- injection of messages on the CAN bus network. The CAN bus lacks the tools and procedures necessary to determine invalid messages. The research will account for the proper verification of both sent messages themselves and message contents.
- the staging of MitM attack on the CAN bus network. This is a very common form of attack, and the CAN bus' design cannot entirely account for its total prevention. Nevertheless, the research will examine a method to mitigate the chances and resulting damages of an attack in this manner.

The research considers a threefold approach to solving the above problems:

- the development of an updated segment structure.
- a streamlined communication process between two MCUs on a network.
- the integration of a SSM to collect traffic and manage the integrity of the network in the case of a breach.

A virtual CAN bus network-based demonstration with custom-built software was developed to serve as a proof of concept. Preliminary results showed that the system model was able to mitigate all of the listed problems, as well as benchmark the efficiency of an important algorithm used to determine potential system breaches.

Finally, this master's thesis outlines the significance and feasibility of this model in today's context, and highlights some potential applications, including possible integration of autonomous technology.

KEYWORDS: automotive electronics, microcontroller units (MCUs), controller area network (CAN) bus, self-driving vehicles, network security, OSI model, session layer, transport layer, block ciphers

Chapter 1

Introduction

Most research on computer networks involve the kind of network most utilized in a home or business. These networks largely follow the same end devices (i.e. desktop computers, laptops, and smart devices), infrastructure (i.e. routers, switches, and hubs), connections (i.e. Ethernet and Wi-Fi), and protocols (i.e. TCP/IP). However, computer networks can have much further applications. There are plenty of wired-network infrastructures that can be applied to various fields.

One possible implementation is in a vehicle. Any type of vehicle, from planes to trains to automobiles to even spacecraft, can utilize a computer network to more efficiently operate itself. The end devices on these networks usually take the form of microcontrollers (MCUs), and they can be connected to each other either through a wired connection, be it Ethernet or a simpler medium, or a wireless connection, whether it's Wi-Fi or radio. The most impactful vehicular network is the one found in standard automobiles.

1.1 Background

Prior to the early 1980s, automotive manufacturers had their own methods of automotive networking. These networks were entirely proprietary in nature, and had very little similarities between brands. This made it harder for mechanics and repairmen to diagnose the same problem type across multiple makes and models.

The CAN bus network was first developed at Bosch starting in 1983. Formalization methods began in 1986, and an ISO standard [1] was created in 1993. [2] The mandate in the United States for all road-legal automobiles starting from MY 1996 to have On-board Diagnostics version two (OBD-II) compatibility further consolidated the use of the CAN bus network by all vehicle manufacturers. [3]

The CAN bus allows for multiple microcontrollers on a network to be connected to each other and exchange information in almost real-time. Average message time for a CAN bus message on a network can be measured in microseconds, and its especially low fault rate ensures that messages are properly sent the first time.

1.2 Existing issues

Because of the era in which the CAN bus was designed, there was no consideration of cybersecurity when the ISO standard for the bus was first drafted and approved. This means that there are multiple opportunities for the traffic on a CAN bus to be affected. A hacker could assume control of all devices on the network, and there would be no safeguards stopping any of their actions from occurring.

Among all possible events and their combinations, a hacker could especially do the following:

- introduce traffic onto the network and pass it to other MCUs without any form of message verification. This action is commonly known in the world of computer networking as *injection*.
- multiple, concentrated attempts to introduce faulty data to a MCU in an attempt to change its expected operation. This action is commonly known as *fuzzing*.
- assume the role of another MCU, or insert itself onto the network directly. This action is traditionally known as a *MitM attack*.

The introduction to the networks of any of these events could spell disaster not just for the vehicle itself, but the occupants inside it as well. An eye-opening research project in 2015 highlighted the sudden need for a better-secure automotive network, especially with the upcoming introductions of partial and fully self-driving vehicles to mass development.

1.3 Brief outline of proposed solutions

The proposed solutions to the issues are threefold:

- the introduction of a new frame structure utilizing multiple protocols that will make message tampering more difficult
- the reorganization of message exchange sequences to meet the needs of real-time efficiency while also maintaining a sense of security
- the integration of a security module that can monitor network traffic and react in the case of a network breach

1.4 Thesis organization

The rest of this research thesis is organized as follows:

- Chapter 2 highlights both prior research and existing efforts to address the issue of automotive security, as well as the specifics of the issues to be addressed.
- Chapter 3 considers a newer system model that can be used to address the problems defined in Section 1.2. The hardware and software designs of this system model are discussed here.
- Chapter 4 shows the establishment of the environment required for proof of concept, as well as detailed explanations for device testing.
- Chapter 5 examines the results collected from the tests defined in Chapter 4, and determines whether or not the results are in line with the desired outcomes.
- Chapter 6 is the thesis' conclusion, which considers how this updated CAN bus protocol can be most effectively and efficiently applied.

In addition to the above sections, there is an appendices section towards the end of the paper, where pseudocode for certain system model algorithms and source code used for testing and development are located and appropriately labeled.

Chapter 2

Model and research issues

This section focuses on the existing knowledge related to the CAN bus network. It also expands the focus on the issues defined in Section 1.2.

2.1 Context of current needs of the automotive industry

Figure 2.1 shows a general incorporation of all technology incorporated into the standard automobile. Every new, showroom-ready vehicle at an auto manufacturer's dealership incorporates all or some form of the technology mentioned in this figure. This research will focus on the network that connects these various devices together (CAN bus), with some emphasis on the devices themselves (MCUs). The research does not specifically focus on a particular brand's existing network implementation, but rather a proof of concept that can be considered in designing future automotive networks.

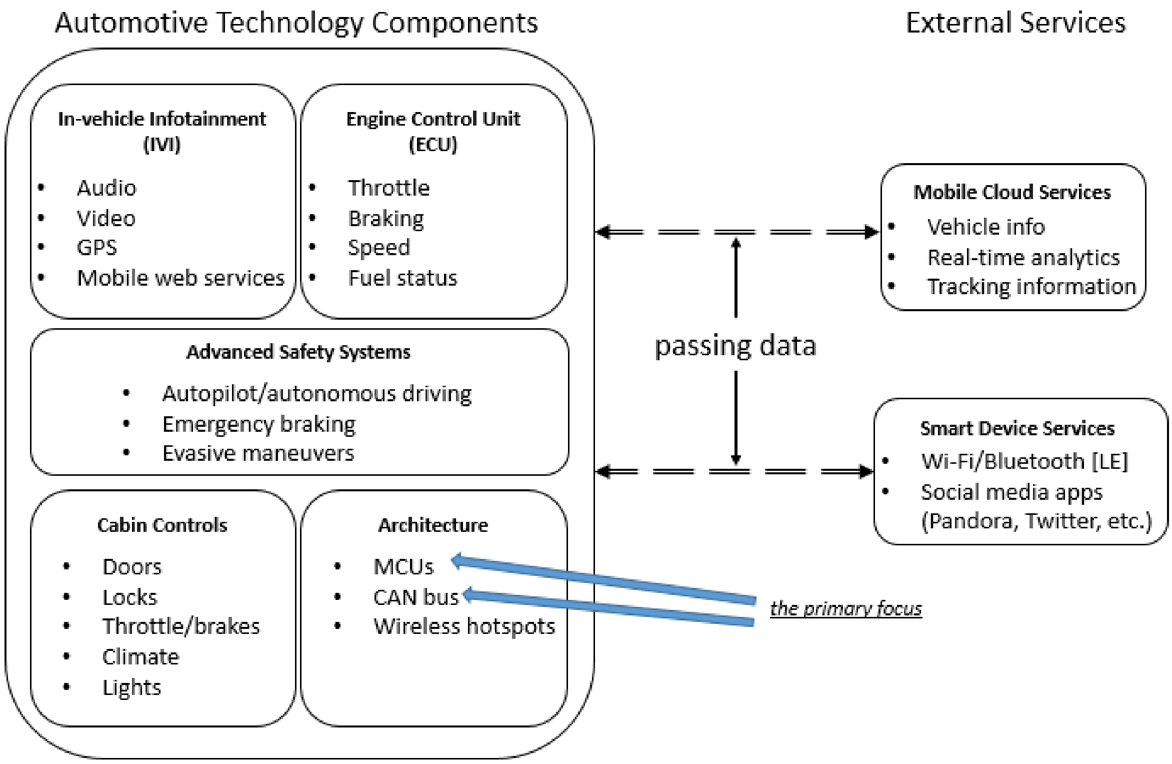


Figure 2.1: Technology incorporated into a new, showroom-ready automobile

The CAN bus plays an especially important role in this system. It is the medium through which, with the exception of the IVI category, all of these features are controlled and monitored. For this reason, any security issues related to the CAN bus have the potential to affect the operation of all of these features.

Figure 2.2 shows an expanded view on the areas of focus in **Figure 2.1**. A MCU of any vehicle can be implemented in a variety of architectures. While past manufacturers and parts suppliers used to rely on their own software to load onto the MCUs, many are starting to transition over to Automotive Grade Linux [4] as a sort of industry standard. Among these various aspects, the focus is further centered on the communication and network security aspects.

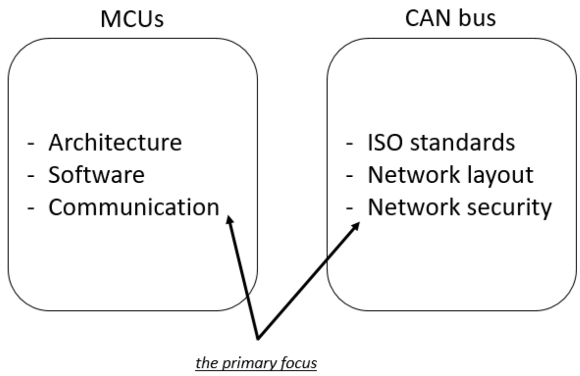


Figure 2.2: Expanded view related to **Figure 2.1**

2.1.1 Recent events

In 2015, Miller and Valasek made international headlines with their research [5] into the hacking of a showroom-ready Jeep Cherokee. The publication of the research was significant to the point where Fiat Chrysler, the manufacturer of the car, issued a firmware update that affected over 1.4 million vehicles. [6] The exploits developed utilized the lack of security on the CAN bus to send messages from the interconnected IVI system in order to control various features of the vehicle, from the lights and radio to the brakes and engine.

Following this event, automotive manufacturers began realizing the need to properly secure their automotive networks. Over the past five years, automotive cybersecurity has become a field of its own, as various conferences and events are held by IEEE organizations and Tier-1 OEM parts suppliers across the globe. Because automotive manufacturers usually outsource the research and development of their electrical systems to third-parties, these suppliers will be largely responsible for the actual implementation of any cybersecurity practices onto their devices and systems.

However, these efforts are still underdeveloped, and are mainly based on theoretical studies. An example of such research efforts is the *autoimmune* vulnerability [7] discovered in 2017, where a hacker can attempt a DDoS attack that can render networked MCUs unable to respond to messages. In addition, no outstanding public research or reporting shows any details regarding feasible implementations of more secure CAN bus-based systems for mass production vehicles.

2.2 Previous research

In this subsection, existing graduate-level research, field surveys, and other organizational efforts are analyzed and briefly summarized.

2.2.1 Field surveys

A few field surveys were published within the last few years with respect to overall CAN bus security. Dariz et al [8] conducted a survey of automotive security on the CAN bus for heavy-duty vehicles. Their research focused on the CAN bus in its current implementation compared to the Ethernet protocol with respect to the CIA security triage, and the authors suggested the implementation of a MAC-based communication method. Ring et al [9] researched various possible security attacks on a CAN bus network. Due to the amount of diagnostic work required to isolate a potential attack, they proposed to create a centralized database where automotive manufacturers and white-hat hackers alike could share information regarding potential exploits.

2.2.2 Definitive research

Masters-level research conducted by Bruton [10] analyzed the implementation of cryptographic algorithms on the CAN bus network. A series of hashing algorithms, such as RC4 and hash-based message authentication code (HMAC), and communication protocols, such as SSL, were benchmarked on a simple hardware-based recreation of the network. This research determined that certain implementations of cryptography would have no major impacts on network performance. However, this conclusion assumes that the implementations are implemented on a larger frame, such as CAN FD.

Masters-level research conducted by Yousef [11] centered on developing a custom protocol implemented on the CAN bus network. This protocol utilized existing protocols such as HMAC Timed Efficient Stream Loss-tolerate Authentication (TESLA) and contained certain security levels, including one to check for compromise of any network nodes. However, this research is incomplete as the same protocol was demonstrated to be compromised and had not yet been implemented on larger frame sizes such as CAN FD.

Research by Moore, Bridges, Combs, Starr, and Powell [12] researched and developed a method of detecting traffic anomalies automatically on a CAN bus network. They established a detection system that could determine the transmission of certain messages at improper times and frequencies. Said system was able to detect abnormalities with high precision and accuracy. Future research plans included more extensive testing on a larger subset of interfaces that utilize the CAN bus.

Dagan and Wool [13] developed a software program called *Parrot* that could be applied as a software patch to a MCU on a CAN bus network. Should that MCU be compromised, the software will force the MCU to disconnect itself from the network until the hacker no longer attempts to interact with the bus.

Woo et al [14] developed their own protocol by modifying the lower layers of the CAN bus to utilize both CAN FD and cryptographic hashing using algorithms such as AES-128 and SHA-256. They were able to successfully transmit traffic over the network within reasonable time and network load parameters with some of the algorithms.

2.2.3 Block ciphers

In terms of simplicity and overhead, block ciphers are one of the most efficient ways of encrypting and decrypting a data value for storage or transfer. A block cipher operates on entire groups of bits in one setting instead of one bit at a time. These block ciphers utilize symmetric key encryption, which uses the same key to encrypt and decrypt a particular set of data. While not as logistically secure as asymmetric key encryption due to the existence of only a combined public and private key and the difficulties of confidential pre-sharing of keys, symmetric key encryption in junction with the block cipher, when implemented properly, is especially useful for embedded systems with less-powerful processors than normal desktop computers. [15] [16]

Block cipher security, however, has been increasingly prone to discovery of vulnerabilities. Earlier block cipher technologies such as 3DES, Blowfish, and lower-bit sizes of the AES block cipher scheme have already been *cracked*, or rendered obsolete due to reported vulnerabilities that make it easy to convert ciphertext to plaintext without having access to the keys that created said ciphertext. In addition, within the past two years, even larger block cipher schemes such as

AES-128 have been cracked via side channel analysis. [17] [18] However, AES-256 still has not been cracked as of the publication of this thesis, meaning that it is still a reliable block cipher that can be used for lightweight encryption and decryption of information.

2.3 Desired research goals and areas of concern

The overall objective of the research is the introduction of a newer solution that can utilize existing hardware technologies to contribute as a possible solution to the issue of automotive security. As noted in Section 2.4.1, the CAN bus remains a lower-scale, efficient network that is best suitable for near-real-time data transmission.

2.3.1 Fuzzing

A very common form of CAN bus hacking is fuzz testing, or *fuzzing*. Fuzzing refers to a constant onslaught of data of any values to an unsuspecting device with the intention of negatively affecting the operation of said device. Devices under a fuzzing attack can be given false data to work with, or the type of data that is being sent may cause a critical system error that could lead to the suspension of any services required by that system. It is a very common type of cybersecurity testing with respect to *pen[etration] testing* [19] [20], but it has also been used in automotive network testing. [21]

The research conducted for this thesis will need to address the issue of fuzzing. **Figure 2.3** shows a sequential flow diagram highlighting how a hacker can perform a fuzzing attack on a CAN bus MCU. This is a simple example showing a communication of the same message [using binary form] between two MCUs on the same network. The actual passing of the data is an automated process; it is common for hundreds if not thousands of data entries to be passed to the MCU per testing cycle.

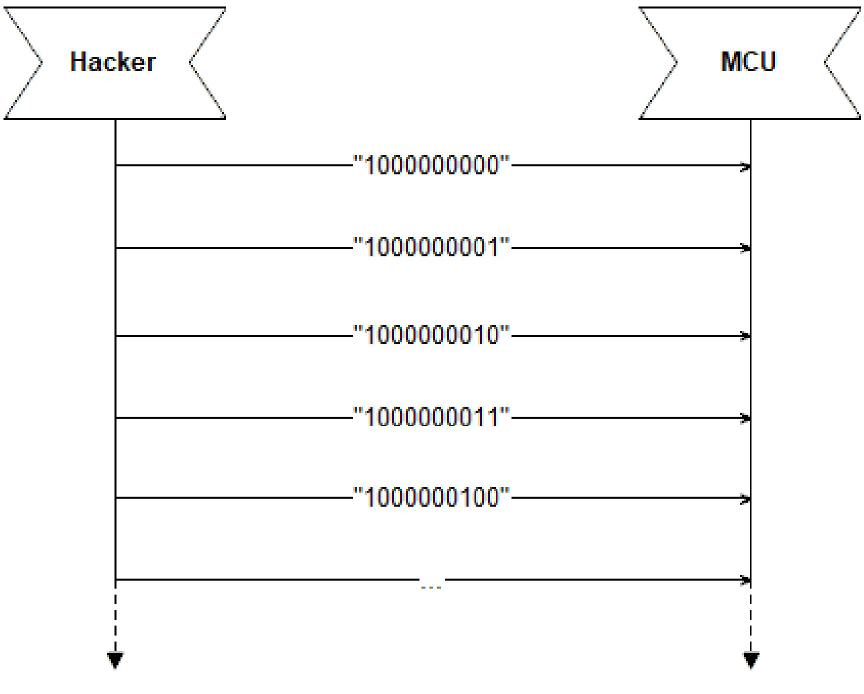


Figure 2.3: Fuzzing over a CAN bus network connection

2.3.2 Message injection

The current implementation of the CAN bus network does not distinguish the sender of a given message, nor does it have the capacity to keep track. This enables a hacker to send message frames over the network to any device that is defined in the control field. This high level of control can impede a network device's expected operation. This type of behavior has been defined by Miller and Valasek [22] as *message injection*.

This action shares a similarity with fuzzing defined in the previous subsection: they both rely on the lack of sender information for their success. However, fuzzing refers to a concentrated effort of sending various input to a specific target. Message injection can be used to attack any device on the network at any frequency of data sending.

The research conducted for this thesis will need to prevent the receipt and transmission of such message frames. It must either utilize a common method of identification, or determine a custom solution meant for the same purpose. This update should further allow a network device to discriminate the information being sent to it based on whether the sender is a valid network device or a hacker's injection.

The below points state the specific areas of concern:

- messages are sent and delivered to a network device without any data encryption or delivery information
- messages are sent and delivered to a network device with fraudulent or faulty identification and/or information

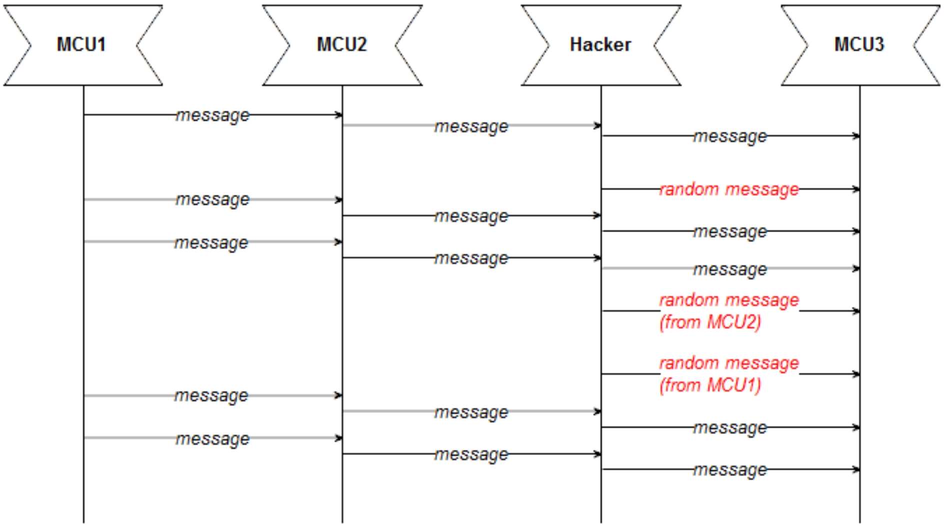


Figure 2.4: Injection of a packet on the CAN bus network

2.3.3 Man-in-the-middle (MitM) interference

A very common form of network infiltration is a practice called *man-in-the-middle* (MitM). The basic concept of MitM is that a third party intercepts and monitors the communications between two parties that are unaware of the existence of said third party. This kind of attack has been analyzed extensively in network security research. [23] [24] According to the layout of the CAN bus, which will be later outlined in Section 2.4.1, the interconnection of CAN bus nodes means that a third-party can be either appropriated from the network in the form of an existing node, or placed directly inside the network by connecting two adjacent nodes to it. With this access, a hacker can have the ability to alter or drop a message that passes its connection.

Figure 2.5 shows a sequential diagram highlighting a possible scenario on a CAN bus network. Because the MCUs on the network are chained together, a hacker can either assume control of an existing MCU on the network through some method (e.g. malignant firmware) or rearrange the network connection between two devices so that it passes along the network traffic.

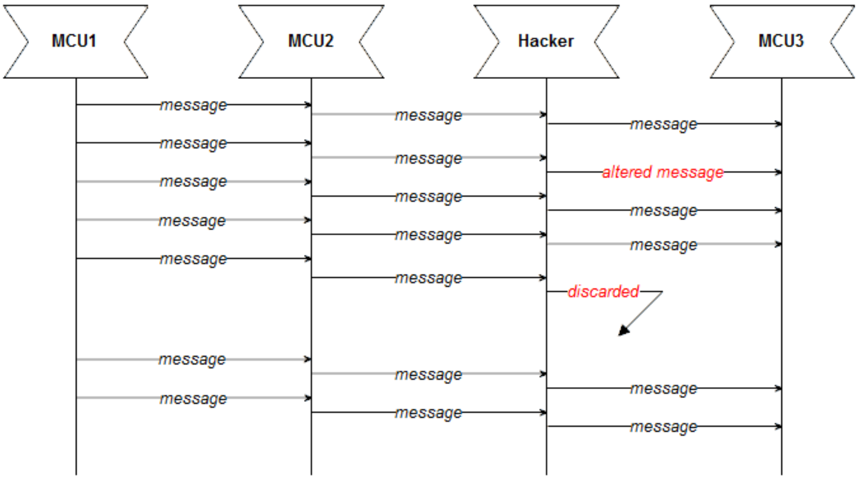


Figure 2.5: An example of a MitM-style attack on a CAN bus network

The research conducted for this thesis will have to ensure that any possible MitM attack is as mitigated as possible. Unfortunately, the overall network structure of the current CAN bus architecture will always guarantee a possible entry point for a MitM-style attack. That being said, there are still possible methods at which severe network calamity can be mitigated or avoided entirely.

Therefore, the areas of concern for this research is as follows:

- an alteration of some or all data passing through the connection of the third party
- the dropping of some or all data passing through the connection of the third party

Both of these scenarios are shown in **Figure 2.5**.

2.4 Existing tools and protocols

This subsection reviews the general layout of an actual CAN bus network, and how data is passed over it.

2.4.1 Overall layout of existing CAN bus protocol

The modern version of the CAN bus, defined under ISO 11898-2, consists of a series of MCUs chained together by twisted-pair wires. One wire is considered positive while the other is considered negative. The positive voltage wire runs at 5 V, while the negative voltage wire runs at 0 V. The wire pairs are terminated at both ends using 120 Ω resistors. [1]

When transmitting a signal over the wires, dominant and recessive voltages are used to determine binary logic. When no signal is being transmitted over the network, the overall voltage read across the wire pair is the recessive voltage, which is 2.5 V. To indicate a ”high” (1) signal, the wire pair transmits a dominant voltage of 3.5 V. To indicate a ”low” (0) signal, the wire pair transmits a dominant voltage of 1.5 V. All MCUs read the transmission voltage on the network as it is being transmitted; those whose CAN ID do not match the voltage simply ignore the message. [1]

There is no specific order for MCU transmission on the bus. If a MCU wants to use the bus, and nothing is being transmitted, it may use the bus. If two MCUs wish to use the bus at the same time, the MCU with the lower numeric CAN ID is allowed to transmit first.

Due to the scope size and complexity, this research will mainly focus on the following CAN bus features:

- There are four types of CAN bus frames: data, remote, error, and overload. Because the primary focus of the research is about message contents and their manipulation, this research will focus on the data frame portion only.
- CAN frames also have a priority identifier field, but for the purposes of this research, this identifier field will not be considered.

Figure 2.6 shows the layout of a standard CAN bus network, and **Figure 2.7** shows an example of network traffic at the physical layer.

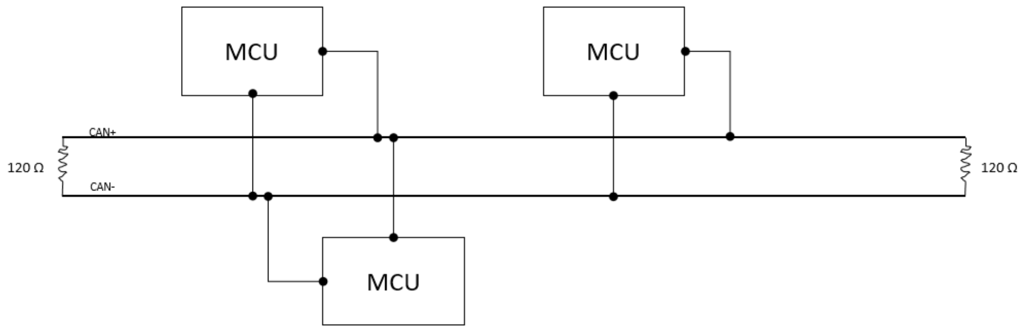


Figure 2.6: An example of a CAN bus network according to ISO 11898-2

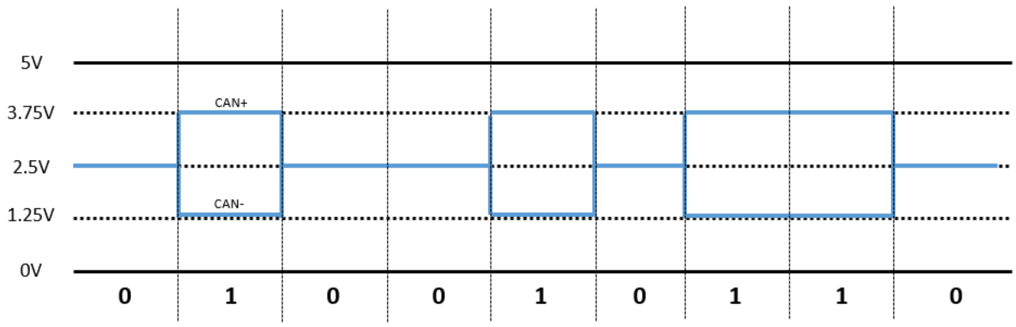


Figure 2.7: A view of data transmission over the CAN bus network at the physical layer

2.4.2 CAN bus speed

Because of the two-wire outline of the CAN bus, and the lack of signal repeaters on the physical layer, the overall speed of the CAN bus is inversely proportional to the length of the network. That is, as the length of the network increases, the maximum possible speed decreases. **Table 2.1** shows the implementation of a CAN bus speed. These figures are according to the ISO standard and are relative to the overall length of the network with respect to the "High Speed" CAN architecture. (The "Low Speed" CAN architecture is restricted to 125 kbps.)

2.4.3 CAN frame types

Modern-day implementations of the CAN bus rely on two different types of CAN messages. The first is CAN 2.0, introduced in 1991. CAN 2.0 allows for 11-bit (CAN 2.0A) and 29-bit

MAXIMUM NETWORK LENGTH (m)	MAXIMUM CAN BUS SPEED (kbps)
40	1000
100	500
200	200
660	100
1000	50
10000	5

Table 2.1: Comparison of CAN bus network speed related to network length between terminators

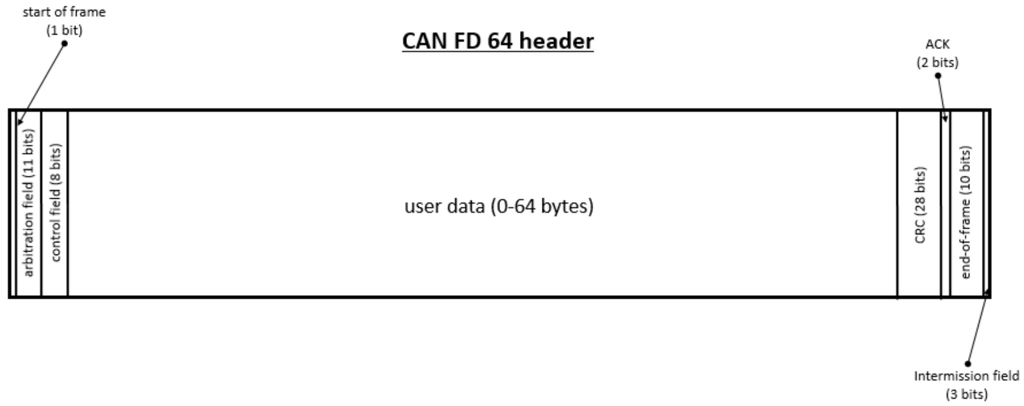


Figure 2.8: The current CAN FD frame layout

(CAN 2.0B) identifiers, used for device identifiers on the network. Both versions of CAN 2.0 contain a data payload field that can support up to eight bytes of data payload.

Bosch, the original creators of the CAN bus protocol, introduced the CAN FD extension in 2012. **Figure 2.8** shows the layout of the CAN FD frame, the largest frame utilized in current CAN-based architecture to date. The CAN FD frame can transmit up to 64 bytes of data, compared to CAN 2.0's 8-byte limit, which makes it possible not only to transfer larger messages with only one frame, but to do so almost four times faster than dividing the data into separate CAN 2.0 frames and then sending them all in order. [25]

2.4.4 Analysis of existing CAN bus implementations

The first vehicle with a CAN bus implemented was the Mercedes-Benz W140 in 1991. Since that time, all existing vehicles for sale on the general market include at least one implementation of the CAN bus. [26]

The CAN bus is one of the interfaces utilized by the OBD standard. The purpose of OBD is to make it easier to collect and analyze vehicle information across different makes and models. Its implementation has been a legal requirement for all roadworthy vehicles for sale in the United States since 1995, and there are similar iterations of OBD for the European and Japanese markets. [3]

What makes the CAN bus especially useful is that it supports transmission of multiple protocols on the same network. It is possible to send a CAN 2.0 message at one point, and then send a CAN FD message the next. This practice is possible as long as the recipient MCU on the network knows how to receive the frame format. This means that any solution derived in this research should theoretically be backwards compatible with existing hardware.

2.4.5 Automotive network security testing

There are a variety of testing software available for automotive network security testing. Synopsys [27] has developed a ”test tool suite” for the CAN bus across multiple ISO versions of the network. Li [28] has developed CANsee, an intrusion detection system (IDS) based on machine learning for usage on a CAN bus network. An open-source example is CANard (now named pyvit) [29], a Python-based CAN bus interface API that can be used to conduct certain security-based attacks such as DDoS.

However, at the time of this research, these tools are lacking in flexibility and features. Much of this software only deals with CAN 2.0 frames and not CAN FD frames. In addition, some of the available hardware or software tools that companies have developed are of a proprietary nature. Usage of these tools is known as *black-box testing*, where the hardware or software is given an input, and gives an output in exchange, without the user knowing anything about how the testing was performed. Some existing tools that are open source come with limited documentation. Usage of these tools is known as *grey-box testing*, where the schematics and/or code are provided to the user, but there is lacking, outdated, or otherwise nonexistent documentation provided with the tools. Furthermore, *white-box testing*, where the schematics and/or code are provided to the user, refers to the usage of tools complete with clear and understandable documentation regarding how the tools work. The effectiveness of each type of testing depends on the specific property of an application being tested. [30] [31] With respect to the above examples, using Synopsys' software would be considered black-box testing, while using CANsee and pyvit would be considered white-box testing.

2.4.6 OSI network model

Most computer networks follow the Open Systems Interconnection (OSI) model, which is a standardization of how these networks should be designed. The model divides a network protocol into layers that can be more easily defined. These layers interconnect with each other to provide a well-structured, efficient network.

The OSI model is divided into seven layers. From top to bottom, they are:

7. *Application* - This layer is where user data is handled by any applications local to the device. Application programming interfaces (APIs), or sets of functions used for manipulating this user data, can be considered a part of this layer.

6. *Presentation* - This layer handles the interpretations between application and session layers. It is the most flexible layer in the model, as both surrounding layers can also perform certain tasks that the presentation layer is responsible for.

5. *Session* - This layer maintains the communication sessions between two devices communicating with each other. It guarantees that requests for data and subsequent responses are appropriately met.

4. *Transport* - This layer ensures that all communication between two specific devices on a network are properly ordered and maintained. It can, and should, handle events such as missing messages, data corrupted during transmission, and acknowledgement of one device's receipt of the other's data.

3. *Network* - This layer is responsible for communication and routing between multiple networks of devices. If two or more devices do not share a network, this layer is responsible for bridging the gap between all devices in terms of device identification and translation.

2. *Data link* - This layer constructs the frames that messages between devices are sent in. It defines how to interpret the various signals being transmitted.

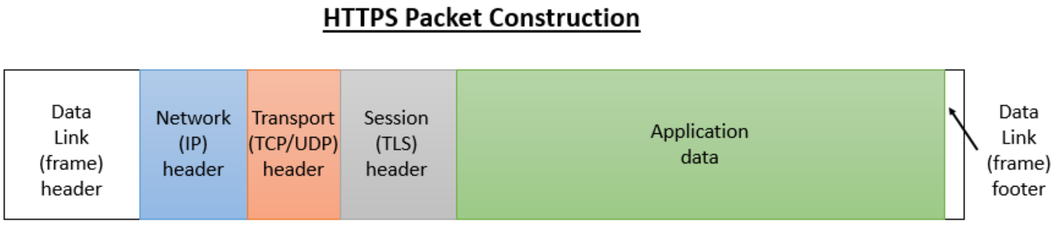


Figure 2.9: The structure of a standard Internet packet

1. *Physical* - This layer focuses on the transmission of bits (0s and 1s) across either a physical medium, such as a wire, or a broadcasted medium, such as a wireless antenna.

Under the OSI model, the CAN bus adheres to the first, second, and seventh layers. [32] This is because, due to the age and simplicity of CAN bus network construction, the other layers are not required for consideration under the ISO standards that define the CAN bus.

2.5 Properties of encapsulation

To address the need for security and ordering credentials, it is necessary to update the existing CAN frame structure to include this information by default. The information from this protocol can be used to maintain a proper message order while preventing messages from randomly being inserted into said order.

To understand how the CAN bus will be used in this protocol, it is important to first understand how a given network packet is structured. Network packets usually utilize a process called *encapsulation*. When a frame is created in the data link layer, a series of bits are arranged to help a *network interface card* (NIC) determine the difference between an actual frame and mere random chatter on a network. Another series of bits ends the message; that is, the NIC knows after it reads this series of bits that the frame has finally and totally arrived at its destination, and the NIC can then start listening again for other messages. Each frame consists of a header, which contains information such as physical addresses, and a payload, which stores the user data to be relayed to the device.

For ascending layers, the layout is almost the same. Each layer's protocol uses a header first, and user data afterwards. The information they provide can be considered ”stackable”, much like a set of building blocks on top of one another. After the frame's header information, the first few bits of the user data are used by the header of the next highest layer. After that layer adds its header, *its* first few bits of user data are used by the header of the next highest layer. All of these layers, however, do not require a series of bits at the end stating that the layer's data is complete; these layers are considered as wholesome up until either the start of the next highest layer's header or the end of the frame itself. **Figure 2.9** shows an example of the structure of a HTTP over TLS (HTTPS) packet, which utilizes all seven layers of the OSI model using encapsulation.

When a certain layer wishes to examine the contents of the frame relevant to its layer, it can interpret the data by starting to read it after a certain offset. Say for example that in the packet of a fictional network protocol, a frame has two layers stacked on top of it, with the lower layer starting at byte d of the frame and the upper layer starting at byte e. For the lower layer to read its data, it starts reading at byte d, as the bytes from 0 to d are relevant to the frame, and are therefore irrelevant to the lower layer. The upper layer follows the same logic, except it starts reading from byte e, bypassing the information of the frame and other layer. This protocol usually assumes that these offsets are constants; that is, each valid packet being transmitted will always have a frame header offset of $d - 1$ bytes and a lower level header offset of $d + e - 1$ bytes.

Chapter 3

System design

This section focuses on the proposed system to address the problems in Section 2. The approaches to address the issues mentioned in Section 2.3 are discussed from a top-level basis.

3.1 Revised frame structure

Figure 3.1 shows the finalized layout of the proposed protocol's frame layout. This layout is over the CAN FD frame outlined in **Figure 2.8**. In the section of the CAN FD frame for user data, a total of 27 bytes are allocated for transport and session layer headers. These headers are not the exact same as the TCP or TLS headers as specified in **Figure 2.9**. They have been updated to better meet the needs of the overall CAN protocol.

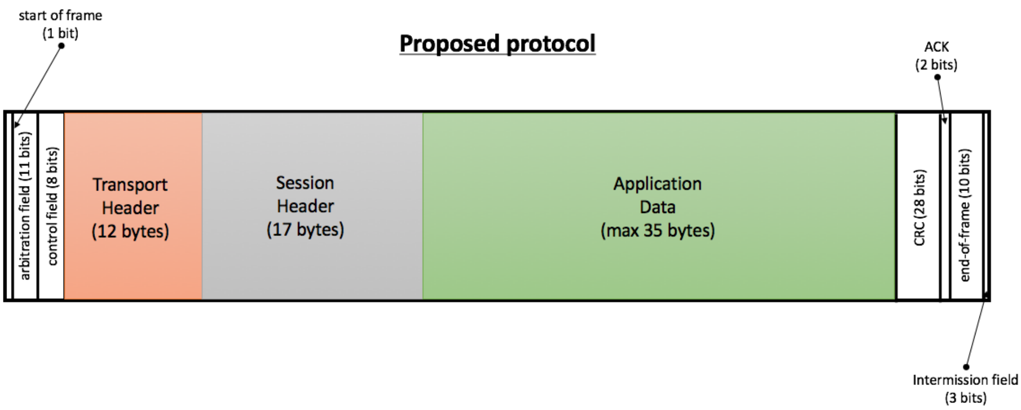


Figure 3.1: The overall proposed frame layout

There also is no specification of the network layer in this new protocol. This is because it is simply not required. The network layer is responsible for routing between multiple networks like the Internet, which is not required for simple local networks like CAN. [32]

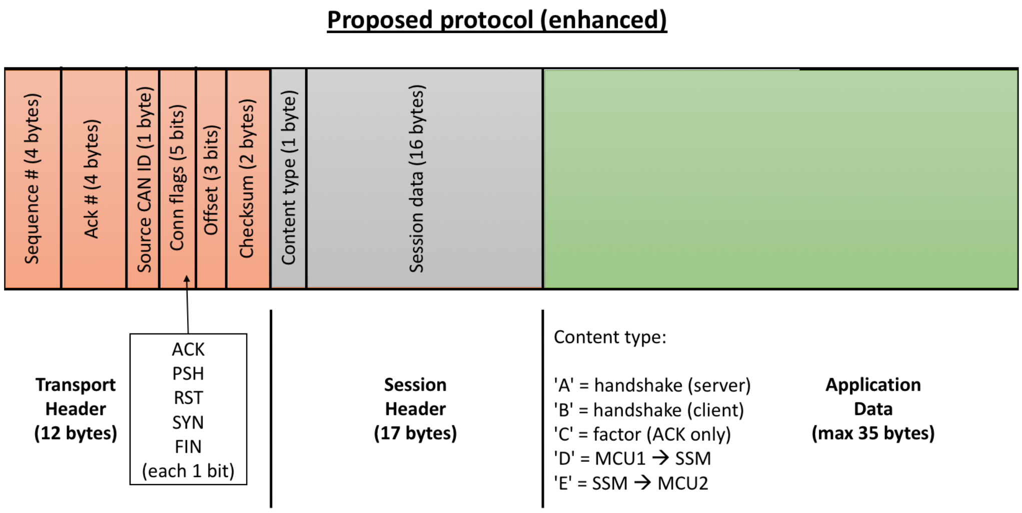


Figure 3.2: Enhanced view of the proposed protocol as specified in Section 3.1

Figure 3.2 shows an enhanced view of the frame's updated structure within the payload field of the original CAN FD frame in **Figure 2.8**. The frame structure uses trimmed down versions of TCP and TLS in order to save space while maintaining sequential orders and content encryption. More information regarding the differences between the transport and session layers defined here are further explained in Section 4.1.1.

Table 3.1 shows an expanded outline of the session layer based on content type as noted in **Figure 3.2**. For the connection between MCU1 and the SSM, the session layer can be used to determine the destination of the application data. For the connection between the SSMand MCU2, the session layer transports the variables required to compute the AFV for the SSM. All session layer data are encrypted, and are tested to work with transmission of four-byte floating point integers.

CONTENT TYPE	CONTENT MEANING	HEADER CONTENTS
'A'	Handshake (server)	Encrypted password
'B'	Handshake (client)	Connection status code (0x19 for invalid, 0x32 for OK)
'C'	Factor (in MCU2 ACK only)	Factor as a result of operation by MCU2
'D'	Application data from MCU1 to SSM	Destination of data
'E'	Application data from SSMto MCU2	Application data from packet *n-1* for factor calculation

Table 3.1: Session layer contents based on content type

3.2 Third-party monitor

The above mentioned protocol was designed to secure the communication between two MCUs on a given network. However, this protocol by itself cannot totally prevent a hacker's attack. Therefore, it is important to provide an independent third-party that can collect and monitor passing traffic over the network. To this end, a SSM is proposed.

The SSM assumes a slightly different role from other MCUs. It listens to network transmission like the other MCUs, but it is not for vehicular module operation. Instead, it serves as a trusted third-party that establishes a connection between two particular MCUs. The original version of the CAN bus has no device like this, and based on the prior research conducted in Section 2, no other proposed solutions have suggested something similar. With this monitor implemented, it decreases the responsibilities of individual MCUs while increasing the amount of security on the overall network.

Consider an airplane's flight recorder, or "black box", as a partial example of how the SSM should work. The black box collects airplane statistics and cockpit interactions to be used for future investigations and analysis. The SSM, in a similar vein, would maintain recent copies of traffic sent in the vehicle between MCUs. It would store this information in a stack-like data structure, with the ability for stack entries containing to be "popped off" in case of an emergency. (Further explanation of stack behavior is explained in Section 3.2.1.)

The SSM also has an additional role. In addition to collecting network information, it must be prepared to assume a "balancing" role in the case of a network breach or other nefarious attack by a hacker. The SSM must be able to prevent the hacker from assuming any more damage to a network, while supplying information as needed to the rest of the network both during the attack and after the attack has taken place. With the SSM in place in the overall system, the overall security of the network strengthens, much like a firewall increases the security of a local area network.

Figure 3.3 shows a view of the SSM through a finite-state machine diagram. There are three main states that the SSM follows:

- *State 0* - This state is for normal operation. When an attack, or "breach", is detected, it goes immediately into state 1.
- *State 1* - This state is for initial detection of a breach on the network. An error timer is set, which initiates a *cooldown period.* If there is a reconnection before the counter expires, it returns to state 0. Else, it moves on to state 2.
- *State 2* - This state is for handing archived data to the remaining connected MCU. Once a safe reconnection is made to the other MCU, it returns to state 0.

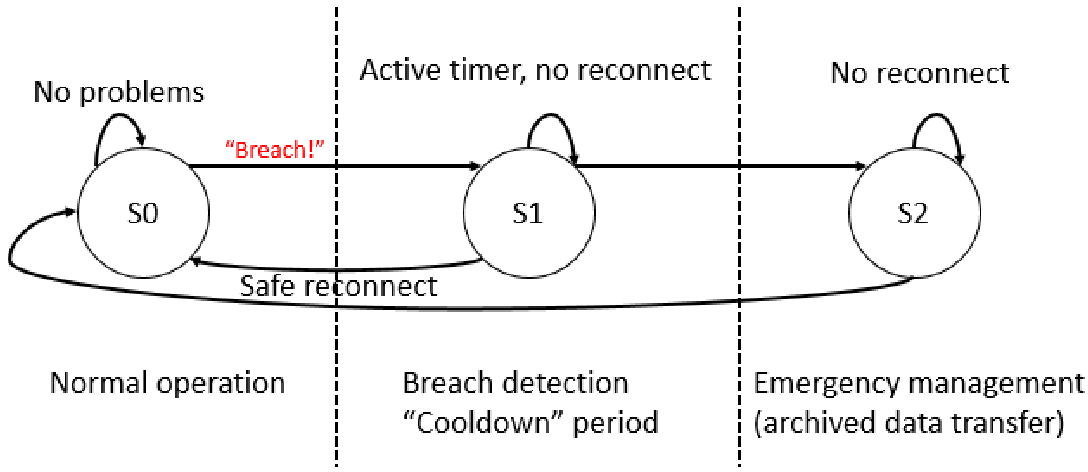


Figure 3.3: Finite-state machine diagram of the SSM assuming its duties of preventing a network breach from continuing

A further explanation of these states is given later on in Section 3.3, as well as in **Figure 3.9**.

A certain degree of network awareness is required that can determine if any malignant activity is being performed or attempted. This role is assumed by the SSM, which serves as a trusted third-party, monitor, and gateway for messages to be transferred over the CAN bus network. The SSM will be able to determine hacking attempts similar to the ones defined in Section 2.3.

3.2.1 Data stack

The SSM's roles outlined in Section 3.2 are realized through the implementation of a stack on the module. Each entry on the stack is for one message that is sent from one MCU to another. The stack takes note of the following attributes:

- the time that the message was received in Unix seconds
- the message's destination
- the message's source
- the actual message being sent [in encrypted form]

The stack can hold up to *n* entries, which would allow the SSM to have the most recent data utilized while not keeping the entire history on the device. In the event where the stack is full, the stack entry at *n* is removed, as such data is not recent and therefore no longer relevant to the overall context of the network. **Figure 3.4** shows an example of what the stack would look like. The value of *n* itself should be determined by the number of devices utilizing the network and the overall rate that messages are passed over the network. Said value can vary on the type of MCUs and variety of data involved.

stack entry	attributes: date (s)	to	from	data
1 (top)	1805965	dev1	dev3	xxxxxxxx
2	1805904	dev2	dev3	yyyyyy
3	1772931	dev3	dev5	74
4	1771110	dev4	dev1	Kansas City
5	1762445	dev2	dev3	2
...				
n	1409065	dev5	dev1	aaaaaaaaa

Figure 3.4: Example of a stack maintained by the SSM

Figure 3.5 shows the general parsing process of the stack at the detection of the breach. There are two important pieces of information utilized for parsing: the time in Unix seconds of detection, and the device which had been compromised by the hack. The SSM checks the stack first for all messages sent after the detection time. (Older messages on the stack do not have to be considered.) It then parses those entries for messages that have been sent by the compromised device. If any entries are found after the detection time with the same sender ID as the compromised device, they are removed from the stack.

In realizing the need for archived data in **Figure 3.9**, the latest message(s) sent from the device before the breach is sent again. This data is used until there is a successful and safe reconnection between the SSM and the compromised device once the hacker is finished. The status of the stack post-breach is shown in **Figure 3.6**.

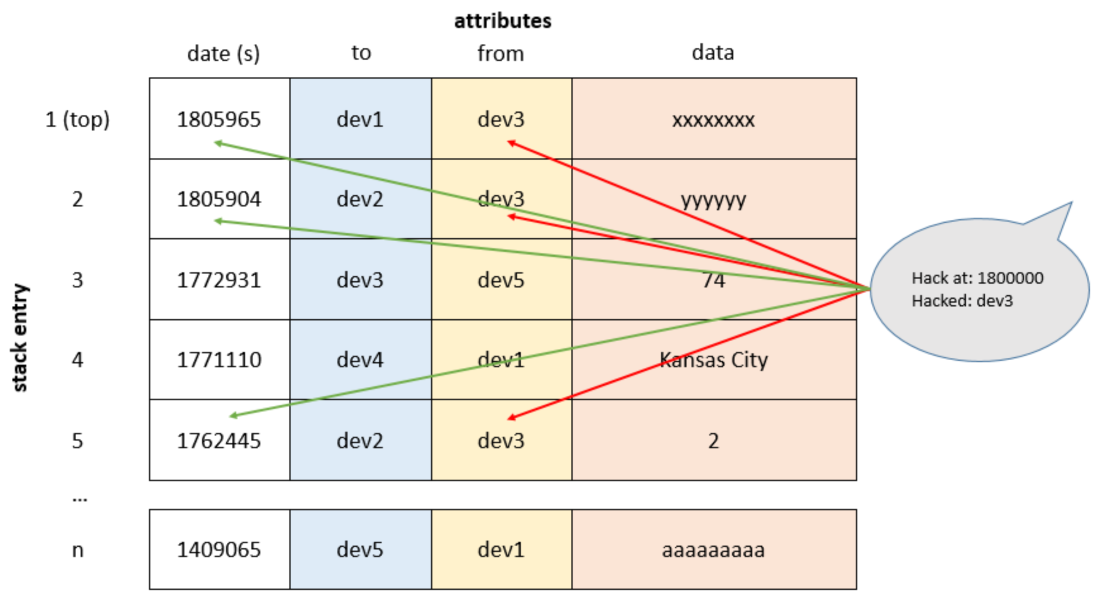


Figure 3.5: Stack parsing based on event data

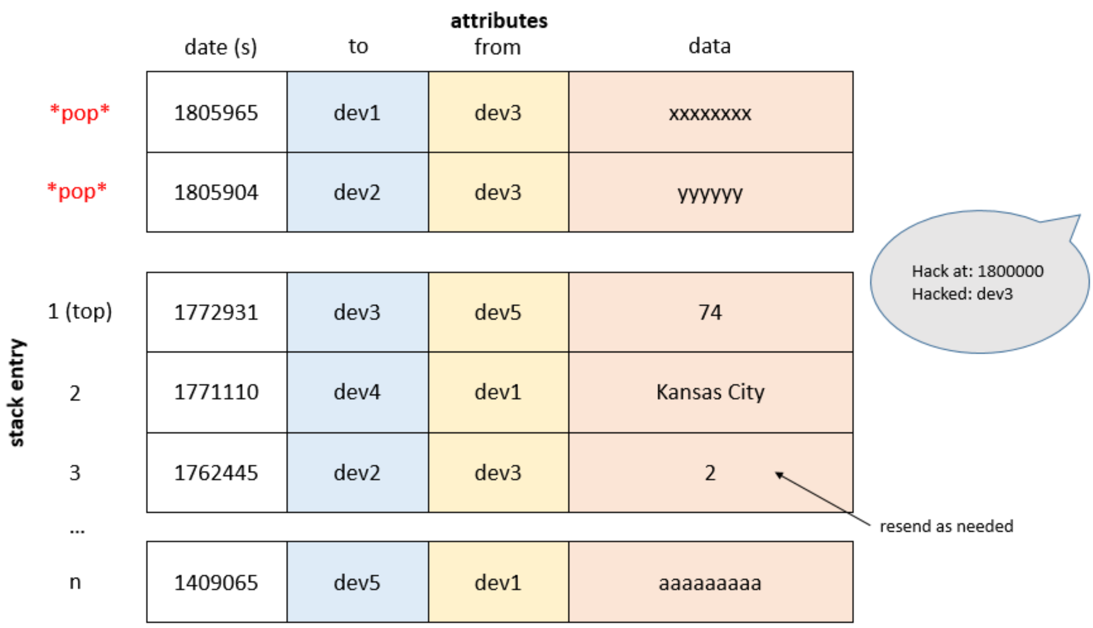


Figure 3.6: Reconfiguration of stack post-event

3.2.2 Clustering algorithm monitoring

In parallel to the above stack creation and maintenance, and to determine that a breach has actually occurred, a simple clustering algorithm is introduced. (The pseudocode for this layout is included in **Appendix A.1**.)

It is not proper protocol design to handle the application data directly, especially when it is first encrypted and has to be decrypted. Therefore, the application data shall be encrypted using a symmetric block cipher with a pre-shared key among all of the MCUs. In this manner, both MCUs, as well as only the SSM awareness application, will be able to encrypt and decrypt their data in a manner using relatively little overhead. In the meantime, the SSM itself will be unaware of the actual data that is being sent over the network.

As data is collected by the SSM, it is stored inside a data set of *a* entries, and the *b* most recent data sets are preserved. From each of these data sets, the factors are calculated between each data entry, and these results are stored in their own data set of *a-1* entries. It should be noted that *a*, *b*, and *n* are all developer-determined. That is, these variables should be chosen by the network engineer responsible for implementing the system based on any and all of the following:

- the number of devices on the network
- the data rate of the devices on the network
- the amount of time to reference when trying to determine a hack

From there, the AFV is calculated as an average from each data set and compared to each other. From these observed values, four distinct patterns can be determined. To explain each pattern, the measurement of speed and acceleration can be used.

- *Consistency* - there have been no overall changes in the data being sent. For example, if a vehicle is travelling at a constant 50 km/h, the AFVs should equal, or come very close to, zero.
- *Gradual change* - a steady, non-zero rate of change is being observed for that particular point in time. For example, if a vehicle is accelerating from 50 to 80 km/h, or decelerating from 50 to 30 km/h, the AFVs should be non-zero, but not numerically far apart from each other.
- *"Blips"* - one or few non-zero AFVs of negligible amount. For example, during a consistency pattern, during one of the many data values being transferred per second, a glitch in the hardware or software may send a single message saying that the vehicle is travelling at 70 km/h instead of 50. The inclusion of this particular data point, or *blip*, will make the resulting AFV for the set that it's in a non-zero number. However, when compared to the consistency of the other average data values, this non-zero value observation may be of no significant meaning or consequence, and can thus be safely ignored.
- *Unreasonable change in data* - one or few non-zero AFVs of a significant amount. For example, through the actions of a hacker, the vehicle's speed, and messages, are suddenly increased from 50 to 100 km/h. Compared to the gradual change pattern, where the observation is being monitored in terms of driver action, this pattern relates to a very sudden change in data, too fast for any driver or automotive component to change to. From the observation of this pattern, it is most likely that the vehicle's network is under attack by an external threat.

The clustering algorithm is responsible for the detection of the hack. In the event that a hack is detected, the algorithm passes the time that the hack was first detected and the connection information to the stack manager, which will then use that information to filter the potentially malignant data out of the stack.

3.3 Message exchange sequences

An updated protocol and third-party monitor alone will not be enough if there isn't a reliable, consistent method of handling device communication. Therefore, this research will focus on two different communication scenarios: connection initialization and hacking scenarios.

3.3.1 Initialization

In TLS, which is what this project's session layer is based on, a rather lengthy handshake process is required to connect two devices with each other. The general process is as follows:

1. The first device sends a packet to the second with the SYN flag set to high.
2. The second device responds to that packet with another packet with the SYN and ACK flags set to high.
3. The first device responds to that packet with another packet with the ACK flag set to high.
4. The first device then sends a *TLS ClientHello* packet to the second device to secure the connection.
5. The second device responds to that packet with a *TLS ServerHello* packet.
6. The second device then sends its certificate information, followed by a *ServerKeyExchange* message and a *ServerHelloDone* message.
7. The first device replies with a *ChangeCipherSpec* message, changing its communication over to an encrypted format.
8. The second device replies to the above with a *ChangeCipherSpec* of its own.

This process is too long and mostly unnecessary for the CAN bus network. This is because the CAN bus is a real-time network, and the theoretical amount of time needed to complete the entire exchange can significantly impact the overall performance of the network. In addition, this process is supposed to be compatible with multiple versions of TLS and the various types of certificates to be exchanged. On this network, for the sake of simplicity and reduced overhead, only one type of cipher suite and protocol is necessary. Therefore, many of these processes and requirements can be cut out, keeping the overall connection establishment time short.

Therefore, the above list requires modification for the system model. **Figure 3.7** shows an example of the standard connection, message exchange, and termination process. The connection process is as follows:

1. The first device sends a packet to the second with the SYN flag set to high. The session layer portion contains an encrypted version of the password to access the device.
2. The second device responds to that packet with another packet with the SYN and ACK flags set to high. If the password is also correct, a success notification will be included in the session layer.
3. The first device responds to that packet with another packet with the PSH and ACK flags set to high.
    - In the instance of the connection between MCU1 and the SSM, MCU1's intended destination (MCU2) is defined in the session header. Before returning the ACK, the SSM makes a full connection to MCU2 first before returning the ACK for MCU1.
4. The second device responds to that packet with another packet with the ACK flag set to high. At this point, application data may be sent between the two parties.

The termination process is as follows:

1. The first device sends a packet to the second with the FIN flag set to high.

2. The second device responds to that packet with another packet with the FIN and ACK flags set to high.

   - The SSM will send its FIN packet to MCU2 before responding to MCU1's FIN packet.

3. The first device responds to that packet with another packet with the ACK flag set to high.

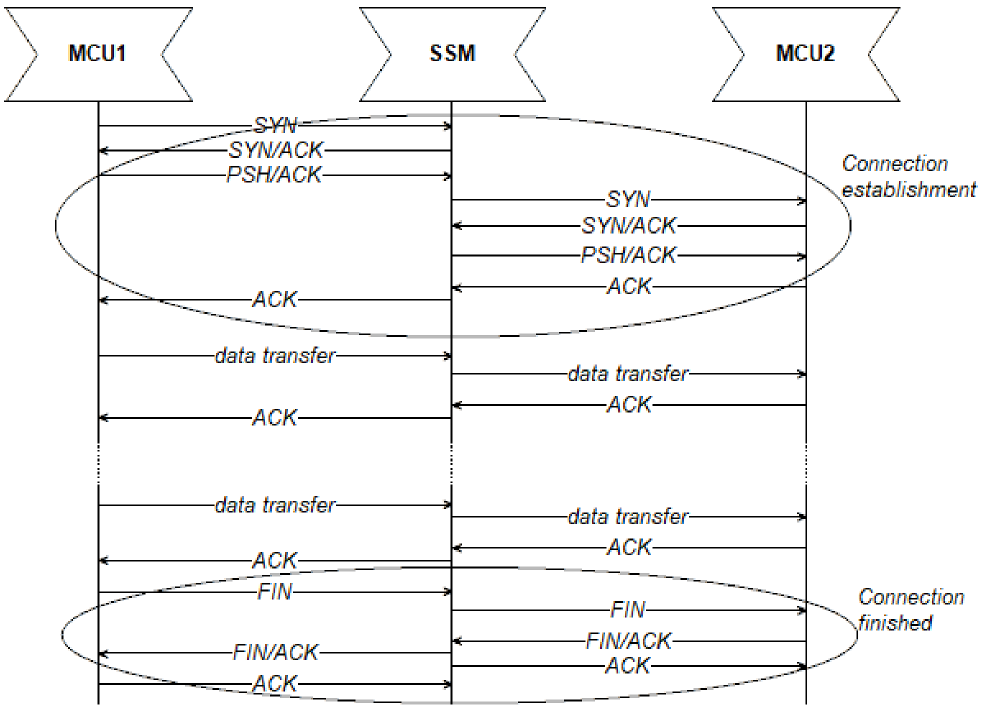


Figure 3.7: Connection initialization, standard message transfer, and connection termination

Figure 3.8 shows the standard behavior of any given MCU, including the SSM, on the network through a finite-state machine diagram with respect to the transport layer. There are six main states that a MCU follows when making and using a connection. (A relationship between one other MCU is assumed.)

- *State 0* - In this state, there is no connection to any other MCU, and the MCU is waiting for either data to send or a request to connect from another MCU. From here, this state can transition either to states 1 or 2.

- *State 1* - This MCU has data to send the other MCU and wants to establish an existing connection with another MCU. Once the connection is established with the other MCU, this state transitions to state 3.

- *State 2* - This MCU is receiving a connection request from another MCU to exchange data. Once the connection is established with the other MCU, it moves to state 3.

- *State 3* - There exists is a connection between the two MCUs, but no data is being exchanged between them. From here, this state can transition either to states 4 or 5.

- *State 4* - This MCU wishes to send data to the other MCU. After it is done sending data, it returns to state 3. However, should this MCU wish to terminate the connection (FIN), the connection is closed and the state transitions back at state 0.

- *State 5* - This MCU is receiving data from the other MCU. After it receives all the data, it returns to state 3, with the exception of a message received with a wish to terminate. Should that happen, the connection is closed and the state transitions back to state 0.

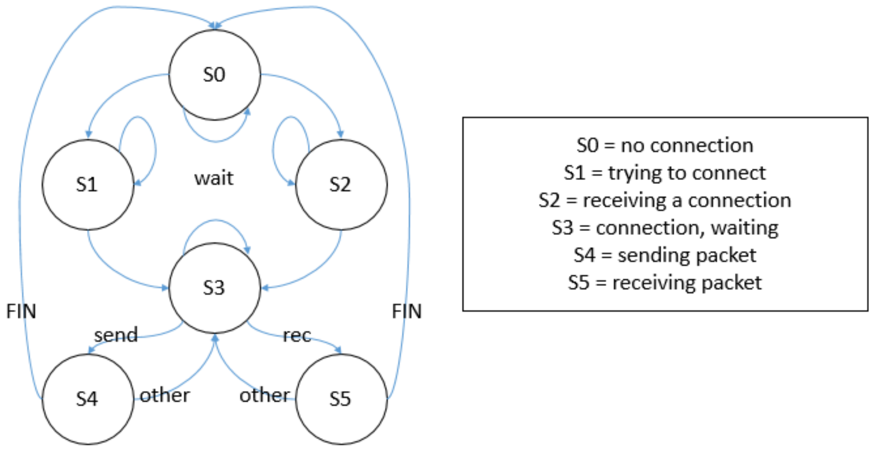


Figure 3.8: Finite-state machine diagram of the connection status of a MCU on the updated CAN bus network

3.3.2 Standard message transfer

The message process, as shown towards the bottom of **Figure 3.7**, shows the general process of sending data from MCU1 to MCU2. The general process follows the below steps:

1. MCU1 sends data for MCU2 to the SSM.
2. The SSM records a copy of the data, in encrypted form, for its monitoring purposes. Should there be no issues with the data or the connection, it sends the data to MCU2.
3. MCU2 sends an ACK back to the SSM to signal that the data transfer was a success.
4. The SSM sends an ACK back to MCU1.

This process repeats for the entire session, until MCU1 wishes to terminate the connection.

3.3.3 Hacking scenario

In the case of a hacker's presence on the network, the system must act fast to address and mitigate the problem. It is realistically impossible for a system to be able to stop every attack at the very instant that it occurs, but it is nevertheless important to detect the attack as soon as possible. **Figure 3.9** looks at the reaction of the SSM in the event of an attack. In this example, through some method, a hacker assumes the identity of MCU1 and tries to send malignant data to MCU2. The SSM, through methods already explained earlier in this section, will detect the actions of the hacker and terminate its connection with the hacker. The SSM then activates an error timer for the cooldown period. By this point, the hacker will notice that their activity has no effect and will stop. If the SSM successfully reconnects to the true MCU1 within this period of time, data transfer will resume as normal. However, as seen in the figure, if the error timer times out before the connection can be remade, it will then resort to sending archived data that it has collected to MCU2. While it is not the absolute most recent data, it is the latest data that the SSM has that was actually sent by MCU1 and not sent by the hacker masquerading as MCU1.

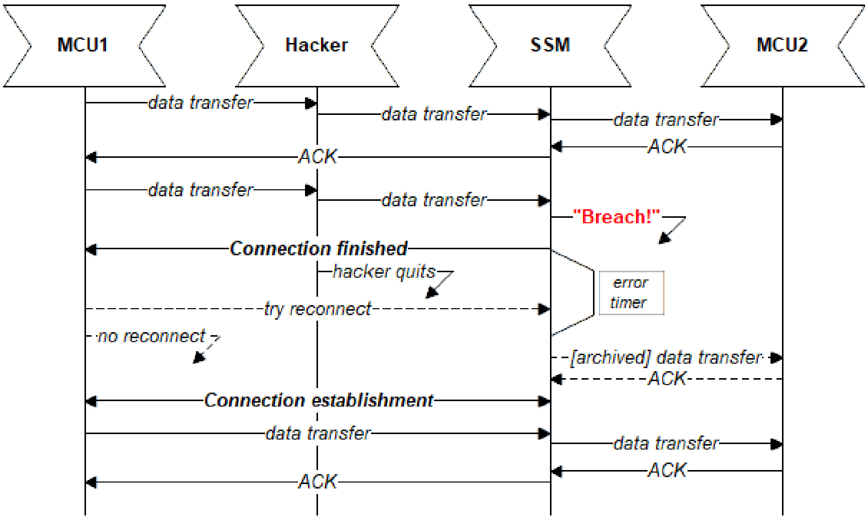


Figure 3.9: SSM reaction scenario in the case of an attempted breach

One use of the ACK frames used in this network will be for factor transferring between the SSM and MCU2. The factor values collected by the SSM will be sent directly to the SSM's awareness application, where they will be decrypted and added to the algorithm. In the case of the hack detection, the awareness application will send the relevant hack information to the data stack application as specified in Section 3.2.1, and that stack application will set aside the data required for the SSM should it be required for retransmission.

3.4 Application of system model to the noted problems

The system model proposed in this section will be able to prevent the problems mentioned in Section 2.3 in multiple aspects. **Table 3.2** shows a simplified version of the solutions proposed by this model.

SOLUTION	Fuzzing	Message injection	MitM
Updated protocol	◯	◯	x
Updated connections	◯	◯	△
SSM	△	x	◯

Table 3.2: Comparison of system model solutions to areas of concern as defined in Section 2.3

The updated protocol from Section 3.1 will include important sender and connection information that will be required before any data is passed to the application layer. This solves the issues of fuzzing and message injection, as both of these issues exploit the lack of sender information and order control. The MitM attack is outside the scope of the updated protocol.

The updated connection procedures from Section 3.3 will contain session information that will prevent any random messages due to fuzzing or unexpected messages due to message injection from occurring on the network. The hacker will require credential information in order to make a valid connection, and cannot send data otherwise. The updated connection process should theoretically prevent a MitM-style attack as well, but because of the nature of MitM attacks, the threat is not entirely eliminable due to the layout of the network and the control that the MitM device could have. At the very least, it will be much harder for that device to make actual changes to the encrypted data (thanks to the properties of asymmetric encryption), or the

SOURCE	Message protocol	CAN FD	Network awareness	Session layer	Trusted intermediary
Bruton	x	x	x	△	x
Yousef	◯	x	◯	x	x
Moore et al	x	x	◯	x	x
Parrot	x	◯	◯	x	△
Woo et al	◯	◯	x	x	x
Putnam	◯	◯	◯	◯	◯

Table 3.3: Comparison of research results to this thesis' research

underlying data in the rest of the CAN frame (depending on the context that a hacker may have prior to any attack).

Finally, the SSM will be able to determine any sudden changes in data in the event that a connection is hijacked, or a hacker manages somehow to create a valid connection. In addition, it should also prevent the effects of fuzzing from wreaking havoc on the system, as the more data the hacker sends, the more information the SSM has to determine that an attack is taking place. However, the SSM has no control over the amount of network traffic, which in its current form could leave open the possibility of a DDoS attack. The message injection attack is outside the scope of the SSM.

3.5 Comparison of model to previous research

Table 3.3 shows the comparison of this system model compared to the facets of other efforts of research. The system model focuses on the following features:

- a customized message protocol
- support for larger frames, mainly of the CAN FD type
- utilization of a session layer
- implementation of a trusted intermediary (in this case, the SSM)

The research mentioned in Chapter 2 was analyzed and their results categorized according to the parameters of **Table 3.3**. The masters-level research and a few external papers were analyzed against the research for this thesis.

Bruton's research experimented with different forms of cryptography on the CAN bus network. One of these methods were what the author referred to as "out-of-band SSL," which is essentially an implementation of TLS. It is possible that while SSL was tested, and proven to work properly within a reasonable amount of time, the CAN bus frame may not have been utilized. The author had no research related to the other categories.

Yousef's research also involved the creation of a customized protocol. The author also went farther to define a series of security levels, and how the network should react at each level. However, this author's research involved using hashing for message verification instead of something connection-oriented like TLS. It also had no trusted intermediary, nor was it tested on the CAN FD frame.

The implementation of *Parrot* satisfies the needs of network awareness and CAN FD support. It does not, however, deal with the network layers directly; at best, it is an application layer device. It also relies on cryptographic hashing instead of TLS.

Finally, the new security architecture suggested by Woo et al utilized an updated frame and FD support. But it also utilized cryptographic hashing instead, including AES-128, which has specifically been proven prone to cracking via side channel analysis. [17] [18] It is also missing network awareness and a trusted intermediary.

The system model proposed in this section seeks to further the research conducted by the above researchers and others. It will do so by building off of the ideas that the researchers have come up with, incorporating a singular solution that will include all five categories. **Table 3.4** shows the comparison of implemented solutions to those proposed by the examined authors. As derivable from the table, the solution proposed by this paper should be the most comprehensive with respect to the problems that it intends to solve.

SOURCE	Custom message protocol	Redesigned connection process	Trusted intermediary
Bruton [5]	x	x	x
Yousef [6]	◯	◯	x
Moore et al [7]	x	x	x
Parrot [8]	x	x	△
Woo et al [9]	◯	x	x
Putnam	◯	◯	◯

Table 3.4: Comparison of implementation results to this thesis' research

Chapter 4

System model and test implementation

With the system model design specified in the previous section, this section will focus on the implementation of the system model as well as the establishment of the testing environment. To test the effectiveness of the system, a virtual CAN bus network recreation without the solution implemented, and then with the solution implemented, will be tested using the exact same tests to determine the performance and effectiveness of the new system.

4.1 Software layout

4.1.1 System model implementation

The implementation of the revised frame structure outlined in Section 3.1 and the updated message exchange practices in Section 3.3 follow the OSI network model as mentioned in Section 2.4.6. According to the network model, this implementation now includes five layers: application, session, transport, data link, and physical.

In TCP-oriented connections, connection information is stored in a Transmission Control Block (TCB). The TCB contains current connection information defined in the transport header. A TCB module manages the TCBs based on incoming and outgoing data as well as the connection timers related to each TCB. **Appendix A.2** shows the control logic for a particular TCB entry.

This research borrows from the type of TCB utilized by TCP-oriented connections. However, there are some differences between the two implementations, which are listed below:

- There are 11 connection states implemented in TCP. This research simplifies it to six states: CLOSED, SYN-RCVD, SYN-SENT, ESTABLISHED, TIME-WAIT, and FIN-WAIT.

- TCP goes through an extensive verification process when receiving packets with the RST flag set to high. This research treats RST as a hard reset; connections on both ends are immediately closed.

- This research does not utilize any sliding windows or header options that TCP uses, partially due to design constraints and the inability to grow the CAN FD frame payload larger than 64 bytes.

- Port numbers are used in a TCP connection, while this implementation will not use them. They are not required for this implementation as it is not necessary for two MCUs to have multiple ports open between each other.

- The sequence and acknowledgement numbers are used to represent the total number of bytes passed between the connection. When a receiver receives a packet, it increments both the sequence and acknowledgement numbers by the same amount, and then adds the length of the ACK packet to the sequence number before transmission. The original sender will still know by the acknowledgement number that its earlier message was received.

Some of the above differences are because of time restrictions or other necessities. They will be discussed more in detail in Section 6.3.

The type of encryption used for the system model's implementation is AES-256 in ECB mode. Any payload entered in less than 16 character bytes will be transformed into a 16-byte ciphertext. No payload greater than 15 bytes is supported, because if the payload is greater than this limit, an additional 16 bytes of ciphertext will be generated, which will make the overall payload too large for successful transmission. A 256-bit key can be generated using the *randGen.cpp* file defined in **Appendix B.1.16**.

Figure 4.1 shows the general flow of data in between two MCUs with respect to the software developed for testing. The contents of each file are shown in **Appendix B.1**. *TCBmodule.cpp/hpp* include the combined transport and session layer implementations, and the logic involving the TCB state is defined in *TCB.cpp/hpp*. Timer logic for the entire implementation is defined in *Timer.cpp/hpp*. *MCUout.cpp* initiates the TCB module and sends data to it for transmission, and *MCUin.cpp* returns data from the TCB module after it has been decrypted. *inputbuffer.cpp* and *outputbuffer.cpp* handle message transmission at the data-link layer according to methods already established by SocketCAN and VirtualCAN.

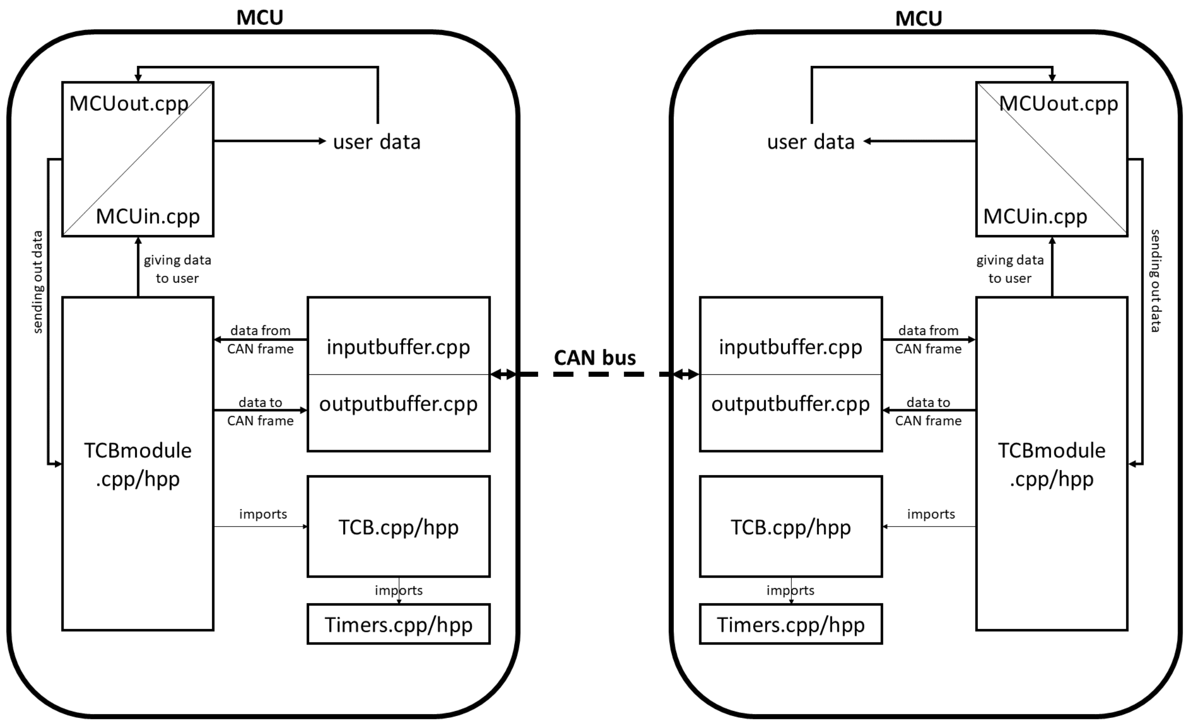


Figure 4.1: General flow of data between two MCUs on the CAN bus network

Figure 4.2 shows the full layout of a connection between two MCUs and the SSM. The SSM also shares the same transport and session layer logic as a regular MCU, but also includes separate applications for the stack and the clustering algorithm (Aware). Both of these applications retrieve new data at the same time. The clustering algorithm will notify the stack of any hacks that it detects, and the stack will provide the most recent valid data if needed. In cases of archived data retransmission, the stack will send the latest valid data entry to the transport and session layer modules.

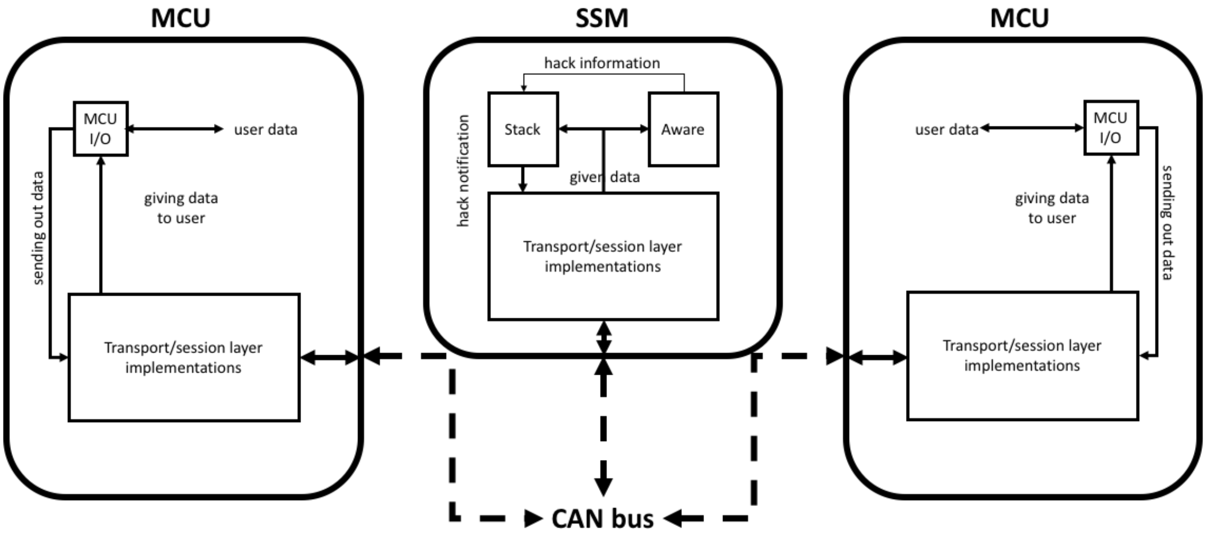


Figure 4.2: General flow of data between MCUs and the SSM on the CAN bus network

4.1.2 Developed testing tools and applications

The areas of concern defined in Section 2.3 will be tested against two types of MCUs: one implemented by the system model, and one implemented by directly interfacing with the CAN bus network. The message injection attack is the simplest form of attack; it only requires the message injection program and the victim to be on the same network. **Figure 4.3** shows the connection layout for the attack.

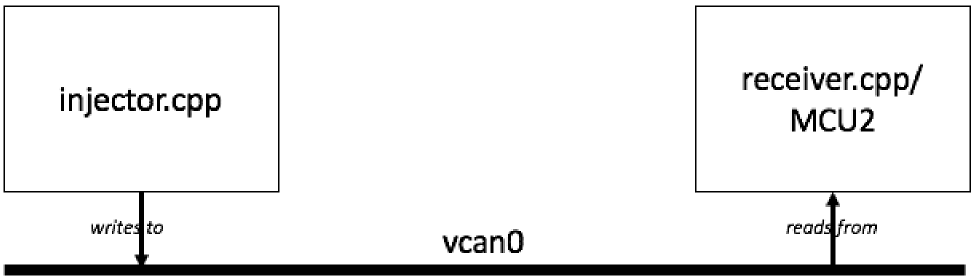


Figure 4.3: Appropriate layout of the CAN bus network for fuzzing testing

The fuzzing test requires an additional program which generates the text files required for the fuzzing software to perform the attack. Otherwise, the network layout here is the same as the message injection network. **Figure 4.4** shows the connection layout for the attack.

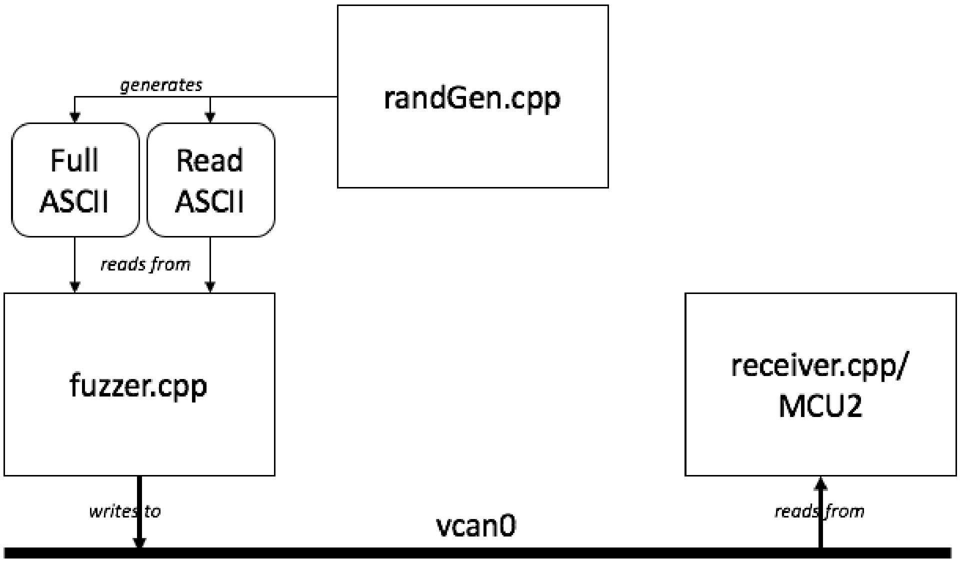


Figure 4.4: Appropriate layout of the CAN bus network for fuzzing testing

The MitM test, unlike the first two tests, requires the establishment of two CAN bus networks. In order for the test to work, the sender and receiver (victim) MCUs must be on opposite networks. The attacker will be the only MCU on the network with a connection to both networks. It will be responsible for listening for any messages on the sender's network so that they can be transferred to the receiver. **Figure 4.5** shows the connection layout for the attack.

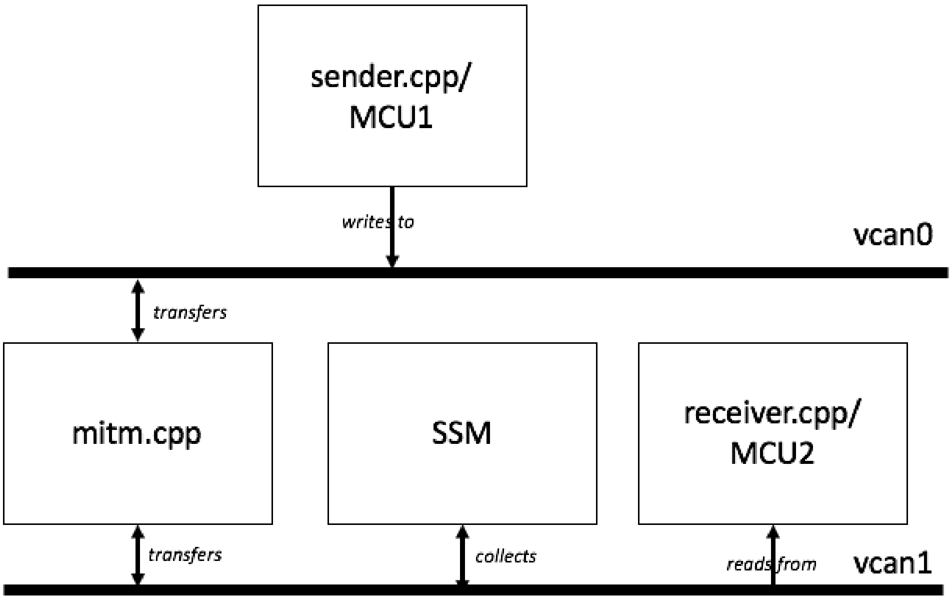


Figure 4.5: Appropriate layout of the CAN bus network for MitM testing

4.1.3 Utilized third-party software and libraries

All testing and development performed for this implementation used free, open-source software available for download. A virtual machine in VirtualBox 5.2.4 with a copy of Lubuntu 17.10 was created for code development and testing. In addition to the developed software described in the previous subsubsections, additional existing libraries were utilized in order to realize the proof of concept. **Table 4.1** lists all external libraries used that were not included as part of a standard Lubuntu installation.

Library	Purpose	DL Location	Version
Crypto++	AES encryption	*apt* package manager	5.6.5
can-utils	virtual CAN implementation	*apt* package manager	0.0+git20161220-1

Table 4.1: List of additional specialized libraries required for system model compilation and operation

4.2 Establishment of test environment

All testing was performed using virtual implementations of the CAN bus network. The source code can be compiled using the Makefile defined in **Appendix B.5.1**. The test network itself is relatively simple to setup. Using the bash script defined in **Appendix B.5.2**, a CAN bus with a speed of 1 Mbit/s and CAN FD support is created. For multiple networks, the bash script in **Appendix B.5.3** will setup more than one network with the same settings. This testing network will be the same for testing in all cases.

Raw data can be observed on the test network using the utilities provided by the *can-utils* package. The command `candump vcan0` prints the contents of a packet in hexadecimal representation every time a CAN frame is transferred on the network `vcan0`, regardless of sender or destination. The command `candump -t a vcan0` is recommended for readability related to this testing. **Figure 4.6** shows an example of the output of this command in a separate terminal.

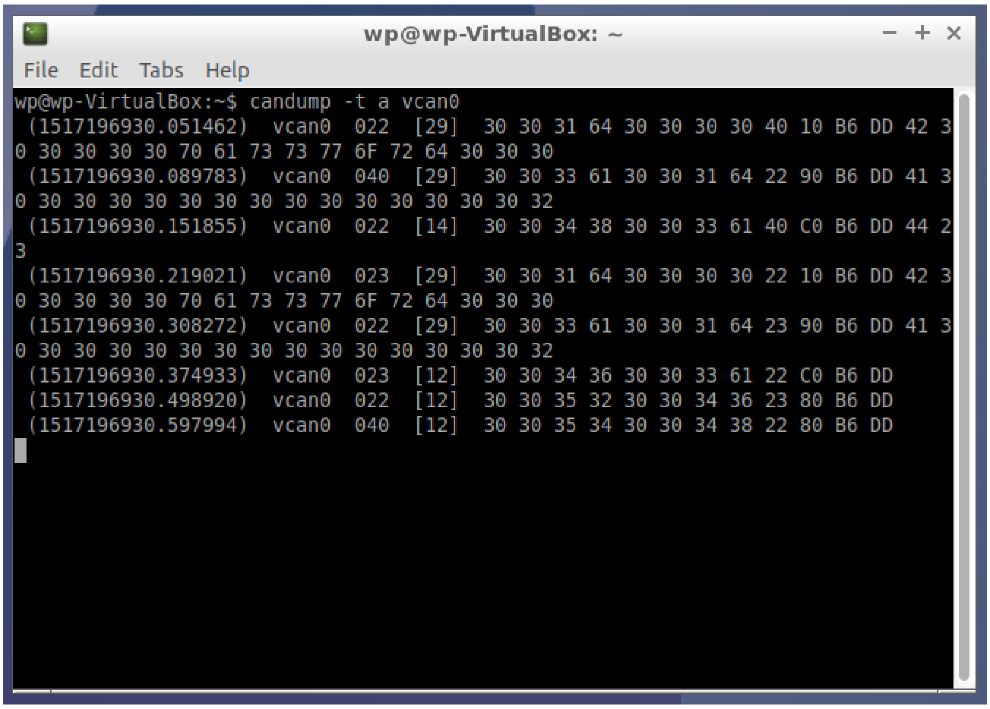


Figure 4.6: Sample output of the `candump` application with the recommended flags for readability

4.3 Test procedures

Due to the unavailability of security testing software that is open-source, white-box compatible, and usable on virtual networks, it is necessary to design custom testing procedures in order to evaluate the effectiveness of the system model. These procedures are based on existing knowledge of automotive network attacks and how they are generally structured. In order to contribute to the spectrum of automotive cybersecurity research, any and all source code used for testing are defined in their entirety in **Appendix B**.

All of these test procedures involve two parties: an attacker (hacker) and a victim. The victim is in charge of storing a particular value in a four-byte integer form. The attacker, meanwhile, must attempt to modify this value as many times as possible within the testing constraints mentioned in Section 4.1.2.

Each test will require five trials, in which each MCU instance is loaded, tested, and then exited before starting a new test. If any external files are involved for reference, they shall be used in both tests to ensure that the same inputs are given.

4.3.1 Message injection

The code files for the message injection attack is defined in **Appendix B.2**. The attacker logic follows the below pattern:

1. A CAN ID is provided at runtime. This will be the victim of the attack.
2. A timer is started to automatically run the program. This timer is set for one hour.
3. Until the timer expires, the following is performed:
   (a) The attacker is asked to enter a data string to be sent to the victim.
   (b) A new CAN FD frame is generated.
   (c) The data payload for the frame is set to the data string entered by the attacker.
   (d) The frame is sent to the victim.
4. The program terminates.

4.3.2 Fuzzing

The code files for the fuzzing attack are defined in **Appendix B.3**. The attacker logic follows the below pattern:

1. Using the separate program for the string generation, a text file containing a list of ASCII strings is created.
2. A CAN ID is provided at runtime. This will be the victim of the attack.
3. The text file generated earlier is loaded into memory.
4. For the number of entries in a file, the following is performed:
   (a) A new CAN FD frame is constructed.
   (b) The data payload for the frame is set to the file entry.
   (c) The frame is sent to the victim.
5. The program terminates.

Two different types of input will be given for this test. The first type is the entire range of American Standard Code for Information Interchange (ASCII) characters from 0 to 127, including non-printable characters. The second type is the range of ASCII characters from 32 to 126, which are all printable characters that can be entered by a standard US-EN layout keyboard. In all tests, both the unprotected and protected MCUs will be tested with the same two files.

4.3.3 Man-in-the-middle

The code files for the MitM attack are defined in **Appendix B.4**. The attacker logic follows the below pattern:

1. The attacker specifies a target MCU (MCU1) on the first network, and a target MCU (SSM) on the second network at runtime.
2. Until the attacker decides to terminate the program, the following is performed:
    (a) The attacker listens on the source MCU's network for any messages meant for the SSM on the other network.
    (b) If said message arrives, a random number between zero and four will be generated to determine whether or not to let the message pass through.
        i. If the number is **not** zero, the message will be passed along. Else, it will be dropped.
        ii. In the case of the former, another random number between zero and two will be generated to determine whether or not to change the payload.
            A. If the number **is** zero, the message will be modified. Else, it will be sent unmodified.
                - In the case of the former, another random number between zero and 125 will be generated.
                - For all instances that the character (int integer form) appears in the message, that character will be incremented by one.
                - Another random number will be generated in a boolean method to determine whether or not to change the calculated checksum for the message as well.
3. The program terminates.

For one trial, one hundred messages will be sent from either the *unprotSender.sh* script (for *receiver.cpp*'s case, see **Appendix B.4.3**) or MCU1 (for MCU2's case).

4.3.4 Victim logic

The victim logic is slightly different for all tests performed. With respect to the system model, testing will be directly performed on an existing instance of MCU2. The code for the victim without the solutions implemented in the system model is fully defined in **Appendix B.5.4**, and follows the below pattern in a constant loop:

1. The victim listens for any incoming messages that match its CAN ID.
2. If a received CAN frame's ID matches that of the victim's, the following is performed:
    (a) The victim extracts the entire data payload from the frame, minus the supposed offset for what would be considered the transport and session layers.
    (b) The victim changes its set value to whatever was extracted.

4.4 Testing criteria

The security testing mentioned in the previous section will produce two types of results: the number of times that the application layer value was changed, if applicable, and the error code returned by the implemented system model due to the detection of an error. The error codes will appear in the system output for the targeted MCU should it be given any input not properly crafted and sent by another MCU.

The error codes are defined below, in numeric order:

1. There was an error in processing the CAN frame input.
2. A valid checksum was not calculated.
3. The frame input was rejected due to a sequence numbering issue.
4. There was a problem with payload decryption.

The most important attribute is the number of times that the value was changed. The lower this number is for the modified network when compared to operation of the normal network, the more effective this system is in preventing that type of attack. Inversely, the more error codes there are detected by the network, the more effective it is considered to be in terms of attack prevention.

4.4.1 Additional benchmarking

In addition to the security testing, additional performance benchmarking will be performed. This benchmarking will focus on an appropriate size *a* entries for the awareness algorithm collection. Modifying the awareness algorithm test only requires changing the value of *N* in *Aware.hpp* as shown in **Appendix B.1.14**. The overall testing procedure is defined below:

1. New instances of MCU1, MCU2, and the SSM are created.
2. MCU1 starts a new connection with MCU2.
3. MCU1 starts sending the same data value repeatedly, until all three data sets in the awareness algorithm for that connection are full. The awareness algorithm should deliver a constant verdict.
4. MCU1 continues sending the same variable for half, or close to half, of the number of times required to fill another data set.
5. MCU1 then begins sending twice the value of the data value being sent, and does so repeatedly until the data set is full.
6. As soon as the verdict is delivered, the test is completed. The time for test completion and the third AFV are recorded for data analysis.

The following points are of note when recreating the test:

- Two time variables from the C++ STL `<chrono>` library are used to keep track of the time elapsed for the test.
- The first time variable is set and the test begins when the first data point is entered into the stack.
- The second time variable is set when a verdict is given. Because two verdicts will be used, the first time value returned may be discarded, and the second one kept for results.

- Data is entered in MCU1's user interface manually. New data is entered as soon as MCU1 receives an ACK.
- Retransmission capabilities have been disabled for collection of this data.

4.5 Technical caveats

It should be noted that because this is a proof of concept being broadcasted on a virtual representation of the CAN bus network, any and all potential technical issues regarding the setup and operation of a real CAN bus network are not observable. This issues include, but are not limited to, the following:

- transmission interference and noise due to voltage variance across the wires connecting the MCUs
- delays in message transmission based on the length of ”wire” between MCUs
- delays in message parsing due to the overhead required by the algorithms and libraries processing the message
- performance statistics of utilized hardware, especially regarding any microprocessor capabilities

Chapter 5

Evaluation results

This chapter details the collected results from the testing performed in Chapter 4. The collected data is compared between the unprotected (no system model protections) and protected (with system model protections) networks.

5.1 Collected results from testing

This section observes the collected results based on the tests outlined in the previous chapter.

5.1.1 Message injection results

Table 5.1 shows the number of times that the stored value in both the unprotected and protected victim MCUs were changed by the attacker across all five rounds as a result of message injection. The breakdown of error codes detected by the protected victim, accumulated across all five trials, are shown in **Table 5.2**.

TEST TYPE	TRIAL 1	TRIAL 2	TRIAL 3	TRIAL 4	TRIAL 5
Unprotected MCU	57	75	66	68	73
Protected MCU	0	0	0	0	0

Table 5.1: Number of changes in stored data value by hacker on unprotected and protected systems as a result of message injection

ERR CODE 1	ERR CODE 2	ERR CODE 3	ERR CODE 4
0	294	0	0

Table 5.2: Number of error code instances experienced as a result of message injection

5.1.2 Fuzzing results

Table 5.3 shows the number of times that the stored value in the unprotected victim MCU was changed by the attacker across all five rounds as a result of fuzzing. **Table 5.4** shows the number of times that the stored value in the protected victim MCU was changed in the same manner. The breakdown of error codes detected by the protected victim, accumulated across all five trials, are shown in **Table 5.5**.

TEST TYPE	TRIAL 1	TRIAL 2	TRIAL 3	TRIAL 4	TRIAL 5
Full ASCII range	1011	1133	893	866	834
Readable ASCII only	886	921	559	834	1072

Table 5.3: Number of changes in stored data value by hacker on unprotected system as a result of fuzzing

TEST TYPE	TRIAL 1	TRIAL 2	TRIAL 3	TRIAL 4	TRIAL 5
Full ASCII range	0	0	0	0	0
Readable ASCII only	0	0	0	0	0

Table 5.4: Number of changes in stored data value by hacker on protected system as a result of fuzzing

TEST TYPE	ERR CODE 1	ERR CODE 2	ERR CODE 3	ERR CODE 4
Full ASCII range	0	9	0	0
Readable ASCII only	0	8	0	0

Table 5.5: Number of error code instances experienced as a result of fuzzing

5.1.3 MitM results

Table 5.6 shows the number of times that the stored value in both the unprotected and protected victim MCUs were changed by the attacker across all five rounds as a result of the MitM attack. The breakdown of error codes detected by the protected victim, accumulated across all five trials, are shown in **Table 5.7**.

TEST TYPE	TRIAL 1	TRIAL 2	TRIAL 3	TRIAL 4	TRIAL 5
Unprotected MCU	82	81	76	81	70
Protected MCU	0	0	0	0	0

Table 5.6: Number of changes in stored data value by hacker on unprotected and protected systems as a result of the MitM attack

ERR CODE 1	ERR CODE 2	ERR CODE 3	ERR CODE 4
0	17	55	7

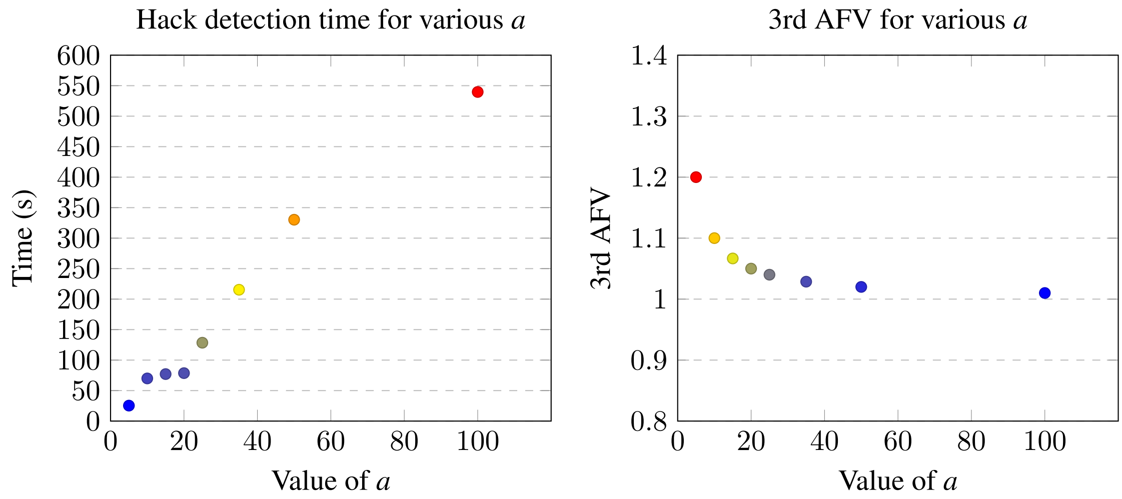
Table 5.7: Number of error code instances experienced as a result of the MitM attack

5.2 Additional performance benchmarking

The results from the test outlined in Section 4.4.1 are displayed below. Because there are only three data sets hard-coded into the proof of concept, the value of a is being tested only with three data sets to utilize. In other words, the value of b is permanently set to three. **Table 5.8** shows the results in tabular form. **Figure 5.1a** shows the time duration required to complete the test, and **Figure 5.1b** shows the third AFV collected at the completion of the test.

a	Time (s)	3rd AFV
5	25.353	1.2
10	69.934	1.1
15	77.049	1.06667
20	78.610	1.05
25	128.473	1.04
35	215.347	1.02857
50	330.170	1.02
100	539.677	1.01

Table 5.8: Hack detection times and 3rd AFVs for various a with $b = 3$



(a) Graphical representation of hack detection times for various a with $b = 3$

(b) Graphical representation of 3rd AFVs for various a with $b = 3$

Figure 5.1: Graphical representations of **Table 5.8**'s results

5.3 Result evaluation

From the message injection results outlined in Section 5.1.1, the protected system model successfully prevented the message injection attack from being implemented on the same system. As noted in **Table 5.2**, only the checksum calculation was responsible for preventing any message injection from reaching MCU2's application layer. Because the checksum is the first attribute to be checked when reading in a packet, and the packet contents were generated at random, the verification process made it impossible for the packet to be accepted.

From the fuzzing results outlined in Section 5.1.2, it is obvious that the proposed system model prevented any changes to the values stored in the application layer. According to **Table 5.5**, the only error that appeared in the entire testing process was the error code for an invalid checksum, similar to what was seen in the message injection testing.

From the MitM results outlined in Section 5.1.3, the MitM testing succeeded on both fronts. In addition to checksum-related errors, it was able to resend packets that were dropped by the MitM device. Even contents that were successfully changed made no significant impact on the entire system's operation, as any ciphertext tampered with was either discarded or rejected.

The benchmarking results in Section 5.2 shows an inversely proportional relationship between the time required to collect the data necessary to reach a verdict and the AFV observed by the system. This relationship highlights a clear advantage for smaller values of a. As a's value increases, the data collection time almost exponentially increases, while the AFV decreases towards an asymptotic limit of one. This observation is most likely because as more and more data points are introduced to each set, the overall impact of the change in data becomes harder to recognize.

The awareness algorithm in the system model's source code, defined in **Appendix B.1.15**, has been hard-coded to determine that if the derivative between any AFV is greater than 0.1, then a hack has been detected. Therefore, according to the results in **Table 5.8**, the AFVs calculated by the algorithm would not consider the sudden change in behavior to be a hack. This renders any and all values of a greater than ten useless, as it takes longer to reach a verdict that won't actually count. Therefore, when applying this system model, it is recommended to use values of a that are less than or equal to ten.

Chapter 6

Conclusion

6.1 Contributions of the thesis

The system model successfully solves the issues mentioned in Chapter 2. The message injection attack prevented messages from penetrating the application layer. The system model also withstood all fuzzing attempts, even as the testing tools were filling the network with data frames faster than the MCUs could read. The system model even successfully managed to handle both dropped and altered packets sent by the network bridging application in the MitM attack.

The system model developed as part of this research serves as a potential model for future CAN bus networks. The model itself is not yet ready for immediate implementation to a system, due to the time required to properly parse the frames compared to the requirement of the CAN bus network to operate in almost real-time conditions. However, the model serves as a working demo that can be improved with future and faster libraries and tools for the system model to utilize. Between the protocol design and implementation of data payload encryption, the system model also makes it much harder in theory for an attacker to disrupt the exchange of data between MCUs on the CAN bus network.

The testing procedures for the model were custom-designed and programmed in order to test the system model. Due to the unavailability of the tools required to test the system on a virtual network, these tools can be used to test virtual representations of the CAN bus network, which will make it easier for researchers to perform security testing on theoretical automotive network models without having to first pay for proper hardware, if such hardware in fact exists for testing purposes.

6.2 Significance of the contributions

The developed system model serves as a proposal for how CAN bus networks should be structured with regards to network security. It can be implemented on top of existing hardware and software solutions with little to no additional cost to a developer or manufacturer. It is based on the OSI network model, which is an important framework in network design. The system model also highlights the importance of balancing real-time network traffic with some kind of independent authority that can intervene in the event of a cyberattack.

With regards to the concept of automotive network security testing, the test procedures performed were well-defined in Chapter 4. All materials and recreation steps were provided to contribute to existing research tools regarding white-box testing. Although the tools are more specific to the designed system model, the release of this information will allow other automotive cybersecurity researchers to peer review the system and tools in order to strengthen the overall security of the system model. The knowledge gained from this review could then be shared with other existing tools in order to standardize the existing solutions.

6.2.1 Potential utilization of machine learning

Machine learning refers to the concept of a system learning to control itself without manual input (e.g. from a human). A system with machine learning capabilities collects data in order to make future predictions on incoming data and expands the overall capabilities of data classification far beyond the capabilities of the human mind.

This authority could be possible given developments in the field of machine learning with respect to cybersecurity [33] [34]. When applied to the developed system model's awareness algorithm, machine learning can learn about the general patterns of a MCU or a connection over time, and can then better predict and detect any external threats from wreaking havoc on the system. Although the realm of machine learning has yet to expand to automotive security, the proposed system model could be enhanced even further with the integration of machine learning techniques.

6.2.2 Autonomous driving and secured vehicles

A significant trend in recent years is the implementation of autonomous driving. There are already vehicles for sale on the general market that come with partially-autonomous driving features, such as *advanced emergency braking* (AED) and *adaptive cruise control* (ACC). These technologies help reduce the risk of human error, which is a cause of at least 94% of preventable accidents. [35]

According to a survey and simulations by Amoozadeh et al [36], self-driving vehicles still maintain significant security risks on multiple levels. This is an issue that is especially on the minds of consumers, especially as there is still significant resistance with respect to the supposed trustworthiness of partially or fully autonomous vehicles. [37]

This system model is meant to serve as part of a solution to the issue of securing a vehicle from external threats. Although it requires refinement, it can serve as a good security system to protect against external signals, much like a firewall protects a standard computer network. However, regardless of the finally selected system model to utilize, the lack of automotive network security is a critical design feature that must and should be implemented before ***any*** large-scale implementation of fully autonomous driving on ***any*** level.

6.3 Future research direction and recommendations

The system model and testing procedures can greatly benefit from additional development work. The three most important fields to expand upon are the expansion of the system model's network implementations, a potential revision of the system model's cryptography capabilities, and the importance of retesting the system model using hardware instead of virtual representations.

6.3.1 Refinement of network implementation

The system model implemented a lightweight transport protocol based on TCP. This was mostly due to the fact that many of the features implemented in TCP are not required, mainly due to the fixed size of the user data payload and lack of network-layer support on the CAN bus network. However, in the interest of completing a workable proof-of-concept, some of the flags, connection states, and exchange practices normally utilized by TCP were revised for simplicity. While full implementation of TCP still remains impractical if not impossible given current CAN bus architecture, it is still possible to refine the connection states and exchange practices to make the system model's connections more reliable. One possible refinement is the addition of a reconnection scheme in the event of network issues not as a result of a cyberattack. Although the CAN bus network is very efficient at delivering messages, the additional protection would prevent the entire network from entering an unknown state should such issues arise during operation.

6.3.2 Expansion of cryptography-based features

The utilization of AES-256 in ECB mode was due to the fact that this version of AES was the only reliable encryption format without the requirement of an IV. When used with a symmetric key, the IV adds an additional layer of security, as it provides much more random ciphertexts that can make it harder for an intruder to decrypt a message. However, the IV must be 16 bytes, and using the same IV more than once is considered a bad form of cryptographic practice, because an attacker can attempt to derive patterns from multiple ciphertexts should those strings have similar byte sequences. Because the IV is required to decrypt the message as well, it needs to be transferred with the ciphertext.

This means that an encrypted four-byte float will require 32 bytes of cryptographic data to be decrypted (16 bytes for the float, and another 16 bytes for the IV). In terms of the session layer implemented by the system model, a transfer of data from the SSM to MCU2 would require 64 bytes, which is the entire user data length of the CAN FD frame. Should there be another extension of available CAN bus frames to 128 bytes of user data payload or more in the future, it would be possible to select another mode of AES, such as CBC or GCM mode. The system model could then be adjusted to allow the transfer of all 64 bytes of cryptographic data while keeping the original structure of the system model's protocol.

6.3.3 Hardware-based versus virtual-based testing

The compatible hardware for CAN FD testing was under development at around the same time that the system design and testing were performed. As of the date of publication of this thesis, the hardware is available for purchase. The system model outlined in this section can finally be tested on an actual network, and the results that this network produces should be compared to that of the virtual network's to determine accuracy compared to real-world expectations. In addition, albeit proprietary in nature, there may be some MCUs for sale on the general market with CAN FD support that can also be used to test this system model.

References

[1] I. S. O. (ISO), *ISO 11898 : Road vehicles : Interchange of digital information : Controller Area Network (CAN) for High-speed Communication*, 1st ed. International Standards Organization (ISO), 1993.

[2] "History of can technology." [Online]. Available: https://www.can-cia.org/can-knowledge/can/can-history/

[3] "Does my car have obd-ii?" 2011. [Online]. Available: http://www.obdii.com/connector.html

[4] C. Hoffman, "Automotive grade linux: An open-source platform for the entire car industry," May 2016. [Online]. Available: https://www.pcworld.com/article/3066913/linux/automotive-grade-linux-an-open-source-platform-for-the-entire-car-industry.html

[5] C. Miller and C. Valasek, *Black Hat USA 2015*. self-published, Aug 2015, ch. nc. [Online]. Available: https://securityzap.com/files/RemoteCarHacking.pdf

[6] A. Press, "Fiat chrysler recalls 1.4m vehicles in wake of jeep hacking revelation," *The Guardian*, Jul 2015. [Online]. Available: https://www.theguardian.com/business/2015/jul/24/fiat-chrysler-recall-jeep-hacking

[7] A. Greenberg, "A deep flaw in your car lets hackers shut down safety features," Aug 2017. [Online]. Available: https://www.wired.com/story/car-hack-shut-down-safety-features/

[8] L. Dariz, M. Ruggeri, G. Costantino, and F. Martinelli, "A survey over low-level security issues in heavy duty vehicles," in *Automotive Cyber Security Conference. ESCAR*, 2016.

[9] M. Ring, J. Drrwang, F. Sommer, and R. Kriesten, "Survey on vehicular attacks - building a vulnerability database," in *2015 IEEE International Conference on Vehicular Electronics and Safety (ICVES)*, Nov 2015, pp. 208–212.

[10] J. A. Bruton, "Securing can bus communication: An analysis of cryptographic approaches," Master's thesis, Open Sensor Network Authentication (OSNA), Aug 2004. [Online]. Available: http://www.osna-solutions.com/wp-content/uploads/Securing-CAN-Bus-Communication-Thesis-Extract.pdf

[11] A. H. Yousef, "Methods of securing in-vehicle networks," Master's thesis, Cairo University, Feb 2013. [Online]. Available: http://eece.cu.edu.eg/~hfahmy/thesis/2013_02_sec.pdf

[12] M. R. Moore, R. A. Bridges, F. L. Combs, M. S. Starr, and S. J. Prowell, "Modeling inter-signal arrival times for accurate detection of can bus signal injection attacks: A data-driven approach to in-vehicle intrusion detection," in *Proceedings of the 12th Annual Conference on Cyber and Information Security Research*, ser. CISRC '17. New York, NY, USA: ACM, 2017, pp. 11:1–11:4. [Online]. Available: http://doi.acm.org/10.1145/3064814.3064816

[13] T. Dagan and A. Wool, “Parrot, a software-only anti-spoofing defense system for the can bus,” in *14th Embedded Security in Cars*. Tel Aviv University, 2016.

[14] S. Woo, H. J. Jo, I. S. Kim, and D. H. Lee, “A practical security architecture for in-vehicle can-fd,” *IEEE Transactions on Intelligent Transportation Systems*, vol. 17, no. 8, pp. 2248–2261, Aug 2016.

[15] M. Cazorla, K. Marquet, and M. Minier, “Survey and benchmark of lightweight block ciphers for wireless sensor networks,” in *2013 International Conference on Security and Cryptography (SECRYPT)*, July 2013, pp. 1–6.

[16] D. Dinu, Y. L. Corre, D. Khovratovich, L. Perrin, J. Großschädl, and A. Biryukov, “Triathlon of lightweight block ciphers for the internet of things,” IACR ePrint archive, Tech. Rep., 2015.

[17] A. A. Pammu, K. S. Chong, W. G. Ho, and B. H. Gwee, “Interceptive side channel attack on aes-128 wireless communications for iot applications,” in *2016 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, Oct 2016, pp. 650–653.

[18] O. Lo, W. J. Buchanan, and D. Carson, “Power analysis attacks on the aes-128 s-box using differential power analysis (dpa) and correlation power analysis (cpa),” *Journal of Cyber Security Technology*, vol. 1, no. 2, pp. 88–107, 2017. [Online]. Available: https://doi.org/10.1080/23742917.2016.1231523

[19] B. Liu, L. Shi, Z. Cai, and M. Li, “Software vulnerability discovery techniques: A survey,” in *2012 Fourth International Conference on Multimedia Information Networking and Security*, Nov 2012, pp. 152–156.

[20] A. Takanen, “Fuzzing: the past, the present and the future,” in *Actes du 7ème symposium sur la sécurité des technologies de l’information et des communications (SSTIC)*, 2009, pp. 202–212.

[21] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage, “Experimental security analysis of a modern automobile,” in *2010 IEEE Symposium on Security and Privacy*, May 2010, pp. 447–462.

[22] C. Miller and C. Valasek, “Can message injection og dynamite edition,” Jun 2016. [Online]. Available: https://dl.packetstormsecurity.net/papers/attack/can-message-injection.pdf

[23] F. Callegati, W. Cerroni, and M. Ramilli, “Man-in-the-middle attack to the https protocol,” *IEEE Security Privacy*, vol. 7, no. 1, pp. 78–81, Jan 2009.

[24] G. N. Nayak and S. G. Samaddar, “Different flavours of man-in-the-middle attack, consequences and feasible solutions,” in *2010 3rd International Conference on Computer Science and Information Technology*, vol. 5, July 2010, pp. 491–495.

[25] T. Lindenkreuz, “Vector congress 2012,” in *6th Vector Congress 2012*. Vector, Nov 2012. [Online]. Available: https://vector.com/portal/medien/cmc/events/commercial_events/VectorCongress_2012/VeCo12_8_NewBusSystems_3_Lindenkreuz_Lecture.pdf

[26] “Mercedes w140: First car with can,” Mar 2016. [Online]. Available: https://can-newsletter.org/engineering/applications/160322_25th-anniversary-mercedes-w140-first-car-with-can/

[27] “Can bus test suite data sheet,” Jan 2018. [Online]. Available: https://www.synopsys.com/software-integrity/security-testing/fuzz-testing/defensics/protocols/can-bus.html

[28] J. Li, "Cansee - an automobile intrusion detection system." [Online]. Available: https://conference.hitb.org/hitbsecconf2016ams/materials/D2T1%20-%20Jun%20Li%20-%20CANSsee%20-%20An%20Automobile%20Intrusion%20Detection%20System.pdf

[29] E. Evenchick, "An introduction to the canard toolkit," in *Blackhat Conference*, 2015.

[30] S. Bayer, T. Enderle, D.-K. Oka, and M. Wolf, "Security crash test - practical security evaluations of automotive onboard it components," in *Automotive - Safety & Security 2014*, H. Klenk, H. B. Keller, E. Pldereder, and P. Dencker, Eds. Bonn: Gesellschaft fr Informatik e.V., 2015, pp. 125–139.

[31] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. L. Traon, "Comparing white-box and black-box test prioritization," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, May 2016, pp. 523–534.

[32] A. Diarra, "Osi layers in automotive networks," Mar 2013. [Online]. Available: http://www.ieee802.org/1/files/public/docs2013/new-tsn-diarra-osi-layers-in-automotive-networks-0313-v01.pdf

[33] E. Kanal, "Machine learning in cybersecurity," Jun 2017. [Online]. Available: https://insights.sei.cmu.edu/sei_blog/2017/06/machine-learning-in-cybersecurity.html

[34] R. Marty, "Ai and machine learning in cyber security towards data science," Jan 2018. [Online]. Available: https://towardsdatascience.com/ai-and-machine-learning-in-cyber-security-d6fbee480af0

[35] N. H. T. S. Administration and S. Singh, *Critical reasons for crashes investigated in the National Motor Vehicle Crash Causation Survey*, ser. Traffic Safety Facts - Crash Stats. The National Academies of Sciences, Engineering, and Medicine, Feb 2015. [Online]. Available: https://crashstats.nhtsa.dot.gov/Api/Public/ViewPublication/812115

[36] M. Amoozadeh, A. Raghuramu, C. n. Chuah, D. Ghosal, H. M. Zhang, J. Rowe, and K. Levitt, "Security vulnerabilities of connected vehicle streams and their impact on cooperative driving," *IEEE Communications Magazine*, vol. 53, no. 6, pp. 126–132, June 2015.

[37] B. Schoettle and M. Sivak, "A survey of public opinion about autonomous and self-driving vehicles in the u.s., the u.k., and australia," Jul 2014. [Online]. Available: https://deepblue.lib.umich.edu/bitstream/handle/2027.42/108384/103024.pdf

Appendices

A Pseudocode

This section shows all pseudocode use to define algorithmic concepts defined in the system model.

A.1 Pseudocode for clustering algorithm data collection

The below pseudocode shows the process of collecting AFVs and organizing them into sets for analysis.

```
data1 := arr[1..n] := {-1}
data2 := arr[1..n] := {-1}
data3 := arr[1..n] := {-1}

procedure GETSETS(newFactor)
    done := False

    for i from 1..n do
        if data1[i] == -1 then
            data1[i] := newFactor
            done := True
            break
        end if
    end for

    if done == False then
        for i from 1..n do
            if data2[i] == -1 then
                data2[i] := newFactor
                done := True
                break
            end if
        end for
    end if

    if done == False then
        for i from 1..n do
            if data3[i] == -1 then
                data3[i] := newFactor
                done := True
                if i == 49 then calcAvgs()
                    data1 := data2
                    data2 := data3
                    data3 := arr[1..n] := {-1}
                end ifbreak
            end if
        end for
    end if
end procedure

procedure CALCAVGS
    d1avg := 0.00
    d2avg := 0.00
    d3avg := 0.00
```

```
    for i from 1 to n-1 do
        d1avg += data1[i]
        d2avg += data2[i]
        d3avg += data3[i]
    end for

    d1avg /= 49
    d2avg /= 49
    d3avg /= 49
    doVerdict(d1avg, d2avg, d3avg)
end procedure

for  do
    newFactor = getFactor()
    getSets(newFactor)
end for
```

A.2 Pseudocode for transport layer TCB

The below pseudocode shows the control logic for a TCB object for each connection.

```
CLOSED := 0
SYN_SENT := 1
SYN_RCVD := 2
ESTABLISHED := 3
FIN_WAIT := 4
TIME_WAIT := 5

for  do
    thisConnState := getConnState()
    if thisConnState == CLOSED then
        if SYNflag == true then
            setConnState(SYN_RCVD)
        else if startConn == true then
            setConnState(SYN_SENT)
        else
            RSTflag = true
        end if
    else if thisConnState == SYN_SENT then
        if timeout then
            setConnState := CLOSED
        else if SYNflag == true and ACKflag == false then
            updateFlags(SYN-ACK)
            setConnState(SYN_RCVD)
        else if SYNflag == true and ACKflag == true then
            updateFlags(ACK)
            setConnState(ESTABLISHED)
        end if
    else if thisConnState == SYN_RCVD then
        if ACKflag == true then
            setConnState(ESTABLISHED)
        else if timeout then
            updateFlags(RST)
            setConnState(TIME_WAIT)
        else if RSTflag == true then
            setConnState(CLOSED)
        end if
    else if thisConnState == ESTABLISHED then
        if FINflag == true then
            updateFlags(FIN-ACK)
            setConnState(FIN_WAIT)
        else if RSTflag == true then
            setConnState(TIME_WAIT)
        end if
    else if thisConnState == FIN_WAIT then
        if ACKflag == true then
            setConnState(TIME_WAIT)
        end if
    else if thisConnState == TIME_WAIT then
        if timeout then
            setConnState(CLOSED)
        end if
    end if
end for
```

B Source code

This section shows all code files used for developing the system model and security testing for posterity. This is the latest version of the files as of the date of final submission of this thesis.

B.1 System model implementation

B.1.1 Timer.hpp

```
#include <ctime>
#include <chrono>
#include <iostream>

class Timer{
typedef std::chrono::high_resolution_clock hrc;
typedef std::chrono::nanoseconds nanosecs;
typedef std::chrono::seconds secs;

private:
  hrc::time_point start;
  int type;

public:
  Timer(int t);
  ~Timer(){}
  void reset(){ start = hrc::now(); }
  bool isExpired();
};
```

B.1.2 Timer.cpp

```
#include "Timer.hpp"

using namespace std;

Timer::Timer(int t){
  type = t;
  if (type == 0){}              //timer for retrans
  else if (type == 1){}         //timer for keepalive
  else if (type == 2){}         //timer for time_wait
  else if (type == 3){}         //error counter (SSM)
  else if (type == 4){}         //hack timer
}

bool Timer::isExpired(){
  if (type == 0 && chrono::duration_cast<secs>(hrc::now() - start).count() > 1) return true;
  else if (type == 1 && chrono::duration_cast<secs>(hrc::now() - start).count() > 10) return true;
  else if (type == 2 && chrono::duration_cast<secs>(hrc::now() - start).count() > 5) return true;
  else if (type == 3 && chrono::duration_cast<secs>(hrc::now() - start).count() > 5) return true;
  else if (type == 4 && chrono::duration_cast<secs>(hrc::now() - start).count() > 10) return true;
  else return false;
}
```

B.1.3 TCB.hpp

```
#ifndef _TCBHPP
#define _TCBHPP

#include <bitset>
#include <csignal>
#include <cstdio>
#include <cstring>
#include <cstdlib>
#include <fstream>
```

```cpp
#include <iomanip>
#include <iostream>
#include <iterator>
#include <memory>
#include <mutex>
#include <string>
#include <sstream>
#include <stdexcept>
#include <thread>
#include <vector>

#include <fcntl.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <unistd.h>

#include <cryptopp/osrng.h>
#include <cryptopp/cryptlib.h>
#include <cryptopp/hex.h>
#include <cryptopp/filters.h>
#include <cryptopp/aes.h>
#include <cryptopp/modes.h>

#include "Timer.hpp"

#define CLOSED 0
#define SYN_SENT 1
#define SYN_RCVD 2
#define ESTABLISHED 3
#define FIN_WAIT 4
#define TIME_WAIT 5

#define SSMID "34"

class TCB{
private:
  bool MCU1 = false, MCU2 = false;
  bool didSYNACK = false, isEstab = false, establishing = false, isVerif = false, killing = false, killed = false, lastPackGarb = false;
  float lastValue = -1;
  unsigned char canID; //for our destination!
  std::string ALpipepath, latestFVenc, lastValueEnc;
  byte key[CryptoPP::AES::MAX_KEYLENGTH];

  const bool SYNonly[5] = {false, false, false, true, false};
  const bool SYNACK[5] = {true, false, false, true, false};
  const bool ACKonly[5] = {true, false, false, false, false};
  const bool RSTonly[5] = {false, false, true, false, false};
  const bool FINACK[5] = {true, false, false, false, true};
  const bool PSHonly[5] = {false, true, false, false, false};
  const bool PSHACK[5] = {true, true, false, false, false};
  const bool FINonly[5] = {false, false, false, false, true};

public:
  unsigned int seq = 0, ackn = 0;
  int thisConnState;
  float latestFV = 0;
  std::unique_ptr<Timer> retrans, keepalive, time_wait;
  bool makeConn = false, isAuth = false, needAck = false, sending = false, hardReset = false, killConn = false;
  std::string currentInput, currentOutput;
  TCB(int s, int a, char c, int MCU) :
    seq(s),
    ackn(a),
    canID(c),
    retrans(std::make_unique<Timer>(0)),
    keepalive(std::make_unique<Timer>(1)),
    time_wait(std::make_unique<Timer>(2)),
    thisConnState(0),
    ALpipepath("/tmp/TCBtoAL")
    { relation(MCU); }
  ~TCB(){std::cerr << "NOT: " << canID << " dying!\n";}

```

```
    bool sameID(char c){ return canID == c; }
    bool isMCU1() const { return MCU1; }
    bool isMCU2() const { return MCU2; }
    std::string diagInfo();
    char getCanID() const { return canID; }
    void relation(int MCU);
    int flagStats(bool f[5]);
    bool prepareSwitch(int s, int a, bool f[5], std::string input);
    void updateStats(unsigned int ds, unsigned int da);
    int checkStats(unsigned int inSeq, unsigned int inAckn, char c);
    void makeOutput(char src, std::string outfilepipepath, std::string al);
    std::string checksum(std::string in);
    char MCU1toSSM_SL(std::string sl, std::string pay);
    char SSMtoMCU2_SL(std::string sl);
    char MCU2dest();
    void SLtoAL(char CANID);

    void TCBswitch(bool f[5]);
    void caseClosed(bool f[5]);
    void caseSynSent(bool f[5]);
    void caseSynRcvd(bool f[5]);
    void caseEstablished(bool f[5]);
    void caseFinWait(bool f[5]);
    void caseTimeWait(bool f[5]);

};

#endif
```

B.1.4 TCB.cpp

```
#include "TCB.hpp"

using namespace std;
using CryptoPP::AES;
using CryptoPP::AutoSeededRandomPool;
using CryptoPP::ECB_Mode;
using CryptoPP::Exception;
using CryptoPP::HexEncoder;
using CryptoPP::HexDecoder;
using CryptoPP::StringSink;
using CryptoPP::StringSource;
using CryptoPP::StreamTransformationFilter;

//Name: diagInfo
//Description: Pipe out packet data.
//Output: Concatnated strings for transport layer
//
    ------------------------------------------------------------------------------------------------

string TCB::diagInfo(){
    stringstream ss, ss2;
    char c = 0;

    ss <<  setw(4) << setfill('0') << hex << seq << setw(4) << setfill('0') << ackn;

    if (didSYNACK && !establishing && !isEstab) c = 0b00010000;
    else if (!didSYNACK && establishing && !isEstab) c = 0b10010000;
    else if (didSYNACK && isEstab && !sending) c = 0b11000000;
    else if (isEstab && isVerif && !sending && !killConn && !killed) c = 0b10000000;
    else if (isEstab && isVerif && sending && !killConn) c = 0b01000000;
    else if (hardReset) c = 0b00100000;
    else if (killConn && !killed) c = 0b00001000;
    else if (!killConn && killed)c = 0b10001000;
    else if (killConn && killed) c = 0b10000000;

    ss2 << '0' << c << "00";
    return ss.str() + ss2.str();
}
//
    ------------------------------------------------------------------------------------------------

//Name: relation
```

```
//Description: Are we MCU1, MCU2, or SSM?
//Output: N/A
//--------------------------------------------------------------------------------

void TCB::relation(int MCU){
  ifstream ifs("my.key", ios::in);
  char readIn[32];
  string tmp;

  if (MCU == 0) MCU1 = true;
  else if (MCU == 1) MCU2 = true;

  cerr << "NOT: Initialized TCBentry for ID " << canID;
  if (MCU1 && !MCU2) cerr << " oriented as MCU1\n";
  else if (!MCU1 && MCU2) cerr << " oriented as MCU2\n";
  else if (!MCU1 && !MCU2) cerr << " oriented as SSM\n";

  if (ifs.is_open()){
    ifs.read(readIn, AES::MAX_KEYLENGTH);
    for (int i = 0; i < AES::MAX_KEYLENGTH; ++i) key[i] = (byte) readIn[i];
    ifs.close();
  }
  else{
    cerr << "ERR: can't open AES key file in stack application\n";
    exit(1);
  }
}
//--------------------------------------------------------------------------------

//Name: updateStats
//Description: Increment the seq and ack numbers for the TCB entry.
//Output: N/A
//--------------------------------------------------------------------------------

void TCB::updateStats(unsigned int ds, unsigned int da){
  seq += ds;
  ackn += da;
}
//--------------------------------------------------------------------------------

//Name: checkStats
//Description: Change TCB entry's connection flags before running
//             the switch.
//Output: 0 if all packet variables match TCB entry, or error code if one does not
//--------------------------------------------------------------------------------

int TCB::checkStats(unsigned int inSeq, unsigned int inAckn, char c){
  if (c != canID) return 1;
  else if (seq > 0 && ((inAckn+currentInput.size()) != inSeq)) return 2;
  else if (ackn != 0 && inAckn != seq) return 3;
  return 0;
}
//--------------------------------------------------------------------------------

//Name: flagStats
//Description: Check the current status of the flags in the TCB entry.
//Output: flag status code
//--------------------------------------------------------------------------------

int TCB::flagStats(bool f[5]){
  //cerr << "FLAGSTATS: " << f[0] << f[1] << f[2] << f[3] << f[4] << endl;
  if (equal(f, f+4, SYNonly)) return 0;
  if (equal(f, f+4, SYNACK)) return 1;
  if (equal(f, f+4, ACKonly)) return 2;
  if (equal(f, f+4, RSTonly)) return 3;
  if (equal(f, f+4, FINACK)) return 4;
```

```
  if (equal(f, f+4, PSHonly)) return 5;
  if (equal(f, f+4, PSHACK)) return 6;
  if (equal(f, f+4, FINonly)) return 7;
  if (f[2]) return 8;
  if (f[1]) return 9;

  return -1;    //unknown
}
//----------------------------------------------------------------------------
//Name: prepareSwitch
//Description: Do input checking here for switch.
//Output: N/A
//----------------------------------------------------------------------------
bool TCB::prepareSwitch(int s, int a, bool f[5], string input){
  int tmp;

  if (input.size() > 0) currentInput = input;
  tmp = checkStats(s, a, canID);
  if (tmp > 0){
    cerr << "ERR: incoming packet for " << canID << " rejected ";
    if (tmp == 1) cerr << "because source CAN_ID is wrong\n";
    else if (tmp > 1) cerr << "due to ERR_CODE_3\n";
    return false;
  }
  else {
    ackn = s;
    seq = s;
    TCBswitch(f);
    return true;
  }
}
//----------------------------------------------------------------------------
//Name: switch
//Description: Do input checking here for switch.
//Output: N/A
//----------------------------------------------------------------------------
void TCB::TCBswitch(bool f[5]){
  switch(thisConnState){
    case CLOSED: { caseClosed(f); break; }
    case SYN_SENT: { caseSynSent(f); break; }
    case SYN_RCVD: { caseSynRcvd(f); break; }
    case ESTABLISHED: { caseEstablished(f); break; }
    case FIN_WAIT: { caseFinWait(f); break; }
    case TIME_WAIT: { caseTimeWait(f); break; }
    default:{ break; }
  }
  cerr << "TCS_OF_" << (int)canID << ": " << thisConnState << endl;
}
//----------------------------------------------------------------------------
//Name: caseClosed
//Description: thisConnState = CLOSED
//Output: N/A
//----------------------------------------------------------------------------
void TCB::caseClosed(bool f[5]){
  if (flagStats(f) == 0 && !makeConn){
    thisConnState = SYN_RCVD;
    establishing = true;
    isVerif = true;
  }
  else if (flagStats(f) == 0 && makeConn){
    thisConnState = SYN_SENT;
```

```
      makeConn = false;
      didSYNACK = true;
    }
    else{
      hardReset = true;
    }
}
//--------------------------------------------------------------------------------
//Name: caseSynSent
//Description: thisConnState = SYN_SENT
//Output: N/A
//--------------------------------------------------------------------------------
void TCB::caseSynSent(bool f[5]){
  if (retrans->isExpired()){
    thisConnState = CLOSED;
  }
  else if (flagStats(f) == 0){
    thisConnState = SYN_RCVD;
  }
  else if (flagStats(f) == 1){
    thisConnState = ESTABLISHED;
    isEstab = true;
    isVerif = true;
  }
}
//--------------------------------------------------------------------------------
//Name: caseSynRcvd
//Description: thisConnState = SYN_RCVD
//Output: N/A
//--------------------------------------------------------------------------------
void TCB::caseSynRcvd(bool f[5]){
  if (flagStats(f) == 6 || flagStats(f) == 2){
    thisConnState = ESTABLISHED;

    if (MCU2){
      establishing = false;
      isEstab = true;
      isVerif = true;
    }
    else if (!MCU1 && !MCU2 && establishing){
      isEstab = true;
      isVerif = true;
    }
  }
  else if (retrans->isExpired()){
    cerr << "expired\n";
  }
  else if (flagStats(f) == 3){
    thisConnState = CLOSED;
    isEstab = false;
    isVerif = false;
  }
}
//--------------------------------------------------------------------------------
//Name: caseEstablished
//Description: thisConnState = ESTABLISHED
//Output: N/A
//--------------------------------------------------------------------------------
void TCB::caseEstablished(bool f[5]){
  //for FIN (dest)
  if (flagStats(f) == 7){
```

```
      thisConnState = FIN_WAIT;
      if (!killConn) killed = true;
    }
    //for SSM's RST
    else if (hardReset){
      killed = true;
      thisConnState = TIME_WAIT;
      time_wait->reset();
    }
    //for RST
    else if (flagStats(f) == 3){
      thisConnState = TIME_WAIT;
      time_wait->reset();
    }
    //for FIN (sender)
    else if (killConn) thisConnState = FIN_WAIT;
}
//------------------------------------------------------------------------------------------
//Name: caseFinWait
//Description: thisConnState = FIN_WAIT
//Output: N/A
//------------------------------------------------------------------------------------------
void TCB::caseFinWait(bool f[5]){
    if (flagStats(f) == 2){
      thisConnState = TIME_WAIT;
      killed = true;
      time_wait->reset();
    }
}
//------------------------------------------------------------------------------------------
//Name: caseTimeWait
//Description: thisConnState = TIME_WAIT
//Output: N/A
//------------------------------------------------------------------------------------------
void TCB::caseTimeWait(bool f[5]){
    if (time_wait->isExpired() || flagStats(f) == 2) thisConnState = CLOSED;
}
//------------------------------------------------------------------------------------------
//Name: makeOutput
//Description: Make our output to give to TCBmodule
//Output: N/A
//------------------------------------------------------------------------------------------
void TCB::makeOutput(char src, string outfilepipepath, string al){
    ofstream ofs(outfilepipepath.c_str(), ios::out | ios::binary);
    stringstream ss;
    string tmp, SLenc, check1, check2;

    currentOutput.clear();
    ss << diagInfo();

    //for MCU1 starting connection to SSM ***OR*** SSM starting connection to MCU2
    if ((MCU1 && !MCU2 && didSYNACK && !isEstab) ||
      (!MCU1 && !MCU2 && didSYNACK && !isEstab)){
        try{
          ECB_Mode< AES >::Encryption e;
          e.SetKey(key, sizeof(key));
          StringSource("mastersthesis", true, new StreamTransformationFilter(e, new StringSink(SLenc)));
          ss << 'B' << SLenc;
        }
        catch(const CryptoPP::Exception& e){
```

```
            cerr << "ERR:␣MAKEOUTPUT␣ENCRYPTION␣GONE␣WRONG\n" << e.what() << "\nEXITING\n"
                ;
            exit(1);
         }
      }
   //for SSM acking MCU1 ***OR *** MCU2 acking SSM (the latter is init only)
   else if ((!MCU1 && !MCU2 && establishing && !isEstab) ||
      (!MCU1 && MCU2 && establishing)){
         try{
            ECB_Mode< AES >::Encryption e;
            e.SetKey(key, sizeof(key));
            StringSource("2", true, new StreamTransformationFilter(e, new StringSink(SLenc
                ))); //(char) 50
            ss << 'A' << SLenc;
         }
         catch(const CryptoPP::Exception& e){
            cerr << "ERR:␣MAKEOUTPUT␣ENCRYPTION␣GONE␣WRONG\n" << e.what() << "\nEXITING\n"
                ;
            exit(1);
         }
      }
   //for MCU1 sending data to SSM
   else if (MCU1 && !MCU2 && didSYNACK && isEstab && isVerif && !killed){
         try{
            ECB_Mode< AES >::Encryption e;
            e.SetKey(key, sizeof(key));
            StringSource(string(1,canID), true, new StreamTransformationFilter(e, new
                StringSink(SLenc)));
            ss << 'D' << SLenc;
         }
         catch(const CryptoPP::Exception& e){
            cerr << "ERR:␣MAKEOUTPUT␣ENCRYPTION␣GONE␣WRONG\n" << e.what() << "\nEXITING\n"
                ;
            exit(1);
         }
      }
   //for MCU2 acking w/ latestFVenc
   else if (!MCU1 && MCU2 && lastValue != -1 && isEstab && isVerif && !killed){
      ss << 'C' << latestFVenc;
   }
   //for SSM sending data to MCU2
   else if (!MCU1 && !MCU2 && isEstab && isVerif && sending){
      if (lastValueEnc.empty()){
         try{
            ECB_Mode< AES >::Encryption e;
            e.SetKey(key, sizeof(key));
            StringSource("0", true, new StreamTransformationFilter(e, new StringSink(SLenc
                )));
            ss << 'E' << SLenc;
            lastValueEnc.assign(al);
         }
         catch(const CryptoPP::Exception& e){
            cerr << "ERR:␣MAKEOUTPUT␣ENCRYPTION␣GONE␣WRONG\n" << e.what() << "\nEXITING\n"
                ;
            exit(1);
         }
      }
      else{
         ss << "E" << lastValueEnc;
         lastValueEnc.assign(al);
      }
   }

   //finally, application layer data
   ss << al;

   currentOutput = ss.str();
   currentOutput[8] = src; //source CAN ID

   if (!hardReset) updateStats(currentOutput.size(), 0);
   else updateStats(currentOutput.size(), seq-ackn);

   ss.str(string(""));
```

```
  ss.clear();
  ss << diagInfo();
  tmp = ss.str().substr(0,4);
  currentOutput.replace(0, 4, tmp);
  tmp.clear();
  if (hardReset){
    tmp = ss.str().substr(4,4);
    currentOutput.replace(4, 4, tmp);
    tmp.clear();
  }

  tmp = checksum(currentOutput);
  check1 = tmp.substr(0,8);
  check2 = tmp.substr(8,8);
  currentOutput[10] = static_cast<char>(std::stoi(check1, nullptr, 2));
  currentOutput[11] = static_cast<char>(std::stoi(check2, nullptr, 2));

  cerr << "OUTPUT OF " << canID << ": " << currentOutput << endl;
  ofs << currentOutput;
  ofs.flush();

  retrans->reset();
  keepalive->reset();
}
//------------------------------------------------------------------------------------------
//Name: checksum
//Description: Performs the calculation of the checksum on the packet.
//Output: the checksum in binary (string) form
//------------------------------------------------------------------------------------------
string TCB::checksum(string in){
  string tmp, check1, check2;
  unsigned int total, t = 0, q = 0;
  vector<int> num;

  //clear checksum in frame
  in[10] = (char) 0;
  in[11] = (char) 0;

  for (int j = 0; j < in.length(); ++j){
    tmp.clear();
    for (int i = 7; i >= 0; --i) tmp += ((in[j] & (1 << i))? '1' : '0');
    ++j;
        if (j+1 == in.length()){
            tmp.append("00000000");
            break;
        }
        for (int i = 7; i >= 0; --i) tmp += ((in[j] & (1 << i))? '1' : '0');
        q = bitset<16>(tmp).to_ulong();
        num.push_back(q);
  }

    for (int i = 0; i < num.size(); ++i) t += num[i];

    while (t>>16) t = (t & 0xffff) + (t >> 16);

  t = 0xffff - t;

  bitset<16> bits (t);
  tmp.clear();
  return bits.to_string();
}
//------------------------------------------------------------------------------------------
//Name: MCU1toSSM_SL
//Description: Handles session layer between MCU1 and SSM.
//Output: see return values
//------------------------------------------------------------------------------------------
```

```
char TCB::MCU1toSSM_SL(string sl, string pay){
  string decoded;

  if (sl.size() == 0) return 30;

  //decrypt and then run the switch
  try{
    ECB_Mode< AES >::Decryption d;
    d.SetKey(key, sizeof(key));
    StringSource s(sl.substr(1,sl.size()-1), true, new StreamTransformationFilter(d, new StringSink(decoded)));
  }
  catch(const CryptoPP::Exception& e){
    cerr << "ERR: MCU1TOSSM_SL_DECRYPTION_GONE_WRONG\n" << e.what() << "\nREJECTING (ERR_CODE 4)\n";
    return 25;
  }
  if (sl[0] == 'A'){
    if (decoded.compare("2") == 0){          //because this is a proof of concept
      isVerif = true;
      return 50;
    }
    else if (decoded.compare("1") == 0){
      cerr << "ERR: Connection rejected by SSM - password incorrect?\n";
    }
  }
  else if (sl[0] == 'B'){//cerr << sl << endl;
    if (decoded.compare("mastersthesis") == 0){
      isVerif = true;
      return 50;
    }
  }
  else if (sl[0] == 'D'){
    if (pay.size() > 0){
      ofstream ofs("/tmp/newstackentry"+string(SSMID), ios::out);
      //send lastvalue to the stack, need four entries: time, to (MCU2), from (MCU1), val
      ofs << time(NULL) << " " << (int)decoded[0] << " " << (int)canID << " " << pay;
    }
    return decoded[0];
  }
  return 25;      //rejected, or not applicable
}
//--------------------------------------------------------------------------------
//Name: SSMtoMCU2_SL
//Description: Handles session layer between SSM and MCU2.
//Output: see return values
//--------------------------------------------------------------------------------
char TCB::SSMtoMCU2_SL(string sl){
  string decoded;

  if (sl.size() == 0) return 30;

  try{
    ECB_Mode< AES >::Decryption d;
    d.SetKey(key, sizeof(key));
    StringSource s(sl.substr(1,sl.size()-1), true, new StreamTransformationFilter(d, new StringSink(decoded)));
  }
  catch(const CryptoPP::Exception& e){
    cerr << "ERR: SSMTOMCU2_SL_ENCRYPTION_GONE_WRONG\n" << e.what();
    if (MCU2 && lastPackGarb){      //for in case decryption of previous packet AL was a failure
      decoded.clear();
      decoded = to_string(latestFV);
      //lastPackGarb = false;
      cerr << "\nBecause last packet AL layer had garbage, use the same FV as last time\n";
```

```
      }
      else{
        cerr  << "\nREJECTING (ERR_CODE 4)\n";
        return 25;
      }
    }
    if (sl[0] == 'A'){
      if (decoded.compare("2") == 0){            //because this is a proof of concept
        isVerif = true;
        return 50;
      }
      else if (decoded.compare("1") == 0){
        cerr << "ERR: Connection rejected by SSM - password incorrect?\n";
      }
    }
    else if (sl[0] == 'B'){//cerr << sl << endl;
      //password check here (plain text for now, sorry)
      if (decoded.compare("mastersthesis") == 0){
        isVerif = true;
        return 50;
      }
    }
    else if (sl[0] == 'C'){
      latestFV = stof(decoded, NULL);
      return 30;
    }
    else if (sl[0] == 'E'){
      lastValue = stol(decoded, NULL, 10);
      return 60;
    }
    return 25;      //rejected, or not applicable
}
//----------------------------------------------------------------------------------------
//Name: MCU2dest
//Description: Look in current input, decrypt destination of MCU1's
//             message, and return it.
//Output: CAN ID of destination
//----------------------------------------------------------------------------------------
char TCB::MCU2dest(){
    string theID, msgenc = currentInput.substr(13, 16);
    try{ //do decrypting here
      ECB_Mode< AES >::Decryption d;
      d.SetKey(key, sizeof(key));
      StringSource s(msgenc, true, new StreamTransformationFilter(d, new StringSink(
          theID)));
    }
    catch(const CryptoPP::Exception& e){
      cerr << "ERR: MCU2DEST DECRYPTION GONE WRONG\n" << e.what() << "\nNOT PASSING
          ALONG MSG (ERR_CODE 4)\n";
      return 25;
    }
    return theID[0];
}
//----------------------------------------------------------------------------------------
//Name: SLtoAL
//Description: Send data to AL and get factor value.
//Output: N/A
//----------------------------------------------------------------------------------------
void TCB::SLtoAL(char CANID){
    //cerr << "ENTERING SLTOAL\n";
    string path = ALpipepath + to_string(int(CANID)), msgenc = currentInput.substr(29,
        currentInput.size()-29), msg, lfv(15, '\0');
    ofstream ofs(path.c_str(), ios::out | ios::binary);
    int resultNow = 1;

```

```
  try{ //do decrypting here
    ECB_Mode< AES >::Decryption d;
    d.SetKey(key, sizeof(key));
    StringSource s(msgenc, true, new StreamTransformationFilter(d, new StringSink(msg)
        ));
    resultNow = stoi(msg, NULL, 10);
  }
  catch(const CryptoPP::Exception& e){
    cerr << "ERR: SLTOAL DECRYPTION GONE WRONG\n" << e.what() << "\nNOT PASSING TO 
        APPLICATION LAYER (ERR CODE 4)\n";
    latestFV = 1;
    resultNow = lastValue;
    lastPackGarb = true;
    return;
  }
  if (lastValue < 1){
    lastValue = resultNow;
    latestFV = 1;
  }
  else if (lastPackGarb){
    lastPackGarb = false;
    latestFV = 1;
  }
  else latestFV = resultNow / lastValue;
  ofs << msg;   //off to AL with ye

  //encrypt latestFV
  try{
    snprintf(&lfv[0], lfv.size(), "%.2f", latestFV);
    cerr << "lfv: " << lfv << endl;
    latestFVenc.clear();
    ECB_Mode< AES >::Encryption e;
    e.SetKey(key, sizeof(key));
    StringSource(lfv, true, new StreamTransformationFilter(e, new StringSink(
        latestFVenc)));
  }
  catch(const CryptoPP::Exception& e){
    cerr << "ERR: SLTOAL ENCRYPTION GONE WRONG\n" << e.what() << "\nEXITING\n";
    exit(1);
  }
}
```

B.1.5 TCBmodule.hpp

```
#ifndef _TCBMODHPP
#define _TCBMODHPP

#include "TCB.hpp"
#include <atomic>
#include <chrono>

struct hackInfo{
  char to, from;
  std::string lastAL;
  std::unique_ptr<Timer> hack_timer, resend;
  hackInfo(char t, char f, std::string las) :
    to(t),
    from(f),
    lastAL(las),
    hack_timer(std::make_unique<Timer>(4)),
    resend(std::make_unique<Timer>(0))
    { hack_timer->reset(); }
};

struct toAndFrom{
  char to, from;
  bool transmitting;
  toAndFrom(char t, char f) :
    to(t),
    from(f),
    transmitting(false)
    {}
};
```

```
class TCBmodule{
private:
  //SSMonly vector is for SSM's connections to MCU1
  std::vector<std::shared_ptr<TCB>> TCBentries, SSMonly;
  std::vector<hackInfo> hacks;
  std::vector<toAndFrom> sessionConns;
  unsigned char CANID;
  std::string helppath = "/tmp/help", infilepipepath, outfilepipepath;
  pid_t ibPid, MCUinPid, awarePid, stackPid;
  bool SSMtoMCU2 = false, isSSM = false;
  std::atomic_bool killAllConn, doneKilling;
  byte key[CryptoPP::AES::MAX_KEYLENGTH];

public:
  TCBmodule(char c);
  ~TCBmodule(){}
  void startInputLoop(std::string s);
  void startMCUin(std::string s);
  void startAware();
  void startStack();
  void stopInputLoop(){ kill(ibPid, SIGKILL); }
  void stopMCUin(){ kill(MCUinPid, SIGKILL); }
  void stopAware(){ kill(awarePid, SIGKILL); }
  void stopStack(){ kill(stackPid, SIGKILL); }
  bool checkConn(char c);
  void acceptInput();
  int entryCheck(char c);
  int entryCheckSSM(char c);
  std::string checksum(std::string in);
  bool checkChecksum(std::string in);
  void giveOutput(int index);
  void msgFromAL(char destID, std::string msg);
  int findMatchingSessConn(char t, char f);
  int findMatchingAck(int a);
  int findMatchingAckSSM(char c);
  void endAllConnections();
  void endConnectionLoop();
  void discoveredHack();
  void checkHacks();
  void retransmissions();
  void retransmissionsSSM(int s);
};

#endif
```

B.1.6 TCBmodule.cpp

```
#include "TCBmodule.hpp"

using namespace std;
using CryptoPP::AES;
using CryptoPP::AutoSeededRandomPool;
using CryptoPP::ECB_Mode;
using CryptoPP::Exception;
using CryptoPP::HexEncoder;
using CryptoPP::HexDecoder;
using CryptoPP::StringSink;
using CryptoPP::StringSource;
using CryptoPP::StreamTransformationFilter;

//Name: [TCBmodule constructor]
//Description: initialize TCB module for the MCU/SSM
//Output: N/A
//
  ------------------------------------------------------------------------------------------------
TCBmodule::TCBmodule(char c){
  int tmpCANID, sysStat;
  stringstream ss;
  ifstream ifs("my.key", ios::in);
  char readIn[32];

```

```
  //get CANID into hex, then initialize our filepipepath
  CANID = c;
      if (c == 34){
        isSSM = true;
        //hack_timer = make_unique<Timer>(4);
        thread thAware(&TCBmodule::startAware, this);
        thAware.detach();
        thread thStack(&TCBmodule::startStack, this);
        thStack.detach();
      }
  infilepipepath = "/tmp/ibToTCB" + to_string(CANID);
  outfilepipepath = "/tmp/TCBToOb" + to_string(CANID);

  killAllConn = false;
  doneKilling = false;

  //load inputbuffer.cpp
  ss << "./ib " << hex << CANID << " &";
  thread th(&TCBmodule::startInputLoop, this, to_string(CANID));
  th.detach();

  //get thread going for input buffer
  thread th2(&TCBmodule::acceptInput, this);
  th2.detach();

  //get thread going for MCUin
  if (!isSSM){
    thread th3(&TCBmodule::startMCUin, this, to_string(CANID));
    th3.detach();
  }

  if (ifs.is_open()){
    ifs.read(readIn, AES::MAX_KEYLENGTH);
    for (int i = 0; i < AES::MAX_KEYLENGTH; ++i) key[i] = (byte) readIn[i];
    ifs.close();
  }
  else{
    cerr << "ERR: can't open AES key file in TCBmodule\n";
    exit(1);
  }
}
//------------------------------------------------------------------------------------------------
//Name: startInputLoop
//Description: function to run the ib executable
//Output: N/A
//------------------------------------------------------------------------------------------------
void TCBmodule::startInputLoop(string s){
  int status;
  ibPid = fork();

  switch(ibPid){
    case -1: { perror("fork"); exit(1); }
    case 0: {
      execl("./ib", s.c_str(), (const char*) NULL);
      perror("execl");
    }
    default: {
      while (waitpid(ibPid, &status, 0) == -1);
      if (!WIFEXITED(status) || WEXITSTATUS(status) != 0){
        cerr << "ERR: input buffer threading problems\n";
        exit(1);
      }
      break;
    }
  }
}
//------------------------------------------------------------------------------------------------
```

```
//Name: startMCUin
//Description: function to run the MCUin executable
//Output: N/A
//---------------------------------------------------------------------------
void TCBmodule::startMCUin(string s){
  int status;
  MCUinPid = fork();

  switch(MCUinPid){
    case -1: { perror("fork"); exit(1); }
    case 0: {
      execl("/usr/bin/xterm", "xterm", "-hold", "-e", "./MCUin", s.c_str(), (const
          char*) NULL);
      perror("execl");
    }
    default: {
      while (waitpid(MCUinPid, &status, 0) == -1);
      if (!WIFEXITED(status) || WEXITSTATUS(status) != 0){
        cerr << "ERR: MCUin threading problems\n";
        exit(1);
      }
      break;
    }
  }
}
//---------------------------------------------------------------------------
//Name: startAware
//Description: function to run the aware executable
//Output: N/A
//---------------------------------------------------------------------------
void TCBmodule::startAware(){
  int status;
  awarePid = fork();

  switch(awarePid){
    case -1: { perror("fork"); exit(1); }
    case 0: {
      execl("/usr/bin/xterm", "xterm", "-hold", "-e", "./aware", (const char*) NULL);
      perror("execl");
    }
    default: {
      while (waitpid(awarePid, &status, 0) == -1);
      if (!WIFEXITED(status) || WEXITSTATUS(status) != 0){
        cerr << "ERR: aware threading problems\n";
        exit(1);
      }
      break;
    }
  }
}
//---------------------------------------------------------------------------
//Name: startStack
//Description: function to run the stack executable
//Output: N/A
//---------------------------------------------------------------------------
void TCBmodule::startStack(){
  int status;
  stackPid = fork();

  switch(stackPid){
    case -1: { perror("fork"); exit(1); }
    case 0: {
      execl("/usr/bin/xterm", "xterm", "-hold", "-e", "./stack", (const char*) NULL);
```

```
            perror("execl");
        }
        default: {
            while (waitpid(stackPid, &status, 0) == -1);
            if (!WIFEXITED(status) || WEXITSTATUS(status) != 0){
                cerr << "ERR: aware threading problems\n";
                exit(1);
            }
            break;
        }
    }
}
//--------------------------------------------------------------------------------
//Name: checkConn
//Description: check to see if we have a connection with the specified device
//             If not, we have to make one. If we are SSM, this connection
//             is for MCU2.
//Output: N/A
//--------------------------------------------------------------------------------
bool TCBmodule::checkConn(char c){
    if (isSSM && SSMonly.empty()){
        cerr << "ERR: SSM cannot set AL-issued connections\n";
        return false;
    }

    else if (c == CANID){
        cerr << "ERR: Cannot establish connection to self\n";
        return false;
    }

    for (int i = 0; i < TCBentries.size(); ++i){
        if (TCBentries[i]->sameID(c)){
            cerr << "CONNECTION EXISTS AT " << i << endl;
            return true;
        }
    }

    //entry does not exist, let's get one going!
    //STEP 1
    int entryN = TCBentries.size();
    bool b[5] = {false, false, false, true, false};
    TCBentries.push_back(make_shared<TCB>(0, 0, c, 0));
    TCBentries[entryN]->makeConn = true;
    TCBentries[entryN]->prepareSwitch(0, 0, b, "");
    TCBentries[entryN]->makeOutput(CANID, outfilepipepath, "");
    TCBentries[entryN]->needAck = true;
    giveOutput(-1);

    return true;
}
//--------------------------------------------------------------------------------
//Name: acceptInput
//Description: loop constantly for anything read in by ib. Parse input
//             based on whether message is for SSM or self.
//Output: N/A
//--------------------------------------------------------------------------------
void TCBmodule::acceptInput(){
    struct stat buf;
    string input;
    int entryN, toSSM2, sessConnEnt;
    unsigned int s, a;
    char c, chksm[2];
    bool newStart[5] = {false, false, false, true, false}, flags[5] = {false, false,
        false, false, false};

```

```
  for(;;){
    if (stat(infilepipepath.c_str(), &buf) != -1){
      ifstream ifs(infilepipepath.c_str(), ios::in | ios::binary);
      for (string line; getline(ifs, line);){
        input.append(line);
        if (ifs.peek() && ifs.eof()) break;
        else input.append("\n");
      }
      cerr << "//----------------------------------------------------------------\n";

      chksm[0] = input[10];
      chksm[1] = input[11];
      try{
        cerr << "INPUT: " << input.substr(0,8) << " " << (int)input[8] << " " <<
            bitset<8>(input[9]) << " " << (int) input[10] << " " << (int)input[11] <<
            " " <<input.substr(12, input.size()-12) << endl;
      } catch (...){
        cerr << "Oops! Got a little ahead of myself there.\n";
        ifs.seekg(0, ios::end);
        if (ifs.tellg() < 1){              //in case there's a file, but it's empty
          input.clear();
          remove(infilepipepath.c_str());      //clear out buffer
        }
        continue;
      }

      //if our checksums match, it's a good packet
      if (checkChecksum(input) && input.size() > 11){
        try{
          //extract other values from the packet
          s = stoi(input.substr(0,4), nullptr, 16);
          a = stoi(input.substr(4,4), nullptr, 16);
          c = input[8];
          bitset<8> connFlags(input[9]);
          flags[0] = connFlags[7];
          flags[1] = connFlags[6];
          flags[2] = connFlags[5];
          flags[3] = connFlags[4];
          flags[4] = connFlags[3];

          if (isSSM){
            entryN = entryCheckSSM(c);
            if (entryN != -1){
              if (SSMonly[entryN]->thisConnState == 4 && connFlags.to_ulong() == 128){
                    //MCU1's last ACK
                SSMonly[entryN]->prepareSwitch(s, a, flags, input);
                input.clear();
                remove(infilepipepath.c_str());      //clear out buffer
                continue;
              }
              //STEP 4
              c = SSMonly[entryN]->MCU1toSSM_SL(input.substr(12, 17), input.substr
                  (29,16));
              if (SSMonly[entryN]->thisConnState == 4 && SSMonly[toSSM2]->flagStats(
                  flags) == 2){
                //for getting ACK from FIN-ACK (going into TIME_WAIT)
                SSMonly[entryN]->prepareSwitch(s, a, flags, input);
                input.clear();
                remove(infilepipepath.c_str());      //clear out buffer
                continue;
              }
              if (c != 25){
                SSMonly[entryN]->needAck = true;  //for after MCU2 init
                //find the new entry, or make one
                toSSM2 = entryCheck(c);
                if (toSSM2 == -1){
                  //notify aware alg of new connection
                  ofstream ofs("/tmp/makeawareentry" + string(SSMID), ios::out);
                  toSSM2 = TCBentries.size();
                  TCBentries.push_back(make_shared<TCB>(0, 0, c, 2));
                  sessionConns.push_back(toAndFrom(c, SSMonly[entryN]->getCanID()));
                  //make aware session here
                  ofs << c << SSMonly[entryN]->getCanID();
```

```
      ofs.close();
      TCBentries[toSSM2]->makeConn = true;
      TCBentries[toSSM2]->prepareSwitch(0, 0, newStart, input);
      TCBentries[toSSM2]->makeOutput(CANID, outfilepipepath, "");
      TCBentries[toSSM2]->needAck = true;
      SSMtoMCU2 = true;
      giveOutput(toSSM2);
      //now update the MCU1/SSM connection
      if (connFlags.to_ulong() == 192){ //if PSHACK
        SSMonly[entryN]->currentInput.clear();
        SSMonly[entryN]->currentInput = input;
        SSMonly[entryN]->needAck = true;
        SSMonly[entryN]->prepareSwitch(s, a, flags, input);
      }
    }
    else if (TCBentries[toSSM2]->thisConnState == 3 && SSMonly[entryN]->thisConnState < 3){
      ofstream ofs("/tmp/makeawareentry" + string(SSMID), ios::out);
      SSMonly[entryN]->currentInput.clear();
      SSMonly[entryN]->currentInput = input;
      SSMonly[entryN]->prepareSwitch(s, a, flags, input);
      SSMonly[entryN]->makeOutput(CANID, outfilepipepath, "");
      giveOutput(entryN);
      sessionConns.push_back(toAndFrom(c, SSMonly[entryN]->getCanID()));
      ofs << c << SSMonly[entryN]->getCanID();
      ofs.close();
    }
    else{ //MCU1 to SSM data
      toSSM2 = entryCheck(c);
      if (!SSMonly[entryN]->prepareSwitch(s, a, flags, input) && isSSM){
        retransmissionsSSM(s);
        input.clear();
        remove(infilepipepath.c_str());     //clear out buffer
        continue;
      }
      SSMonly[entryN]->needAck = true;
      if (TCBentries[toSSM2]->flagStats(flags) == 7){
        TCBentries[toSSM2]->killConn = true;
        TCBentries[toSSM2]->prepareSwitch(TCBentries[toSSM2]->seq, TCBentries[toSSM2]->ackn, flags, "");
      }
      else TCBentries[toSSM2]->sending = true;
      TCBentries[toSSM2]->makeOutput(CANID, outfilepipepath, input.substr(29, input.size()-29));
      TCBentries[toSSM2]->needAck = true;
      SSMtoMCU2 = true;
      giveOutput(toSSM2);
      sessConnEnt = findMatchingSessConn(TCBentries[toSSM2]->getCanID(), SSMonly[entryN]->getCanID());
      sessionConns[sessConnEnt].transmitting = true;
    }
  }
}
else if (entryCheck(c) != -1){
  //STEP 6
  entryN = entryCheck(c);
  char q = TCBentries[entryN]->SSMtoMCU2_SL(input.substr(12, 17));
  if (q == 50){
    TCBentries[entryN]->prepareSwitch(s, a, flags, input);
    TCBentries[entryN]->makeOutput(CANID, outfilepipepath, "");
    TCBentries[entryN]->needAck = true;
    SSMtoMCU2 = true;
    giveOutput(entryN);
    TCBentries[entryN]->needAck = false;
  }
  //STEP 8
  if (q == 30 && connFlags.to_ulong() == 128){  //if ACK
    toSSM2 = entryN;
    entryN = findMatchingAckSSM(c);
    sessConnEnt = findMatchingSessConn(TCBentries[toSSM2]->getCanID(), SSMonly[entryN]->getCanID());
    if (sessionConns[sessConnEnt].transmitting){
      ofstream ofs("/tmp/newawareentry"+string(SSMID), ios::out);
```

```
        ofs << TCBentries[toSSM2]->getCanID() << SSMonly[entryN]->getCanID() << " " << TCBentries[toSSM2]->latestFV; //send factor to the awareness
        ofs.close();
        sessionConns[sessConnEnt].transmitting = false;
      }
      if (entryN != -1){
        SSMonly[entryN]->makeOutput(CANID, outfilepipepath, "");
        giveOutput(entryN);
      }
      //update TCB entry for SSM to MCU2 conn
      TCBentries[toSSM2]->needAck = false;
      TCBentries[toSSM2]->currentInput.clear();
      TCBentries[toSSM2]->currentInput = input;
      TCBentries[toSSM2]->prepareSwitch(s, a, flags, input);
    }
    if (q == 30 && connFlags.to_ulong() == 136){  //if FIN-ACK
      toSSM2 = entryN;
      entryN = findMatchingAckSSM(c);
      if (entryN != -1){
        cerr << "FIN-ACK\n";            //SSMonly[entryN]->killed
        SSMonly[entryN]->makeOutput(CANID, outfilepipepath, "");
        giveOutput(entryN);
      }
      //update TCB entry for SSM to MCU2 conn
      TCBentries[toSSM2]->needAck = false;
      TCBentries[toSSM2]->currentInput.clear();
      TCBentries[toSSM2]->currentInput = input;
      TCBentries[toSSM2]->prepareSwitch(s, a, flags, input);
      //***WARNING!*** This sleep function is needed!
      //ob needs enough time to deliver MCU1's message first!
      this_thread::sleep_for(chrono::microseconds(40000));
      //if you remove this sleep line, MCU1 gets whatever MCU2 gets as well
      //This delay can be removed when this code is weaned off file pipes
      TCBentries[toSSM2]->makeOutput(CANID, outfilepipepath, "");
      SSMtoMCU2 = true;
      giveOutput(toSSM2);
    }
  }
  else{
    //STEP 2
    entryN = SSMonly.size();
    SSMonly.push_back(make_shared<TCB>(input.size(), 0, c, 2));
    if (SSMonly[entryN]->MCU1toSSM_SL(input.substr(12, input.size()-12), "") == 50){
      SSMonly[entryN]->prepareSwitch(s, a, flags, input);
      SSMonly[entryN]->makeOutput(CANID, outfilepipepath, "");
      SSMonly[entryN]->needAck = true;
      giveOutput(entryN);
      SSMonly[entryN]->needAck = false;
    }
  }
}
else{ //NOT SSM
  //STEP 3 (MCU1), STEP 5 (MCU2)
  //we are assuming that the newest entry on the stack is for the waiting connection
  entryN = entryCheck(c);
  if (entryN == -1){
    if (c == 34){ //for MCU1!
      entryN = findMatchingAck(a);
      if (connFlags.to_ulong() == 32){        //RST flag
        for (int i = 0; i < TCBentries.size(); ++i){
          if (TCBentries[i]->seq == a){      //go by ACK number for now
            TCBentries[i]->prepareSwitch(s, a, flags, input);
          }
        }
      }
      else if (TCBentries.size() == 0 || entryN == -1){ //for MCU2
        entryN = TCBentries.size();
        TCBentries.push_back(make_shared<TCB>(input.size(), 0, c, 1));
        cerr << "TCB pointer: " << TCBentries[entryN] << endl;
        if (TCBentries[entryN]->SSMtoMCU2_SL(input.substr(12, input.size()
```

```
                        -12)) == 50){
                    if (TCBentries[entryN]->isMCU1()) TCBentries[entryN]->makeConn = true;
                    TCBentries[entryN]->prepareSwitch(s, a, flags, input);
                    TCBentries[entryN]->makeOutput(CANID, outfilepipepath, "");
                    TCBentries[entryN]->needAck = true;
                    giveOutput(-1);
                  }
                }
                //after step 8, stop the initializing chain
                else if (TCBentries[entryN]->flagStats(flags) == 2){
                  if (TCBentries[entryN]->prepareSwitch(s, a, flags, input)){
                    input.clear();
                    remove(infilepipepath.c_str());
                    continue;
                  }
                  else cerr << "dun goofed\n";
                }
                else if(TCBentries[entryN]->prepareSwitch(s, a, flags, input)){
                  TCBentries[entryN]->makeOutput(CANID, outfilepipepath, "");
                  TCBentries[entryN]->needAck = true;
                  giveOutput(entryN);
                }
              }
              else{
                cerr << "SOMETHING ODD\n";
              }
            }
            else{//STEP 7 (MCU2), MCU2 also receives PSH data here
              if (TCBentries[entryN]->needAck && TCBentries[entryN]->prepareSwitch(s, a, flags, input)){
                char q = TCBentries[entryN]->SSMtoMCU2_SL(input.substr(12, 17));
                if (TCBentries[entryN]->isMCU2() && TCBentries[entryN]->thisConnState == 3 && q == 60) TCBentries[entryN]->SLtoAL(CANID);
                else if (q == 25){
                  input.clear();
                  remove(infilepipepath.c_str());
                  continue;
                }
                if (TCBentries[entryN]->isMCU2() && TCBentries[entryN]->thisConnState == 5 && connFlags.to_ulong() == 128){ //MCU2 getting ACK after FIN-ACK
                  input.clear();
                  remove(infilepipepath.c_str());
                  continue;
                }
                else {
                  TCBentries[entryN]->makeOutput(CANID, outfilepipepath, "");
                  TCBentries[entryN]->needAck = true;
                  giveOutput(-1);
                }
              }
            }
          }
        } catch(std::out_of_range& e){
          cerr << "ERR_CODE 1\n";   //for the ugly chars, testing eval
        }
        catch(exception& ex){
          cerr << "GENERAL ERROR\n" << ex.what() << endl;      //for other
        }
      }
      else{
        cerr << "ERR_CODE 2\n";
      }
      input.clear();
      remove(infilepipepath.c_str());      //clear out buffer
    }
    //logic here for any necessary retransmissions
    retransmissions();
    //logic here for erasing entries when time_wait expires
    for (int i = 0; i < SSMonly.size(); ++i){
      if (SSMonly[i]->time_wait->isExpired() && SSMonly[i]->thisConnState == 5){
        cerr << "NOT: Connection " << SSMonly[i]->getCanID() << " in SSMonly vector 
```

```
            removed\n";
      SSMonly.erase(SSMonly.begin()+i);
    }
  }
  for (int i = 0; i < TCBentries.size(); ++i){
    if (TCBentries[i]->time_wait->isExpired() && TCBentries[i]->thisConnState == 5){
      cerr << "NOT: Connection " << TCBentries[i]->getCanID() << " in TCBentries vector removed\n";
      TCBentries.erase(TCBentries.begin()+i);
    }
  }
  //logic here for if a hack has been discovered
  if (isSSM && (stat(helppath.c_str(), &buf) != -1)) discoveredHack();
  //logic here to check on the hacks
  if (isSSM) checkHacks();
  //logic here for if it's time to end the program
  if (killAllConn) endConnectionLoop();
 }
}
//------------------------------------------------------------------------------------------------
//Name: retransmissions
//Description: MCU1 only. If no ACK was received, send data again.
//Output: N/A
//------------------------------------------------------------------------------------------------
void TCBmodule::retransmissions(){
 for (int i = 0; i < TCBentries.size(); ++i){
  if (TCBentries[i]->isMCU1() && TCBentries[i]->needAck && TCBentries[i]->retrans->isExpired()){
   ofstream ofs(outfilepipepath.c_str(), ios::out | ios::binary);
   ofs << TCBentries[i]->currentOutput;
   ofs.flush();
   giveOutput(-1);
   TCBentries[i]->retrans->reset();
   cerr << "NOT: had to resend packet to " << TCBentries[i]->getCanID() << " b/c timer expired\n";
   //***WARNING!*** This sleep function is needed!
   //ob needs enough time to deliver MCU1's message first!
   this_thread::sleep_for(chrono::microseconds(40000));
   //if you remove this sleep line, and there are multiple expired timers, not all messages may be sent
   //This delay can be removed when this code is weaned off file pipes
  }
 }
}
//------------------------------------------------------------------------------------------------
//Name: retransmissionsSSM
//Description: SSM only. If MCU1 never got that previous ACK, it will show in the retransmission input.
//Output: N/A
//------------------------------------------------------------------------------------------------
void TCBmodule::retransmissionsSSM(int s){
 for (int i = 0; i < SSMonly.size(); ++i){
  if (SSMonly[i]->ackn == s){
   ofstream ofs(outfilepipepath.c_str(), ios::out | ios::binary);
   ofs << SSMonly[i]->currentOutput;
   ofs.flush();
   giveOutput(i);
   SSMonly[i]->retrans->reset();
   cerr << "NOT: had to resend ACK packet to " << TCBentries[i]->getCanID() << " b/c timer expired\n";
   //***WARNING!*** This sleep function is needed!
   //ob needs enough time to deliver MCU1's message first!
   this_thread::sleep_for(chrono::microseconds(40000));
   //if you remove this sleep line, and there are multiple expired timers, not all messages may be sent
```

```
      //This delay can be removed when this code is weaned off file pipes
      break;
    }
  }
}
//-----------------------------------------------------------------------------

//Name: findMatchingSessConn
//Description: SSM only. Check for matching session connection.
//Output: Array index of connection, or -1 if it doesn't exist
//-----------------------------------------------------------------------------

int TCBmodule::findMatchingSessConn(char t, char f){
  for (int i = 0; i < sessionConns.size(); ++i){
    if (sessionConns[i].to == t && sessionConns[i].from == f) return i;
  }
  cerr << "WARN: returning -1 from findMatchingSessConn(), either new entry or this could be bad!\n";
  return -1;
}
//-----------------------------------------------------------------------------

//Name: findMatchingAck
//Description: MCU1 only. (?) Check existing connections to match up
//             entry from SSM to entry for MCU2.
//Output: Array index of connection, or -1 if it doesn't exist
//-----------------------------------------------------------------------------

int TCBmodule::findMatchingAck(int a){
  int entryN = -1;
  for (int i = 0; i < TCBentries.size(); ++i){
    if (TCBentries[i]->seq == a && TCBentries[i]->needAck){
      entryN = i;
      TCBentries[entryN]->needAck = false;
      break;
    }
  }
  if (entryN == -1) cerr << "WARN: returning -1 from findMatchingAck(), either new entry or this could be bad!\n";
  return entryN;
}
//-----------------------------------------------------------------------------

//Name: findMatchingAckSSM
//Description: SSM only. Check existing connections to match up
//             entry from SSM to entry for MCU1.
//Output: Array index of connection, or -1 if it doesn't exist
//-----------------------------------------------------------------------------

int TCBmodule::findMatchingAckSSM(char c){
  int entryN = -1;
  for (int i = 0; i < SSMonly.size(); ++i){
    if (SSMonly[i]->needAck && SSMonly[i]->MCU2dest() == c){
      entryN = i;
      SSMonly[entryN]->needAck = false;
      break;
    }
  }
  if (entryN == -1) cerr << "WARN: returning -1 from findMatchingAckSSM(), either new entry or this could be bad!\n";
  return entryN;
}
//-----------------------------------------------------------------------------

//Name: entryCheck
//Description: Check to see if we have a TCB connection entry in our TCBentries
```

```
//            vector.
//Output: Array index of connection, or -1 if it doesn't exist
//------------------------------------------------------------------------------------------------------------
int TCBmodule::entryCheck(char c){
  int entryN = -1;
  for (int i = 0; i < TCBentries.size(); ++i){
    if (TCBentries[i]->sameID(c)){
      entryN = i;
      break;
    }
  }
  return entryN;
}
//------------------------------------------------------------------------------------------------------------
//Name: entryCheckSSM
//Description: Same as entryCheck, but for the SSM and SSMonly vector.
//Output: Array index of connection, or -1 if it doesn't exist.
//------------------------------------------------------------------------------------------------------------
int TCBmodule::entryCheckSSM(char c){
  int entryN = -1;
  for (int i = 0; i < SSMonly.size(); ++i){
    if (SSMonly[i]->sameID(c)){
      entryN = i;
      break;
    }
  }
  return entryN;
}
//------------------------------------------------------------------------------------------------------------
//Name: giveOutput
//Description: Send our packets out to ob. System command depends on
//            whether or not we are SSM, and if we are, whether or not
//            we are sending to MCU1 or MCU2.
//Output: N/A
//------------------------------------------------------------------------------------------------------------
void TCBmodule::giveOutput(int index){
  stringstream ss;
  int sysStat;

  //MCU1/2 to SSM
  if (!isSSM && index == -1) ss << "./ob " << dec << 34 << " " << (int)CANID << " &";
  //SSM to MCU1
  else if (isSSM && !SSMtoMCU2) ss << "./ob " << dec << (unsigned) SSMonly[index]->
      getCanID() << " " << 34 << " &";
  //SSM to MCU2
  else if (isSSM && SSMtoMCU2){
    //cerr << "whoops\n";
    ss << "./ob " << dec << (unsigned) TCBentries[index]->getCanID() << " " << 34 <<
        " &";
    SSMtoMCU2 = false;
  }
  else ss << "./ob " << dec << 34 << " " << (int)CANID << " &";

  sysStat = system(ss.str().c_str());
  if (sysStat < 0 && sysStat >= 127){
    cerr << "ERR: unable to run command " << ss.str() << ", errno " << errno << endl;
  }
}
//------------------------------------------------------------------------------------------------------------
//Name: msgFromAL
//Description: Our message from MCUout to be sent. (application layer)
```

```
//Output: N/A
//-----------------------------------------------------------------------------------------
void TCBmodule::msgFromAL(char destID, string msg){
  int entryN = entryCheck(destID);
  string msgenc;

  try{
    ECB_Mode< AES >::Encryption e;
    e.SetKey(key, sizeof(key));
    StringSource(msg, true, new StreamTransformationFilter(e, new StringSink(msgenc)));
    TCBentries[entryN]->sending = true;
    TCBentries[entryN]->makeOutput(CANID, outfilepipepath, msgenc);
    TCBentries[entryN]->needAck = true;
    giveOutput(-1);
  }
  catch(const CryptoPP::Exception& e){
    cerr << "ERR: MSGFROMAL ENCRYPTION GONE WRONG\n" << e.what() << "\nNO PACKET SENT\n";
  }
}
//-----------------------------------------------------------------------------------------
//Name: checkChecksum
//Description: Check to see if the checksum for the incoming packet
//             is valid. (deprecated for now)
//Output: true if valid checksum, false otherwise
//-----------------------------------------------------------------------------------------
bool TCBmodule::checkChecksum(string in){
  string check1, check2, tmp = checksum(in);
  check1 = tmp.substr(0,8);
  check2 = tmp.substr(8,8);

  return (static_cast<char>(std::stoi(check1, nullptr, 2)) == in[10]) && (static_cast<char>(std::stoi(check2, nullptr, 2)) == in[11]);
}
//-----------------------------------------------------------------------------------------
//Name: checksum
//Description: Performs the calculation of the checksum on the packet.
//Output: the checksum in binary (string) form
//-----------------------------------------------------------------------------------------
string TCBmodule::checksum(string in){
  string tmp, check1, check2;
  unsigned int total, t = 0, q = 0;
  vector<int> num;

  //clear checksum in frame
  in[10] = (char) 0;
  in[11] = (char) 0;

  for (int j = 0; j < in.length(); ++j){
    tmp.clear();
    for (int i = 7; i >= 0; --i) tmp += ((in[j] & (1 << i))? '1' : '0');
    ++j;
        if (j+1 == in.length()){
          tmp.append("00000000");
          break;
        }
        for (int i = 7; i >= 0; --i) tmp += ((in[j] & (1 << i))? '1' : '0');
        q = bitset<16>(tmp).to_ulong();
        num.push_back(q);
  }

    for (int i = 0; i < num.size(); ++i) t += num[i];
```

```
    while (t>>16) t = (t & 0xffff) + (t >> 16);

  t = 0xffff - t;

  bitset<16> bits (t);
  tmp.clear();
  return bits.to_string();
}
//------------------------------------------------------------------------------------------------
//Name: endAllConnections
//Description: When disconnecting, first gracefully terminate all connections.
//Output: N/A
//------------------------------------------------------------------------------------------------
void TCBmodule::endAllConnections(){
  for(;;){
    killAllConn = true;
    if (doneKilling) return;  //done with ending all connections
  }
}
//------------------------------------------------------------------------------------------------
//Name: endConnectionLoop
//Description: Going through our TCBentries vector, gracefully terminate each
    connection.
//Output: N/A
//------------------------------------------------------------------------------------------------
void TCBmodule::endConnectionLoop(){
  struct stat buf;
  string input;
  int entryN, toSSM2, tmp;
  unsigned int s, a;
  char c, chksm[2];
  bool newStart[5] = {false, false, false, true, false}, flags[5] = {false, false,
      false, false, false};

  bool f[5] = {false, false, false, false, true};
  for (int i = 0; i < TCBentries.size(); ++i){
    if (TCBentries[i]->thisConnState == 3){
      TCBentries[i]->killConn = true;
      TCBentries[i]->TCBswitch(f);
      TCBentries[i]->makeOutput(CANID, outfilepipepath, "");
      TCBentries[i]->needAck = true;
      giveOutput(-1);
      for (;;){
        if (stat(infilepipepath.c_str(), &buf) != -1){
          ifstream ifs(infilepipepath.c_str(), ifstream::in | ios::binary);
          ifs >> input;
          cerr << "//--------------------------------------------------------\n";
          cerr << "END_INPUT: " << input.substr(0,8) << " " << (int)input[8] << " " <<
                bitset<8>(input[9]) << endl;

          //if our checksums match, it's a good packet
          chksm[0] = input[10];
          chksm[1] = input[11];
          if (checkChecksum(input)){
            //extract other values from the packet
            s = stoi(input.substr(0,4), nullptr, 16);
            a = stoi(input.substr(4,4), nullptr, 16);
            c = input[8];
            bitset<8> connFlags(input[9]);
            f[0] = connFlags[7];
            f[1] = connFlags[6];
            f[2] = connFlags[5];
            f[3] = connFlags[4];
            f[4] = connFlags[3];
```

```
          if (connFlags.to_ulong() == 136 && TCBentries[i]->prepareSwitch(s, a, f,
              input)){
            TCBentries[i]->makeOutput(CANID, outfilepipepath, "");
            giveOutput(-1);
            usleep(50);       //give ob a chance to work!
            break;            //done with this entry
          }
        }
        retransmissions();
        input.clear();
        remove(infilepipepath.c_str());     //clear out buffer
      }
    }
  }
}
  input.clear();
  remove(infilepipepath.c_str());     //clear out buffer
  doneKilling = true;
  return;
}
//------------------------------------------------------------------------------------------------
//Name: discoveredHack
//Description: If the stack notifies us of a hack, do something.
//Output: N/A
//------------------------------------------------------------------------------------------------
void TCBmodule::discoveredHack(){
  ifstream ifs(helppath.c_str(), ios::in | ios::binary);
  char culprit, victim;
  string input, lastData;
  cerr << "~~~HACK_DETECTED~~~\n";

  for (string line; getline(ifs, line);){
    input.append(line);
    if (ifs.peek() && ifs.eof()) break;
    else input.append("\n");
  }
  victim = input[0];
  culprit = input[1];
  lastData = input.substr(3, 16);
  //do RST here
  for (int i = 0; i < SSMonly.size(); ++i){
    if (SSMonly[i]->getCanID() == culprit){
      bool b[5] = {false, false, true, false, false};
      SSMonly[i]->hardReset = true;
      SSMonly[i]->TCBswitch(b);
      SSMonly[i]->makeOutput(CANID, outfilepipepath, "");
      giveOutput(i);
      hacks.push_back(hackInfo(victim, culprit, lastData));
      for (int j = 0; j < sessionConns.size(); ++j){
        if (sessionConns[i].from == culprit){
          ofstream ofs("/tmp/delaware"+string(SSMID), ios::out);
          ofs << sessionConns[i].to << sessionConns[i].from;
          sessionConns.erase(sessionConns.begin()+i);
        }
      }
    }
  }
  remove(helppath.c_str());
}
//------------------------------------------------------------------------------------------------
//Name: checkHacks
//Description: Check on status of [un]resolved hacks.
//Output: N/A
//------------------------------------------------------------------------------------------------
void TCBmodule::checkHacks(){
```

```
  for (int i = 0; i < hacks.size(); ++i){
    if (hacks[i].hack_timer->isExpired() && hacks[i].resend->isExpired()){  //do we
        need archived data?
      bool b[5] = {false, true, false, false, false};
      int toSSM2 = entryCheck(hacks[i].to);
      TCBentries[toSSM2]->TCBswitch(b);
      TCBentries[toSSM2]->makeOutput(CANID, outfilepipepath, hacks[i].lastAL);
      SSMtoMCU2 = true;
      giveOutput(toSSM2);
      //***WARNING!*** This sleep function is needed!
      //ob needs enough time to deliver MCU1's message first!
      this_thread::sleep_for(chrono::microseconds(40000));
      //if you remove this sleep line, MCU1 will not get the RST message
      //This delay can be removed when this code is weaned off file pipes
      hacks[i].resend->reset();
    }
    else{
      for (int j = 0; j < SSMonly.size(); ++j){
        for (int k = 0; k < hacks.size(); ++k){
          if (SSMonly[j]->getCanID() == hacks[k].from && SSMonly[j]->thisConnState !=
              5){
            cerr << "hack threat gone\n";
            hacks.erase(hacks.begin()+k);
            break;
          }
        }
      }
    }
  }
}
```

B.1.7 outputbuffer.cpp

```
#include <linux/can.h>
#include <linux/can/raw.h>

#include <net/if.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>

#include <cerrno>
#include <csignal>
#include <cstdio>
#include <cstring>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <string>
#include <sstream>

using namespace std;

int main (int argc, char** argv){

  const char *intrf = "vcan0";
  string msg;
  struct canfd_frame frame;
  stringstream ss;
  int destID = strtol(argv[1], NULL, 10);
  int srcID = strtol(argv[2], NULL, 10);
  ifstream ifs;
  int rc, sockfd, opt, bytes, enable = 1;
  struct sockaddr_can canaddr;
  struct ifreq ifr;
  string fifo = "/tmp/TCBToOb" + to_string(srcID), out;
  const char *filepipepath = fifo.c_str();

  sockfd = socket(PF_CAN, SOCK_RAW, CAN_RAW);
  if (sockfd == -1){
    cerr << "Can't open socket, errno: " << errno << endl;
```

```
      return 1;
    }

    rc = setsockopt(sockfd, SOL_CAN_RAW, CAN_RAW_FD_FRAMES, &enable, sizeof(enable));
    if (rc == -1){
      cerr << "Can't set socket options\n";
      return 1;
    }

    std::strncpy(ifr.ifr_name, intrf, IFNAMSIZ);
    if (ioctl(sockfd, SIOCGIFINDEX, &ifr) == -1){
      cerr << "Can't interact with network interface, errno " << errno << endl;
      return 1;
    }

    canaddr.can_family = AF_CAN;
    canaddr.can_ifindex = ifr.ifr_ifindex;
    fcntl(sockfd, F_SETFL, O_NONBLOCK);
    rc = bind(sockfd, (struct sockaddr *)&canaddr, sizeof(canaddr));

    if (rc == -1){
      cerr << "Can't bind socket\n";
      return 1;
    }

    //get packet from pipe
    ifs.open(filepipepath, ios::in | ios::binary);
    if (ifs.is_open()){
      for (string line; getline(ifs, line);){ //watch out for whitespaces!
        out.append(line);
        if (ifs.peek() && ifs.eof()) break;
        else out.append("\n");
      }
      frame.can_id = destID;
      frame.len = out.size();
      for (int i = 0; i < 64; ++i) frame.data[i] = (int)out[i];
      bytes = write(sockfd, &frame, sizeof(struct canfd_frame));

      //destroy file pipe
      ifs.close();
      remove(filepipepath);
    }

    return 0;
}
```

B.1.8 inputbuffer.cpp

```
#include <linux/can.h>
#include <linux/can/raw.h>

#include <net/if.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>

#include <cerrno>
#include <csignal>
#include <cstdio>
#include <cstring>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <string>
#include <sstream>

using namespace std;

int main(int argc, char** argv) {

```

```
  const char *intrf = "vcan0";
  int listeningID = strtol(argv[argc-1], NULL, 10);
  int rc, sockfd, opt, enable = 1;
  struct sockaddr_can canaddr;
  struct ifreq ifr;
  string filepipepath = "/tmp/ibToTCB" + to_string(listeningID);
  sockfd = socket(PF_CAN, SOCK_RAW, CAN_RAW);
  if (sockfd == -1){
    cerr << "Can't open socket for listeningID " << listeningID << endl;
    return 1;
  }

  rc = setsockopt(sockfd, SOL_CAN_RAW, CAN_RAW_FD_FRAMES, &enable, sizeof(enable));
  if (rc == -1){
    cerr << "Can't set socket options for listeningID " << listeningID << endl;
    return 1;
  }

  std::strncpy(ifr.ifr_name, intrf, IFNAMSIZ);
  if (ioctl(sockfd, SIOCGIFINDEX, &ifr) == -1){
    cerr << "Can't interact with network interface for listeningID " << listeningID << dec << ", errno " << errno << endl;
    return 1;
  }

  canaddr.can_family = AF_CAN;
  canaddr.can_ifindex = ifr.ifr_ifindex;
  fcntl(sockfd, F_SETFL, O_NONBLOCK);
  rc = bind(sockfd, (struct sockaddr *)&canaddr, sizeof(canaddr));
  if (rc == -1){
    cerr << "Can't bind socket for listeningID "<< listeningID << endl;
    return 1;
  }

  //loop for a new frame, collect it, and put payload in pipe for TCBmodule
  for(;;){
    struct canfd_frame fr;

    int bytes = read(sockfd, &fr, CANFD_MTU);
    if (bytes > 8 && fr.can_id == listeningID){
      //cout << bytes << endl;
      ofstream ofs;
      ostringstream oss("");
      int fd;
      struct stat buf;

        for (int i = 0; i < fr.len; ++i) oss << fr.data[i];
        //cerr << "got packet: " << oss.str() << endl;

        if (stat(filepipepath.c_str(), &buf) == -1) ofs.open(filepipepath.c_str(), ios::out | ios::binary);
          ofs << oss.str();
          ofs.close();
          //cerr << "got packet: " << oss.str() << endl;

          //cerr << oss.str().size() << endl;
          oss.str(string(""));
          oss.flush();
    }
    fr.can_id = 0;
    fr.len = 0;
    for (int i = 0; i < 64; ++i) fr.data[i] = '\0';
  }

  return 0;

}
```

B.1.9 MCUin.cpp

```
#include <sys/stat.h>
#include <fstream>
#include <iostream>
```

```
#include <thread>
#include <cerrno>
#include <csignal>
#include <cstdio>
#include <string>
#include <cstring>
#include <cstdlib>

using namespace std;

int main(int argc, char** argv) {

  struct stat buf;
  string latestVal, currentVal;

  int CANID = strtol(argv[1], NULL, 10);
  string datafilepipepath = "/tmp/TCBtoAL"+to_string(CANID);

  cout << "MCUIN␣FOR␣CAN␣ID␣" << CANID << endl << endl;
  cout << "CURRENT␣VALUE:␣N/A";

  for (;;){
    if (stat(datafilepipepath.c_str(), &buf) != -1){
      ifstream in(datafilepipepath.c_str(), ios::in);
      latestVal.clear();
      in >> latestVal;
      if (latestVal.compare(currentVal) != 0){
        cout << endl;   //to prove that the value has changed
        currentVal.clear();
        currentVal = latestVal;
      }
      remove(datafilepipepath.c_str());
    }
    cout << "\rCURRENT␣VALUE:␣" << currentVal;  //keep same values on one line
  }
}
```

B.1.10 MCUout.cpp

```
#include "TCBmodule.hpp"

using namespace std;

int main (int argc, char** argv){

  string msg;
  char destID, srcID;

  if (argc != 2){
    cerr << "Usage:␣./tcbm␣[source␣CAN␣ID]";
    exit(1);
  }
  srcID = argv[1][0];
  TCBmodule TCBm(srcID);

  if (srcID == 34){
    cout << "OPERATING␣IN␣SSM␣MODE\nTYPE␣\"^q\"␣(no␣quotes)␣TO␣QUIT\n";
    for(;;){
      cin >> msg;
      if (msg.compare("^q") == 0) break;
    }
    TCBm.stopAware();
    TCBm.stopStack();
    TCBm.stopInputLoop(); //ib is never killed unless this is here
  }

  else{
    while (msg.compare("^q") != 0){
      for(;;){
        if (msg.compare("^q") == 0) break;
        cout << "Enter␣destination␣CAN␣ID␣in␣ASCII␣form.\n";
        cin >> destID;
        if (TCBm.checkConn(destID)){
```

```
            cout << "Enter message to send. ('^q' to quit)\n";
            cin >> msg;
            if (msg.compare("^q") == 0) break;

            //send off to TCBmodule
            TCBm.msgFromAL(destID, msg);
          }
        }
      }
    }
    //close all open connections

    TCBm.endAllConnections();

    TCBm.stopInputLoop();
    TCBm.stopMCUin();

    return 0;
}
```

B.1.11 SSMstack.hpp

```
#ifndef __SSMSTACKHPP
#define __SSMSTACKHPP

#include <cstring>
#include <ctime>
#include <fstream>
#include <iostream>
#include <iterator>
#include <string>
#include <sstream>
#include <vector>

#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

#define N 200 //how large should the stack be?
#define SSMID "34"

struct entry{
  time_t time;
  char to, from;
  std::string data;
};

#endif
```

B.1.12 SSMstack.cpp

```
#include "SSMstack.hpp"

using namespace std;

int main(int argc, char** argv){

  char hackFrom;
  struct stat buf;
  vector<entry> stack;
  string tmp, datafilepipepath = "/tmp/newstackentry"+string(SSMID),
    errfilepipepath = "/tmp/hack"+string(SSMID), help = "/tmp/help",
    forSSM = "/tmp/forSSM"+string(SSMID);
  stringstream ss;
  int hackTime;

  cerr << "SSM stack application\n\n";

  for(;;){
    //read in new stack entry from file
    if (stat(datafilepipepath.c_str(), &buf) != -1){
      ifstream in(datafilepipepath.c_str(), ios::in | ios::binary);
```

```
      string token;

      for (string line; getline(in, line);){
        tmp.append(line);
        if (in.peek() != ' ' && in.eof()) break;
        else tmp.append("\n");
      }
      ss << tmp;
      tmp.clear();
      vector<string> tokens;
      while (getline(ss, token, ' ')){
        tokens.push_back(token);
      }
      if (!tokens.empty()){
        entry e;
        e.time = (time_t) strtol(tokens[0].c_str(), NULL, 10);
        e.to = (char) strtol(tokens[1].c_str(), NULL, 10);
        e.from = (char) strtol(tokens[2].c_str(), NULL, 10);
        if (tokens[3].size() < 16 && tokens.size() > 4){      //in case a space was a string char, which was treated as a delimiter
          for (int i = 4; i < tokens.size(); ++i){
            tokens[3].append(" ");
            tokens[3].append(tokens[i]);
          }
        }
        e.data = tokens[3];
        stack.push_back(e);
        cerr << "NOT: Stack has latest " << stack.size() << " entries\n";
        remove(datafilepipepath.c_str());
      }
    }

    //delete oldest stack size entry when its size is greater than N
    if (stack.size() > N){
      stack.erase(stack.begin());
      cerr << "NOT: Removed oldest data entry from stack\n";
    }

    tmp.clear();
    ss.str(string(""));
    ss.clear();

    //check for notification of hack, and if there is one, read it in
    if (stat(errfilepipepath.c_str(), &buf) != -1){
      cerr << "here\n";
      ifstream in(errfilepipepath.c_str(), ios::in);
      string token;

      getline(in, tmp);
      ss << tmp;
      vector<string> tokens;
      while (getline(ss, token, ' ')) tokens.push_back(token);
      if (!tokens.empty()){
        hackFrom = tokens[0][0];
        hackTime = strtol(tokens[1].c_str(), NULL, 10);
        for (int i = stack.size()-1; i > -1; --i){
          cerr << i << " " << stack[i].time << " " << hackTime << endl;
          if (stack[i].time > hackTime){
            if (hackFrom == stack[i].from){
              cerr << "NOT: Hacked data entry found, removing\n";
              stack.erase(stack.begin()+i);
            }
          }
          else if (hackFrom == stack[i].from){
            ofstream ofs(help.c_str(), ios::out);
            ofs << stack[i].to << stack[i].from << " " << stack[i].data;
            ofs.close();
            break;
          }
        }
        remove(errfilepipepath.c_str());
      }
    }
```

```
      tmp.clear();
      ss.str(string(""));
      ss.clear();
    }
}
```

B.1.13 SSMaware.cpp

```
#include "Aware.hpp"

using namespace std;

int main(int argc, char** argv){

  struct stat buf;
  string datafilepipepath = "/tmp/newawareentry"+string(SSMID), makeentry = "/tmp/
      makeawareentry"+string(SSMID), deleteentry = "/tmp/delaware"+string(SSMID);
  vector<shared_ptr<Aware>> dataSources;

  cerr << "SSM awareness algorithm\n\n";

  for(;;){
    if (stat(makeentry.c_str(), &buf) != -1){
      ifstream in(makeentry.c_str(), ios::in);
      string tmp;

      getline(in, tmp);
      dataSources.push_back(make_shared<Aware>(tmp[0], tmp[1]));
      remove(makeentry.c_str());
    }
    if (stat(datafilepipepath.c_str(), &buf) != -1){
      ifstream in(datafilepipepath.c_str(), ios::in);
      stringstream ss;
      vector<string> tokens;
      string tmp, token;
      char destCANID, srcCANID;
      int entryN = -1;
      float factor;

      getline(in, tmp);
      ss << tmp;
      while (getline(ss, token, ' ')){
        tokens.push_back(token);
      }
      if (!tokens.empty()){      //if tokens is empty, program will hang on the next
          line!
        destCANID = tokens[0][0];
        srcCANID = tokens[0][1];
        factor = stof(tokens[1].c_str(), NULL);
        for (int i = 0; i < dataSources.size(); ++i){
          if (dataSources[i]->sameToID(destCANID) && dataSources[i]->sameFromID(
              srcCANID)){
            entryN = i;
            break;
          }
        }
        dataSources[entryN]->getSets(factor);
        remove(datafilepipepath.c_str());
      }
    }
    if (stat(deleteentry.c_str(), &buf) != -1){
      ifstream in(deleteentry.c_str(), ios::in);
      string tmp;

      getline(in, tmp);
      for (int i = 0; i < dataSources.size(); ++i){
        if (dataSources[i]->sameToID(tmp[0]) && dataSources[i]->sameFromID(tmp[1])){
          cerr << "removing terminated connection\n";
          dataSources.erase(dataSources.begin()+i);
          break;
        }
      }
      remove(deleteentry.c_str());
```

```
    }
  }

}
```

B.1.14 Aware.hpp

```
#ifndef __SSMAWAREHPP
#define __SSMAWAREHPP

#include <ctime>
#include <cstdlib>
#include <cstring>
#include <fstream>
#include <iostream>
#include <iterator>
#include <memory>
#include <string>
#include <sstream>
#include <vector>

#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

#define N 5 //this is a
#define SSMID "34"

class Aware{
private:
  char toID, fromID;
  float data1[N], data2[N], data3[N];
  std::time_t start1, start2, start3; //when was data set "created?"
  std::string errfilepipepath = "/tmp/hack" + std::string(SSMID);

public:
  Aware(char t, char f);
  ~Aware(){ std::cerr << "end of connection entry for " << toID << " from " << fromID
      << std::endl; }
  void calcAvgs();
  void decide(float avg1, float avg2, float avg3);
  void getSets(float newFactor);
  bool sameToID(char c) const{ return c == toID; }
  bool sameFromID(char c) const{ return c == fromID; }
};

#endif
```

B.1.15 Aware.cpp

```
#include "Aware.hpp"

using namespace std;

Aware::Aware(char t, char f){
  toID = t;
  fromID = f;
  cerr << "to and from: " << int(toID) << " " << int(fromID) << endl;
  for (int i = 0; i < N; ++i){
    data1[i] = -1;
    data2[i] = -1;
    data3[i] = -1;
  }
}

void Aware::getSets(float newFactor){
  bool done = false;

  if (data1[0] == -1) start1 = time(NULL);
  for (int i = 0; i < N; ++i){
    if (data1[i] == -1){
      data1[i] = newFactor;
```

```
      cout << "data1 " << i << endl;
      done = true;
      break;
    }
  }

  if (data2[0] == -1 && !done) start2 = time(NULL);
  if (!done){
    for (int j = 0; j < N; ++j){
      if (data2[j] == -1){
        data2[j] = newFactor;
        cout << "data2 " << j << endl;
        done = true;
        break;
      }
    }
  }

  if (data3[0] == -1 && !done) start3 = time(NULL);
  if (!done){
    for (int k = 0; k < N; ++k){
      if (data3[k] == -1){
        data3[k] = newFactor;
        cout << "data3 " << k << endl;
        done = true;
        if (k == N-1){
          calcAvgs();
          memcpy(data1, data2, N*sizeof(float));
          memcpy(data2, data3, N*sizeof(float));
          for (int q = 0; q < N; ++q) data3[q] = -1;
          start1 = start2;
          start2 = start3;
          start3 = 0;
        }
        return;
      }
    }
  }
}

void Aware::calcAvgs(){
  cerr << "CALCAVGS\n";
  float d1avg = 0, d2avg = 0, d3avg = 0;

  for (int j = 0; j < N; ++j){
    d1avg += data1[j];
    d2avg += data2[j];
    d3avg += data3[j];
  }

  d1avg /= N;
  d2avg /= N;
  d3avg /= N;

  decide(d1avg, d2avg, d3avg);
}

void Aware::decide(float avg1, float avg2, float avg3){
  cerr << avg1 << " " << avg2 << " " << avg3 << endl;
  cerr << "verdict: ";

  if (avg1 == avg2 && avg1 == avg3){
    //no change in values
    cerr << "constant\n";
  }

  else if (abs(avg1-avg2) > 0.1 || abs(avg2-avg3) > 0.1){
    //hack
    cerr << "HACK\n";
    ofstream ofs("/tmp/hack"+string(SSMID), ios::out);
    ofs << fromID << " " << start2;
  }

```

```
  else if (avg1 != 1 && avg2 != 1 && avg3 != 1){
    //standard increase/decrease
    cerr << "gradual change\n";
  }

  else if (avg1 == 1 && (abs(avg1-avg2) < 0.1 || abs(avg2-avg3) < 0.1) && avg3 == 1){
    //blip
    cerr << "blip\n";
  }
  else {
    //transitionary behavior between data sets
    cerr << "N/A\n";
  }
}
```

B.1.16 keyGen.cpp

```
#include <cryptopp/osrng.h>
#include <cryptopp/cryptlib.h>
#include <cryptopp/hex.h>
#include <cryptopp/filters.h>
#include <cryptopp/aes.h>
#include <cryptopp/modes.h>

#include <iostream>
#include <string>
#include <cstdlib>
#include <fstream>

using namespace std;
using CryptoPP::AES;
using CryptoPP::AutoSeededRandomPool;
using CryptoPP::ECB_Mode;
using CryptoPP::Exception;
using CryptoPP::HexEncoder;
using CryptoPP::HexDecoder;
using CryptoPP::StringSink;
using CryptoPP::StringSource;

int main(int arcg, char** argv){
  AutoSeededRandomPool prng;
  byte key[AES::MAX_KEYLENGTH];
  string keyStr;
  ofstream ofs("my.key", ios::out);

  prng.GenerateBlock(key, sizeof(key));

  StringSource(key, sizeof(key), true, new HexEncoder(new StringSink(keyStr)));

  cout << "new AES-256 key: " << keyStr << endl;

  if (ofs.is_open()){
    ofs << key;
    ofs.close();
  }
  else cerr << "ERR: key could not be written to file\n";

  return 0;
}
```

B.2 Security testing - Message injection

B.2.1 injector.cpp

```
#include <linux/can.h>
#include <linux/can/raw.h>

#include <endian.h>
#include <net/if.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <sys/types.h>
```

```
#include <unistd.h>
#include <fcntl.h>

#include <cerrno>
#include <chrono>
#include <csignal>
#include <cstdint>
#include <cstdio>
#include <cstring>
#include <ctime>
#include <string>
#include <thread>
#include <iostream>
#include <sstream>

using namespace std;
typedef std::chrono::high_resolution_clock hrc;
typedef std::chrono::hours hrs;

int main (int argc, char** argv){

  const char *intrf = "vcan0";
  int rc, sockfd, opt, enable = 1;
  struct sockaddr_can canaddr;
  struct ifreq ifr;
  string output;
  hrc::time_point start;

  srand(time(NULL)); //keep our initial seed as random as possible

  sockfd = socket(PF_CAN, SOCK_RAW, CAN_RAW);
  if (sockfd == -1){
    cerr << "Can't open socket, errno: " << errno << endl;
    return 1;
  }

  rc = setsockopt(sockfd, SOL_CAN_RAW, CAN_RAW_FD_FRAMES, &enable, sizeof(enable));
  if (rc == -1){
    cerr << "Can't set socket options\n";
    return 1;
  }

  std::strncpy(ifr.ifr_name, intrf, IFNAMSIZ);
  if (ioctl(sockfd, SIOCGIFINDEX, &ifr) == -1){
    cerr << "Can't interact with network interface, errno " << errno << endl;
    return 1;
  }

  canaddr.can_family = AF_CAN;
  canaddr.can_ifindex = ifr.ifr_ifindex;
  fcntl(sockfd, F_SETFL, O_NONBLOCK);
  rc = bind(sockfd, (struct sockaddr *)&canaddr, sizeof(canaddr));

  if (rc == -1){
    cerr << "Can't bind socket\n";
    return 1;
  }

  cout << "Established injector - let's inject!\n\n";
  start = hrc::now();

  for(;;){
    struct canfd_frame frame;
    int i = rand() % 63;
    frame.can_id = rand() % 100 + 1;  //any int possible, but let's limit to first 100
    frame.len = i;
    for (int i = 0; i < frame.len; ++i) frame.data[i] = rand() % 127;
    int bytes = write(sockfd, &frame, sizeof(struct canfd_frame));
    this_thread::sleep_for(std::chrono::milliseconds(rand() % 1000));
    //return when test time is completed
    if (chrono::duration_cast<hrs>(hrc::now() - start).count() > 0) return 0;
  }

```

```
}
```

B.2.2 Makefile

```
CXX = g++
CFLAGS = --std=c++11 -O2 -g

ifeq ($(shell uname),Linux)
all: injector receiver

injector: injector.o
	$(CXX) $(CFLAGS) -o injector injector.o
receiver: receiver.o
	$(CXX) $(CFLAGS) -o receiver receiver.o

injector.o: injector.cpp
	$(CXX) -c injector.cpp
receiver.o: receiver.cpp
	$(CXX) -c receiver.cpp

else
all:
	@echo "ERR: need Linux to compile this project."
endif

.PHONY: clean

clean:
	rm -f *.o injector receiver
```

B.3 Security testing - Fuzzing

B.3.1 fuzzer.cpp

```cpp
#include <linux/can.h>
#include <linux/can/raw.h>

#include <endian.h>
#include <net/if.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>

#include <chrono>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <thread>
#include <bitset>

#include <cerrno>
#include <csignal>
#include <cstdint>
#include <cstdio>
#include <string>
#include <sstream>
#include <cstring>

using namespace std;

int main (int argc, char** argv){

  const char  *intrf = "vcan0";
  int rc, CANID, sockfd, opt, enable = 1;
  struct sockaddr_can canaddr;
  struct ifreq ifr;
  ifstream ifile("inputs.txt");
  string input;

  if (argc != 2){
```

```
      cerr << "Usage: ./fuzzer [target_CAN_ID]\n";
      return 1;
    }
    else CANID = strtol(argv[argc-1], NULL, 16);

    sockfd = socket(PF_CAN, SOCK_RAW, CAN_RAW);
    if (sockfd == -1){
      cerr << "Can't open socket, errno: " << errno << endl;
      return 1;
    }

    rc = setsockopt(sockfd, SOL_CAN_RAW, CAN_RAW_FD_FRAMES, &enable, sizeof(enable));
    if (rc == -1){
      cerr << "Can't set socket options\n";
      return 1;
    }

    std::strncpy(ifr.ifr_name, intrf, IFNAMSIZ);
    if (ioctl(sockfd, SIOCGIFINDEX, &ifr) == -1){
      cerr << "Can't interact with network interface, errno " << errno << endl;
      return 1;
    }

    canaddr.can_family = AF_CAN;
    canaddr.can_ifindex = ifr.ifr_ifindex;
    fcntl(sockfd, F_SETFL, O_NONBLOCK);
    rc = bind(sockfd, (struct sockaddr *)&canaddr, sizeof(canaddr));

    if (rc == -1){
      cerr << "Can't bind socket\n";
      return 1;
    }


    cout << "Established fuzzer - let's fuzz!\n\n";

    while(getline(ifile,input)){
      struct canfd_frame frame;
      frame.can_id = CANID;
      //cout << input << endl;
      frame.len = input.length();
      for (int i = 0; i < frame.len; ++i) frame.data[i] = (int)input[i];

      int bytes = write(sockfd, &frame, sizeof(struct canfd_frame));
      input = "";
    }

}
```

B.3.2 randGen.cpp

```
#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <string>
#include <fstream>

using namespace std;

int main(int argc, char** argv) {

  string str;
  int option, offset, mod;
  ofstream ofs("inputs.txt", ios::out);

  if (argc != 2){
      cerr << "Usage: ./randGen [0 for all ASCII, 1 for readable ASCII only]\n";
      return 1;
  }
  else option = strtol(argv[1], NULL, 10);

    if (option == 0){
      offset = 0;
```

```
        mod = 127;
        cerr << "Printing 10000 strings using all ASCII characters\n";
      }
    else if (option == 1){
        offset = 32;
        mod = 95;
        cerr << "Printing 10000 strings using only readable ASCII characters\n";
      }

      else{
        cerr << "Usage: ./randGen [0 for all ASCII, 1 for readable ASCII only]\n";
        return 1;
    }

    srand (time(NULL)); //not truly random, but we're trying to BREAK the system, not BE it

    for (int i = 0; i < 10000; ++i){
        int j = rand() % 63;  //for random string lengths between 0 and 64 bytes long
        for (; j < 64; ++j){
          unsigned char c = rand() % mod + offset;
          if (offset == 1 && c == 127) c = 126; //because DEL is at the end of the range
          str += c;
        }
        ofs << str << endl;
        str.clear();
    }
    return 0;
}
```

B.3.3 Makefile

```
CXX = g++
CFLAGS = --std=c++11 -O2 -g

ifeq ($(shell uname),Linux)
all: fuzzer randGen receiver

fuzzer: fuzzer.o
	$(CXX) $(CFLAGS) -o fuzzer fuzzer.o
randGen: randGen.o
	$(CXX) $(CFLAGS) -o randGen randGen.o
receiver: receiver.o
	$(CXX) $(CFLAGS) -o receiver receiver.o

fuzzer.o: fuzzer.cpp
	$(CXX) -c fuzzer.cpp
randGen.o: randGen.cpp
	$(CXX) -c randGen.cpp
receiver.o: receiver.cpp
	$(CXX) -c receiver.cpp

else
all:
	@echo "ERR: need Linux to compile this project."
endif

.PHONY: clean

clean:
	rm -f *.o fuzzer randGen receiver
```

B.4 Security testing - MitM

B.4.1 mitm.cpp

```
#include <linux/can.h>
#include <linux/can/raw.h>

#include <endian.h>
#include <net/if.h>
#include <sys/ioctl.h>
```

```cpp
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>

#include <chrono>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <thread>
#include <bitset>
#include <vector>

#include <cerrno>
#include <csignal>
#include <cstdint>
#include <cstdio>
#include <cstdlib>
#include <string>
#include <sstream>
#include <cstring>

using namespace std;


//Name: fixChecksum
//Description: Recalculate checksum so that it matches the rest
//         of the string.
//Output: Entire string output with the fixed checksum
//------------------------------------------------------------------------
string fixChecksum(string in){
  string msg = in, tmp, check1, check2;
  unsigned int total, t = 0, q = 0;
  vector<int> num;

  cerr << "Altering the checksum, too\n";

  //clear checksum in frame
  msg[10] = (char) 0;
  msg[11] = (char) 0;

  for (int j = 0; j < msg.length(); ++j){
    tmp.clear();
    for (int i = 7; i >= 0; --i) tmp += ((msg[j] & (1 << i))? '1' : '0');
    ++j;
        if (j+1 == msg.length()){
          tmp.append("00000000");
          break;
        }
        for (int i = 7; i >= 0; --i) tmp += ((msg[j] & (1 << i))? '1' : '0');
        q = bitset<16>(tmp).to_ulong();
        num.push_back(q);
  }

    for (int i = 0; i < num.size(); ++i) t += num[i];

    while (t>>16) t = (t & 0xffff) + (t >> 16);

  t = 0xffff - t;

  bitset<16> bits (t);
  tmp.clear();
  tmp = bits.to_string();
  check1 = tmp.substr(0,8);
  check2 = tmp.substr(8,8);
  msg[10] = static_cast<char>(std::stoi(check1, nullptr, 2));
  msg[11] = static_cast<char>(std::stoi(check2, nullptr, 2));
  return msg;
}
//------------------------------------------------------------------------
```

```
//Name: alterString
//Description: Slightly alter the message contents.
//Output: Entire string output.
//------------------------------------------------------------------------------------
string alterString(string in){
  int checksumToo, modMe = rand() % 126;
  string tmp = in;

  cerr << "Altering all chars w/ ASCII ID " << modMe << " to " << modMe+1 << endl;

  for (int i = 0; i < tmp.size(); ++i){
    if (modMe == (int)tmp[i]) ++tmp[i];
  }

  checksumToo = rand() % 2;   //one in two chance of adjusting checksum
  if (checksumToo) return fixChecksum(tmp);
  else return tmp;
}
//------------------------------------------------------------------------------------
//Name: main
//Description: where the fun happens
//Output: N/A
//------------------------------------------------------------------------------------
int main (int argc, char** argv){

  const char  *intrf1 = "vcan0", *intrf2 = "vcan1";
  int rc1, rc2, CANID1, CANID2, sockfd1, sockfd2, opt, enable = 1;
  struct sockaddr_can canaddr1, canaddr2;
  struct ifreq ifr1, ifr2;
  ifstream ifile("inputs.txt");
  string input;

  if (argc != 3){
    cerr << "Usage: ./fuzzer [target_CAN_ID, network 1] [target_CAN_ID, network 2]\n";
    return 1;
  }
  CANID1 = strtol(argv[1], NULL, 16);
  CANID2 = strtol(argv[2], NULL, 16);

  srand(time(NULL));

  sockfd1 = socket(PF_CAN, SOCK_RAW, CAN_RAW);
  if (sockfd1 == -1){
    cerr << "Can't open socket, errno: " << errno << endl;
    return 1;
  }

  rc1 = setsockopt(sockfd1, SOL_CAN_RAW, CAN_RAW_FD_FRAMES, &enable, sizeof(enable));
  if (rc1 == -1){
    cerr << "Can't set socket options\n";
    return 1;
  }

  std::strncpy(ifr1.ifr_name, intrf1, IFNAMSIZ);
  if (ioctl(sockfd1, SIOCGIFINDEX, &ifr1) == -1){
    cerr << "Can't interact with network interface, errno " << errno << endl;
    return 1;
  }

  canaddr1.can_family = AF_CAN;
  canaddr1.can_ifindex = ifr1.ifr_ifindex;
  fcntl(sockfd1, F_SETFL, O_NONBLOCK);
  rc1 = bind(sockfd1, (struct sockaddr *)&canaddr1, sizeof(canaddr1));

  if (rc1 == -1){
    cerr << "Can't bind socket\n";
```

```
    return 1;
  }

  //now vcan1
  sockfd2 = socket(PF_CAN, SOCK_RAW, CAN_RAW);
  if (sockfd2 == -1){
    cerr << "Can't open socket, errno: " << errno << endl;
    return 1;
  }

  rc2 = setsockopt(sockfd2, SOL_CAN_RAW, CAN_RAW_FD_FRAMES, &enable, sizeof(enable));
  if (rc2 == -1){
    cerr << "Can't set socket options\n";
    return 1;
  }

  std::strncpy(ifr2.ifr_name, intrf2, IFNAMSIZ);
  if (ioctl(sockfd2, SIOCGIFINDEX, &ifr2) == -1){
    cerr << "Can't interact with network interface, errno " << errno << endl;
    return 1;
  }

  canaddr2.can_family = AF_CAN;
  canaddr2.can_ifindex = ifr2.ifr_ifindex;
  fcntl(sockfd2, F_SETFL, O_NONBLOCK);
  rc2 = bind(sockfd2, (struct sockaddr *)&canaddr2, sizeof(canaddr2));

  if (rc2 == -1){
    cerr << "Can't bind socket\n";
    return 1;
  }

  cout << "Established MitM - let's, uh, you know\n\n";

  for (;;){
    //listen on vcan0 for CANID2
    struct canfd_frame fr;
    int passThrough, alter;
    int from1 = read(sockfd1, &fr, CANFD_MTU);
    if (from1 > 8 && fr.can_id == (int) CANID2){
      passThrough = rand() % 5;
      if (passThrough){      //one in five shot of no dropping
        alter = rand() % 3;   //one in three (in five) shot of altering
        if (!alter){
          string tmp, str = "";
          for (int q = 0; q < fr.len; ++q) str += (char)fr.data[q];
          tmp = alterString(str);
          for (int q = 0; q < fr.len; ++q) fr.data[q] = (int)tmp[q];
        }
        int to2 = write(sockfd2, &fr, sizeof(struct canfd_frame));
        if (to2 > 0) cerr << "Transferred packet to vcan1 at " << time(NULL) << endl;
      }
      else cerr << "Packet to vcan1 was dropped at " << time(NULL) << endl;
    }
    fr.can_id = 0;
    fr.len = 0;
    for (int i = 0; i < 64; ++i) fr.data[i] = '\0';

    //listen on vcan1 for CANID1
    int from2 = read(sockfd2, &fr, CANFD_MTU);
    if (from2 > 8 && fr.can_id == (int) CANID1){
      passThrough = rand() % 10;
      if (passThrough){
        int to1 = write(sockfd1, &fr, sizeof(struct canfd_frame));
        if (to1 > 0) cerr << "Transferred packet to vcan0 at " << time(NULL) << endl;
      }
      else cerr << "Packet to vcan0 was dropped at " << time(NULL) << endl;
    }
    fr.can_id = 0;
    fr.len = 0;
    for (int i = 0; i < 64; ++i) fr.data[i] = '\0';
  }

```

```
}
```

B.4.2 Makefile

```
CXX = g++
CFLAGS = --std=c++11 -O2 -g

ifeq ($(shell uname),Linux)
all: mitm receiver

mitm: mitm.o
	$(CXX) $(CFLAGS) -o mitm mitm.o
receiver: receiver.o
	$(CXX) $(CFLAGS) -o receiver receiver.o

mitm.o: mitm.cpp
	$(CXX) -c mitm.cpp
receiver.o: receiver.cpp
	$(CXX) -c receiver.cpp

else
all:
	@echo "ERR: need Linux to compile this project."
endif

.PHONY: clean

clean:
	rm -f *.o mitm receiver
```

B.4.3 unprotSender.sh

```
#/!bin/bash
# NOTE: Start this application with "bash" instead of "sh", or the for loop will not work!!!

# write out our target in string form, because it's easier to parse
CANTARGETHEX="050"
TARGETNET="vcan0"

for i in {1..100}
do
  # [pseudo] randomly generate string length
  NUMBER=$(shuf -i 1-64 -n 1)
  [ $((NUMBER % 2)) -eq 1 ] && NUMBER=$(expr $NUMBER + 1)
  # randomly generate a string
  STR=$(cat /dev/urandom | tr -dc 'a-f0-9' | fold -w $NUMBER | head -n 1)
  # send our string out to our destination
  cansend $TARGETNET $CANTARGETHEX##3$STR
  # regain some entropy for /dev/urandom
  sleep 1
done

echo Test completed
```

B.5 Miscellaneous

B.5.1 Main Makefile to compile the source code (with tests)

```
CXX = g++
CFLAGS = --std=c++11 -O2 -g -pthread
LIBS = -lcryptopp -lpthread
SUBDIRS = fuzzTest injectionTest mitmTest

ifeq ($(shell uname),Linux)
all: aware ib keyGen MCUin ob stack tcbm

aware: Aware.o SSMaware.o
	$(CXX) $(CFLAGS) -o aware Aware.o SSMaware.o
ib: inputbuffer.o
	$(CXX) $(CFLAGS) -o ib inputbuffer.o
```

```
keyGen: keyGen.o
	$(CXX) $(CFLAGS) -o keyGen keyGen.o $(LIBS)
MCUin: MCUin.o
	$(CXX) $(CFLAGS) -o MCUin MCUin.o
ob: outputbuffer.o
	$(CXX) $(CFLAGS) -o ob outputbuffer.o
stack: SSMstack.o
	$(CXX) $(CFLAGS) -o stack SSMstack.o
tcbm: MCUout.o TCBmodule.o TCB.o Timer.o
	$(CXX) $(CFLAGS) -o tcbm MCUout.o TCBmodule.o TCB.o Timer.o $(LIBS)

Aware.o: Aware.cpp Aware.hpp
	$(CXX) -c Aware.cpp
inputbuffer.o: inputbuffer.cpp
	$(CXX) -c inputbuffer.cpp
keyGen.o: keyGen.cpp
	$(CXX) -c keyGen.cpp
MCUin.o: MCUin.cpp
	$(CXX) -c MCUin.cpp
MCUout.o: MCUout.cpp TCBmodule.hpp TCB.hpp Timer.hpp
	$(CXX) -c MCUout.cpp
outputbuffer.o: outputbuffer.cpp
	$(CXX) -c outputbuffer.cpp
SSMaware.o: SSMaware.cpp Aware.hpp
	$(CXX) -c SSMaware.cpp
SSMstack.o: SSMstack.cpp SSMstack.hpp
	$(CXX) -c SSMstack.cpp
TCB.o: TCB.cpp TCB.hpp Timer.hpp
	$(CXX) -c TCB.cpp
TCBmodule.o: TCBmodule.cpp TCBmodule.hpp TCB.hpp Timer.hpp
	$(CXX) -c TCBmodule.cpp
Timer.o: Timer.cpp Timer.hpp
	$(CXX) -c Timer.cpp

else
all:
	@echo "ERR: need Linux to compile this project."
endif

.PHONY: tests clean

tests:
	for dir in $(SUBDIRS); do \
	  $(MAKE) -C $$dir; \
	done

clean:
	rm -f *.o aware ib keyGen MCUin ob stack tcbm /tmp/ibToTCB* /tmp/TCBToOb* /tmp/TCBtoAL* /tmp/newawareentry34 /tmp/newstackentry34 /tmp/makeawareentry34 /tmp/delaware34; \
	for dir in $(SUBDIRS); do \
	  $(MAKE) -C $$dir -f Makefile $@; \
	done
```

B.5.2 Bash script to initialize a single CAN network in a virtual environment

```
#!bin/bash/

sudo modprobe can
sudo modprobe can_raw
sudo modprobe vcan
sudo ip link add dev vcan0 type vcan
sudo ip link set vcan0 mtu 72
sudo ip link set up vcan0
ip link show vcan0
```

B.5.3 Bash script to initialize multiple CAN networks in a virtual environment

```
#!bin/bash/

sudo modprobe can
sudo modprobe can_raw
sudo modprobe vcan
sudo ip link add dev vcan0 type vcan
sudo ip link add dev vcan1 type vcan
sudo ip link set vcan0 mtu 72
sudo ip link set vcan1 mtu 72
sudo ip link set up vcan0
sudo ip link set up vcan1
ip link show vcan0
ip link show vcan1
```

B.5.4 Victim logic for MCU without system model protections

```
#include <linux/can.h>
#include <linux/can/raw.h>

#include <endian.h>
#include <net/if.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>

#include <chrono>
#include <iomanip>
#include <iostream>
#include <thread>
#include <bitset>

#include <cerrno>
#include <climits>
#include <csignal>
#include <cstdint>
#include <cstdio>
#include <string>
#include <cstring>

using namespace std;

int main(int argc, char** argv) {

  sig_atomic_t interrupt = 0;
  const char *intrf = "vcan0";
  int rc, CANID, sockfd, opt, enable = 1, count = 0;
  struct sockaddr_can canaddr;
  struct ifreq ifr;
  string statusStr = "test string";

  if (argc != 2){
    cerr << "Usage: ./receiver [CAN_ID]\n";
    return 1;
  }
  else CANID = strtol(argv[argc-1], NULL, 16);

  sockfd = socket(PF_CAN, SOCK_RAW, CAN_RAW);
  if (sockfd == -1){
    cerr << "Can't open socket\n";
    return 1;
  }

  rc = setsockopt(sockfd, SOL_CAN_RAW, CAN_RAW_FD_FRAMES, &enable, sizeof(enable));
  if (rc == -1){
```

```
    cerr << "Can't set socket options\n";
    return 1;
  }

  std::strncpy(ifr.ifr_name, intrf, IFNAMSIZ);
  if (ioctl(sockfd, SIOCGIFINDEX, &ifr) == -1){
    cerr << "Can't interact with network interface, errno " << errno << endl;
    return 1;
  }

  canaddr.can_family = AF_CAN;
  canaddr.can_ifindex = ifr.ifr_ifindex;
  fcntl(sockfd, F_SETFL, O_NONBLOCK);
  rc = bind(sockfd, (struct sockaddr *)&canaddr, sizeof(canaddr));
  if (rc == -1){
    cerr << "Can't bind socket\n";
    return 1;
  }

  cout << "Established receiver for unprotected test\n\n";
  cout << "Value: " << statusStr << endl;

  for(;;){
    struct canfd_frame fr;
    int bytes = read(sockfd, &fr, CANFD_MTU);
    if (bytes > 0 && fr.can_id == CANID){
      stringstream ss("");
        for (int temp = 12; temp < fr.len; temp++) ss << fr.data[temp];
      statusStr = ss.str().c_str();
      cout << "Value: " << statusStr << endl;
      ss.str(string());
      ss.clear();
      ++count;
      cout << "Number of times changed: " << count << endl;
    }
  }

  return 0;

}
```