BRADLEY University

KOMATSU

ECE 499 – Senior Project Report
Komatsu Sponsored – Performance of an ECU CAN Network With and Without On-The-Fly Data Encryption

Team Members: Grant Abella, John Greifzu
Advisor: Aleksander Malinowski

Department of Electrical and Computer Engineering

May 12, 2019

Acknowledgments

We would like to formally thank several individual who made this project possible. We would like to acknowledge all of the Bradley Electrical Engineering professors for instilling in us the knowledge required to complete a project of this scale. Specifically, we wish to extend our gratitude to Dr. Aleksander Malinowski. Not only did he guide us through this entire project with his suggestions and advice, but he was always available to field any questions we had. In addition to this, Professor Malinowski assisted us with the design of many deliverables including this report and the project website. Next, we would like to thank Dr. Lu for coordinating many of the background tasks that enabled the Electrical Engineering Senior Project program to run so smoothly. Dr. Lu also accompanied Dr. Malinowski and us to all of the meetings we had with Komatsu engineers. Finally, we would like to thank Komatsu America Corporation for providing us with this valuable opportunity. The knowledge and experience that we gained through the completion of this project will accompany us in our future endeavors. Specifically, we would like to thank Komatsu engineers Joshua Rohman, Jason Schepler, Paul Maynard, and Tim Imig for meeting with us throughout the school year. With their assistance and guidance we were able to complete all of the goals set for this project.

Abstract

This project implements an interconnection of electronic control units (ECUs) through a controller area network (CAN) bus. The goal of this project is to incorporate encryption and measure its impact on the system hardware along with the increase in communication latency. The data transmission needs to be encrypted in order to ensure that Komatsu's proprietary software is secure. The Blowfish encryption algorithm is used for testing, and the performance metrics of data packet round-trip time and CPU usage are recorded. In addition to CAN, Ethernet performance is also investigated. Ethernet technology offers both increased transmission speed and cost reduction. Like in the case of CAN bus analysis, the performance of Ethernet with and without encryption is tested and compared.

Table of Contents

I. Introduction

In the automotive industry, ECUs are used to control and monitor electrical systems in vehicles and other electronic equipment. As the complexity of automotive systems increases, the number of required ECUs increases as well. This is because it is not uncommon for a single ECU to be responsible for each important subsystem of the vehicle. For example, a modern car will usually contain separate ECUs dedicated to engine control, airbags, power steering, and more. All of these systems require the ability to communicate with each other, and the CAN communication protocol was created to solve this problem. This provided a standard method for different subsystems to send and receive messages between each other via a connection comprised of two wires.

Although the CAN bus communication was widely adopted after its creation, it carried with it some small disadvantages that have grown as vehicles and machinery have become more and more complex. The main problems with the protocol are centered around speed, reliability, and security.

For some companies in the industry, this has led to the desire for an alternative communication protocol. The Ethernet standard, being newer, faster and more reliable than CAN, seems, for many firms, to be the likely upgrade option. For this reason, some companies are considering upgrading their internal infrastructures from CAN to Ethernet. As with any system upgrade, the impact of new or different hardware on the system must be considered.

Security is another concern for any communication protocol. If the signals sent between ECUs are not secure, the intellectual property on the system is at risk of being stolen, copied, or modified. The solution to this is encryption. Encryption allows the information held within messages to be systematically scrambled before transmission, and unscrambled once the messages are received. This ensures that even if messages are intercepted between ECUs, their contents will be meaningless to whoever has seized them.

Just like with hardware upgrades, changes in system security also bring changes in system performance. Impacts in system resource utilization and message latency should be examined in order to fully understand the effects of implementing security measures like encryption.

II. Problem Statement

Komatsu has requested the extension of a project completed last year. That project simulated an interconnection of ECUs through a CAN bus. The goal of this year's project is to implement encryption on the signals sent between ECUs, and to measure the impact that this has on system utilization and communication latency. Communication over Ethernet with and without encryption will be implemented and measured as well.

The following are the required functionality that will need to be implemented in order to test network performance:

1. Implement communication between two ECUs over CAN bus.
2. Interface with the encryption algorithm to encrypt CAN messages before sending them, and decrypt the messages after they have been received.

3. Implement communication between two ECUs over Ethernet.
4. Interface with the encryption algorithm to encrypt Ethernet packets before sending them, and decrypt the packets after they have been received.
5. Develop a method of obtaining system performance metrics for both plaintext packets and messages, and encrypted packets and messages.
6. Create a method of comparison for all of the cases being tested.

For the ECUs in this project, we will be using two Raspberry Pi 3Bs. As the Raspberry Pi does not have an integrated CAN bus, each Pi will be fitted with a PiCAN2 shield which will allow the ECUs to communicate with each other using the CAN protocol. Two 120 ohm terminating resistors will be require in order for the CAN shield to work properly. Both Pis will need individual power supplies, and for communication over Ethernet, a single Ethernet cable will be connected between the systems. For the encryption and decryption of signals, we will use the Blowfish encryption algorithm. A Linux software solution will need to be used for performance metric gathering.

III. Background

i. CAN Communication

The CAN communication protocol is a bus standard used in vehicles, equipment, and machinery. Communication is achieved by setting the voltages high and low on a pair of twisted wires connected to the bus. Two 120 ohm terminating resistors are required in order to return the wire pair to their nominal differential voltages of 0 V. Because of this interface, the CAN protocol allows devices to communicate without the use of a host computer. The top speed of the protocol is 1 Mbit/s over short distances, with speeds of 250 kbits/s and 125 kbits/s for longer distances. A combination of signals are sent along the wires to make up a full CAN message. The structure of messages consists of an arbitration or identifier field (11 bits), a remote transmission request (1 bit), a data length code (4 bits), and a data field (0 to 64 bits). This means that up to 8 bytes of data can be stored in a single CAN frame.

In addition to this, there are also sections of the message to mark the beginning and end of frames, and a cyclic redundancy check (CRC) field. The purpose of the CRC field is to represent errors that could arise during communication. Possible errors associated with CAN include (but are not limited to) identifier errors, transmission errors, and receiving errors. The microchip embedded in the PiCAN2 controller follows the ISO 11898-2 CAN lower-layer standard.

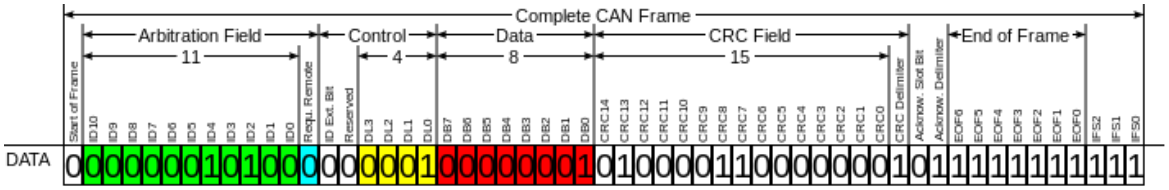

Figure 1 – Structure of a CAN frame.

To give a better understanding of how messages are sent over CAN bus, an example of a partial CAN messages is shown below.

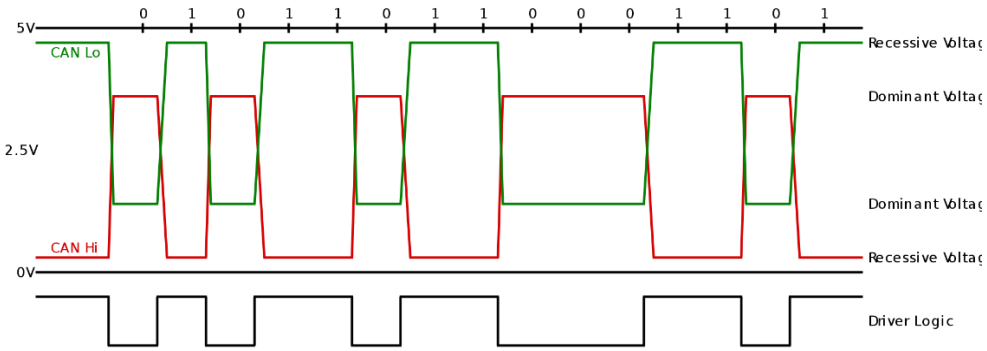

Figure 2 - Low speed CAN signaling.

All of the data contained in CAN messages are made up of individual bits, with each bit holding a value of 0 or 1. These bits are conveyed by the CAN controller through the toggling of voltage levels on the CAN low and CAN high wires. The dominant voltage levels on the low and high wires are 1.25 V and 3.75 V respectively. Conversely, the recessive voltage levels on the low and high wires are 5 V and 0 V respectively. By synchronously setting the low and high wires to their respective dominant and recessive voltage levels, the CAN protocol has the ability to signal either a 0 or a 1 for that specified bit.

This system can be seen in the figure above. The green signal represents CAN low, and the red signal represents CAN high. The black signal labeled “Driver Logic” represents the bit resulting from the CAN low and CAN high voltage levels. Following this logic, points in the figure where high and low CAN signals are both dominant result in a logical value of 0 for that bit, while points in which both signals are recessive result in a logical value of 1.

This process takes place until every bit of the full CAN message is sent.

ii. Ethernet Communication

Ethernet is a computer networking technology often used in local area networks (LAN). Communication is achieved via twisted wires and in some cases, fiber optics. Commonly, speeds of 100 Mbits/s can be achieved using this standard, although this varies depending on the size of the packet (or frame) being sent. Ethernet frames consist of a MAC header, the data, and a checksum.

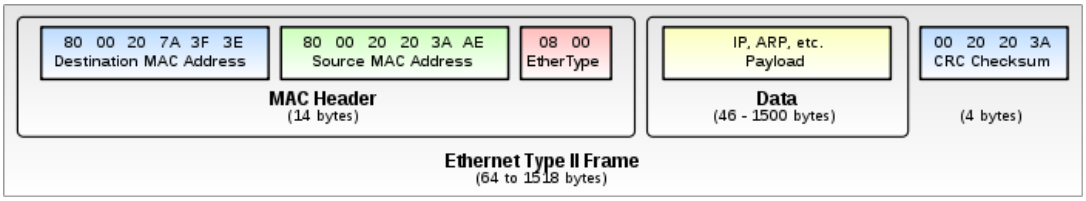

Figure 3 – Structure of an Ethernet frame.

The MAC header is made up of destination and source MAC addresses, and a 16-bit EtherType identifier. The destination MAC address contains the unique address of the device that the packet will be delivered to. The source MAC address is the address of the device that is sending the packet. And the EtherType identifier is used to denote the specific Ethernet protocol being used to send the packet.

Following the MAC header is the section of the frame that carries the data. This portion has a size specification of 46 to 1500 bytes. In cases where the data is less than 46 bytes, padding bytes are appended to the data until the minimum data size is reached.

Finally, a CRC checksum makes up the last 4 bytes of the Ethernet packet. This frame portion, like in CAN, serves as an error-checking precaution.

iii. Encryption

Encryption is the process of encoding messages into a form, known as a ciphertext, which can only be decoded by authorized parties. In computing, the encoding process usually consists of performing a series of XOR and bit-shift operations on the plaintext in combination with a key. The decoding process performs the reverse of the encoding operations on the ciphertext, using the same, or in some cases a different, key. Generally there are two types of encryption schemes, symmetric-key encryption, and public-key (asymmetric-key) encryption.

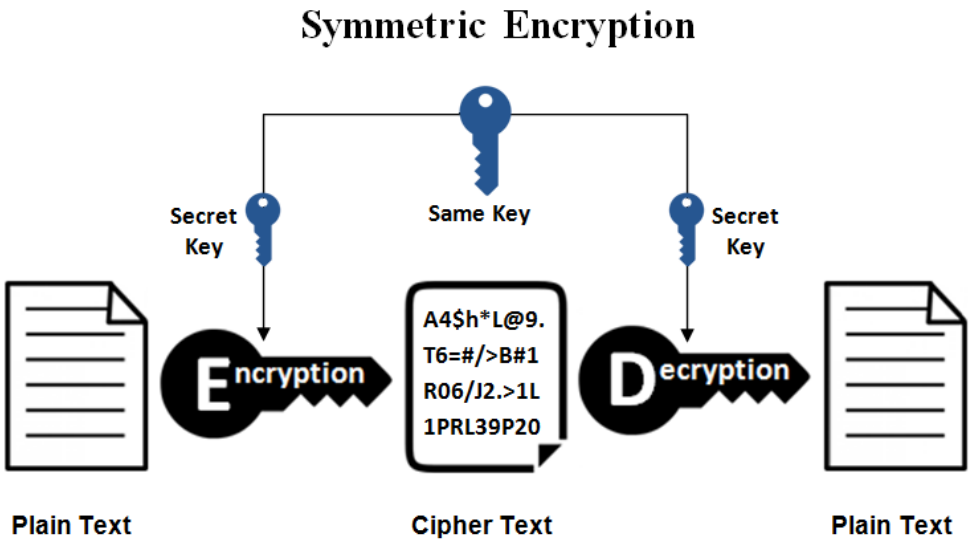

Figure 4 – Representation of symmetric-key encryption.

In symmetric-key encryption, the keys used in the encryption and decryption procedures are the same. This means that the sending and recipient parties must have access to the same key in order for successful encryption and decryption to be achieved. Without the correct key, the decrypted text will be meaningless. In symmetric-key encryption, both parties either possess the key in their program code or in a separate file located on each system, which is then loaded into the program before encryption or decryption can take place. A visual representation of this system is shown above.

In public-key encryption, keys used for encryption are published or accessible for any party to encrypt a plaintext string. However, only authorized parties possess the separate key (or keys) required to decrypt the ciphertext. A visual representation of this system is shown below.

Both encryption schemes have their own advantages and disadvantages, and their applications depend on the situation in which they are being used.

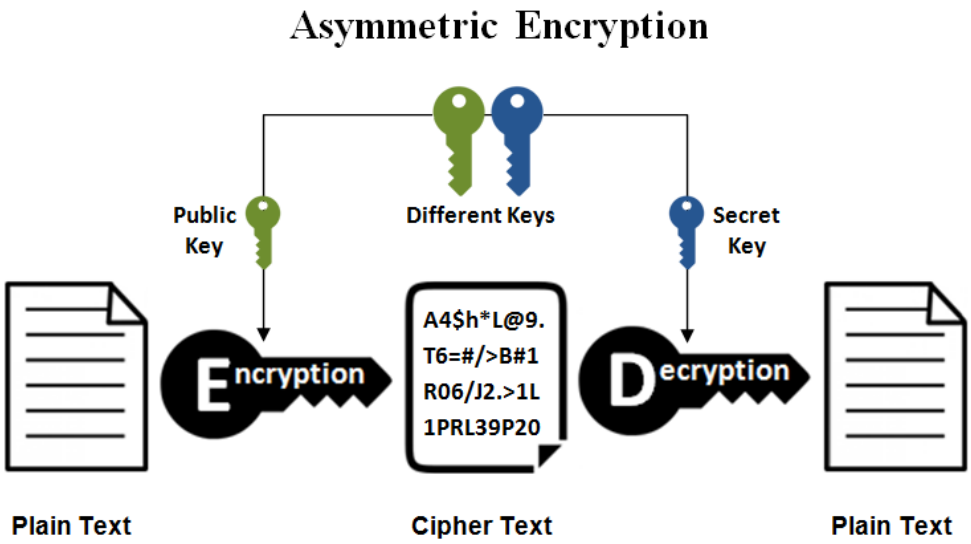

Figure 5 – Representation of public-key encryption.

IV. Implementation

i. Hardware

a. Raspberry Pi 3B

The Raspberry Pi 3B is a small form factor single-board computer designed by the Raspberry Pi Foundation. The Pi utilizes an Advanced RISC Machine (ARM) architecture which is ideal for smaller 32-bit and 64-bit machines.

Figure 6 – Raspberry Pi model 3B.

Significant hardware components utilized in this project include the following:

1. **Quad Core 1.2GHz Broadcom BCM2837 64bit CPU**: The central processing unit (CPU) is arguably the most important component of the system. It is responsible for executing all program code including the operating system and external programs. All logic, arithmetic, and input/output (I/O) operations are performed by the CPU. Within the scope of this project, the CPU ran each of the programs we wrote for communication between the two systems. Because the process of encryption is essentially a series of cascaded logic operations, the CPU handled this procedure. As the amount of concurrent

operations performed by the CPU increases, the load on the CPU also increases. By analyzing the amount of stress a program places on the CPU, it is possible to see how much of an impact a program's logic has on the system. We specifically isolated the CPU load for the programs that were run on ECU #1 in order to compare the difference in load values for programs sending plaintext messages and programs sending encrypted messages.

2. **1 gigabyte RAM**: Random access memory (RAM) is the component of the system that stores all of the machine code and data being used by the system at a given time. RAM differs from other storage mediums like hard drives and memory cards in that it is very high speed memory. Similar to the CPU, RAM usage increases when programs require more temporary data storage. This was another factor that we looked at in our testing, as it is another measure of the impact that a program has on the system.
3. **BCM43438 wireless LAN on board**: Wireless local area network (WLAN) chips allow systems to interface with routers, and by extension, other devices that are connected to that same router. This was of particular importance to our data collection process because it allowed us to connect into the systems with an external laptop. This meant that while the main communication programs were running on the systems, we were able to run a separate program to collect CPU and RAM data pertaining to those programs. Aside from this, having the Pis connected to the internet allowed us to keep the systems up-to-date with the latest releases of programs and settings.
4. **100 Mbits/s Ethernet**: The Ethernet port on both systems is what allowed us to connect the two together for User Datagram Protocol (UDP) communication. This port supported a maximum speed of 100 Mbits/s which was advantageous as it is the speed that Komatsu had requested us to use in our testing.
5. **40-pin extended GPIO**: General purpose input output (GPIO) allows the system to interface with external devices, and enabled us to utilize the CAN shield in our tests. Although the CAN shield was designed to sit on top of all 40 GPIO pins, we only wired the necessary pins to their respective ports on the shield. By doing this, we had access to the remaining pins on the system. This allowed us to use the system's serial capabilities while also enabling the full functionality of the CAN shield. Serial communication is a method of data transmission. In this project, we used two serial-to-USB cables in order to connect the Pis to a laptop. From this point, we were able to run two terminal emulators on the laptop using PuTTy. Through PuTTy, we had full control over the systems.
6. **MicroSD port**: This port enables the user to provide two key elements to the Pi. The first of these is a bootable operating system, provided it is loaded onto a microSD card. Since the operating system is the software that all other programs run on, having microSD ports on the systems was critical. Additionally, any storage space not used by the operating system's files can be used as storage for programs and other files. The specific microSD cards used in our tests were 8 gigabytes in size. This left us with roughly 4 gigabytes of extra storage, which was more than enough room for us to develop our programs.
7. **Switched Micro-USB power source up to 2.5A**: Like all electronic devices, the Pi needs a source of power in order to function. The Pi is designed to accept power via a Micro-USB connector. The Raspberry Pi Foundation recommends a power supply of 5 V at 2.5 A. Power supplied outside of these specifications can lead to unreliable system performance and may damage the Pi. In our development and testing, the power supplies we used comply with the recommended values.

b. PiCAN2

The PiCAN2 shield-board provides the Raspberry Pi with CAN bus capabilities. It sits atop the Pi so that the system's GPIO pins connect to the 40 pin ports on the PiCAN2.

Figure 7 – The PiCAN2 shield-board attached to a Raspberry Pi.

The PiCAN interacts with the Pi through the use of the Raspberry Pi's Serial Peripheral Interface (SPI) bus. SPI is a synchronous serial communication interface ideal for short-range communication. It utilizes master-slave architecture using a single master.

SPI functions using four pins:

1. **Serial Clock (SCLK)**: This is the clock signal supplied by the master device, in this case, the Pi. This sets the pace at which data transmission between the devices will occur. The SCLK frequency when the PiCAN2 is connected to the Pi is 10 MHz.
2. **Master Output Slave Input (MOSI)**: Every cycle of the SCLK signal, the master sends a bit to the slave, and the slave receives it.
3. **Master Input Slave Output (MISO)**: Every cycle of SCLK, the slave sends a bit to the master, and the master receives it.
4. **Slave Select (SS)**: This is the signal used by the master device to select which slave device to interface with. Master-slave topology allows for multiple slave devices to be strung together in a daisy chain. However, for each slave associated with a master, the master must have a corresponding SS pin. It is common for the SS signal to be active-low, meaning that when a low voltage is present, the slave is selected. In this case, the PiCAN2 is the only slave, so only a single pin is needed.

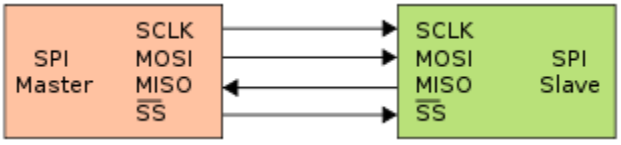

Figure 8 – SPI with a single slave.

In addition to these four pins, the shield also requires a 5 V pin, 3.3 V pin, and a ground pin to power the CAN controller (MCP2515) and CAN transceiver (MCP2551) chips. Both of these chips enable the shield, and by extension the Pi, to send, receive, and interpret CAN signals.

A PiCAN2 was connected to both ECUs, along with a twisted wire pair to connect the CAN high and low terminals of each shield to each other. 120 Ohm resistors were positioned at the shield terminals. We initially installed the shield in the standard manner, atop each Pi, but later chose to wire the shields externally using jumpers.

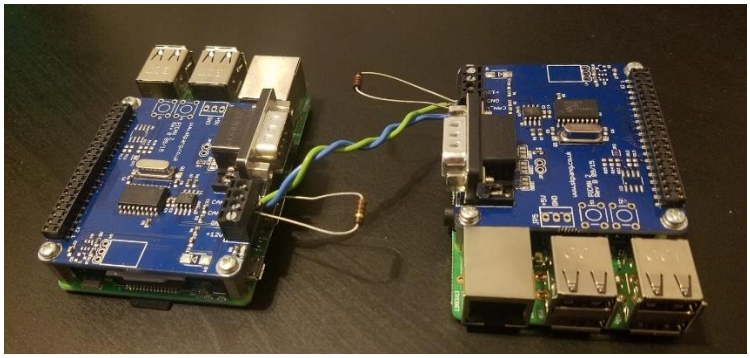

Figure 9 – ECUs #1 and #2 with CAN shields connected by a twisted wire pair.

c. Ethernet Cable

In our testing, an Ethernet cable was required for analysis of UDP communication.

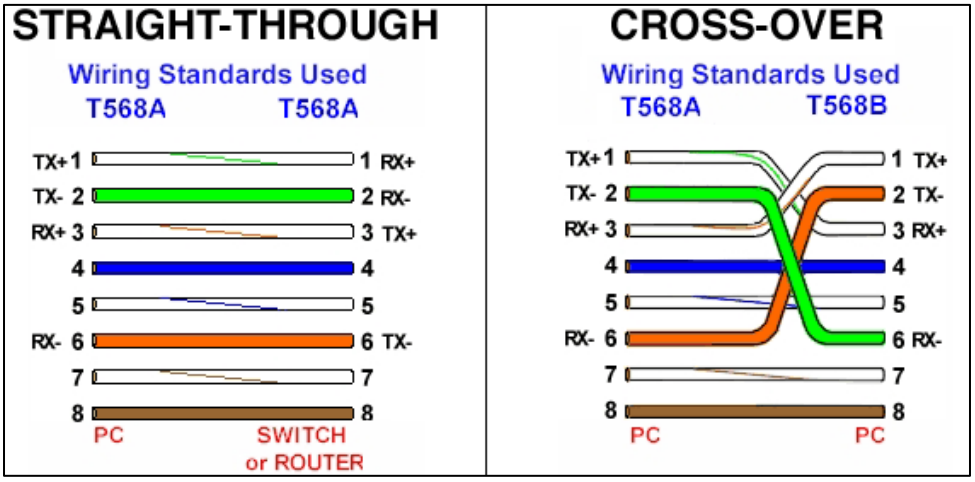

Figure 10 – Standard and crossover Ethernet pinouts.

Typically, when directly connecting together two devices of the same family, an Ethernet crossover cable is the desired medium of linkage. As can be seen in the above figure, a crossover cable is essentially a standard Ethernet cable with the wires rearranged to provide direct linking of the respective pins on each Ethernet port. This is in contrast to a standard cable, which is intended for the connection of a device to a router, switch, or another intermediary system. Fortunately, the use of a crossover cable was not necessary for this project. The reason for this is that the pins on the Raspberry Pi's Ethernet port automatically reconfigure themselves to create the same effect as a crossover cable when needed.

ii. Software

a. Raspbian Linux

In this project, both Raspberry Pis were running Raspbian Linux. Raspbian Linux is a Debian-based operating system, specifically configured for the Raspberry Pi. The latest Raspbian image was flashed onto the microSD cards used in the systems. For some of the early development, we used the operating system's graphical desktop environment, PIXEL. The advantage of the interface was that it allowed for convenient file navigation and editing. However, this came at the cost of increased system utilization because the Pi require the allocation of more resources in order to display the graphics. For the later portions of the project, specifically the testing phase, we opted out of using the graphical interface, and strictly used the terminal. This was to ensure that no extra system resources were being needlessly allocated towards graphics.

b. PuTTY, WinSCP, and SSH server

As previously explained, we interfaced with the Pis through serial and Secure Shell (SSH) connections. We achieved this through the use of a program called PuTTY. PuTTY is an open-source terminal emulator, serial console, and network file transfer application. Our programs used for testing were run using the serial console, and additional programs used to monitor resource usage were run using an SSH connection.

WinSCP, a Windows application, was used to transfer files between the Raspberry Pis and the development computers. The program uses the Secure Copy Protocol (SCP) to facilitate the transfer of files over SSH. It allowed for the concurrent editing of files on both Raspberry Pis using a single Windows computer. This was very convenient because the alternative was to edit program code using the console interface on the Pis. For small changes to the programs, local editing of program files was performed using the Linux Nano application. However, when dealing with files that were hundreds of lines in length, navigating and editing code in Windows was the practical choice. WinSCP also allowed easy access to the data that was generated from the programs. We were able to quickly transfer the .csv files to Excel for further analysis.

c. The Blowfish Encryption Algorithm

Blowfish is an encryption algorithm that uses a symmetric-key block cipher system to encrypt data. It was designed in 1993 by Bruce Schneier, and is open-source. The algorithm supports key-sizes ranging from 32 bits up to 448 bits, with blocks 64 bits in size. Like other encryption protocols, Blowfish uses a complex series of digital logic operations to obfuscate data. Specifically, a Feistel network structure is employed, meaning that relative to algorithms using different structures, the code for Blowfish encryption and decryption logic is small.

Before encryption can occur, the algorithm requires the data to be split into left and right halves. In addition to this, two large arrays (P and S) are used in the encryption process and must be defined ahead of time. At certain points, a function is used to split a 32-bit input into four 8-bit chunks. This function is known as the F-function. The entire encryption process consists of 16 rounds ($r \in 1\text{-}16$), with four steps per round.

The steps are as follows:

Step	Logic
1	XOR the left half of the data with the r^{th} P-array element
2	Use the XORed data from Step 1 as input to the F-function
3	XOR the F-function's output with the right half of the data
4	Swap the left and right halves of the data

Table 1 – Blowfish logic.

d. Programming in C

All the programs designed in this project were written in C, specifically GNU C. This choice was made because in an actual ECU on a vehicle, Embedded C would be the language used. GNU C differs greatly from Embedded C in many ways, but of the language options at our disposal, we chose the language that was most similar to Embedded C. Programming in C also allowed us to have low-level access to the hardware and socket configurations used for the CAN and Ethernet tests.

Another advantage of using C was that the Blowfish encryption algorithm was also written in C. While there do exist ports of the algorithm for languages like Python, there was much more documentation for the C implementation than for other languages. Also, Komatsu provided us a very useful program that generated the arrays needed for the Blowfish algorithm to function. These were the P and S arrays used throughout the XOR and bit shifting operations of the encryption process. The arrays are constructed using their own algorithm which relies on the encryption key. Komatsu's program made this process simple and straightforward.

Finally, the libraries used for CAN and Ethernet communication were also written in C. Linux provides an open-source, socket-based C library known as SocketCAN, which was authored by Volkswagen Research. Among other things, the library provides a plethora of convenient features including CAN-related structures and data frame parsing functions.

Linux also natively packages the C libraries for TCP and UDP communication. For test programs using Ethernet, we utilized these libraries for the configuration of sockets and the handling of UDP communication.

More information regarding all code discussed in this section can be found in Appendix B.

V. Testing and Results

Our testing method consisted of two parts. The first was implementing and testing communication over CAN, and the second was testing communication over Ethernet. In order to accurately gauge the performance impact of encryption, we needed to test and measure system performance without encryption. For all cases tested, we chose to analyze round trips for messages. The reason for this was that we believed this method would round out the data gathered in the testing process, and would give a better view of overall performance.

i. CAN Testing and Results

In our testing, we chose to first implement and examine CAN communication. This was because Komatsu asked us to prioritize CAN over Ethernet. The results of the plaintext and encrypted

implementations will be given in parts a and b of this subsection, and a discussion and comparison of the results will be given in part c.

a. Plaintext Communication

We first wrote programs to send and receive messages in plaintext.

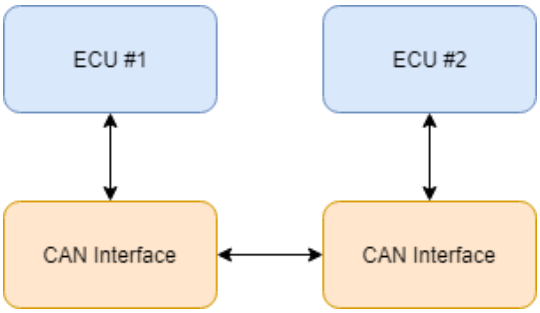

Figure 11 – Block diagram of CAN plaintext implementation.

Each ECU was equipped with a separate program to handle the sending and receiving of messages. A very basic outline of the process is as follows:

1. ECU #1 sends a message to ECU #2.
2. ECU #2 receives the message, constructs a new message using the same data.
3. ECU #1 receives the new message and ensures that it is correct.

To show a more in-depth outline of the logic behind each ECU's program, see the figure below.

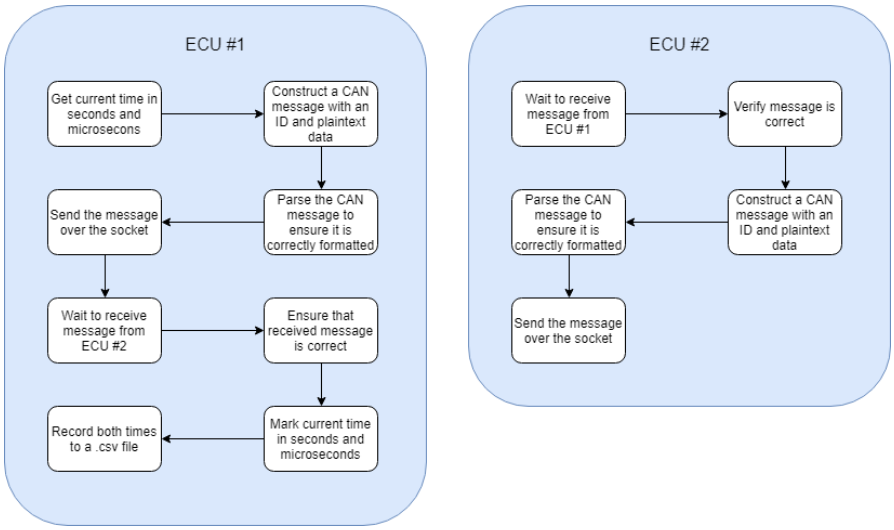

Figure 12 – Program logic for ECU #1 and ECU #2 plaintext CAN communication.

Since we were interested in the time it took for messages to make the complete round trip, ECU #1 took note of two specific times in the communication process. By recording the times at which the message sending process begins and ends, the total elapsed round trip time for each message could be found. This data was recorded in an external .csv file for analysis in Matlab and Microsoft Excel.

Since we were also interested in the CPU and memory usage of the systems, a method was put in place to gather this data throughout the testing process. Data of this nature was collected using the Linux command line program, **top**. **Top** is a utility that allows users to monitor system resource usage statistics. It also allows statistics to be isolated to a specific process, which was very useful in our case.

Initially, we attempted to call **top** from within ECU #1's program using C's system call functionality. However, we found that elapsed message times were greatly impacted when we used this method. This was because every time **top** was called, the program would hang for an inconsistent amount of time, causing the data to be unreliable. Following this, we opted to implement an external approach. We found that by connecting to the system through SSH, **top** could be called from a separate terminal on ECU #1. Having the programs and **top** running in parallel produced much better results. The CPU usage, CPU time, and memory usage for ECU #1's program were all saved to an external .txt file for further analysis in Excel. A series of 50,000 8-byte CAN messages were sent in this test, and the collected data is shown below.

Latency Data			**System Utilization Data**	
Total Time Elapsed (s)	64.83		Average CPU Usage (%)	4.89
Average Time Elapsed (us)	1289		Peak CPU Usage (%)	20
Median Time Elapsed (us)	1277		Total CPU Time (s)	3.10
Mode (us)	1274			
Standard Deviation (us)	132.2			
Kurtosis	146.2			
Skewness	11.54			
Maximum Time Elapsed (us)	4560			
Minimum Time Elapsed (us)	1227			
Range (us)	3333			

Table 2 – Latency and System Utilization statistics for plaintext CAN communication.

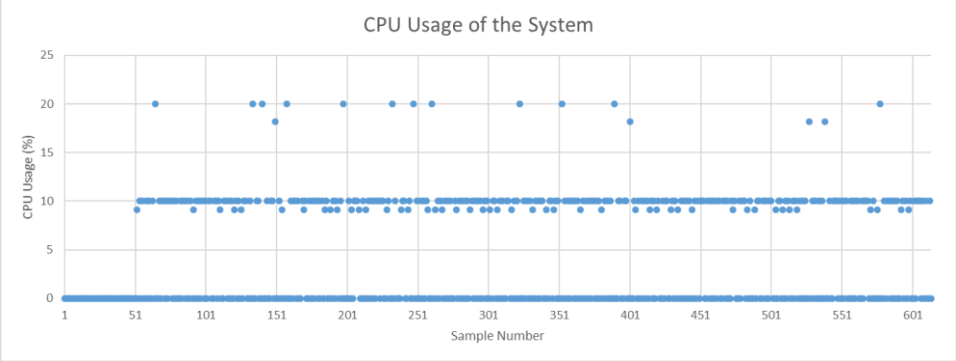

Figure 13 – CPU usage of the program run by ECU #1 for 50,000 plaintext CAN messages.

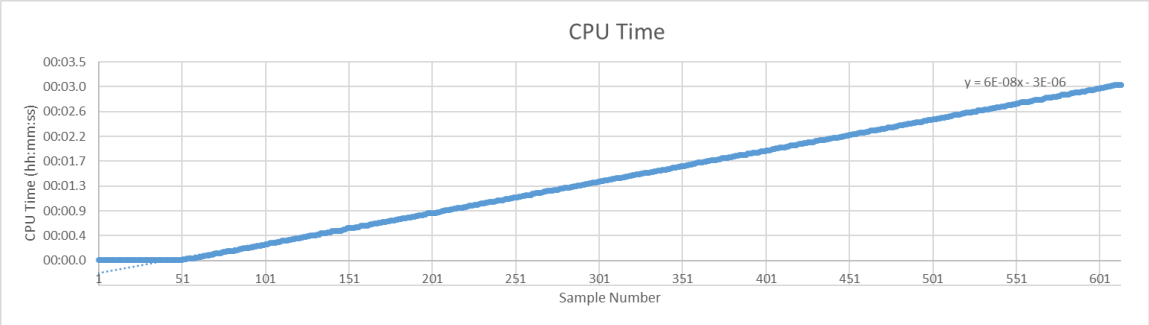

Figure 14 – CPU Time for the program run by ECU #1 for 50,000 plaintext CAN messages.

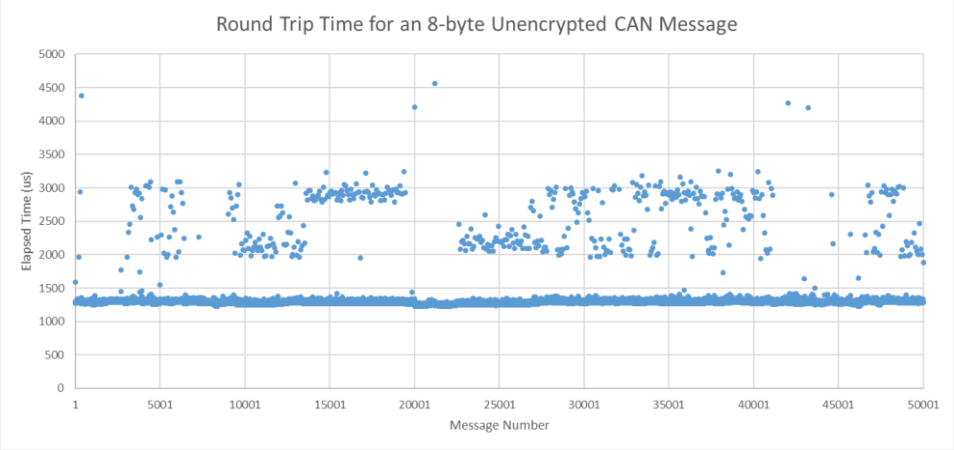

Figure 15 – Round trip time for 50,000 plaintext CAN messages.

Unfortunately, during the data collection process, we found that the system memory usage did not vary enough to be of any use in the analysis.

b. Encrypted Communication

A similar process was performed for the analysis of CAN communication with encryption.

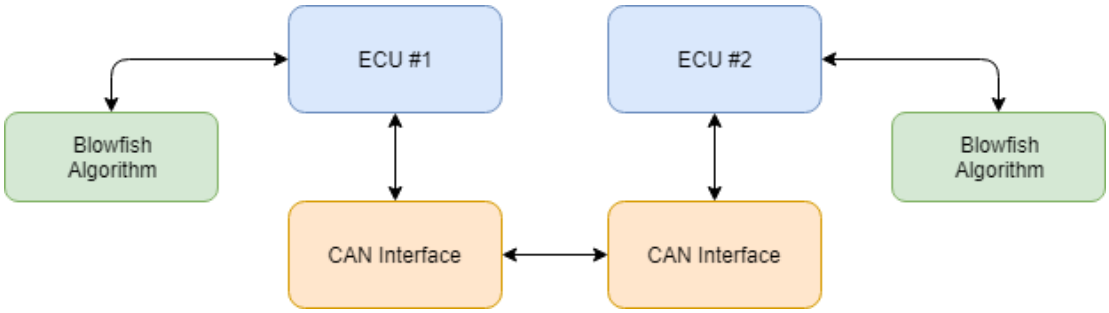

Figure 16 – Block diagram of encrypted CAN implementation.

The programs used in this procedure were similar to those written for the plaintext tests, except that the Blowfish encryption algorithm was used for the encryption and decryption of the data carried within the CAN messages. All of the frames contained data that was 8-bytes in both plaintext and encrypted form. The communication process consisted of the following high-level steps:

1. ECU #1 sends an encrypted message to ECU #2.
2. ECU #2 receives the message, decrypts it, and constructs a new encrypted message using the decrypted data.
3. ECU #1 receives the new encrypted message, decrypts it, and ensures that it is correct.

A lower-level representation of the program logic for both ECUs is shown below:

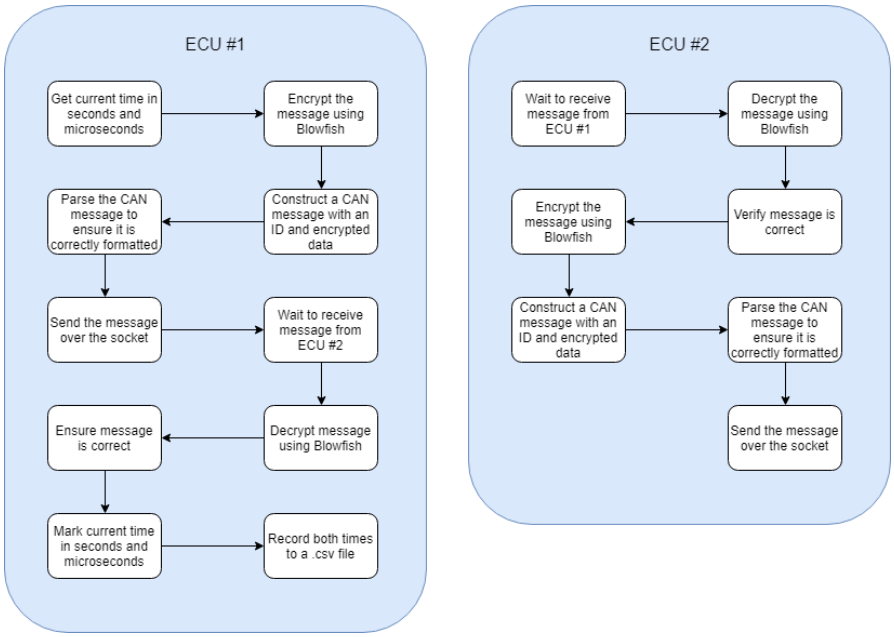

Figure 17 – Program logic for ECU #1 and ECU #2 encrypted CAN communication.

Again, latency was of concern in this test, so the elapsed time for the entire process to take place was gathered. Times were recorded just before the ECU #1's encryption of the first message, and just after ECU #1's decryption and verification of the second message, sent by ECU #2. This gave a full view of the total time for the entire process to transpire. These times were recorded in an external .csv file.

System utilization data was recorded in a manner identical to that in part a, with an external SSH connection calling **top** with the individual program ID. The data was recorded to an external .txt file. Using Excel's column data importer, the CPU usage and CPU time were extracted from this file. A series of 50,000 8-byte encrypted CAN messages were sent in this test, and the collected data is shown below.

Latency Data			System Utilization Data	
Total Time Elapsed (s)	68.8		Average CPU Usage (%)	7.30
Average Time Elapsed (us)	1368.29396		Peak CPU Usage (%)	20
Median Time Elapsed (us)	1355		Total CPU Time (s)	4.70
Mode (us)	1358			
Standard Deviation (us)	165.335782			
Kurtosis	1177.18567			
Skewness	22.3880983			
Maximum Time Elapsed (us)	15388			
Minimum Time Elapsed (us)	1308			
Range (us)	14080			

Table 3 – Latency and System Utilization statistics for encrypted CAN communication.

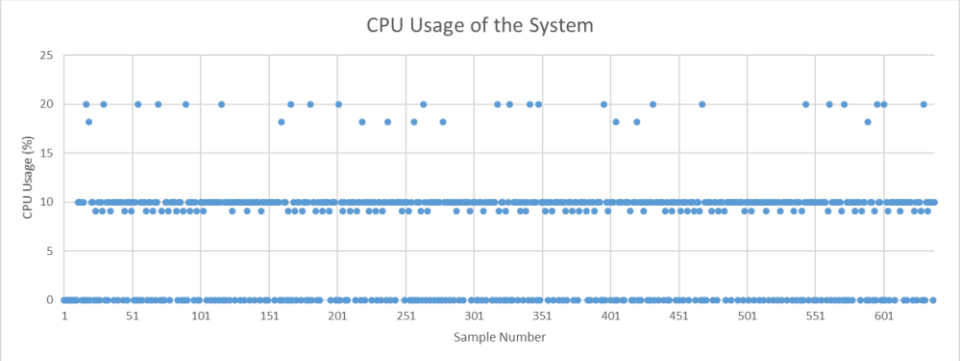

Figure 18 – CPU usage of the program executed by ECU #1 for 50,000 encrypted CAN messages.

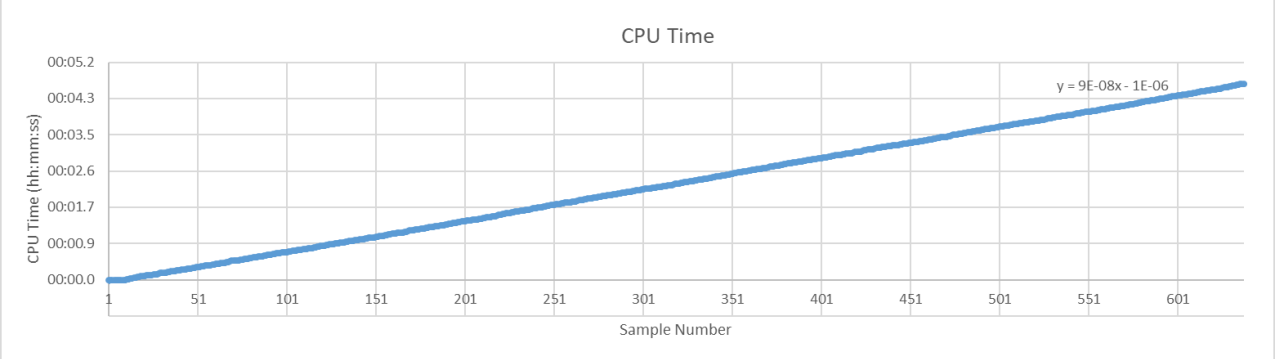

Figure 19 – CPU Time for the program executed by ECU #1 for 50,000 encrypted CAN messages.

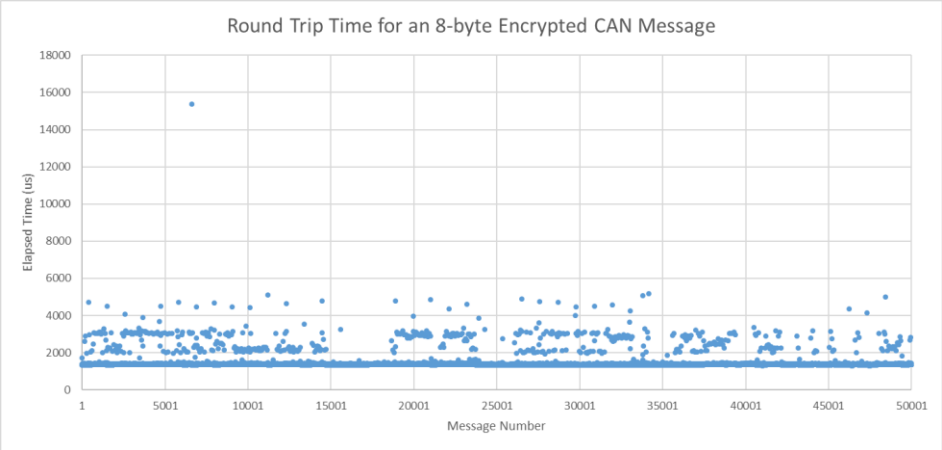

Figure 20 – Round trip time for 50,000 encrypted CAN messages.

c. The Cost of Encryption

Our expectations were met once we observed that the elapse time for encrypted messages was greater than that of plaintext message. For the CAN tests, the entire encryption and decryption process added about 80 microseconds to the overall round trip time. From this, it can be inferred that it takes about 40 microseconds to decrypt and encrypt an 8-byte message. This is because each Pi performed both tasks. The extra 80 microseconds increased overall round trip times by about 6 percent when compared to the plaintext test.

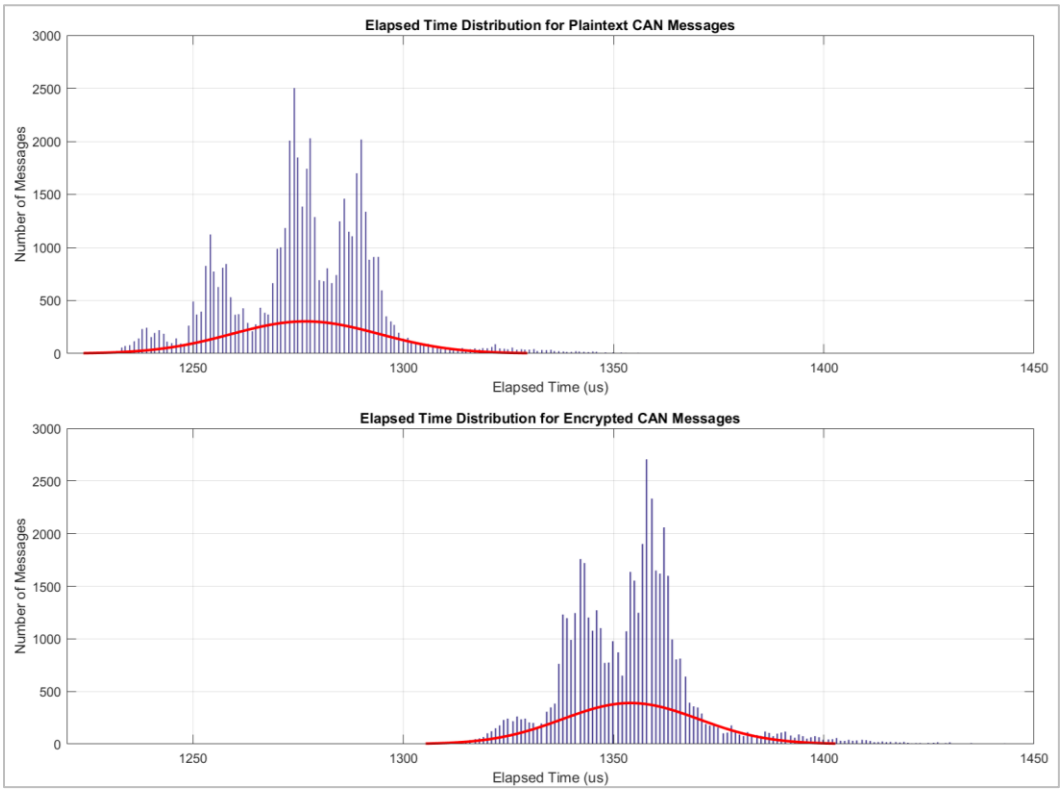

Figure 21 – Distribution of plaintext and encrypted CAN communication times.

In the above figure, it can be seen that both distributions are similar, with an 80-microsecond shift in the elapsed times. The plaintext distribution is clustered around 1290 microseconds, while the encrypted distribution is clustered around 1370 microseconds.

The CPU load was also measured for both the plaintext and encryption cases. It was found that the average CPU load for 50,000 plaintext messages was 4.89 percent. The average CPU load for the encryption implementation was slightly higher, at 7.30 percent. This means that when adding encryption, a roughly 50 percent increase in CPU usage is added as well. The difference in CPU times for both cases displayed a similar increase. The total time in which the processor handled program logic increased by about 50 percent.

ii. Ethernet Testing and Results

After testing CAN communication, we were able to implement and test communication over Ethernet. In software, the only major changes made from the CAN test programs were adjusting the type of socket used for communication and eliminating the message parsing function calls. The results of plaintext and encrypted implementations will be given in parts a and b of this subsection, and a discussion and comparison of the results will be given in part c.

Before beginning the testing process, we expected the overall time difference between plaintext and encrypted Ethernet messages to be similar to that in CAN testing. This was because the same 8 bytes were to be encrypted in this implementation, meaning that the algorithm should cost the same amount of time.

a. Plaintext Communication

As with CAN, for Ethernet, we first constructed programs for plaintext communication. Both Pis were connected together by a standard CAT5 Ethernet cable.

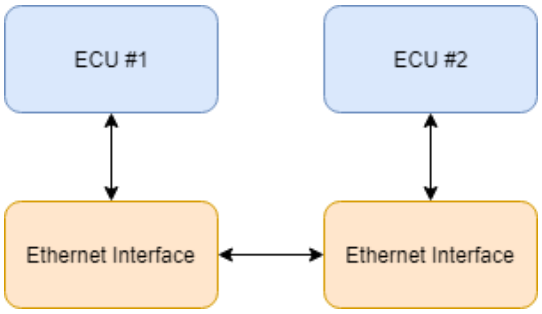

Figure 22 – Block diagram of plaintext Ethernet implementation.

Each ECU was equipped with a separate program to handle the sending and receiving of messages. A basic outline of the process is as follows:

1. ECU #1 sends a message to ECU #2.
2. ECU #2 receives the message, constructs a new message using the same data.
3. ECU #1 receives the new message and ensures that it is correct.

To gain a more in-depth understanding of the logic behind each ECUs program, see the figure below.

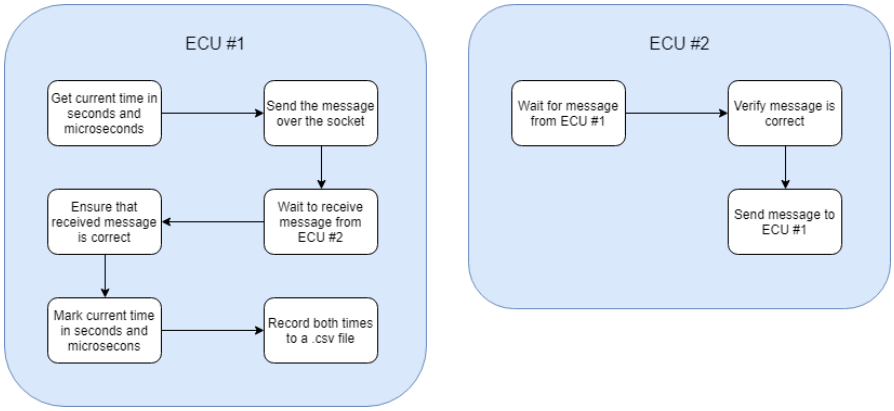

Figure 23 – Program logic for ECU #1 and ECU #2 plaintext Ethernet communication.

In order for us to analyze the latency in the communication process, the program on ECU #1 recorded two specific times in the process. The places in which the times were recorded were the same as in the CAN implementation. This ensured that comparisons between CAN and Ethernet were accurate, which allowed for a more accurate analysis.

The process for collecting system utilization data was the same as in the CAN data collection process. The results of 50,000 plaintext Ethernet messages are shown below.

Latency Data			**System Utilization Data**	
Total Time Elapsed (s)	26.55		Average CPU Usage (%)	10.95
Average Time Elapsed (us)	519		Peak CPU Usage (%)	20
Median Time Elapsed (us)	512		Total CPU Time (s)	2.40
Mode (us)	505			
Standard Deviation (us)	88.91273			
Kurtosis	363.0925			
Skewness	18.49953			
Maximum Time Elapsed (us)	3995			
Minimum Time Elapsed (us)	445			
Range (us)	3550			

Table 4 – Latency and System Utilization statistics for plaintext Ethernet communication.

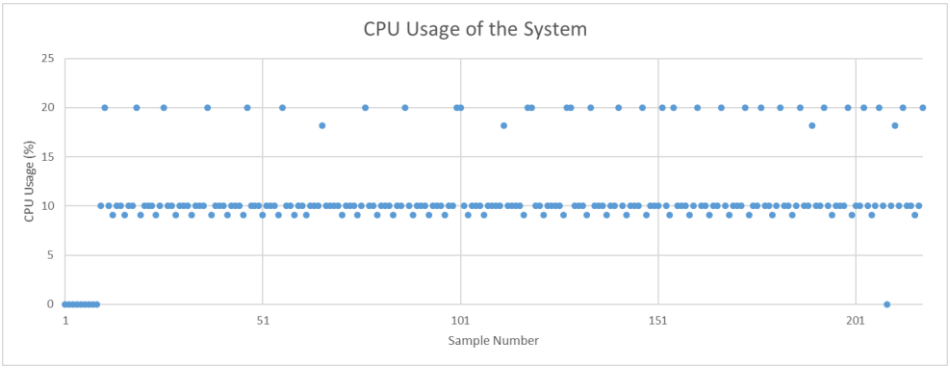

Figure 24 – CPU usage of the program executed by ECU #1 for 50,000 plaintext Ethernet messages.

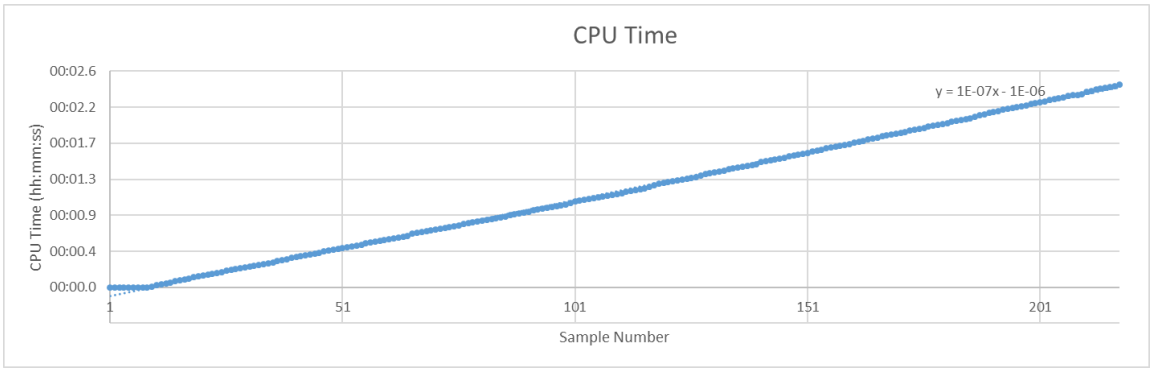

Figure 25 – CPU Time for the program executed by ECU #1 for 50,000 plaintext Ethernet messages.

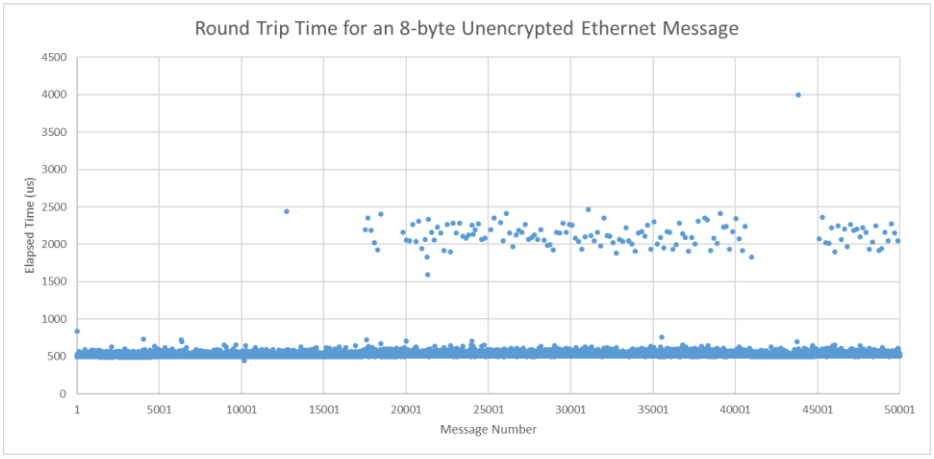

Figure 26 – Round trip time for 50,000 plaintext Ethernet messages.

b. Encrypted Communication

A similar process was performed for the analysis of CAN communication with encryption.

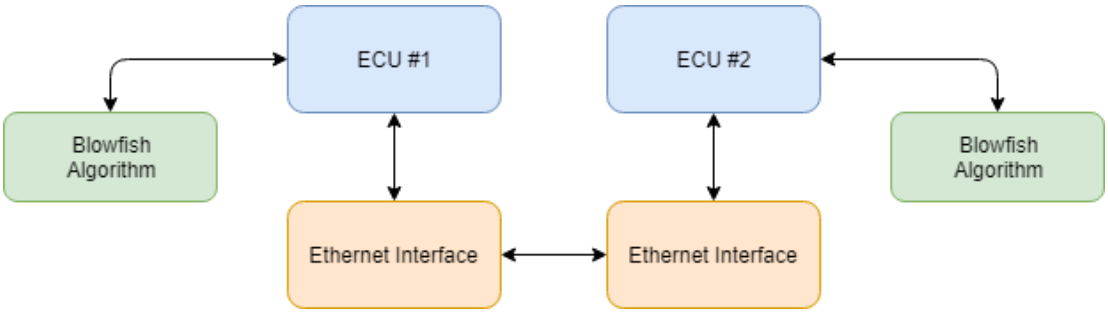

Figure 27 – Block diagram of encrypted Ethernet implementation.

The programs used in this procedure were similar to those written for the encrypted CAN tests. This was important to the data collection process because we wanted the CAN and Ethernet implementation logic to be as close to each other as possible. We felt that in doing this, the results for both cases would be more comparable. All messages contained data that was 8-bytes in both plaintext and encrypted form. The communication process consisted of the following high-level steps:

1. ECU #1 sends an encrypted message to ECU #2.
2. ECU #2 receives the message, decrypts it, and constructs a new encrypted message using the decrypted data.
3. ECU #1 receives the new encrypted message, decrypts it, and ensures that it is correct.

A lower-level representation of the program logic for both ECUs is shown below in figure 28.

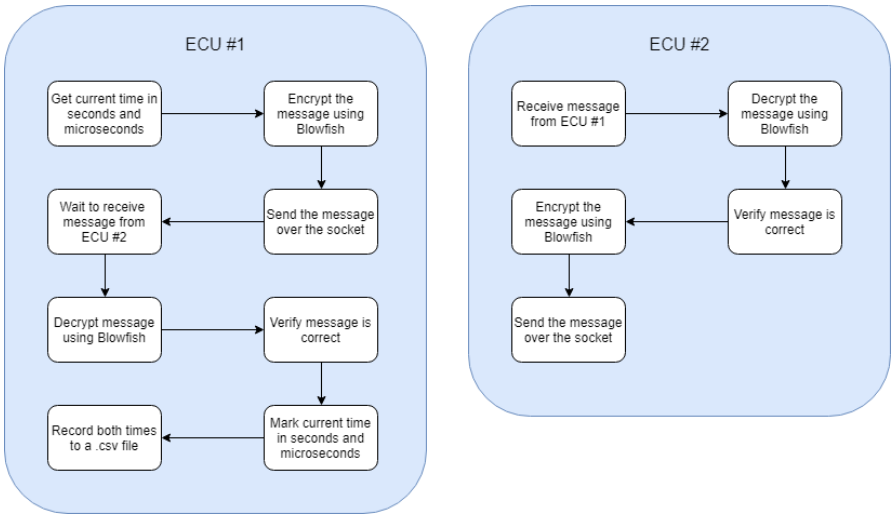

Figure 28 – Program logic for ECU #1 and ECU #2 encrypted Ethernet communication.

Again, the areas in which the times were gathered are the same as in the CAN tests. This was to ensure maximum comparability between CAN and Ethernet time values.

The data from 50,000 encrypted Ethernet messages is shown below.

Latency Data			**System Utilization Data**	
Total Time Elapsed (s)	31.04		Average CPU Usage (%)	17.29
Average Time Elapsed (us)	613		Peak CPU Usage (%)	30
Median Time Elapsed (us)	615		Total CPU Time (s)	4.50
Mode (us)	618			
Standard Deviation (us)	59.6			
Kurtosis	653.976			
Skewness	24.4597			
Maximum Time Elapsed (us)	2951			
Minimum Time Elapsed (us)	572			
Range (us)	2379			

Table 5 – Latency and System Utilization statistics for encrypted Ethernet communication.

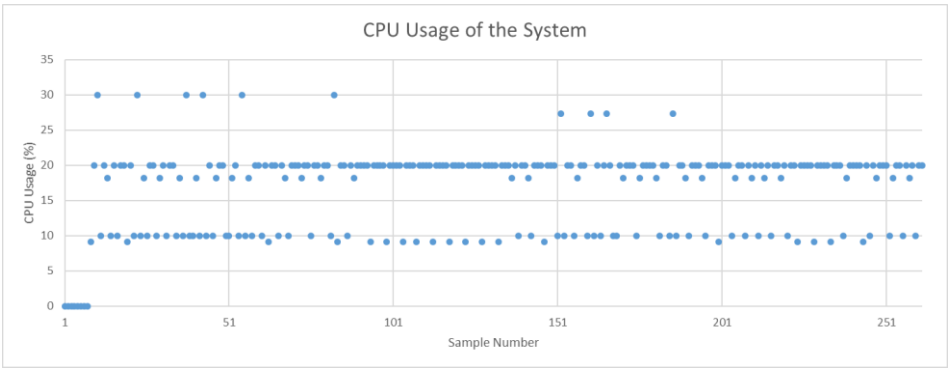

Figure 29 – CPU usage of the program executed by ECU #1 for 50,000 encrypted Ethernet messages.

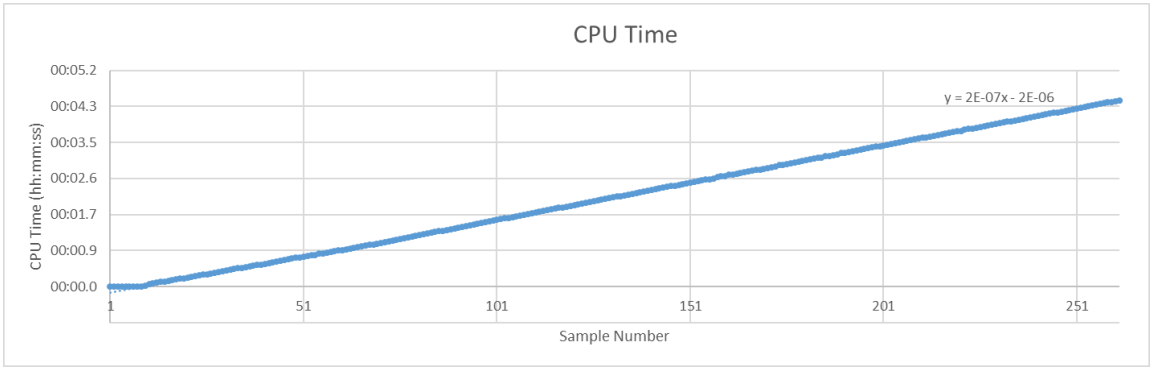

Figure 30 – CPU Time for the program executed by ECU #1 for 50,000 encrypted Ethernet messages.

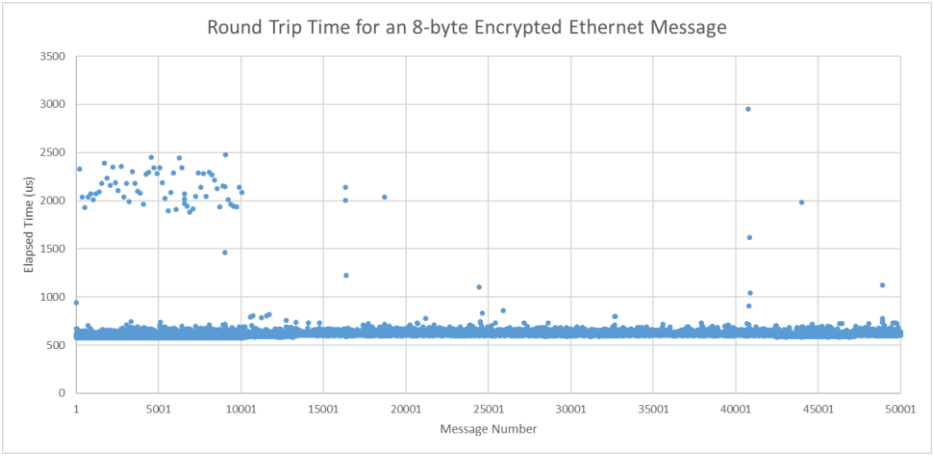

Figure 31 – Round trip time for 50,000 encrypted Ethernet messages.

c. The Cost of Encryption

Adding encryption to the Ethernet implementation had a performance impact similar to that of the CAN tests. During Ethernet testing, the encryption of messages added about 90 microseconds to the overall round trip time when compared to the plaintext Ethernet data. The average times were 519 microseconds for plaintext and 613 microseconds for encrypted messages, meaning that there was an 18 percent difference in elapsed times. This can be seen in figure 32 on the next page.

The average CPU usage for both cases were 10.95 percent and 17.29 percent respectively. This was similar to the CAN results in that there was roughly a 50 percent difference in usage. With this said, the average CPU usage was much higher in this test than in the same test performed with CAN. We believe that this is a result of Ethernet being a faster protocol. Since in both cases, 50,000 8-byte messages were sent and received, the same amount of CPU processing power should be used. However, the Ethernet test required this processing power in a shorter amount of time, leading to a higher system CPU utilization value.

The total CPU times used were similar to the CAN as well. Times ranged from 2.40 seconds for plaintext to 4.50 seconds for encryption. These numbers are slightly smaller than the CAN CPU times. This is due to some CPU time being used for polling for new messages continuously. The Ethernet driver spent much less time checking the socket for new messages.

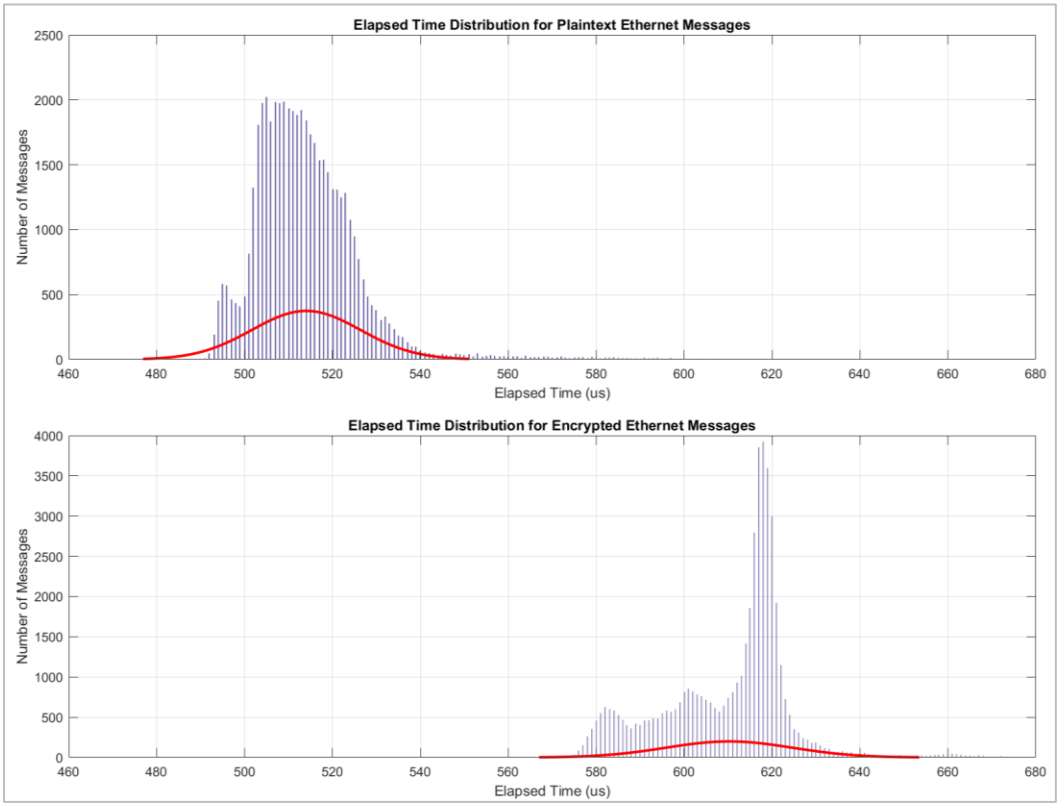

Figure 32 – Distribution of plaintext and encrypted Ethernet communication times.

VI. Conclusion and Future Work

All of the goals that Komatsu set for us were reached, and the collected data has been neatly placed within well-formatted Excel workbooks and Matlab scripts. Through our testing, we were able to create an accurate comparison of all four ECU communication cases. We believe that this research has provided sufficient proof of concept for the implementation of encryption on control units within machinery. With this said, there are several ways in which this project could be extended and improved in the future.

Perhaps the largest improvement that could be made on this project would be using the actual microcontroller present on the ECU. This would be an improvement over the Raspberry Pi implementation in this project because it would provide a much more accurate representation of the ECU hardware. This would lead to more accurate results, and an easier transition to implementing any project testing code onto the actual vehicles.

The Raspberry Pi is not a very good representation of an ECU for several reasons. One of the biggest issues is that Raspbian Linux is not a real-time operating system. Because of this, there are a plethora of background tasks running at any given time. This can bog down the system by stealing CPU and memory from more crucial processes. In a real-time operating system, tasks are executed on a very tight schedule to ensure that no unnecessary tasks are being performed. In addition to this, memory allocation is much more critical in real-time operating systems. This is because, in order to ensure system stability, there can be no memory leaks.

Another way this project could be expanded upon would be testing round-trip performance for a multi-ECU network. This could be accomplished by combining last year's project with this project's testing programs and some minor modifications. The advantage of a setup such as this is that it would more accurately simulate a real-world implementation.

Finally, different types of encryption algorithms could be tested on the systems. From here, a comparison of the system utilization and message latency costs could be made. Different algorithms provide different levels of security, and it would be interesting to see the security-to-cost ratio of a variety of algorithms.

VII. References

https://www.raspberrypi.org/
https://copperhilltech.com/pican-2-can-interface-for-raspberry-pi-2-3/
http://www.cables-solutions.com/difference-between-straight-through-and-crossover-cable.html
https://www.ssl2buy.com/wiki/symmetric-vs-asymmetric-encryption-what-are-differences
https://www.schneier.com/academic/blowfish/

Appendix A – Bill of Materials

All of the parts used in this project were recycled from last year's project. Because of this, the material cost for the project was zero dollars.

Quantity	Component
2	Raspberry Pi Model 3B
2	PiCAN2
2	8 Gigabyte MicroSD Card
2	5 V, 2.5 A Micro-USB Power Supply
2	USB to Serial Cable
1	Ethernet Cable

Table 6 – Project parts list.

Appendix B – Source Code

The C code for both the CAN and Ethernet implementations are split into two parts. The first section is for unencrypted messages used for baseline benchmarks. The second section is for encrypted messages.

blowfish.c: The file containing functions for encryption and decryption. P and S arrays were generated using Komatsu's proprietary key generation program written by Jason Schepler. This file is present for CAN and Ethernet encryption implementations on both ECUs.

```
// Modified blowfish.c file.
// P and S arrays generated using BlowfishArrayGen program by Jason Schepler.
#ifdef little_endian   /* Eg: Intel */
   #include <dos.h>
   #include <graphics.h>
   #include <io.h>
#endif

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#ifdef little_endian   /* Eg: Intel */
   #include <alloc.h>
#endif

#include <ctype.h>

#ifdef little_endian   /* Eg: Intel */
   #include <dir.h>
   #include <bios.h>
#endif

#ifdef big_endian
   #include <Types.h>
#endif

#include "blowfish.h"

#define N               16
#define noErr            0
#define DATAERROR         -1
#define KEYBYTES         8

unsigned long P[18] =
{
  0x706D9FCC, 0x1792D23A, 0x2DB9D714, 0x966E1439, 0xAC21A76D, 0x8324E988,
  0xAC0DC9DD, 0x2C38F6B3, 0x70619520, 0xFA23ECBE, 0x17B2F676, 0xEBA13A04,
  0x8B949E61, 0x7A147CAF, 0x56CCC6B6, 0x4461B24D, 0x7361E6A1, 0x196A7C43
};
unsigned long S[4][256] =
{
  {
    0xBC29E0A1, 0xCC73A6D2, 0x7B255A38, 0x16E42BB5, 0xB9AD572D, 0x0AABAC90,
    0xC5505974, 0x9B05C7A3, 0x45024F2D, 0x96CB9B09, 0xC8F75E4E, 0x11CBB634,
    0xEC5E0CCB, 0x3CE49F7A, 0x3B7F2579, 0xF89EB915, 0xEEAEEFC7, 0x9547FE50,
    0x4AB4FD2D, 0x095CF9EC, 0x507C983A, 0x47242D2F, 0xA5B2CEA3, 0x7B124BB9,
    0x5BACC696, 0xC795928B, 0x4B951626, 0x80BCD0C5, 0x97919EA7, 0x0C29D22A,
    0x67F67B4C, 0xB89CB1E1, 0xA8695041, 0x3034D67C, 0xE7F15452, 0x0C45FA3C,
    0xDA38E7D8, 0x44B9AD0B, 0xEB0479AF, 0xF92EABE8, 0xF8DBFDCD, 0x7DF2748B,
    0x9015E377, 0x2B2B85D5, 0x93682A96, 0x7BB9C921, 0x5D2D72DA, 0xDB12E508,
    0x1D097F2D, 0x7102BFC6, 0x6F701C92, 0x808C6F71, 0x64FCBFA8, 0x7120112C,
    0x63DA8486, 0x0CD6CC8E, 0x7DBD60CF, 0xBE3B7BD5, 0xC9C2A32D, 0x764B5B2E,
    0x0929D362, 0x7B3DB3AC, 0x8B47AB61, 0x1193D865, 0x4E5DBA00, 0xC97642EA,
    0x2E2EDDFE, 0x2813E1A0, 0xF6C170F4, 0xA7A726CA, 0xF92A07A6, 0x77086DC2,
    0x6B8A4B6D, 0x87B174FD, 0x587EDC05, 0xF9CDAF5C, 0x7E3C925D, 0x7261F4A0,
    0x827D1078, 0x220377D4, 0x2CFA2533, 0x8010A5A1, 0xF26EE83C, 0x78A9C1E3,
    0xC35FB910, 0xEABD6C42, 0x7AEBB2C7, 0x9BAA026B, 0x9D09484C, 0xF1D18D6C,
    0x1CDAAFF7, 0x31D85DFB, 0x3C2F1DCF, 0xEB074DEA, 0x5E351D0D, 0x1C937C72,
    0xC5066A62, 0xF1AC19D5, 0x16032745, 0x77DAD6D7, 0x95088FD3, 0x4038CD86,
    0xD6D6F831, 0x59BF7B2D, 0x6D9208A7, 0x04EB91D4, 0xF6C0AE79, 0x4C6E7DFB,
    0xFB2E72D7, 0x210B7FA9, 0xEC5F4546, 0x6656B4A2, 0x5FED6EB1, 0xFE559498,
    0xE78A2D2F, 0xF870CD06, 0x6B439BA6, 0x037E0C72, 0x53BC605A, 0x4024810C,
    0xFE52A29C, 0xE67DC240, 0x641E21CC, 0x8ED29AB6, 0x17A922B3, 0xB7EFBEAE,
    0xD94AF07B, 0xCBA4CC18, 0x361DB726, 0x84DB6066, 0xBE89AEF6, 0xC85C8101,
    0x64F0B9F0, 0x09631C57, 0xD83207A2, 0x697DDFEF, 0x4C0A823D, 0x68B6D11A,
    0x529AF12A, 0x851F697E, 0xA8DE5997, 0xE805E196, 0x67B1650C, 0x9C03B246,
    0x0C7A8507, 0x769EA020, 0xEC463E5A, 0x209B82C8, 0xD11339BC, 0x6340775E,
    0x1FBED4BF, 0xBDD93416, 0x9B62F11B, 0xA7E5729A, 0x403FD5BB, 0xDE883F87,
    0xFE002D1C, 0xC9D9AA5A, 0x31EC47CA, 0x837DCA0C, 0xC09114CF, 0x0246E876,
    0x1A9FE4AF, 0xB054DA48, 0x3B0CA5FF, 0x7AB24415, 0x396CA045, 0x4883A58A,
    0x5C0C9979, 0xCEF065FA, 0xEB9858D4, 0x92A71DDF, 0x2DB558F4, 0xF4D697DC,
    0x438BF05A, 0xAA01B428, 0xDF81DD23, 0x074E3B94, 0x7BDFF85C, 0x74517E4A,
    0x61960DA6, 0x85900821, 0x60007082, 0x66D03DEE, 0x35CB3A43, 0x1BFF2085,
    0xDFAC0758, 0xBA1BB5D2, 0xDB2C9F41, 0x9F824A41, 0x6A320C91, 0x9C0C2A90,
    0x72B35ACE, 0x031B2685, 0xC1884629, 0x28B46EEE, 0xE5B8FA3A, 0x5E5C26B2,
```

```
    0xCEE6D2E9, 0x40959425, 0x191CDD50, 0xB52F9664, 0xE40D0FB0, 0x961567FF,
    0x0855B0DB, 0x0FF145EC, 0xDEAD649D, 0x12185985, 0x21C4E91A, 0x9BE13EBD,
    0x1B699849, 0x98C3E7BB, 0xC6A6FF47, 0x3A5FDED3, 0x52AECDF2, 0xF6BC0C78,
    0x2DD45A5A, 0x5FFF159D, 0x2EECF9BC, 0x134A5CC1, 0xC4933785, 0x4C85375C,
    0x913860C3, 0xCF035D93, 0xF9D2BF06, 0x8346123A, 0x1A52E98B, 0xE559AD7D,
    0xF279B69B, 0x12277A56, 0x012A3B50, 0x09D3A2D4, 0x2B778C56, 0xBB1D65D0,
    0x9CDB593D, 0x25A067F7, 0x323E81F9, 0x9F3148B7, 0x829295C4, 0x5A49B5F1,
    0x0EF2C289, 0xEF03C824, 0x499B4F82, 0x1B84EA13, 0xBFFC7779, 0x08FF68E6,
    0xE53CABAC, 0xED355F38, 0x3855899E, 0x28D80BCA, 0x5A031F37, 0xD72D4BA7,
    0x38E54857, 0xD6DDA34A, 0xC3465D76, 0x09A5C1ED
  },
  {
    0x7E867B3B, 0xC1E2F578, 0x4CCECC4F, 0xC0D71E5A, 0x77E4818F, 0xF07C7B91,
    0x9625C477, 0xD816E24C, 0x965167D4, 0x02B28771, 0xFC5B13C5, 0x50253220,
    0xF78E1953, 0x813B24A6, 0x8E1D9287, 0x0A58A319, 0x0EF0AA73, 0xC836E168,
    0x075DFA61, 0xC1C7C447, 0xA4CD9C56, 0x3F2353CB, 0x495F0572, 0x49289D93,
    0x5EF5FC55, 0x88F01DB4, 0xEABD6979, 0xDDC6A38D, 0xF172E37F, 0xBC535D4D,
    0x834BE3C3, 0xDE4E53B1, 0x446AEF8C, 0x59FF8B44, 0xA461FB6B, 0x008C0EF8,
    0x5E952E88, 0x8007C2BC, 0xE1DFCB6C, 0x8BC91DF5, 0x0886FDAC, 0x61EC03FF,
    0x92907748, 0xC7CB1952, 0x235929E8, 0xBAD38578, 0xA40293D6, 0x2CE28A0D,
    0xE2AC431C, 0x9C0578CE, 0xC8B3BB04, 0xB3BDF1DC, 0xB5538FCF, 0x383147D9,
    0xD1516C3A, 0xE9D28332, 0xC0A42E09, 0x1BD9B349, 0xAAC8C0D0, 0xF1BB3BF1,
    0x48EB2B0F, 0xD3F862A2, 0x5DBAFEBB, 0x850B1A12, 0xD368F5B9, 0xFA4B1274,
    0x15DDFC69, 0x9C646E66, 0x54C9B089, 0xDF326714, 0xB462A23E, 0xB3B7DF45,
    0xC072D39C, 0x82CB3014, 0x69B5CB3E, 0x8CF9A3FF, 0x44C6D7A6, 0x52156F5B,
    0x2BABD959, 0x9F9F12DF, 0x0740286B, 0xA08D6D25, 0x892EAD4C, 0x4D6E9EBB,
    0x900463AE, 0xBFF4F886, 0x9CAB0BE8, 0x0FA626CA, 0x77D73598, 0x44AA9FAC,
    0x72493624, 0x20A98E7C, 0xE679D315, 0xD2E0A30A, 0xD896FDC6, 0x6A62AEA4,
    0x9482078C, 0xC605EC23, 0xA82872CC, 0x6265B663, 0xB6FD526E, 0x742E9AF7,
    0x81FE71F7, 0x854BCD82, 0x6003ABD8, 0x1D71D60A, 0x5D980B3D, 0x7EBE5E8E,
    0xF0F5DA3F, 0xCFF83918, 0x8A1D62AA, 0x8DE22414, 0x3D33A505, 0x7EEE5CC6,
    0x3AE91A5B, 0x876102FB, 0x9FC91427, 0xD9568A58, 0xB5A10516, 0x96268186,
    0x2519B32E, 0x313EE250, 0x156061ED, 0xDE660333, 0xEC9B272E, 0x0EEDE729,
    0x467FFEA8, 0x67EFFD41, 0x00AE6738, 0x54423211, 0xD184392A, 0x2436ED4D,
    0xEBF06527, 0x6E776D5F, 0xCE3ECF74, 0x5C13F32B, 0x1BD2F97A, 0x2C9DE423,
    0x269BFB10, 0x89B339A6, 0x7D817B4E, 0x9421AC5E, 0x916095E4, 0xFA00591A,
    0x7CDF744D, 0x1BEF2E37, 0xBE41F67D, 0xA4CE9B09, 0xBB5BC79B, 0x3631E9A9,
    0xBCF42705, 0x987FE159, 0x09CB14F6, 0x39981B19, 0xE9B7E212, 0x7D4B1A24,
    0xF5DA7134, 0xE68388D8, 0x61652345, 0x271BD494, 0xBB318401, 0x08046D67,
    0x23CE5A3F, 0x4A381FC2, 0x5AC32585, 0x5FE1A2A1, 0x47C9D8A6, 0xC5ABBA8F,
    0x5ABC86A6, 0x23FFD45B, 0xB289313F, 0x1AEFE82B, 0x47CDF935, 0x22B279A2,
    0xCBD5B23D, 0x3864DB5F, 0xA9AE7A32, 0xBFAF601E, 0x0D4A1879, 0x0C844243,
    0x1D6277A3, 0xA630C3FA, 0xD7BBC07C, 0x6EE545F8, 0x4746DBB7, 0x240ACA48,
    0xCDAE49A6, 0x4E6EA014, 0x7CDEDDA4, 0x7AE4B5FF, 0x853382F8, 0x9116738E,
    0x3406A5F1, 0xD1F7465A, 0x260A3B87, 0x8C11422F, 0x9805B9FA, 0x7D403572,
    0x4252684F, 0xF6EEF2E4, 0x5F950ECF, 0x781F641F, 0xF891F8EC, 0x112F14A7,
    0x5DECDED0, 0x171F5D39, 0x7B9E6563, 0x76E93FD5, 0xF2BC6EE9, 0xCDBA0103,
    0xDF3F5F72, 0xDFEB7DDE, 0x9895F9C5, 0x554BC788, 0x13CE43CA, 0xAEB37587,
    0xB1B19144, 0xDF9B4E36, 0x22D161A8, 0x964FA04B, 0x9AF039CA, 0x8A04FE4E,
    0x4A4601AD, 0x249507EA, 0x51BE4063, 0x9AAC44C0, 0x476DCEF9, 0xA2B7FAFA,
    0x6EF3A8D5, 0xF5865DEA, 0x71931CAA, 0x5DC98E00, 0xD5EB1229, 0xE822B634,
    0x469C0AC5, 0xCCD2BEFD, 0x7582C931, 0xB1F06CF2, 0x279DF8F6, 0xCF6F3ADC,
    0xFD9BB818, 0x4CF68584, 0xD85556BC, 0xA5452EC5, 0x35A48179, 0xEAF52FCC,
    0x146F2AE8, 0x74F73771, 0x5BC7224D, 0x2DB1B066, 0xEA00280C, 0xBB2247D6,
    0xEAB87905, 0x0890EAF3, 0x8D4A4A77, 0x4EC72E6E
  },
  {
    0x8C43EA8E, 0x5378268F, 0x05D8585F, 0xA4439F6D, 0x6F7A809D, 0xE64EA9B3,
    0x5DCE752D, 0x179313C8, 0xF48BD418, 0x94593E7A, 0x358E81A2, 0x5401C66E,
    0xCB44ED8C, 0x26F5029B, 0xE5A4E250, 0x7918A687, 0x508437C6, 0x94FCF372,
    0x8F215DBD, 0x6F55B34B, 0x5A3B650A, 0xD599DCE4, 0xB56710AD, 0x08271EB9,
    0x83416D76, 0xEC67921F, 0x14CCC650, 0xEF233D89, 0xBD48FC3E, 0x1D9CEBD0,
    0xD392F53C, 0x65170345, 0xDA13A370, 0x5D0EE68F, 0xDFCDAEB1, 0x0123B50C,
    0xBEC44671, 0xFB055735, 0x933D9ABD, 0x87C91FBB, 0xAC6E1280, 0x9F171D0F,
    0xED6FA263, 0x90A4E1E2, 0x483823C1, 0xE0080101, 0x39CE66DA, 0xD932AED7,
    0x22D70D20, 0x8EC4B8B4, 0xF0625243, 0xF0756A5D, 0x3AA72992, 0xAE678117,
    0x44C85448, 0xBC116403, 0x1A7E0FAB, 0xAFE0764B, 0xCD5EBABF, 0x07E90F5D,
    0xED5053BA, 0x89FCA659, 0xBE472ECA, 0x53606610, 0x02808C27, 0xD5B81709,
    0xCA444EB3, 0xB8469D58, 0x7E1A6C1C, 0xD54337A9, 0x8B53D406, 0xC32E9B04,
    0x551A4F6E, 0x538BCBC2, 0xDA68E96B, 0x3496E4FA, 0xFD2D8862, 0x45D77D77,
    0x58FCB8AD, 0x9109487C, 0x76AF0B06, 0xB78CE58E, 0xB7FBE3B1, 0xE7FEE525,
    0x25DCC4AA, 0xF145006A, 0x181995B9, 0x9419EEE5, 0x450E0D6F, 0xC7099EE5,
    0x0FA80667, 0xB02868A0, 0x0B49EA78, 0xE128155C, 0xA5BDED74, 0x83C3C89A,
    0xD3EF638F, 0x57A1BE8F, 0x41F1962D, 0xB705C2A5, 0xCE2DB25F, 0x598070E3,
    0x549B5B39, 0x50ECAD57, 0xD113AE86, 0x75CA5880, 0x17085017, 0x2FF75D1B,
    0x29F9291D, 0x0BB53727, 0x58C90800, 0x8F63179C, 0xC856E55B, 0x7EE15473,
    0xDE52107D, 0x1F73AA59, 0x1A4E56E4, 0x7CDC134C, 0xCFECD32C, 0x91BB1347,
    0x76A7CDE3, 0x3C9FB67E, 0xA8ADE78F, 0x6B230B91, 0xB9371A0B, 0x50145379,
    0x5C0D8504, 0xA502410A, 0x10841967, 0xA4894FA2, 0x758239DD, 0xECBE1677,
    0x0F94112A, 0x0E14B339, 0x2A553747, 0x278D9D54, 0xB20E6318, 0x289DB523,
    0xE07C5C90, 0x97E8685C, 0x945FDAC4, 0x940FC1E4, 0xB07A6B23, 0x017EBB91,
    0x257F4E32, 0x589AFFCF, 0x6AA41816, 0xFBFC9B44, 0xCF273FAE, 0xA342E5DE,
    0x418A75C4, 0xEFEE3C8A, 0xF98B9E59, 0x24C098C8, 0x3F468386, 0xA7005941,
    0x3B36A56B, 0x7803E167, 0x72242634, 0x1D961A47, 0x257F7825, 0x3977854B,
    0x4709FA07, 0x140E258F, 0x7BF4231F, 0x40C8BDEF, 0xAAA896B8, 0xADD3BD76,
    0xB100DF78, 0x76B415DD, 0xA0ECBF70, 0x79689957, 0xF3467D2A, 0x434DF493,
    0xBA0A3447, 0x30D9638A, 0x0C62BA1B, 0x10D870B2, 0x3F1C0302, 0x66A56B4D,
    0xCC572C1F, 0x2E42847B, 0xCC5F7544, 0x2855B6CF, 0x28F53F4A, 0x2AFF6E01,
    0x427DB358, 0xA72017FE, 0x6071428A, 0x24F246FF, 0xC745B3DD, 0xACE52832,
    0x620527A7, 0x1BF09DAD, 0xC418627E, 0x30A2C846, 0xEB3AEB56, 0xCC2210E8,
    0x0176E124, 0xF80CDEFC, 0x20FF27E2, 0xA17B5E10, 0x3599B22F, 0x0F219934,
    0xCD833A91, 0xF8F255C3, 0x401E0D2D, 0x80AAEF91, 0xFBB1BA5A, 0x28FD2F70,
    0x9C24873A, 0xFC1922F8, 0xEF5C4D66, 0x40F2A183, 0xB1DE4D8E, 0xB356036F,
    0xDD466F7C, 0x31CC7719, 0xB832F4A1, 0xE0C73651, 0x40A3D553, 0x277ACEDF,
    0xF306FA0C, 0x8F1F275B, 0x222C5DC1, 0xA94C0B42, 0x9262FF89, 0x01FA21E1,
```

```
    0xD56AA168, 0x5A488654, 0x40409C48, 0x4D80EAF4, 0x155008D7, 0xCF6ADCEE,
    0x92D69025, 0xDACC637E, 0xD6062CBD, 0x4920EE74, 0xD75C5852, 0x806B84EE,
    0x64ED0D24, 0x0ACDAABA, 0xB33AF9D0, 0x31F94B66, 0x387E534A, 0x5D49A53C,
    0x8223C610, 0x15F00B65, 0xCB715FCA, 0x33EEAB1E, 0xF11D5983, 0xFA2BB7EF,
    0x24DD6194, 0x2CB751AE, 0x3B75BF9A, 0xE56ADFCF
  },
  {
    0x1B736708, 0xB61E15F6, 0x43ABDE45, 0x04AAFF72, 0x5EF1D215, 0x5BFF7C02,
    0x2D5DB7AC, 0x97A29757, 0x5EA85D53, 0x0BB17DB3, 0x9796259A, 0x25CE9062,
    0xA9018CCF, 0x9D544BC5, 0x25E617F4, 0xAC8324E3, 0x24851F54, 0x70C43D03,
    0x3DF1AC5C, 0x2A7F8A99, 0x5990CDA3, 0xCA920C72, 0x33E627D4, 0x41CFD5E0,
    0xAB367680, 0x66FA53A0, 0x4D7411E6, 0x9D251FD5, 0x5565F4C4, 0x6B739893,
    0xB66C630E, 0x6681915F, 0x236D47A9, 0x40DADD61, 0xB6D82796, 0xC9489914,
    0x840D74BD, 0x26A43010, 0x103083E9, 0xECBB020F, 0x0642E110, 0x08C12D9F,
    0xB63DD35C, 0xE6E4E3F8, 0x153D50FD, 0x5DD82A6C, 0x3B0DE90D, 0x357C91E3,
    0x939ABD33, 0x2EBFD8CD, 0x239B591C, 0x384F7815, 0x34FB0AB8, 0x8F4DE4FF,
    0xA8891EE6, 0xEFF1D6C1, 0x5BD1794B, 0x14E83212, 0x48C68BBF, 0x73F13DBA,
    0x5E7157C4, 0x4D1C9876, 0x052B6C2E, 0xA0B25A50, 0xDCF2DFD9, 0x60D639F9,
    0xB7084DD6, 0xAE6035CD, 0xD196F1C7, 0xE7FEC55A, 0x83C84F2D, 0x15EF929D,
    0xEF0ADF5A, 0x3D38A449, 0xCDA26407, 0x6B5B5DB1, 0x4800F95A, 0x56BBD267,
    0x3C6B6E7B, 0x22BBF74C, 0xC466090A, 0x14C2631A, 0xD2FAC7F6, 0x8FB596C0,
    0xD1078CCA, 0x0DB6556B, 0x17DE8B7A, 0x29A57E3E, 0x038DB0E4, 0x0B3B2E5E,
    0xE48F05AE, 0x55796BA5, 0xF643FD88, 0xE4EAAA22, 0x44FE6CB0, 0xE56302BA,
    0x901895E4, 0x92A42FAC, 0xDF41EAEA, 0xA2F74B0D, 0xF21EDA01, 0x41E83E7D,
    0x2095234D, 0x9864832E, 0x1CB1F673, 0xAE9DF2AD, 0xEC059051, 0x39411920,
    0xB3A17F4F, 0xEA0DB827, 0x14FA97A4, 0x2B9000B0, 0xF2B7B55A, 0xF14C10FD,
    0x92DA65A7, 0xFE583EE4, 0xB2A119DF, 0xA85861B9, 0x560016C3, 0x920F1F31,
    0x498309A8, 0x24817CD3, 0xD66A9072, 0x1A1D66CF, 0xAEB8E5B4, 0x4C370940,
    0x1AAC5BDF, 0x3215047F, 0x79A30BB4, 0x32ECA027, 0x87BA2859, 0x595A28A2,
    0xCEF4F510, 0x99EEAEAB, 0x18C98F06, 0x02C19A18, 0x2BDE1B6D, 0x0B5DE67D,
    0x567CA114, 0x6ABA55E4, 0xCB7FCA0D, 0x51AFF32C, 0x2494C8DC, 0xAF436241,
    0xBDAD4523, 0x284AF720, 0xE0FC3990, 0x0F412B2F, 0x1279DAC3, 0xFB0BC2D0,
    0xD01D0F36, 0xBF44CCF7, 0x225B2540, 0x7FDA7FDE, 0x87518696, 0x3B52162A,
    0x8B96CD83, 0x2052AB1F, 0x2E97D148, 0x10EF8380, 0xA76427D0, 0x9B28F7A7,
    0xEBB8EE36, 0x62F003DD, 0xB1CE3550, 0x7B061BA6, 0xDADCE4AE, 0x95DDF917,
    0x5C0F8E51, 0x1859F08B, 0x7E7CD404, 0x86ED6A2A, 0x205FD455, 0x73C9366C,
    0x19455B7E, 0xCF80B853, 0x53A95CFD, 0x8863ABDE, 0xEE79BFEE, 0x2C25EFFE,
    0x1A9E80BE, 0x9370D190, 0xAE69CA13, 0x9B6D75BF, 0x2BDA1D98, 0xA015F7F3,
    0x65582889, 0x94CE4475, 0xF2D4832E, 0xD7E8D3A7, 0xB25E7FB4, 0xC2467DD4,
    0x5E738E4B, 0xCB79BD05, 0x1772A216, 0x34353920, 0xCC0F7B2C, 0x8E460BCE,
    0xFEE146B0, 0x38B17369, 0x84B097BF, 0x415A9B12, 0x63AF69AC, 0xA9D4568C,
    0x416774F2, 0x9AB034D3, 0x36C48605, 0x1D04C155, 0x408C207E, 0x54B2613D,
    0x32248E16, 0x2643A0F9, 0x8CBD2212, 0x4613CE5E, 0x13ADCED3, 0x0E20069E,
    0xA015203F, 0x6346DE6C, 0xD2D6E22C, 0xDB5930D8, 0x63F20C2B, 0xB0C6955B,
    0x53353A3F, 0x44ECD412, 0x425D7C14, 0xC15AE582, 0x3ED8BD01, 0x208DA797,
    0x9220675B, 0xA4279A67, 0x03DE8E53, 0x9C8A6866, 0xF592E2E7, 0xE645C52F,
    0xEDD42F59, 0xEDA5B8C3, 0x00AC09D3, 0x71A7C674, 0xC9CADBA5, 0x41C36B7E,
    0x2A897B04, 0xD1C53A6E, 0xCD9C6662, 0xB779C287, 0xD5C13F33, 0x417AE0AE,
    0x20FE91A0, 0x9D18C93C, 0x845FB9C2, 0x323C00B6, 0x9C4CF4A8, 0x2C1F6676,
    0x26E4953B, 0x7119E67D, 0xD6FB1FCE, 0x265939B0
  }
};
unsigned long blowfishsubkey[] =
{
    0x243F6A88, 0x85A308D3, 0x13198A2E, 0x03707344,
    0xA4093822, 0x299F31D0, 0x082EFA98, 0xEC4E6C89,
    0x452821E6, 0x38D01377, 0xBE5466CF, 0x34E90C6C,
    0xC0AC29B7, 0xC97C50DD, 0x3F84D5B5, 0xB5470917,
    0x9216D5D9, 0x8979FB1B, 0xD1310BA6, 0x98DFB5AC,
    0x2FFD72DB, 0xD01ADFB7, 0xB8E1AFED, 0x6A267E96,
    0xBA7C9045, 0xF12C7F99, 0x24A19947, 0xB3916CF7,
    0x0801F2E2, 0x858EFC16, 0x636920D8, 0x71574E69,
    0xA458FEA3, 0xF4933D7E, 0x0D95748F, 0x728EB658,
    0x718BCD58, 0x82154AEE, 0x7B54A41D, 0xC25A59B5,
    0x9C30D539, 0x2AF26013, 0xC5D1B023, 0x286085F0,
    0xCA417918, 0xB8DB38EF, 0x8E79DCB0, 0x603A180E,
    0x6C9E0E8B, 0xB01E8A3E, 0xD71577C1, 0xBD314B27,
    0x78AF2FDA, 0x55605C60, 0xE65525F3, 0xAA55AB94,
    0x57489862, 0x63E81440, 0x55CA396A, 0x2AAB10B6,
    0xB4CC5C34, 0x1141E8CE, 0xA15486AF, 0x7C72E993,
    0xB3EE1411, 0x636FBC2A, 0x2BA9C55D, 0x741831F6,
    0xCE5C3E16, 0x9B87931E, 0xAFD6BA33, 0x6C24CF5C,
    0x7A325381, 0x28958677, 0x3B8F4898, 0x6B4BB9AF,
    0xC4BFE81B, 0x66282193, 0x61D809CC, 0xFB21A991,
    0x487CAC60, 0x5DEC8032, 0xEF845D5D, 0xE98575B1,
    0xDC262302, 0xEB651B88, 0x23893E81, 0xD396ACC5,
    0x0F6D6FF3, 0x83F44239, 0x2E0B4482, 0xA4842004,
    0x69C8F04A, 0x9E1F9B5E, 0x21C66842, 0xF6E96C9A,
    0x670C9C61, 0xABD388F0, 0x6A51A0D2, 0xD8542F68,
    0x960FA728, 0xAB5133A3, 0x6EEF0B6C, 0x137A3BE4,
    0xBA3BF050, 0x7EFB2A98, 0xA1F1651D, 0x39AF0176,
    0x66CA593E, 0x82430E88, 0x8CEE8619, 0x456F9FB4,
    0x7D84A5C3, 0x3B8B5EBE, 0xE06F75D8, 0x85C12073,
    0x401A449F, 0x56C16AA6, 0x4ED3AA62, 0x363F7706,
    0x1BFEDF72, 0x429B023D, 0x37D0D724, 0xD00A1248,
    0xDB0FEAD3, 0x49F1C09B, 0x075372C9, 0x80991B7B,
    0x25D479D8, 0xF6E8DEF7, 0xE3FE501A, 0xB6794C3B,
    0x976CE0BD, 0x04C006BA, 0xC1A94FB6, 0x409F60C4,
    0x5E5C9EC2, 0x196A2463, 0x68FB6FAF, 0x3E6C53B5,
    0x1339B2EB, 0x3B52EC6F, 0x6DFC511F, 0x9B30952C,
    0xCC814544, 0xAF5EBD09, 0xBEE3D004, 0xDE334AFD,
    0x660F2807, 0x192E4BB3, 0xC0CBA857, 0x45C8740F,
    0xD20B5F39, 0xB9D3FBDB, 0x5579C0BD, 0x1A60320A,
    0xD6A100C6, 0x402C7279, 0x679F25FE, 0xFB1FA3CC,
    0x8EA5E9F8, 0xDB3222F8, 0x3C7516DF, 0xFD616B15,
```

```
    0x2F501EC8, 0xAD0552AB, 0x323DB5FA, 0xFD238760,
    0x53317B48, 0x3E00DF82, 0x9E5C57BB, 0xCA6F8CA0,
    0x1A87562E, 0xDF1769DB, 0xD542A8F6, 0x287EFFC3,
    0xAC6732C6, 0x8C4F5573, 0x695B27B0, 0xBBCA58C8,
    0xE1FFA35D, 0xB8F011A0, 0x10FA3D98, 0xFD2183B8,
    0x4AFCB56C, 0x2DD1D35B, 0x9A53E479, 0xB6F84565,
    0xD28E49BC, 0x4BFB9790, 0xE1DDF2DA, 0xA4CB7E33,
    0x62FB1341, 0xCEE4C6E8, 0xEF20CADA, 0x36774C01,
    0xD07E9EFE, 0x2BF11FB4, 0x95DBDA4D, 0xAE909198,
    0xEAAD8E71, 0x6B93D5A0, 0xD08ED1D0, 0xAFC725E0,
    0x8E3C5B2F, 0x8E7594B7, 0x8FF6E2FB, 0xF2122B64,
    0x8888B812, 0x900DF01C, 0x4FAD5EA0, 0x688FC31C,
    0xD1CFF191, 0xB3A8C1AD, 0x2F2F2218, 0xBE0E1777,
    0xEA752DFE, 0x8B021FA1, 0xE5A0CC0F, 0xB56F74E8,
    0x18ACF3D6, 0xCE89E299, 0xB4A84FE0, 0xFD13E0B7,
    0x7CC43B81, 0xD2ADA8D9, 0x165FA266, 0x80957705,
    0x93CC7314, 0x211A1477, 0xE6AD2065, 0x77B5FA86,
    0xC75442F5, 0xFB9D35CF, 0xEBCDAF0C, 0x7B3E89A0,
    0xD6411BD3, 0xAE1E7E49, 0x00250E2D, 0x2071B35E,
    0x226800BE, 0x57B8E0AF, 0x2464369B, 0xF009B91E,
    0x5563911D, 0x59DFA6AA, 0x78C14389, 0xD95A537F,
    0x207D5BA2, 0x02E5B9C5, 0x83260376, 0x6295CFA9,
    0x11C81968, 0x4E734A41, 0xB3472DCA, 0x7B14A94A,
    0x1B510052, 0x9A532915, 0xD60F573F, 0xBC9BC6E4,
    0x2B60A476, 0x81E67400, 0x08BA6FB5, 0x571BE91F,
    0xF296EC6B, 0x2A0DD915, 0xB6636521, 0xE7B9F9B6,
    0xFF34052E, 0xC5855664, 0x53B02D5D, 0xA99F8FA1,
    0x08BA4799, 0x6E85076A, 0x4B7A70E9, 0xB5B32944,
    0xDB75092E, 0xC4192623, 0xAD6EA6B0, 0x49A7DF7D,
    0x9CEE60B8, 0x8FEDB266, 0xECAA8C71, 0x699A17FF,
    0x5664526C, 0xC2B19EE1, 0x193602A5, 0x75094C29,
    0xA0591340, 0xE4183A3E, 0x3F54989A, 0x5B429D65,
    0x6B8FE4D6, 0x99F73FD6, 0xA1D29C07, 0xEFE830F5,
    0x4D2D38E6, 0xF0255DC1, 0x4CDD2086, 0x8470EB26,
    0x6382E9C6, 0x021ECC5E, 0x09686B3F, 0x3EBAEFC9,
    0x3C971814, 0x6B6A70A1, 0x687F3584, 0x52A0E286,
    0xB79C5305, 0xAA500737, 0x3E07841C, 0x7FDEAE5C,
    0x8E7D44EC, 0x5716F2B8, 0xB03ADA37, 0xF0500C0D,
    0xF01C1F04, 0x0200B3FF, 0xAE0CF51A, 0x3CB574B2,
    0x25837A58, 0xDC0921BD, 0xD19113F9, 0x7CA92FF6,
    0x94324773, 0x22F54701, 0x3AE5E581, 0x37C2DADC,
    0xC8B57634, 0x9AF3DDA7, 0xA9446146, 0x0FD0030E,
    0xECC8C73E, 0xA4751E41, 0xE238CD99, 0x3BEA0E2F,
    0x3280BBA1, 0x183EB331, 0x4E548B38, 0x4F6DB908,
    0x6F420D03, 0xF60A04BF, 0x2CB81290, 0x24977C79,
    0x5679B072, 0xBCAF89AF, 0xDE9A771F, 0xD9930810,
    0xB38BAE12, 0xDCCF3F2E, 0x5512721F, 0x2E6B7124,
    0x501ADDE6, 0x9F84CD87, 0x7A584718, 0x7408DA17,
    0xBC9F9ABC, 0xE94B7D8C, 0xEC7AEC3A, 0xDB851DFA,
    0x63094366, 0xC464C3D2, 0xEF1C1847, 0x3215D908,
    0xDD433B37, 0x24C2BA16, 0x12A14D43, 0x2A65C451,
    0x50940002, 0x133AE4DD, 0x71DFF89E, 0x10314E55,
    0x81AC77D6, 0x5F11199B, 0x043556F1, 0xD7A3C76B,
    0x3C11183B, 0x5924A509, 0xF28FE6ED, 0x97F1FBFA,
    0x9EBABF2C, 0x1E153C6E, 0x86E34570, 0xEAE96FB1,
    0x860E5E0A, 0x5A3E2AB3, 0x771FE71C, 0x4E3D06FA,
    0x2965DCB9, 0x99E71D0F, 0x803E89D6, 0x5266C825,
    0x2E4CC978, 0x9C10B36A, 0xC6150EBA, 0x94E2EA78,
    0xA5FC3C53, 0x1E0A2DF4, 0xF2F74EA7, 0x361D2B3D,
    0x1939260F, 0x19C27960, 0x5223A708, 0xF71312B6,
    0xEBADFE6E, 0xEAC31F66, 0xE3BC4595, 0xA67BC883,
    0xB17F37D1, 0x018CFF28, 0xC332DDEF, 0xBE6C5AA5,
    0x65582185, 0x68AB9802, 0xEECEA50F, 0xDB2F953B,
    0x2AEF7DAD, 0x5B6E2F84, 0x1521B628, 0x29076170,
    0xECDD4775, 0x619F1510, 0x13CCA830, 0xEB61BD96,
    0x0334FE1E, 0xAA0363CF, 0xB5735C90, 0x4C70A239,
    0xD59E9E0B, 0xCBAADE14, 0xEECC86BC, 0x60622CA7,
    0x9CAB5CAB, 0xB2F3846E, 0x648B1EAF, 0x19BDF0CA,
    0xA02369B9, 0x655ABB50, 0x40685A32, 0x3C2AB4B3,
    0x319EE9D5, 0xC021B8F7, 0x9B540B19, 0x875FA099,
    0x95F7997E, 0x623D7DA8, 0xF837889A, 0x97E32D77,
    0x11ED935F, 0x16681281, 0x0E358829, 0xC7E61FD6,
    0x96DEDFA1, 0x7858BA99, 0x57F584A5, 0x1B227263,
    0x9B83C3FF, 0x1AC24696, 0xCDB30AEB, 0x532E3054,
    0x8FD948E4, 0x6DBC3128, 0x58EBF2EF, 0x34C6FFEA,
    0xFE28ED61, 0xEE7C3C73, 0x5D4A14D9, 0xE864B7E3,
    0x42105D14, 0x203E13E0, 0x45EEE2B6, 0xA3AAABEA,
    0xDB6C4F15, 0xFACB4FD0, 0xC742F442, 0xEF6ABBB5,
    0x654F3B1D, 0x41CD2105, 0xD81E799E, 0x86854DC7,
    0xE44B476A, 0x3D816250, 0xCF62A1F2, 0x5B8D2646,
    0xFC8883A0, 0xC1C7B6A3, 0x7F1524C3, 0x69CB7492,
    0x47848A0B, 0x5692B285, 0x095BBF00, 0xAD19489D,
    0x1462B174, 0x23820E00, 0x58428D2A, 0x0C55F5EA,
    0x1DADF43E, 0x233F7061, 0x3372F092, 0x8D937E41,
    0xD65FECF1, 0x6C223BDB, 0x7CDE3759, 0xCBEE7460,
    0x4085F2A7, 0xCE77326E, 0xA6078084, 0x19F8509E,
    0xE8EFD855, 0x61D99735, 0xA969A7AA, 0xC50C06C2,
    0x5A04ABFC, 0x800BCADC, 0x9E447A2E, 0xC3453484,
    0xFDD56705, 0x0E1E9EC9, 0xDB73DBD3, 0x105588CD,
    0x675FDA79, 0xE3674340, 0xC5C43465, 0x713E38D8,
    0x3D28F89E, 0xF16DFF20, 0x153E21E7, 0x8FB03D4A,
    0xE6E39F2B, 0xDB83ADF7, 0xE93D5A68, 0x948140F7,
    0xF64C261C, 0x94692934, 0x411520F7, 0x7602D4F7,
    0xBCF46B2E, 0xD4A20068, 0xD4082471, 0x3320F46A,
    0x43B7D4B7, 0x500061AF, 0x1E39F62E, 0x97244546,
```

```
    0x14214F74, 0xBF8B8840, 0x4D95FC1D, 0x96B591AF,
    0x70F4DDD3, 0x66A02F45, 0xBFBC09EC, 0x03BD9785,
    0x7FAC6DD0, 0x31CB8504, 0x96EB27B3, 0x55FD3941,
    0xDA2547E6, 0xABCA0A9A, 0x28507825, 0x530429F4,
    0x0A2C86DA, 0xE9B66DFB, 0x68DC1462, 0xD7486900,
    0x680EC0A4, 0x27A18DEE, 0x4F3FFEA2, 0xE887AD8C,
    0xB58CE006, 0x7AF4D6B6, 0xAACE1E7C, 0xD3375FEC,
    0xCE78A399, 0x406B2A42, 0x20FE9E35, 0xD9F385B9,
    0xEE39D7AB, 0x3B124E8B, 0x1DC9FAF7, 0x4B6D1856,
    0x26A36631, 0xEAE397B2, 0x3A6EFA74, 0xDD5B4332,
    0x6841E7F7, 0xCA7820FB, 0xFB0AF54E, 0xD8FEB397,
    0x454056AC, 0xBA489527, 0x55533A3A, 0x20838D87,
    0xFE6BA9B7, 0xD096954B, 0x55A867BC, 0xA1159A58,
    0xCCA92963, 0x99E1DB33, 0xA62A4A56, 0x3F3125F9,
    0x5EF47E1C, 0x9029317C, 0xFDF8E802, 0x04272F70,
    0x80BB155C, 0x05282CE3, 0x95C11548, 0xE4C66D22,
    0x48C1133F, 0xC70F86DC, 0x07F9C9EE, 0x41041F0F,
    0x404779A4, 0x5D886E17, 0x325F51EB, 0xD59BC0D1,
    0xF2BCC18F, 0x41113564, 0x257B7834, 0x602A9C60,
    0xDFF8E8A3, 0x1F636C1B, 0x0E12B4C2, 0x02E1329E,
    0xAF664FD1, 0xCAD18115, 0x6B2395E0, 0x333E92E1,
    0x3B240B62, 0xEEBEB922, 0x85B2A20E, 0xE6BA0D99,
    0xDE720C8C, 0x2DA2F728, 0xD0127845, 0x95B794FD,
    0x647D0862, 0xE7CCF5F0, 0x5449A36F, 0x877D48FA,
    0xC39DFD27, 0xF33E8D1E, 0x0A476341, 0x992EFF74,
    0x3A6F6EAB, 0xF4F8FD37, 0xA812DC60, 0xA1EBDDF8,
    0x991BE14C, 0xDB6E6B0D, 0xC67B5510, 0x6D672C37,
    0x2765D43B, 0xDCD0E804, 0xF1290DC7, 0xCC00FFA3,
    0xB5390F92, 0x690FED0B, 0x667B9FFB, 0xCEDB7D9C,
    0xA091CF0B, 0xD9155EA3, 0xBB132F88, 0x515BAD24,
    0x7B9479BF, 0x763BD6EB, 0x37392EB3, 0xCC115979,
    0x8026E297, 0xF42E312D, 0x6842ADA7, 0xC66A2B3B,
    0x12754CCC, 0x782EF11C, 0x6A124237, 0xB79251E7,
    0x06A1BBE6, 0x4BFB6350, 0x1A6B1018, 0x11CAEDFA,
    0x3D25BDD8, 0xE2E1C3C9, 0x44421659, 0x0A121386,
    0xD90CEC6E, 0xD5ABEA2A, 0x64AF674E, 0xDA86A85F,
    0xBEBFE988, 0x64E4C3FE, 0x9DBC8057, 0xF0F7C086,
    0x60787BF8, 0x6003604D, 0xD1FD8346, 0xF6381FB0,
    0x7745AE04, 0xD736FCCC, 0x83426B33, 0xF01EAB71,
    0xB0804187, 0x3C005E5F, 0x77A057BE, 0xBDE8AE24,
    0x55464299, 0xBF582E61, 0x4E58F48F, 0xF2DDFDA2,
    0xF474EF38, 0x8789BDC2, 0x5366F9C3, 0xC8B38E74,
    0xB475F255, 0x46FCD9B9, 0x7AEB2661, 0x8B1DDF84,
    0x846A0E79, 0x915F95E2, 0x466E598E, 0x20B45770,
    0x8CD55591, 0xC902DE4C, 0xB90BACE1, 0xBB8205D0,
    0x11A86248, 0x7574A99E, 0xB77F19B6, 0xE0A9DC09,
    0x662D09A1, 0xC4324633, 0xE85A1F02, 0x09F0BE8C,
    0x4A99A025, 0x1D6EFE10, 0x1AB93D1D, 0x0BA5A4DF,
    0xA186F20F, 0x2868F169, 0xDCB7DA83, 0x573906FE,
    0xA1E2CE9B, 0x4FCD7F52, 0x50115E01, 0xA70683FA,
    0xA002B5C4, 0x0DE6D027, 0x9AF88C27, 0x773F8641,
    0xC3604C06, 0x61A806B5, 0xF0177A28, 0xC0F586E0,
    0x006058AA, 0x30DC7D62, 0x11E69ED7, 0x2338EA63,
    0x53C2DD94, 0xC2C21634, 0xBBCBEE56, 0x90BCB6DE,
    0xEBFC7DA1, 0xCE591D76, 0x6F05E409, 0x4B7C0188,
    0x39720A3D, 0x7C927C24, 0x86E3725F, 0x724D9DB9,
    0x1AC15BB4, 0xD39EB8FC, 0xED545578, 0x08FCA5B5,
    0xD83D7CD3, 0x4DAD0FC4, 0x1E50EF5E, 0xB161E6F8,
    0xA28514D9, 0x6C51133C, 0x6FD5C7E7, 0x56E14EC4,
    0x362ABFCE, 0xDDC6C837, 0xD79A3234, 0x92638212,
    0x670EFA8E, 0x406000E0, 0x3A39CE37, 0xD3FAF5CF,
    0xABC27737, 0x5AC52D1B, 0x5CB0679E, 0x4FA33742,
    0xD3822740, 0x99BC9BBE, 0xD5118E9D, 0xBF0F7315,
    0xD62D1C7E, 0xC700C47B, 0xB78C1B6B, 0x21A19045,
    0xB26EB1BE, 0x6A366EB4, 0x5748AB2F, 0xBC946E79,
    0xC6A376D2, 0x6549C2C8, 0x530FF8EE, 0x468DDE7D,
    0xD5730A1D, 0x4CD04DC6, 0x2939BBDB, 0xA9BA4650,
    0xAC9526E8, 0xBE5EE304, 0xA1FAD5F0, 0x6A2D519A,
    0x63EF8CE2, 0x9A86EE22, 0xC089C2B8, 0x43242EF6,
    0xA51E03AA, 0x9CF2D0A4, 0x83C061BA, 0x9BE96A4D,
    0x8FE51550, 0xBA645BD6, 0x2826A2F9, 0xA73A3AE1,
    0x4BA99586, 0xEF5562E9, 0xC72FEFD3, 0xF752F7DA,
    0x3F046F69, 0x77FA0A59, 0x80E4A915, 0x87B08601,
    0x9B09E6AD, 0x3B3EE593, 0xE990FD5A, 0x9E34D797,
    0x2CF0B7D9, 0x022B8B51, 0x96D5AC3A, 0x017DA67D,
    0xD1CF3ED6, 0x7C7D2D28, 0x1F9F25CF, 0xADF2B89B,
    0x5AD6B472, 0x5A88F54C, 0xE029AC71, 0xE019A5E6,
    0x47B0ACFD, 0xED93FA9B, 0xE8D3C48D, 0x283B57CC,
    0xF8D56629, 0x79132E28, 0x785F0191, 0xED756055,
    0xF7960E44, 0xE3D35E8C, 0x15056DD4, 0x88F46DBA,
    0x03A16125, 0x0564F0BD, 0xC3EB9E15, 0x3C9057A2,
    0x97271AEC, 0xA93A072A, 0x1B3F6D9B, 0x1E6321F5,
    0xF59C66FB, 0x26DCF319, 0x7533D928, 0xB155FDF5,
    0x03563482, 0x8ABA3CBB, 0x28517711, 0xC20AD9F8,
    0xABCC5167, 0xCCAD925F, 0x4DE81751, 0x3830DC8E,
    0x379D5862, 0x9320F991, 0xEA7A90C2, 0xFB3E7BCE,
    0x5121CE64, 0x774FBE32, 0xA8B6E37E, 0xC3293D46,
    0x48DE5369, 0x6413E680, 0xA2AE0810, 0xDD6DB224,
    0x69852DFD, 0x09072166, 0xB39A460A, 0x6445C0DD,
    0x586CDECF, 0x1C20C8AE, 0x5BBEF7DD, 0x1B588D40,
    0xCCD2017F, 0x6BB4E3BB, 0xDDA26A7E, 0x3A59FF45,
    0x3E350A44, 0xBCB4CDD5, 0x72EACEA8, 0xFA6484BB,
    0x8D6612AE, 0xBF3C6F47, 0xD29BE463, 0x542F5D9E,
    0xAEC2771B, 0xF64E6370, 0x740E0D8D, 0xE75B1357,
    0xF8721671, 0xAF537D5D, 0x4040CB08, 0x4EB4E2CC,
```

```
    0x34D2466A, 0x0115AF84, 0xE1B00428, 0x95983A1D,
    0x06B89FB4, 0xCE6EA048, 0x6F3F3B82, 0x3520AB82,
    0x011A1D4B, 0x277227F8, 0x611560B1, 0xE7933FDC,
    0xBB3A792B, 0x344525BD, 0xA08839E1, 0x51CE794B,
    0x2F32C9B7, 0xA01FBAC9, 0xE01CC87E, 0xBCC7D1F6,
    0xCF0111C3, 0xA1E8AAC7, 0x1A908749, 0xD44FBD9A,
    0xD0DADECB, 0xD50ADA38, 0x0339C32A, 0xC6913667,
    0x8DF9317C, 0xE0B12B4F, 0xF79E59B7, 0x43F5BB3A,
    0xF2D519FF, 0x27D9459C, 0xBF97222C, 0x15E6FC2A,
    0x0F91FC71, 0x9B941525, 0xFAE59361, 0xCEB69CEB,
    0xC2A86459, 0x12BAA8D1, 0xB6C1075E, 0xE3056A0C,
    0x10D25065, 0xCB03A442, 0xE0EC6E0E, 0x1698DB3B,
    0x4C98A0BE, 0x3278E964, 0x9F1F9532, 0xE0D392DF,
    0xD3A0342B, 0x8971F21E, 0x1B0A7441, 0x4BA3348C,
    0xC5BE7120, 0xC37632D8, 0xDF359F8D, 0x9B992F2E,
    0xE60B6F47, 0x0FE3F11D, 0xE54CDA54, 0x1EDAD891,
    0xCE6279CF, 0xCD3E7E6F, 0x1618B166, 0xFD2C1D05,
    0x848FD2C5, 0xF6FB2299, 0xF523F357, 0xA6327623,
    0x93A83531, 0x56CCCD02, 0xACF08162, 0x5A75EBB5,
    0x6E163697, 0x88D273CC, 0xDE966292, 0x81B949D0,
    0x4C50901B, 0x71C65614, 0xE6C6C7BD, 0x327A140A,
    0x45E1D006, 0xC3F27B9A, 0xC9AA53FD, 0x62A80F00,
    0xBB25BFE2, 0x35BDD2F6, 0x71126905, 0xB2040222,
    0xB6CBCF7C, 0xCD769C2B, 0x53113EC0, 0x1640E3D3,
    0x38ABBD60, 0x2547ADF0, 0xBA38209C, 0xF746CE76,
    0x77AFA1C5, 0x20756060, 0x85CBFE4E, 0x8AE88DD8,
    0x7AAAF9B0, 0x4CF9AA7E, 0x1948C25C, 0x02FB8A8C,
    0x01C36AE4, 0xD6EBE1F9, 0x90D4F869, 0xA65CDEA0,
    0x3F09252D, 0xC208E69F, 0xB74E6132, 0xCE77E25B,
    0x578FDFE3, 0x3AC372E6, 0xB83ACB02, 0x2002397A,
    0x6EC6FB5B, 0xFFCFD4DD, 0x4CBF5ED1, 0xF43FE582,
    0x3EF4E823, 0x2D152AF0, 0xE718C970, 0x59BD9820,
    0x1F4A9D62, 0xE7A529BA, 0x89E1248D, 0x3BF88656,
    0xC5114D0E, 0xBC4CEE16, 0x034D8A39, 0x20E47882,
    0xE9AE8FBD, 0xE3ABDC1F, 0x6DA51E52, 0x5DB2BAE1,
    0x01F86E7A, 0x6D9C68A9, 0x2708FCD9, 0x293CBC0C,
    0xB03C86F8, 0xA8AD2C2F, 0x00424EEB, 0xCACB452D,
    0x89CC71FC, 0xD59C7F91, 0x7F0622BC, 0x6D8A08B1,
    0x834D2132, 0x6884CA82, 0xE3AACBF3, 0x7786F2FA,
    0x2CAB6E3D, 0xCE535AD1, 0xF20AC607, 0xC6B8E14F,
    0x5EB4388E, 0x775014A6, 0x656665F7, 0xB64A43E4,
    0xBA383D01, 0xB2E41079, 0x8EB2986F, 0x909E0CA4,
    0x1F7B3777, 0x2C126030, 0x85088718, 0xC4E7D1BD,
    0x4065FFCE, 0x8392FD8A, 0xAA36D12B, 0xB4C8C9D0,
    0x994FB0B7, 0x14F96818, 0xF9A53998, 0xA0A178C6,
    0x2684A81E, 0x8AE972F6, 0xB8425EB6, 0x7A29D486,
    0x551BD719, 0xAF32C189, 0xD5145505, 0xDC81D53E,
    0x48424EDA, 0xB796EF46, 0xA0498F03, 0x667DEEDE,
    0x03AC0AB3, 0xC497733D, 0x5316A891, 0x30A88FCC,
    0x9604440A, 0xCEEB893A, 0x7725B82B, 0x0E1EF69D,
    0x302A5C8E, 0xE7B84DEF, 0x5A31B096, 0xC9EBF88D,
    0x512D788E, 0x7E4002EE, 0x87E02AF6, 0xC358A1BB,
    0x02E8D7AF, 0xDF9FB0E7, 0x790E942A, 0x3B3C1ABA,
    0xC6FFA7AF, 0x9DF796F9, 0x321BB994, 0x0174A8A8,
    0xED22162C, 0xCFF1BB99, 0xDAA8D551, 0xA4D5E44B,
    0xECDDE3EC, 0xA80DC509, 0x0393EEF2, 0x72523D31,
    0xD48E3A1C, 0x224EB65E, 0x6052C3A4, 0x2109C32F,
    0x052EE388, 0xED9F7EA9, 0x91C62F97, 0x77B55BA0,
    0x150CBCA3, 0x3AEC6525, 0xDF318383, 0x43A9CE26,
    0x9362AD8B, 0x0134140B, 0x8DF5CF81, 0x1E9FF559,
    0x167F0564, 0x3812F4E0, 0x588A52B0, 0xCBB8E944,
    0xEF5B16A3, 0x73C4EDA1, 0x7DFCFEEA, 0xF54BCBBE,
    0x8773E3D2, 0xC531DCD0, 0x55C46729, 0x52774F3A,
    0x57CA6BC0, 0x467D3A3B, 0x24778425, 0xB7991E9A,
    0xDD825C26, 0xE452C8EE, 0xFCACDE1E, 0x84833AF3,
    0x61211D03, 0x1732C131, 0xCCADB247, 0xE606BE8C,
    0x712B39F1, 0x88B4EF39, 0x3A9FCDC5, 0xC5755169,
    0x1FF6994F, 0x39829CB0, 0x11016573, 0x3343CBEB,
    0x61D3D0B4, 0x44F30AEF, 0xA8AE7375, 0x2A3A1C9D,
    0xB4B70914, 0xD6AB250C, 0x853B7328, 0x495F948F,
    0xD2A4ED8E, 0x6CF751E4, 0xC320BB75, 0xD9CAA0B3,
    0x8BA56262, 0x4E84B03F, 0xEEA8076E, 0x74A07FE5,
    0x8039E00C, 0x36FFDAF8, 0x03731358, 0xB9E671B9,
    0xDAC4CE1C, 0xB25B10ED, 0x4DD3D5B1, 0xFCF2B480,
    0x4634F579, 0x25EAC400, 0xA9AC55EA, 0x728932DF,
    0x06041D05, 0x5D31F502, 0xC539C2E3, 0x2B89D9DB,
    0x5BCC0A98, 0xC05BFD6F, 0x1B250622, 0x2E21BE0E,
    0x60973B04, 0xECD54A67, 0xB54FE638, 0xA6ED6615,
    0x981A910A, 0x5D92928D, 0xAC6FC697, 0xE73C63AD,
    0x456EDF5F, 0x457A8145, 0x51875A64, 0xCD3099F1,
    0x69B5F18A, 0x8C73EE0B, 0x5E57368F, 0x6C79F4BB,
    0x7A595926, 0xAAB49EC6, 0x8AC8FCFB, 0x8000
};

unsigned long F(unsigned long x)
{
    unsigned short a;
    unsigned short b;
    unsigned short c;
    unsigned short d;
    unsigned long  y;

    d = x & 0x00FF;
    x >>= 8;
    c = x & 0x00FF;
```

```
    x >>= 8;
    b = x & 0x00FF;
    x >>= 8;
    a = x & 0x00FF;
    //y = ((S[0][a] + S[1][b]) ^ S[2][c]) + S[3][d];
    y = S[0][a] + S[1][b];
    y = y ^ S[2][c];
    y = y + S[3][d];

    return y;
}

void Blowfish_encipher(unsigned long *xl, unsigned long *xr)
{
    unsigned long  Xl;
    unsigned long  Xr;
    unsigned long  temp;
    short          i;

    Xl = *xl;
    Xr = *xr;

    for (i = 0; i < N; ++i) {
        Xl = Xl ^ P[i];
        Xr = F(Xl) ^ Xr;

        temp = Xl;
        Xl = Xr;
        Xr = temp;
    }

    temp = Xl;
    Xl = Xr;
    Xr = temp;

    Xr = Xr ^ P[N];
    Xl = Xl ^ P[N + 1];

    *xl = Xl;
    *xr = Xr;
}

void Blowfish_decipher(unsigned long *xl, unsigned long *xr)
{
    unsigned long  Xl;
    unsigned long  Xr;
    unsigned long  temp;
    short          i;

    Xl = *xl;
    Xr = *xr;

    for (i = N + 1; i > 1; --i) {
        Xl = Xl ^ P[i];
        Xr = F(Xl) ^ Xr;

        /* Exchange Xl and Xr */
        temp = Xl;
        Xl = Xr;
        Xr = temp;
    }

    /* Exchange Xl and Xr */
    temp = Xl;
    Xl = Xr;
    Xr = temp;

    Xr = Xr ^ P[1];
    Xl = Xl ^ P[0];

    *xl = Xl;
    *xr = Xr;
}

short InitializeBlowfish(char key[], short keybytes)
{
    short          h = 0; //Added by JS for loop control inside blowfishsubkey array
    short          i;
    short          j;
    short          k;
    short          error = 0;
    short          numread = 1;
    unsigned long  data;
    unsigned long  datal;
    unsigned long  datar;

    /* First, open the file containing the array initialization data */
    for (i = 0; i < N + 2; ++i)
    {
        data = blowfishsubkey[h];
    #ifdef little_endian      /* Eg: Intel   We want to process things in byte   */
        /*   order, not as rearranged in a longword          */
        data = ((data & 0xFF000000) >> 24) |
                ((data & 0x00FF0000) >>  8) |
                ((data & 0x0000FF00) <<  8) |
```

```
                ((data & 0x000000FF) << 24);
    #endif

        if (numread != 1)
        {
            return DATAERROR;
        }
        else
        {
            P[i] = data;
        }
        h++;
    } // end for LOOP N+2

    for (i = 0; i < 4; ++i)
    {
        for (j = 0; j < 256; ++j, h++)
        {

            data = blowfishsubkey[h];
      #ifdef little_endian      /* Eg: Intel   We want to process things in byte   */
            /*   order, not as rearranged in a longword          */
            data = ((data & 0xFF000000) >> 24) |
                    ((data & 0x00FF0000) >>  8) |
                    ((data & 0x0000FF00) <<  8) |
                    ((data & 0x000000FF) << 24);
      #endif

            if (numread != 1)
            {
                return DATAERROR;
            }
            else
            {
                S[i][j] = data;
            }
        } // end for LOOP 256
    } //end for LOOP 4

    j = 0;
    for (i = 0; i < N + 2; ++i)
    {
        data = 0x00000000;
        for (k = 0; k < 4; ++k)
        {
            data = (data << 8);
            data |= (unsigned long) key[j]&0xFF; //DONE: CHOKE ISSUE AS NOTED ON WEBSITE
            j = j + 1;
            if (j >= keybytes)
            {
                j = 0;
            }
        }
        P[i] = P[i] ^ data;
    }

    datal = 0x00000000;
    datar = 0x00000000;

    for (i = 0; i < N + 2; i += 2)
    {
        Blowfish_encipher(&datal, &datar);

        P[i] = datal;
        P[i + 1] = datar;
    }

    for (i = 0; i < 4; ++i)
    {
        for (j = 0; j < 256; j += 2)
        {

            Blowfish_encipher(&datal, &datar);

            S[i][j] = datal;
            S[i][j + 1] = datar;
        }
    }

    return error;
}
```

i. CAN Implementation Code

The files relating to the CAN implementation without encryption are as follows:

SRBenchmark.c: The program file responsible for sending messages to the receiver in plaintext over CAN. The program runs on ECU #1.

```
#include <stdio.h>
```

```c
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <time.h>
#include <signal.h>
#include <ctype.h>
#include <libgen.h>
#include <net/if.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <sys/uio.h>
#include <sys/ioctl.h>
#include <net/if.h>
#include <linux/can.h>
#include <linux/can/raw.h>
#include "lib.h"

// Message length global
#define MESSAGE_LEN 20

// Print things like CAN frames and hex values. Set to 1 to enable printing to terminal.
#define PRINTING 0

// Global variables for socket used
struct sockaddr_can addrS;
struct ifreq ifrS;
int nbytesS;
int s;

// Structures for time gathering
struct timeval timeSent;
struct timeval timeRec;

// FILE pointer to times.csv
FILE *times;

// Misc globals
int i;

void sendFrame()
{
    // Message and CAN-related vars
    char temp[3];
    unsigned char MESSAGE[MESSAGE_LEN];
    strcpy(MESSAGE, "000#");
    struct can_frame frameS;

    // Blowfish-related vars
    u_int8_t entered_value[8] = {0xD,0xE,0xA,0xD,0xB,0xE,0xE,0xF};

    // Get the time before encryption process begins
    gettimeofday(&timeSent, NULL);

    // Construct the final CAN message
    for(i = 7; i >= 0; i--)
    {
        sprintf(temp, "%02X", entered_value[i]);
        strcat(MESSAGE, temp);
    }
    if(PRINTING)
    {
        printf("\nMESSAGE CAN: %s\n", MESSAGE);
    }

    // Parse and send the frame
    if (parse_canframe(MESSAGE, &frameS)){
        printf("Error parsing CAN frame\n");
        return;
    }
    if ((nbytesS = write(s, &frameS, sizeof(frameS))) != sizeof(frameS))
    {
        perror("write");
        return;
    }
    return;
}

void recFrame()
{
    // Read the socket
    struct can_frame frameR;
    size_t readSize = 0;
    while(readSize < 1)
    {
        readSize = read(s, &frameR, sizeof(struct can_frame));
    }

    // Message-related vars
    char canMes[20];

    // Place frame data into canMes array
    sprint_canframe(canMes, &frameR, 0);
```

```
    // Record the time received
    gettimeofday(&timeRec, NULL);

    // Save time sent and received to times.csv file
    fprintf(times, "%ld,%ld,%ld,%ld\n", timeSent.tv_sec, timeSent.tv_usec, timeRec.tv_sec, timeRec.tv_usec);
}

int main(int argc, char **argv)
{
    pid_t id = getpid();
    printf("ID: %d", id);

    // Prepare the socket
    s = socket(PF_CAN, SOCK_RAW, CAN_RAW);
    strcpy(ifrS.ifr_name, "can0" );
    if(ioctl(s, SIOCGIFINDEX, &ifrS) < 0)
        printf("Error sender");
    addrS.can_family = AF_CAN;
    addrS.can_ifindex = ifrS.ifr_ifindex;
    bind(s, (struct sockaddr *)&addrS, sizeof(addrS));

    // Set program to run for given number of times (default is 5000 runs)
    int numRuns, count;
    if (argc == 2)
        numRuns = atoi(argv[1]);
    else
        numRuns = 5000;

    // Open times.csv file for time-recording, add headings
    times = fopen("times.csv", "w+");
    fprintf(times, "Time Sent (s),Time Sent (us), Time Received (s),Time Received (us)\n");
    getchar();
    printf("Starting...\n");

    // Run for the given number of times
    count = 0;
    while(count < numRuns)
    {
        sendFrame();
        recFrame();
        count++;
    }

    // Close socket and file. Exit the program
    close(s);
    fclose(times);
    printf("Done! Times saved to times.csv file.\n");
    return 0;
}
```

The makefile for compilation is shown below:

```
all : SRBenchmark

clean :
    rm *.o SRBenchmark

SRBenchmark : lib.o SRBenchmark.o
    gcc -o SRBenchmark lib.o SRBenchmark.o

# The next lines generate the various object files

lib.o : lib.c
    gcc -c lib.c

SRBenchmark.o : SRBenchmark.c
    gcc  -c SRBenchmark.c
```

SRBenchmark_rec.c: The program file responsible for receiving plaintext messages from the sender over CAN. The program runs on ECU #2.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <time.h>
#include <signal.h>
#include <ctype.h>
#include <libgen.h>
#include <net/if.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <sys/uio.h>
#include <sys/ioctl.h>
#include <net/if.h>
#include <linux/can.h>
#include <linux/can/raw.h>
#include "lib.h"

// Print things like CAN frames and hex values. Set to 1 to enable printing to terminal.
#define PRINTING 0
```

```
// Global variables for socket used
struct sockaddr_can addrS;
struct ifreq ifrS;
struct can_frame frameS;
int nbytesS;
int s;

// Misc globals
int i;

void sendFrame(struct can_frame frameS)
{
    if ((nbytesS = write(s, &frameS, sizeof(frameS))) != sizeof(frameS))
    {
        perror("write");
        return;
    }
}

void recFrame()
{
    // Read the socket
    struct can_frame frameR;
    size_t readSize = 0;
    while(readSize < 1)
    {
        readSize = read(s, &frameR, sizeof(struct can_frame));
    }

    // Send the decrypted text to sendFrame for encryption and sending
    sendFrame(frameR);
}

int main(int argc, char **argv)
{
    // Prepare the socket
    s = socket(PF_CAN, SOCK_RAW, CAN_RAW);
    strcpy(ifrS.ifr_name, "can0" );
    if(ioctl(s, SIOCGIFINDEX, &ifrS) < 0)
        printf("Error sender");
    addrS.can_family = AF_CAN;
    addrS.can_ifindex = ifrS.ifr_ifindex;
    bind(s, (struct sockaddr *)&addrS, sizeof(addrS));

    // Set program to run for given number of times (default is 5000 runs)
    int numRuns, count;
    if (argc == 2)
        numRuns = atoi(argv[1]);
    else
        numRuns = 5000;

    printf("Starting...\n");

    // Run for the given number of times
    count = 0;
    while(count < numRuns)
    {
        recFrame();
        count++;
    }

    // Close socket and exit
    close(s);
    printf("Done!\n");
    return 0;
}
```

The makefile for compilation is shown below:

```
all : SRBenchmark_rec

clean :
    rm *.o SRBenchmark_rec

SRBenchmark_rec : lib.o SRBenchmark_rec.o
    gcc -o SRBenchmark_rec lib.o SRBenchmark_rec.o

# The next lines generate the various object files

lib.o : lib.c
    gcc -c lib.c


SRBenchmark_rec.o : SRBenchmark_rec.c
    gcc  -c SRBenchmark_rec.c
```

The files relating to the CAN implementation with encryption are as follows:

encSR_bf.c: The program file responsible for sending encrypted messages to the receiver over CAN. The program runs on ECU #1.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <time.h>
#include <signal.h>
#include <ctype.h>
#include <libgen.h>
#include <net/if.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <sys/uio.h>
#include <sys/ioctl.h>
#include <net/if.h>
#include <linux/can.h>
#include <linux/can/raw.h>
#include "lib.h"
#include "blowfish.h"

// Message length global
#define MESSAGE_LEN 20

// Print things like CAN frames and hex values. Set to 1 to enable printing to terminal.
#define PRINTING 0

// Global variables for socket used
struct sockaddr_can addrS;
struct ifreq ifrS;
int nbytesS;
int s;

// Structures for time gathering
struct timeval timeSent;
struct timeval timeRec;

// FILE pointer to times.csv
FILE *times;

// Misc globals
int i;

void sendFrame()
{
    // Message and CAN-related vars
    char temp[3];
    unsigned char MESSAGE[MESSAGE_LEN];
    strcpy(MESSAGE, "000#");
    struct can_frame frameS;

    // Blowfish-related vars
    u_int8_t entered_value[8] = {0xD,0xE,0xA,0xD,0xB,0xE,0xE,0xF};
    u_int8_t result_value[8] = {0};
    unsigned long left_block = 0;
    unsigned long right_block = 0;

    // Get the time before encryption process begins
    gettimeofday(&timeSent, NULL);

    // Encrypt the message
    left_block  = ((entered_value[0] << 24) | (entered_value[1] << 16) | (entered_value[2] << 8) | entered_value[3]);
    right_block = ((entered_value[4] << 24) | (entered_value[5] << 16) | (entered_value[6] << 8) | entered_value[7]);
    Blowfish_encipher(&left_block, &right_block);
    for(i=0; i<4; i++)
    {
        result_value[i]   = ((right_block & (0xFF << i*8)) >> (i*8));
        result_value[i+4] = ((left_block & (0xFF << i*8)) >> (i*8));
    }

    // Construct the final CAN message
    for(i = 7; i >= 0; i--)
    {
        sprintf(temp, "%02X", result_value[i]);
        strcat(MESSAGE, temp);
    }
    if(PRINTING)
    {
        printf("Encrypted Text (hex) = %02X %02X %02X %02X %02X %02X %02X %02X\n",
                        result_value[7], result_value[6], result_value[5], result_value[4],
                        result_value[3], result_value[2], result_value[1], result_value[0]);
        printf("\nMESSAGE CAN: %s\n", MESSAGE);
    }

    // Parse and send the frame
    if (parse_canframe(MESSAGE, &frameS)){
        printf("Error parsing CAN frame\n");
        return;
    }
    if ((nbytesS = write(s, &frameS, sizeof(frameS))) != sizeof(frameS))
    {
        perror("write");
        return;
    }
    return;
```

```
}

void recFrame()
{
    // Read the socket
    struct can_frame frameR;
    size_t readSize = 0;
    while(readSize < 1)
    {
        readSize = read(s, &frameR, sizeof(struct can_frame));
    }
    // Blowfish-related vars
    unsigned long left_block = 0;
    unsigned long right_block = 0;
    u_int8_t result_value[8] = {0};

    // Message-related vars
    char canMes[20];
    char *mesPtr = &canMes[0];

    // Place frame data into canMes array
    sprint_canframe(canMes, &frameR, 0);

    // Variable containing data characters in the message (after the message ID)
    char *b = canMes + 4;

    // Convert canMes into u_int8_t hex array
    int test[17];
    u_int8_t numHex[8];
    char str1[3],str2[3];
    for(i = 0; i<16; i++)
    {
        test[i] = *(b+i)-'0';
        switch(test[i])
        {
            case 17:
                test[i] = 10;
                break;
            case 18:
                test[i] = 11;
                break;
            case 19:
                test[i] = 12;
                break;
            case 20:
                test[i] = 13;
                break;
            case 21:
                test[i] = 14;
                break;
            case 22:
                test[i] = 15;
                break;
        }
    }
    for(i = 0; i<8; i++)
    {
        sprintf(str1,"%X",test[2*i]);
        sprintf(str2,"%X",test[2*i+1]);
        strcat(str1,str2);
        numHex[i] = (int)strtol(str1,NULL,16);
    }

    // Decrypt the ciphertext
    left_block  = ((numHex[0] << 24) | (numHex[1] << 16) | (numHex[2] << 8) | numHex[3]);
    right_block = ((numHex[4] << 24) | (numHex[5] << 16) | (numHex[6] << 8) | numHex[7]);
    Blowfish_decipher(&left_block, &right_block);
    for(i=0; i<4; i++)
    {
            result_value[i]   = ((right_block & (0xFF << i*8)) >> (i*8));
            result_value[i+4] = ((left_block & (0xFF << i*8)) >> (i*8));
    }
    if(PRINTING)
    {
        printf("Number recieved in hex paird form: ");
        for(i = 0;i<8;i++)
            printf("%X ",numHex[i]);
        printf("\nClear Text (hex) = %02X %02X %02X %02X %02X %02X %02X %02X\n",
                        result_value[7], result_value[6], result_value[5], result_value[4],
                        result_value[3], result_value[2], result_value[1], result_value[0]);
    }

    // Record the time received
    gettimeofday(&timeRec, NULL);
    // Save time sent and received to times.csv file
    fprintf(times, "%ld,%ld,%ld,%ld\n", timeSent.tv_sec, timeSent.tv_usec, timeRec.tv_sec, timeRec.tv_usec);
}

int main(int argc, char **argv)
{
    pid_t id = getpid();
    printf("ID: %d", id);

    // Prepare the socket
    s = socket(PF_CAN, SOCK_RAW, CAN_RAW);
```

```
    strcpy(ifrS.ifr_name, "can0" );
    if(ioctl(s, SIOCGIFINDEX, &ifrS) < 0)
        printf("Error sender");
    addrS.can_family = AF_CAN;
    addrS.can_ifindex = ifrS.ifr_ifindex;
    bind(s, (struct sockaddr *)&addrS, sizeof(addrS));

    // Set program to run for given number of times (default is 5000 runs)
    int numRuns, count;
    if (argc == 2)
        numRuns = atoi(argv[1]);
    else
        numRuns = 5000;

    // Initialize Blowfish with key
    char key[16] = "0000000000000000";
    InitializeBlowfish((char*)key, 16);

    // Open times.csv file for time-recording, add headings
    times = fopen("times.csv", "w+");
    fprintf(times, "Time Sent (s),Time Sent (us), Time Received (s),Time Received (us)\n");
    getchar();
    printf("Starting...\n");

    // Run for the given number of times
    count = 0;
    while(count < numRuns)
    {
        sendFrame();
        recFrame();
        count++;
    }

    // Close socket and file. Exit the program
    close(s);
    fclose(times);
    printf("Done! Times saved to times.csv file.\n");
    return 0;
}
```

The makefile for compilation is show below:

```
all : encSR_bf

clean :
    rm *.o encSR_bf

encSR_bf : lib.o blowfish.o encSR_bf.o
    gcc -o encSR_bf lib.o blowfish.o encSR_bf.o

# The next lines generate the various object files

lib.o : lib.c
    gcc -c lib.c

blowfish.o : blowfish.c
    gcc -c blowfish.c

encSR_bf.o : encSR_bf.c
    gcc  -c encSR_bf.c
```

encSR_rec_bf.c: The program file responsible for receiving encrypted messages from the sender over CAN. The program runs on ECU #2.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <time.h>
#include <signal.h>
#include <ctype.h>
#include <libgen.h>
#include <net/if.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <sys/uio.h>
#include <sys/ioctl.h>
#include <net/if.h>
#include <linux/can.h>
#include <linux/can/raw.h>
#include "lib.h"
#include "blowfish.h"

// Print things like CAN frames and hex values. Set to 1 to enable printing to terminal.
#define PRINTING 0

// Global variables for socket used
struct sockaddr_can addrS;
```

```
struct ifreq ifrS;
struct can_frame frameS;
int nbytesS;
int s;

// Misc globals
int i;

void sendFrame(u_int8_t to_send[8])
{
    // Message and CAN-related vars
    char temp[3];
    unsigned char message[20];
    strcpy(message, "000#");
    struct can_frame frameS;

    // Reverse the to_send array so that it is correct
    u_int8_t fixedSend[8] = {to_send[7], to_send[6], to_send[5], to_send[4], to_send[3], to_send[2], to_send[1], to_send[0]};

    // Blowfish-related vars
    unsigned long left_block = 0;
    unsigned long right_block = 0;
    u_int8_t result_value[8] = {0};

    // Encrypt the message
    left_block  = ((fixedSend[0] << 24) | (fixedSend[1] << 16) | (fixedSend[2] << 8) | fixedSend[3]);
    right_block = ((fixedSend[4] << 24) | (fixedSend[5] << 16) | (fixedSend[6] << 8) | fixedSend[7]);
    Blowfish_encipher(&left_block, &right_block);
    for(i=0; i<4; i++)
    {
        result_value[i]   = ((right_block & (0xFF << i*8)) >> (i*8));
        result_value[i+4] = ((left_block & (0xFF << i*8)) >> (i*8));
    }

    // Construct the final CAN message
    for(i = 7; i >= 0; i--)
    {
        sprintf(temp, "%02X", result_value[i]);
        strcat(message, temp);
    }
    if(PRINTING)
    {
        printf("Encrypted Text (hex) = %02X %02X %02X %02X %02X %02X %02X %02X\n",
                        result_value[7], result_value[6], result_value[5], result_value[4],
                        result_value[3], result_value[2], result_value[1], result_value[0]);
        printf("\nMESSAGE CAN: %s\n", message);
    }
    // Parse and send the frame
    if (parse_canframe(message, &frameS)){
        printf("Error parsing CAN frame\n");
        return;
    }
    if ((nbytesS = write(s, &frameS, sizeof(frameS))) != sizeof(frameS))
    {
        perror("write");
        return;
    }
}

void recFrame()
{
    // Read the socket
    struct can_frame frameR;
    size_t readSize = 0;
    while(readSize < 1)
    {
        readSize = read(s, &frameR, sizeof(struct can_frame));
    }
    // Blowfish-related vars
    unsigned long left_block = 0;
    unsigned long right_block = 0;
    u_int8_t result_value[8] = {0};

    // Message-related vars
    u_int8_t rec[8];
    char canMes[20];
    char *mesPtr = &canMes[0];

    // Place frame data into canMes array
    sprint_canframe(canMes, &frameR, 0);

    // Variable containing data characters in the message (after the message ID)
    char *b = canMes + 4;

    // Convert canMes into u_int8_t hex array
    int test[17];
    u_int8_t numHex[8];
    char str1[3],str2[3];
    for(i = 0; i<16; i++)
    {
        test[i] = *(b+i)-'0';
        switch(test[i])
        {
            case 17:
                test[i] = 10;
```

```
                break;
            case 18:
                test[i] = 11;
                break;
            case 19:
                test[i] = 12;
                break;
            case 20:
                test[i] = 13;
                break;
            case 21:
                test[i] = 14;
                break;
            case 22:
                test[i] = 15;
                break;
        }
    }
    for(i = 0; i<8; i++)
    {
        sprintf(str1,"%X",test[2*i]);
        sprintf(str2,"%X",test[2*i+1]);
        strcat(str1,str2);
        numHex[i] = (int)strtol(str1,NULL,16);
    }

    // Decrypt the ciphertext
    left_block  = ((numHex[0] << 24) | (numHex[1] << 16) | (numHex[2] << 8) | numHex[3]);
    right_block = ((numHex[4] << 24) | (numHex[5] << 16) | (numHex[6] << 8) | numHex[7]);
    Blowfish_decipher(&left_block, &right_block);
    for(i=0; i<4; i++)
    {
            result_value[i]   = ((right_block & (0xFF << i*8)) >> (i*8));
            result_value[i+4] = ((left_block & (0xFF << i*8)) >> (i*8));
    }

    if(PRINTING)
    {
        printf("Number recieved in hex paird form: ");
        for(i = 0;i<8;i++)
            printf("%X ",numHex[i]);
        printf("\nClear Text (hex) = %02X %02X %02X %02X %02X %02X %02X %02X\n",
                        result_value[7], result_value[6], result_value[5], result_value[4],
                        result_value[3], result_value[2], result_value[1], result_value[0]);
    }

    // Send the decrypted text to sendFrame for encryption and sending
    sendFrame(result_value);
}

int main(int argc, char **argv)
{
    // Prepare the socket
    s = socket(PF_CAN, SOCK_RAW, CAN_RAW);
    strcpy(ifrS.ifr_name, "can0" );
    if(ioctl(s, SIOCGIFINDEX, &ifrS) < 0)
        printf("Error sender");
    addrS.can_family = AF_CAN;
    addrS.can_ifindex = ifrS.ifr_ifindex;
    bind(s, (struct sockaddr *)&addrS, sizeof(addrS));

    // Set program to run for given number of times (default is 5000 runs)
    int numRuns, count;
    if (argc == 2)
        numRuns = atoi(argv[1]);
    else
        numRuns = 5000;

    // Initialize Blowfish with key
    char key[16] = "0000000000000000";
    InitializeBlowfish((char*)key, 16);

    printf("Starting...\n");

    // Run for the given number of times
    count = 0;
    while(count < numRuns)
    {
        recFrame();
        count++;
    }

    // Close socket and exit
    close(s);
    printf("Done!\n");
    return 0;
}
```

The makefile for compilation is shown below:

```
all : encSR_rec_bf

clean :
    rm *.o encSR_rec_bf
```

```
encSR_rec_bf : lib.o blowfish.o encSR_rec_bf.o
    gcc -o encSR_rec_bf lib.o blowfish.o encSR_rec_bf.o

# The next lines generate the various object files

lib.o : lib.c
    gcc -c lib.c

blowfish.o : blowfish.c
    gcc -c blowfish.c

encSR_rec_bf.o : encSR_rec_bf.c
    gcc  -c encSR_rec_bf.c
```

ii. Ethernet Implementation Code

The files relating to the Ethernet implementation without encryption are as follows:

client.c: The program file responsible for sending plaintext messages to the receiver over Ethernet. This program runs on ECU #1.

```
#ifndef unix
#include <winsock2.h>
#include <ws2tcpip.h>
/* also include Ws2_32.lib library in linking options */
#else
#define closesocket close
#define SOCKET int
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/time.h> // for timeout option
/* also include xnet library for linking; on command line add: -lxnet */
#endif

#include <stdio.h>
#include <string.h>

#define PORT "60000"                /* default protocol port number */
#define HOST "169.254.205.0"        /* default destination address  */

// Socket-related structs
struct  addrinfo addr_req;   /* default address parameters (hints) */
struct  addrinfo *addr_res;  /* ptr to the  address for connection */
SOCKET  sd;

// Time-related structs
struct timeval timeSent;
struct timeval timeRec;
FILE *times;

void sendMsg()
{
        int  n;                                 /* number of characters received       */
    int  m;                                 /* number of characters sent back      */
    char buf[16]="DEADBEEF";      /* buffer for data from the server     */

    /* Send data to socket in order to request reply. */
    m = 16;                                 /* sned zero bytes */
        gettimeofday(&timeSent, NULL);
    m = sendto(sd,buf,m,0,addr_res->ai_addr,addr_res->ai_addrlen);
    if(m<0)
    {
        fprintf(stderr,"Error in sending");
    }
}

void recMsg()
{
    struct sockaddr_storage src_addr;
  socklen_t src_addr_len = sizeof(src_addr);
    char buf[16];
    int n;
   /* Read data from socket and write to user's screen. */
  n = recvfrom(sd,buf,sizeof(buf),0,(struct sockaddr*)&src_addr,&src_addr_len);
    gettimeofday(&timeRec, NULL);
    fprintf(times, "%ld,%ld,%ld,%ld\n", timeSent.tv_sec, timeSent.tv_usec, timeRec.tv_sec, timeRec.tv_usec);
  if (n >= 0)
            buf[n]='\0';    /* just in case place the termination at the end of the string */
    else
      fprintf(stderr, "Receiving timeout");
}

int main()
{
    // Get process ID and output to terminal
```

```
    pid_t id = getpid();
    printf("ID: %d", id);

    int count = 0;
    int numRuns = 50000;

  /* Convert host name and port name and address hints to the address */
  memset(&addr_req, 0, sizeof(addr_req));
  addr_req.ai_socktype = SOCK_DGRAM;
  addr_req.ai_family = AF_INET; // Use: AF_INET6 or AF_INET or AF_UNSPEC
  if (0 != getaddrinfo(HOST, PORT, &addr_req, &addr_res))
    {
      fprintf(stderr, "cannot set up the destination address\n");
      exit(1);
  }

  /* Create a socket. */
  sd = socket(addr_res->ai_family, addr_res->ai_socktype, addr_res->ai_protocol);
  if (sd < 0)
    {
      fprintf(stderr, "socket creation failed\n");
      exit(1);
  }

{

/* Set timeout option for the socket */
#ifndef unix
        int timeout = 100; // in milliseconds
#else
        struct timeval timeout;
        timeout.tv_sec = 0;
        timeout.tv_usec = 100000L;
#endif
        if (setsockopt(sd, SOL_SOCKET, SO_RCVTIMEO, (char*)&timeout, sizeof(timeout)) < 0)
                fprintf(stderr, "Warning! setting timeout failed\n");
}
    // Open times file for recording
    times = fopen("times.csv", "w+");
    fprintf(times, "Time Sent (s),Time Sent (us), Time Received (s),Time Received (us)\n");
    getchar();
    printf("Starting!");

    while(count < numRuns)
    {
        sendMsg();
        recMsg();
        count++;
  }
    printf("Done!");

    /* Close the socket. */
    closesocket(sd);

#ifdef WIN32
    WSACleanup();                       /* release use of winsock.dll */
#endif

  /* Terminate the client program gracefully. */
  return(0);
}
```

echo_server.c: The program file responsible for receiving plaintext messages from the sender over Ethernet. This program runs on ECU #2.

```
#ifndef unix
#include <winsock2.h>
/* also include Ws2_32.lib library in linking options */
#else
#define closesocket close
#define SOCKET int
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>
/* also include xnet library for linking; on command line add: -lxnet */
#endif

#include <stdio.h>
#include <stdlib.h>

#define PROTOPORT       60000

int main(int argc, char *argv[])
{
    // Socket-related structs and vars
    struct  protoent *ptrp;  /* pointer to a protocol table entry   */
    struct  sockaddr_in sad; /* structure to hold server's address  */
    struct  sockaddr_in cad; /* structure to hold client's address  */
    SOCKET  s, rc;           /* socket descriptors                  */
    int     port;            /* protocol port number                */
```

```
#ifdef WIN32
    WSADATA wsaData;
    if(WSAStartup(0x0101, &wsaData)!=0)
    {
        fprintf(stderr, "Windows Socket Init failed: %d\n", GetLastError());
        exit(1);
    }
#endif

    if (argc > 1)                   /* if argument specified        */
        port = atoi(argv[1]);       /* convert argument to binary   */
    else
        port = PROTOPORT;           /* use default port number      */

    if (port <= 0)                  /* test for illegal value       */
    {                               /* print error message and exit */
        fprintf(stderr,"bad port number %s\n",argv[1]);
        exit(1);
    }

    sad.sin_port = htons((u_short)port); /* set server port number  */
    sad.sin_family=AF_INET;
    sad.sin_addr.s_addr=INADDR_ANY;

    /* Map UDP transport protocol name to protocol number */
    ptrp = getprotobyname("udp");
    if ( ptrp == 0)
    {
        fprintf(stderr, "cannot map \"udp\" to protocol number");
        exit(1);
    }

    s=socket(AF_INET, SOCK_DGRAM, ptrp->p_proto);
    if(s<0)
    {
        fprintf(stderr,"Socket creation failed\n");
        return 1;
    }

    rc=bind(s, (struct sockaddr *)&sad, sizeof(sad));
    if(rc<0)
    {
        fprintf(stderr,"Bind,failed\n");
        return 1;
    }

    while(1)
    {
        int     alen;           /* length of address                    */
        char    buf[1000];      /* buffer for string the echoing        */
        int     n;              /* number of characters received        */
        int     m;              /* number of characters sent back       */

        alen = sizeof(cad);
        n = recvfrom(s,buf,sizeof(buf),0,(struct sockaddr*)&cad,&alen);
        if (n<0)
        {
            fprintf(stderr,"Error in receiving\n");
            continue;
        }
        else if (n==0) {
            fprintf(stderr,"Remote sent an empty packet\n");
            continue;
        }
        else if(n>0)
        {
            m = sendto(s,buf,n,0,(struct sockaddr*)&cad,alen);
            if(m<0)
            {
                fprintf(stderr,"Error in sending");
                continue;
            }
        }
    }

#ifdef WIN32
    WSACleanup();
#endif
    return (0);
}
```

The files relating to the Ethernet implementation with encryption are as follows:

client_bf.c: The program file responsible for sending encrypted messages to the receiver over Ethernet. This program runs on ECU #1.

```
#ifndef unix
#include <winsock2.h>
```

```
#include <ws2tcpip.h>
/* also include Ws2_32.lib library in linking options */
#else
#define closesocket close
#define SOCKET int
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>
#include <sys/time.h> // for timeout option
/* also include xnet library for linking; on command line add: -lxnet */
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "blowfish.h"
#define PORT "60000"                /* default protocol port number */
#define HOST "169.254.205.0"           /* default destination address  */

// Socket-related structs
struct  addrinfo addr_req;   /* default address parameters (hints) */
struct  addrinfo *addr_res;  /* ptr to the  address for connection */
SOCKET  sd;

// Time-related structs
struct timeval timeSent;
struct timeval timeRec;
FILE *times;

void sendMsg()
{
    int  n;              /* number of characters received       */
  int  m;
    int i;
    char temp[3];
    /* number of characters sent back      */
    char buf[1000] = "";

    // Blowfish-related vars
    u_int8_t entered_value[8] = {0xD,0xE,0xA,0xD,0xB,0xE,0xE,0xF};
    u_int8_t result_value[8] = {0};
    unsigned long left_block = 0;
    unsigned long right_block = 0;

    // Get the time before encryption process begins
    gettimeofday(&timeSent, NULL);

    // Encrypt the message
    left_block  = ((entered_value[0] << 24) | (entered_value[1] << 16) | (entered_value[2] << 8) | entered_value[3]);
    right_block = ((entered_value[4] << 24) | (entered_value[5] << 16) | (entered_value[6] << 8) | entered_value[7]);
    Blowfish_encipher(&left_block, &right_block);
    for(i=0; i<4; i++)
    {
        result_value[i]   = ((right_block & (0xFF << i*8)) >> (i*8));
        result_value[i+4] = ((left_block & (0xFF << i*8)) >> (i*8));
    }

    // Construct the final  message
    for(i = 7; i >= 0; i--)
    {
        sprintf(temp, "%02X", result_value[i]);
        strcat(buf, temp);
    }
  /* buffer for data from the server     */

  /* Send data to socket in order to request reply. */
  m = 16;
  m = sendto(sd,buf,m,0,addr_res->ai_addr,addr_res->ai_addrlen);
  if(m<0)
      fprintf(stderr,"Error in sending");
}
void recMsg()
{
    struct sockaddr_storage src_addr;
  socklen_t src_addr_len = sizeof(src_addr);
    char buf[16];
    int n;

  /* Read data from socket */
  n = recvfrom(sd,buf,16,0,(struct sockaddr*)&src_addr,&src_addr_len);
    char *b = buf;
    int i;

    // Blowfish-related vars
    unsigned long left_block = 0;
    unsigned long right_block = 0;
    u_int8_t result_value[8] = {0};

    // Convert message into u_int8_t hex array
    int test[17];
    u_int8_t numHex[8];
    char str1[3],str2[3];
```

```
    for(i = 0; i<16; i++)
    {
        test[i] = *(b+i)-'0';
        switch(test[i])
        {
            case 17:
                test[i] = 10;
                break;
            case 18:
                test[i] = 11;
                break;
            case 19:
                test[i] = 12;
                break;
            case 20:
                test[i] = 13;
                break;
            case 21:
                test[i] = 14;
                break;
            case 22:
                test[i] = 15;
                break;
        }
    }
    for(i = 0; i<8; i++)
    {
        sprintf(str1,"%X",test[2*i]);
        sprintf(str2,"%X",test[2*i+1]);
        strcat(str1,str2);
        numHex[i] = (int)strtol(str1,NULL,16);
    }

    // Decrypt the ciphertext
    left_block  = ((numHex[0] << 24) | (numHex[1] << 16) | (numHex[2] << 8) | numHex[3]);
    right_block = ((numHex[4] << 24) | (numHex[5] << 16) | (numHex[6] << 8) | numHex[7]);
    Blowfish_decipher(&left_block, &right_block);
    for(i=0; i<4; i++)
    {
            result_value[i]   = ((right_block & (0xFF << i*8)) >> (i*8));
            result_value[i+4] = ((left_block & (0xFF << i*8)) >> (i*8));
    }
    gettimeofday(&timeRec, NULL);
    fprintf(times, "%ld,%ld,%ld,%ld\n", timeSent.tv_sec, timeSent.tv_usec, timeRec.tv_sec, timeRec.tv_usec);
    if (n >= 0)
                buf[n]='\0';    /* just in case place the termination at the end of the string */
    else
        fprintf(stderr, "Receiving timeout");
}
int main()
{
    // Get process ID and output to terminal
    pid_t id = getpid();
    printf("ID: %d", id);

    int count = 0;
    int numRuns = 50000;

    // Initialize Blowfish
    char key[16] = "0000000000000000";
    InitializeBlowfish((char*)key, 16);

  /* Convert host name and port name and address hints to the address */
  memset(&addr_req, 0, sizeof(addr_req));
  addr_req.ai_socktype = SOCK_DGRAM;
  addr_req.ai_family = AF_INET; // Use: AF_INET6 or AF_INET or AF_UNSPEC
  if (0 != getaddrinfo(HOST, PORT, &addr_req, &addr_res))
    {
      fprintf(stderr, "cannot set up the destination address\n");
      exit(1);
  }

  /* Create a socket. */
  sd = socket(addr_res->ai_family, addr_res->ai_socktype, addr_res->ai_protocol);
  if (sd < 0)
    {
      fprintf(stderr, "socket creation failed\n");
      exit(1);
  }

{
/* Set timeout option for the socket */
#ifndef unix
        int timeout = 100; // in milliseconds
#else
        struct timeval timeout;
        timeout.tv_sec = 0;
        timeout.tv_usec = 100000L;
#endif
        if (setsockopt(sd, SOL_SOCKET, SO_RCVTIMEO, (char*)&timeout, sizeof(timeout)) < 0) {
            fprintf(stderr, "Warning! setting timeout failed\n");
        }
}

    // Open times file for recording
```

```
    times = fopen("times.csv", "w+");
    fprintf(times, "Time Sent (s),Time Sent (us), Time Received (s),Time Received (us)\n");
    getchar();

    printf("Starting!\n");
    while(count <numRuns)
    {
        sendMsg();
        recMsg();
        count++;
  }
  printf("Done!\n");

  /* Close the socket. */
  closesocket(sd);

#ifdef WIN32
    WSACleanup();                       /* release use of winsock.dll */
#endif

  return(0);
}
```

The makefile for compilation is shown below:

```
all : client_bf

clean :
    rm *.o client_bf

client_bf : blowfish.o client_bf.o
    gcc -o client_bf blowfish.o client_bf.o

# The next lines generate the various object files

blowfish.o : blowfish.c
    gcc -c blowfish.c

client_bf.o : client_bf.c
    gcc  -c client_bf.c
```

echo_server_bf.c: The program file responsible for receiving encrypted messages from the sender over Ethernet. This program runs on ECU #2.

```
#ifndef unix
#include <winsock2.h>
/* also include Ws2_32.lib library in linking options */
#else
#define closesocket close
#define SOCKET int
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>
#include <string.h>
/* also include xnet library for linking; on command line add: -lxnet */
#endif

#include <stdio.h>
#include <stdlib.h>

#define PROTOPORT       60000

// Socket-related structs
struct  protoent *ptrp;  /* pointer to a protocol table entry   */
struct  sockaddr_in sad; /* structure to hold server's address  */
struct  sockaddr_in cad; /* structure to hold client's address  */
SOCKET  s, rc;           /* socket descriptors                  */
int     port;            /* protocol port number                */

void recMsg()
{
    int     alen;          /* length of address                   */
    char    buf[16];       /* buffer for string the echoing       */
    int     n;             /* number of characters received       */
    int     m,i;           /* number of characters sent back      */

    alen = sizeof(cad);
    n = recvfrom(s,buf,sizeof(buf),0,(struct sockaddr*)&cad,&alen);

    if (n<0)
        fprintf(stderr,"Error in receiving\n");
    else if (n==0)
        fprintf(stderr,"Remote sent an empty packet\n");

    // Blowfish-related vars
    char *b = buf;
    unsigned long left_block = 0;
    unsigned long right_block = 0;
    u_int8_t result_value[8] = {0};
```

```
    // Convert message into u_int8_t hex array
    int test[17];
    u_int8_t numHex[8];
    char str1[3],str2[3];
    for(i = 0; i < 16; i++)
    {
        test[i] = *(b+i)-'0';
        switch(test[i])
        {
            case 17:
                test[i] = 10;
                break;
            case 18:
                test[i] = 11;
                break;
            case 19:
                test[i] = 12;
                break;
            case 20:
                test[i] = 13;
                break;
            case 21:
                test[i] = 14;
                break;
            case 22:
                test[i] = 15;
                break;
        }
    }
    for(i = 0; i < 8; i++)
    {
        sprintf(str1,"%X",test[2*i]);
        sprintf(str2,"%X",test[2*i+1]);
        strcat(str1,str2);
        numHex[i] = (int)strtol(str1,NULL,16);
    }

    // Decrypt the ciphertext
    left_block  = ((numHex[0] << 24) | (numHex[1] << 16) | (numHex[2] << 8) | numHex[3]);
    right_block = ((numHex[4] << 24) | (numHex[5] << 16) | (numHex[6] << 8) | numHex[7]);
    Blowfish_decipher(&left_block, &right_block);
    for(i=0; i<4; i++)
    {
            result_value[i]   = ((right_block & (0xFF << i*8)) >> (i*8));
            result_value[i+4] = ((left_block & (0xFF << i*8)) >> (i*8));
    }

    u_int8_t fixedSend[8] = {result_value[7], result_value[6], result_value[5], result_value[4], result_value[3],
result_value[2], result_value[1], result_value[0]};

    // Begin re-encryption process for sendback
    left_block = 0;
    right_block = 0;
    u_int8_t result_value2[8] = {0};

    left_block  = ((fixedSend[0] << 24) | (fixedSend[1] << 16) | (fixedSend[2] << 8) | fixedSend[3]);
    right_block = ((fixedSend[4] << 24) | (fixedSend[5] << 16) | (fixedSend[6] << 8) | fixedSend[7]);
    Blowfish_encipher(&left_block, &right_block);
    for(i=0; i<4; i++)
    {
        result_value2[i]   = ((right_block & (0xFF << i*8)) >> (i*8));
        result_value2[i+4] = ((left_block & (0xFF << i*8)) >> (i*8));
    }

    // Construct the final message
    char enc_send[16] = "";
    char temp[3];
    for(i = 7; i >= 0; i--)
    {
        sprintf(temp, "%02X", result_value2[i]);
        strcat(enc_send, temp);
    }

    m = sendto(s,enc_send,16,0,(struct sockaddr*)&cad,alen);
    if(m<0)
        fprintf(stderr,"Error in sending");
}


int main(int argc, char *argv[])
{
#ifdef WIN32
    WSADATA wsaData;
    if(WSAStartup(0x0101, &wsaData)!=0)
    {
        fprintf(stderr, "Windows Socket Init failed: %d\n", GetLastError());
        exit(1);
    }
#endif

    if (argc > 1)                   /* if argument specified        */
        port = atoi(argv[1]);       /* convert argument to binary   */
    else
        port = PROTOPORT;           /* use default port number      */
```

```
    if (port <= 0)                  /* test for illegal value       */
    {                               /* print error message and exit */
        fprintf(stderr,"bad port number %s\n",argv[1]);
        exit(1);
    }

    sad.sin_port = htons((u_short)port); /* set server port number  */
    sad.sin_family=AF_INET;
    sad.sin_addr.s_addr=INADDR_ANY;

    /* Map UDP transport protocol name to protocol number */
    ptrp = getprotobyname("udp");
    if ( ptrp == 0) {
        fprintf(stderr, "cannot map \"udp\" to protocol number");
        exit(1);
    }

    s=socket(AF_INET, SOCK_DGRAM, ptrp->p_proto);
    if(s<0)
    {
        fprintf(stderr,"Socket creation failed\n");
        return 1;
    }

    rc=bind(s, (struct sockaddr *)&sad, sizeof(sad));
    if(rc<0)
    {
        fprintf(stderr,"Bind,failed\n");
        return 1;
    }

        // Initialize Blowfish
        char key[16] = "0000000000000000";
        InitializeBlowfish((char*)key, 16);

    while(1)
    {
        recMsg();
    }

#ifdef WIN32
    WSACleanup();
#endif
    return (0);
}
```

The makefile for compilation is shown below:

```
all : echo_server_bf

clean :
    rm *.o echo_server_bf

client_bf : blowfish.o echo_server_bf.o
    gcc -o echo_server_bf blowfish.o echo_server_bf.o

# The next lines generate the various object files

blowfish.o : blowfish.c
    gcc -c blowfish.c

echo_server_bf.o : echo_server_bf.c
    gcc  -c echo_server_bf.c
```

Appendix C – Raspberry Pi CAN Configuration Setup

Configuration of the PiCAN2 is split into two parts, the hardware setup and the software configuration.

i. Hardware Configuration

The hardware configuration for CAN on the Pi is straightforward. The PiCAN ships with a set of metal screws and standoffs. After ensuring that the Raspberry Pi is powered off, line up the four standoffs to the four corresponding screw holes in each corner of the Pi's PCB. After this, attach the PiCAN to the Pi by lining up the 40 pin port to the Pi's GPIO pins and pushing down until full contact is made. Finally, insert the screws through the PiCAN's corner holes so that they can be screwed into the standoffs.

To connect two or more Raspberry Pis through CAN, simply connect a twisted wire pair from one PiCAN's CAN_L and CAN_H ports to the others'. When connecting more than two devices together, simply apply a daisy chain wiring method like the one shown in the figure below.

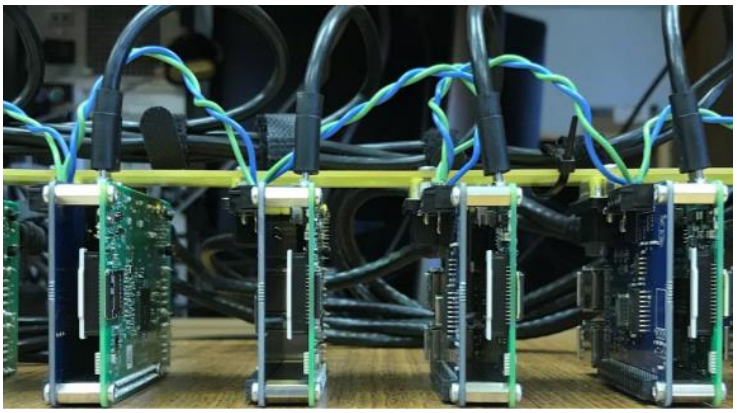

Figure 33 - Several Raspberry Pi CAN configurations connected in parallel.

After the desired number of devices have been wired together, add a 120 ohm resistor in between the CAN_L and CAN_H terminals on each end of the network. Alternatively, the PiCAN2 contains a 120 ohm resistor in their boards. To activate this terminator, simply solder a 2-way header pin to the JP3 header and then insert a jumper.

ii. Software Configuration

After the hardware has been installed, each system must be configured to recognize and interface with the shield. Boot up the Pi, then log in. For best results, log in with superuser privileges.

First, the Pi's configuration file must be edited. Run the command

sudo nano /boot/config.txt

This will open the system's config file in the nano text editor. Add the following 3 lines to the end of the document

dtparam=spi=on
dtoverlay=mcp2515-can0, oscillator=16000000, interrupt=25
dtoverlay=spi-bcm2835-overlay

These lines enable the Raspberry Pi's SPI bus. They also set the bus frequency and the RX (receiving) interrupt to GPIO pin 25. Save the file, close the editor, and reboot the system.

Next, run the command

sudo /sbin/ip link set can0 up type can bitrate 500000

This command initializes the CAN interface to appear as can0 and run at a frequency of 500 kbits/s. This frequency can be set to any value that does not exceed 1 Mbit/s. Please note that this command needs to be run every time the system reboots. To avoid having to hand-type this, consider adding the command to the system's cron table to run at boot.

To test the configuration, a set of programs can be downloaded in .zip form from

http://www.skpang.co.uk/dl/can-test_pi2.zip

The candump program can be run on systems to receive CAN signals by running

./candump can0

from within the unzipped directory. The program will wait for incoming messages and output them to the terminal when they are received.

On another system, the cansend program can be used to send a message to the other device(s). An example command for this is the command

./cansend can0 001#DEADBEEF

which sends a message with the identification code 001 and the hex data 0xDEADBEEF.