THESIS

SECURE CAN LOGGING AND DATA ANALYSIS

Submitted by

Duy Van

Department of Systems Engineering

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Fall 2020

Master's Committee:

Advisor: Jeremy Daily

Steve Simske
Christo Papadopoulos
Stephen Hayne

ABSTRACT

SECURE CAN LOGGING AND DATA ANALYSIS

Controller Area Network (CAN) communications are an essential element of modern vehicles, particularly heavy trucks. However, CAN protocols are vulnerable from a cybersecurity perspective in that they have no mechanism for authentication or authorization. Attacks on vehicle CAN systems present a risk to driver privacy and possibly driver safety. Therefore, developing new tools and techniques to detect cybersecurity threats within CAN networks is a critical research topic. A key component of this research is compiling a large database of representative CAN data from operational vehicles on the road. This database will be used to develop methods for detecting intrusions or other potential threats. In this paper, an open source CAN logger was developed that used hardware and software following the industry security standards to securely log and transmit heavy vehicle CAN data. A hardware prototype demonstrated the ability to encrypt data at over 6 Megabits per second (Mbps) and successfully log all data at 100% bus load on a 1 Mbps baud CAN network in a laboratory setting. An AES-128 Cipher Block Chaining (CBC) encryption mode was chosen. A Hardware Security Module (HSM) was used to generate and securely store asymmetric key pairs for cryptographic communication with a third-party cloud database. It also implemented Elliptic-Curve Cryptography (ECC) algorithms to perform key exchange and sign the data for integrity verification. This solution ensures secure data collection and transmission because only encrypted data is ever stored or transmitted, and communication with the third-party cloud server uses shared, asymmetric secret keys as well as Transport Layer Security (TLS).

ACKNOWLEDMENT

Throughout the writing of this thesis, I have received a great deal of assistance.

I would like to first thank my mentor, Dr. Jeremy Daily, who has guided me through this project with his valuable knowledge and dedication. I would also like to thank Dr. Steve Simske, Dr. Christo Papadopoulos, and Dr. Stephen Hayne for being on my thesis committee. Thanks to all my research colleagues who have helped me throughout the project. More importantly, I would like to thank my parents and my wife for their wise counsel and sympathetic ear.

In addition, I would like to thank the National Motor Freight Traffic Association, especially Urban Jonson and Ben Gardiner within the organization for greatly supporting the project during the project development and in the data collection process. I would like to thank Colorado State University as well as The University of Tulsa for providing the facilities and resources to conduct the research. And lastly, I would like to extend my sincere thanks to the National Science Foundation as the project sponsor.

PROJECT DISCLAIMER

This material is based upon work supported by the National Science Foundation under Grant No. 1715409, with the title of “SaTC: CORE: Small: Collaborative: GOALI: Detecting and Reconstructing Network Anomalies and Intrusions in Heavy Duty Vehicles.” Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

TABLE OF CONTENTS

Chapter 1. Introduction

A. Background

Historically, heavy trucks have been made of various mechanical and thermal systems that convert energy from fuel to kinetic energy. However, modern heavy trucks incorporate many Electronic Control Units (ECU) communicating over an internal vehicle network called the Controller Area Network (CAN). These ECUs carry commands, such as testing the brakes, produce more torque, etc. or sharing sensor data, such as vehicle speed, engine speed, fuel levels, etc. While the additional electronic control systems have enabled increases in fuel efficiency, vehicle reliability, and business effectiveness, the added systems create new levels of complexity. Figure 1-1 illustrates a generic picture of an electronic control system by listing and identifying the major ECUs, such as the anti-lock braking system, electronic stability control, engine control module, etc. within a heavy truck.

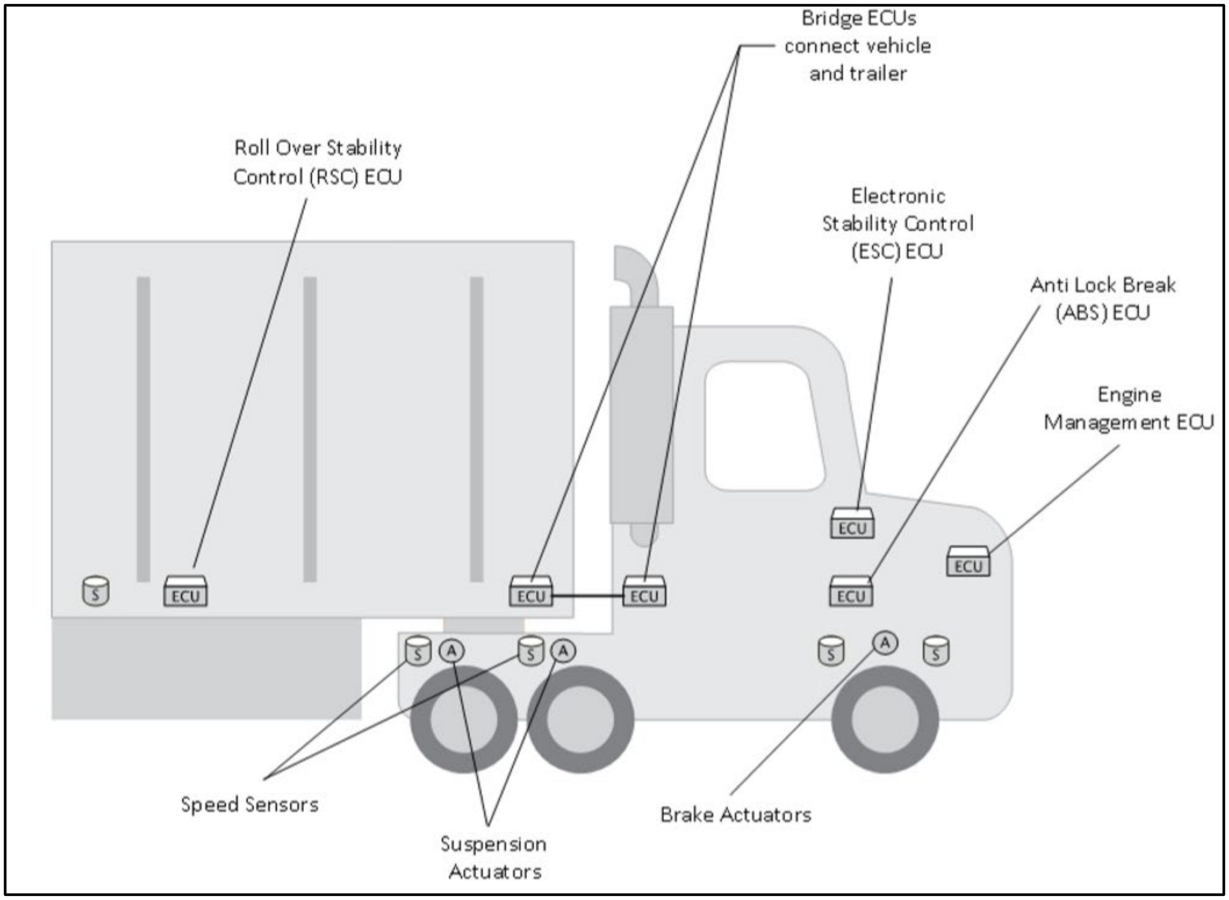


Figure 1-1. A heavy truck system [1]

The National Motor Freight Traffic Association (NMFTA) has published a whitepaper regarding the heavy vehicle cybersecurity [1]. The paper describes why the technologies on these vehicles has progressed (the good), the flaws inherent with such architectures (the bad), and how those flaws can be easily exploited (the ugly).

With so many interconnected ECUs and integrated sensors available in a modern vehicle, safety and comfort features are more robust and well-implemented. Most vehicles now have Anti-lock Braking System, Traction Control System, Roll-over Stability Control, and Electronic Stability Control as standard safety features that significantly improve the driver ability to gain back vehicle control during times when accidents are likely to happen. In addition, some heavy vehicles even include an integrated airbag module to minimize impacting damage on the driver if accidents do occur. The safety of vehicles has been greatly improved over the years, and automotive companies continue to design and optimize these systems. In addition to safety, comfort is an important design consideration. Depending on the consumer's desires, different models or trims now possess some features. Some basic features include door ajar indicator, infotainment sound level adjustment based on vehicle speed, automatic headlights, etc. Higher levels of automation available today include complex systems such as Adaptive Cruise Control, Lane Departure Warning, Lane Keeping Assist, Automated Parking Assist, etc. The automotive industry is heading toward connected vehicles where Vehicle-to-Vehicle, Vehicle-to-Infrastructure, or self-driving vehicles are being developed and tested. As a result, safety, comfort, and automation are the key elements in successful vehicle design, and the evolution in computerization within vehicles has provided a big leap in the industry.

Heavy trucks and passenger cars use CAN for internal network communications. Developed by Bosch in the early 1980s, CAN has been used by the automotive industry progressively since then. The most common implementations of CAN versions used are CAN 2.0A with 11-bit device identifiers for passenger cars, and CAN 2.0B with 29-bit device identifiers often found on heavy trucks, as specified by J1939-21 Data Link Layer [2]. The CAN bus is made up of multiple nodes, primarily ECUs, that communicate with differential signaling through two wires: CAN high (CANH) and CAN low (CANL). The CAN protocol is fundamentally flawed from a data security perspective and has been heavily researched. The NMFTA whitepaper [1] lists some vulnerabilities associated with the architecture of CAN protocol:

- Any node can listen, and any node can talk. There is no order or permission required for a node to start communicating, provided it is on the CAN bus.
- Any node can assert priority. CAN protocol handles message collision with arbitration, in which the message with highest priority wins.
- There is no encryption or validation within the CAN bus communication. The messages are sent in clear text; all received messages are assumed to have been sent from an authorized sender.
- The limit of 8 bytes per CAN frame eliminates the use of any modern block cipher to encrypt the data to ensure confidentiality.

CAN is a high speed, robust communication protocol; however, it was made in the time where cybersecurity was not in the mindset and the vehicle connectivity was not considered. The only security in the CAN messages is through obscurity which means each manufacturer designs its own proprietary message IDs and data fields without publishing it. Nevertheless, as seen by the mentioned characteristics above, data availability, integrity, and confidentiality can be easily exploited. If the network or a node is compromised by an attack, the vehicle safety mechanisms can malfunction. Figure 1-2 shows the CAN protocol and its vulnerability if any CAN node is attacked.

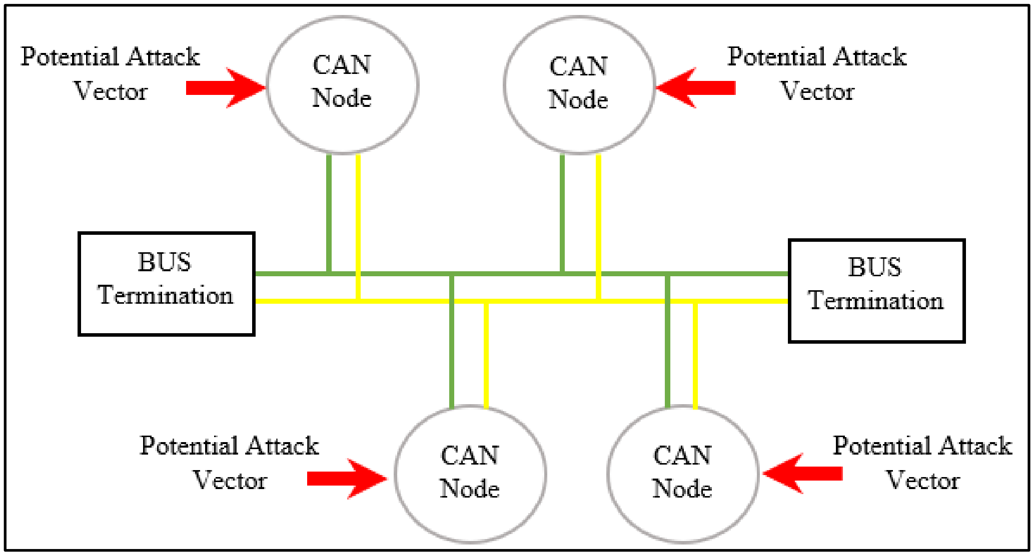


Figure 1-2. Abstracted CAN bus with vulnerabilities

There are a few common attacks that have been done, either by actual hackers or in lab testing, as described in the NMFTA whitepaper:

- Denial of Service – sending messages with the highest priority as fast as possible will overtake other legitimate messages with arbitration and hence, overwhelm the CAN bus. This leads to ECUs being unable to communicate with each other; as a result, the vehicle can behave unpredictably and/or cannot function at all. This is a typical basic attack that affects data availability.

- Middleperson or Man-In-The-Middle (MITM) – a malicious device is inserted between two or more communicating parties where it can observe and modify messages transmitting in between them. Moreover, a CAN bus node can be taken over and become the middleperson, where it sends out modified messages to drown out the original sender. Data integrity and confidentiality can be exploited, and commands can be changed.

- Diagnostic Packets – if attackers have access to the CAN bus, they may also be able to access the diagnostic functions that automotive technicians use for troubleshooting. These functions are mainly intended to be run in a controlled environment and may involve important safety features. If they are exploited and used incorrectly, they will do more harm than good.
- ECUs Firmware – the firmware is the memory and commands for the brain of the vehicle operation. Sometimes, it needs to be updated or debugged by the manufacturer, and this process usually takes place through the diagnostic port, which involves the CAN bus. Hackers can download, reverse-engineer the firmware to assembly level or a C-Code representation, and determine the proprietary data structure designed by the manufacturers. They may have enough information to creatively exploit the vehicle or even rewrite their modified firmware back to the ECUs.
- Fuzzing – this is a method where messages are injected randomly into the CAN bus to determine how the vehicle behaves. Different functions can be identified and tied to what message parameters using fuzzing techniques. Therefore, proprietary information is at risk of being exposed. Fuzzing and also lead to unintended cyber-physical reactions and even physical damage.

A good model against cybersecurity threats can be measured by the CIA Triad: confidentiality, integrity, and availability. Confidentiality means sensitive information should be protected against unauthorized access. Enforcing confidentiality usually involves cryptographic methods. Integrity means that the data has not been altered by unauthorized users and the originator of the data can be verified. Current methods to protect data integrity use cryptographic hashing and digital signatures. Lastly, availability means authorized users can freely access the data. Protecting data availability depends on the system infrastructure and model that can quickly detect threats or failures and be resilient when such circumstances occur. For CAN network systems, any additional cryptographic implementations could be pursued to increase the CIA Triad benchmark.

B. Objective

Automobiles have cybersecurity concerns based on the characteristics of the CAN protocol as there are many attack vectors and methods that can be implemented to exploit automotive systems. However, heavy trucks or commercial vehicles are exposed to cybersecurity risks differently comparing to passenger vehicles due to some major factors. The primary distinguishing feature is that heavy trucks follow the SAE J1939 standard [3], which is a recommended practice for communication and diagnostics among vehicle components. The manufacturers are not obligated to abide by the standard; however, they do implement many parts of SAE J1939 on a heavy vehicle network. The second difference is that heavy trucks are built with accommodations for horizontal integration to allow customers to customize the vehicles based on their needs. This means that customers have many options from which to choose for various components, including engines, brake controllers, transmissions, telematics units, infotainment systems. An unified communications standard, such as SAE J1939, is necessary to support interoperability and "plug and play" functionality between these disparate hardware systems. However, with open standards, heavy vehicles are easy targets because hackers can easily look for weaknesses within the network structure from the publicly available information. The CAN protocol security through obscurity strategy will continue to fail on top of its existing vulnerability to some attacks.

The last difference between passenger vehicles and heavy trucks is the prevalent use of third-party telematics devices. These telematics companies provide equipment that is installed on the vehicle network to keep track of information such as location, speed, fuel status, diagnostic trouble codes, etc. Telematics units can be seen in big fleets where monitoring hundreds or thousands of trucks is essential for business operations and compliance with regulations. A cybersecurity challenge is these telematics units are connected wirelessly, which introduces a new attack vector to the previously air-gapped vehicle network.

With these threats, heavy vehicles may be at high risk of being exposed to cyber-attacks. Therefore, the heavy vehicle industries should realize that increasing cybersecurity posture and mitigating risk and potential threats are important objectives in not only designing and building new commercial vehicles, but also maintaining current trucks on the road. Preventing attacks from occurring is always preferable to mitigating an attack once it takes place. Thus, intrusion and anomaly detection mechanisms need to be developed and deployed in the CAN bus system. A large pool of data from heavy vehicle CAN buses in the form of log files from normally operating trucks is essential for development and testing of vehicle network based cybersecurity controls. This data will consist of various types of CAN messages that take place on the bus, which can be periodic from normal operation or aperiodic from responding to special events.

The purpose of this thesis is to find a solution to build such a data pool securely and efficiently. In addition, the data collected will also be made available by request for references; therefore, it will be beneficial for use by the trucking industry.

C. Motivation

According to the latest Federal Motor Carrier Safety Administration (FMCSA) 2019 Large Truck and Bus Statistics [4], there were approximately 12.2 million registered heavy vehicles in the U.S alone in 2017 and approximately 730,000 new trucks on the road each year. Moreover, commercial vehicles often carry high-risk or high-value cargo. These transportation and freight services play an important role in the national and global economy. A mass cyber-attack on commercial vehicles nationwide will lead to devastating consequences: economic recession, shortage of supplies, public endangerment, lack of essential services, etc. As an engineer, the safety of the public is paramount. Such disasters are not tolerable and need to be prevented. As a result, the motivation for this thesis lies on the responsibilities and ethics of an engineer.

This thesis is one part of a two-part project funded by the National Science Foundation (NSF), with the support of the National Motor Freight Traffic Association, Inc. (NMFTA), who has been providing access to trucks from many volunteering companies. The title of the funded project is “SaTC: CORE: Small: Collaborative: GOALI: Detecting and Reconstructing Network Anomalies and Intrusions in Heavy Duty Vehicles” with the grant number of 1715409 from the National Science Foundation. The industry has taken the matter of cybersecurity seriously by providing resources to achieve the objective of detecting threats rather than responding to damage. Given such support, this is a great and unique opportunity to reach the objective.

D. Related Research

There are many public papers regarding different vehicle hacking techniques that exploit the CAN security posture. One of them is the infamous Jeep hack back in 2015, performed by Charlie Miller and Chris Valasek [5]. Charlie and Chris were able to find a way to gain access to deep level networks where sensitive signals are transmitted via the infotainment system. The firmware of this head unit was modified to execute malicious commands to critical ECUs. The result was that the vehicle was disabled. Data integrity and confidentiality have been exploited with this technique. In another paper, Subhojeet Mukherjee described how he made a denial of service attack on embedded networks in commercial vehicles [6]. With his testbed consisting of a single, high-speed CAN bus of 250 kbps, he has successfully shown that by sending a large number of request messages for a specific parameter, the number of regular messages dropped significantly due to the high computational load. Understanding of the limit of the system performance, Subhojeet exploited data availability here. In a different paper, Kyong-Tak Cho and Kang Shin took advantage of the error handling feature of the CAN protocol to shutdown ECU nodes from the network [7]. When an ECU tried to communicate, they injected attack messages to trigger the error flag to increase the victim Transmit Error Counter (TEC). When the TEC is above 255, the node is forced to shut down, hence the so-called bus-off mode. They can then send messages with forged ID and data to impersonate the node. Again, data integrity has been violated using the CAN data protocol. These attacks are no longer hard to implement, especially with the current publicly available information and technology. The question is how well we mitigate risk and potential threats to prevent cyber-attacks.

Several CAN projects to gather or monitor vehicle data have been pursued. A group of students from the University of Michigan have attempted to build a standalone embedded system to collect CAN messages, while filtering important ones with the purpose of warning drivers [8]. Adnan Shaout, Dhanush Mysuru, and Karthik Raghupathy described in the paper that their setup consisted of Vector software CANoe and Vector 1610 CAN hardware for CAN simulation, an Arduino UNO with ATMega328p processor and a CAN Shield hardware for CAN interface, a display for warning driver, and a Teensy 3.6 with SD card slot for memory storage. During the experiment, the Arduino UNO sniffed all the CAN messages with the help of the CAN Shield. This processor filtered out the messages with appropriate addresses and sent a copy of the data to the Teensy 3.6 for storage on the SD card. The display showed error messages if the messages contain undesired sensor values. The design functioned as intended but encountered computing power problems that caused the system to drop messages with an interval less than 50ms. Progress has been made on this problem, as stated in the paper, where the modified system can handle up to inter message time of 5ms. However, when a vehicle is under denial of service attack, the messages can be injected at a much faster rate, which can pose a challenging issue. If the system cannot capture all messages, it will not meet the requirements of a CAN monitoring design. The system cybersecurity was deemed to be out of scope and thus not addressed in the paper. However, if there is any cybersecurity threat happens to the vehicle, this system will not likely to detect such attack and even fail to operate.

In another project, Manthias Johanson and Lennart Karlsson discussed their wireless diagnostic system, where CAN messages are captured and monitored over an Internet connection [9]. This is interesting because the project involved the Internet of Things (IoT), which led to more complications in the system. The design was a wireless Diagnostic Read-out (DRO) system, which consisted of the Vehicle Information and Diagnostic for Aftersales (VIDA) device as a DRO system, a custom-built Dynamically Linked Library (DLL) for tunneling CAN frames over the Internet, a mobile unit equipped with an embedded Linux OS computer for CAN interface, an Internet connection through a General Packet Radio Services (GPRS) modem, and a server for dispatching requests. The DRO process involved a manual initiation with a button on the mobile device. An encrypted Transmission Control Protocol (TCP) connection was established on the server, with a public IP address reachable from the mobile unit. After that, specific diagnostic CAN messages were sent to the mobile unit from the server, where they were relayed onto the CAN bus. The responses were captured and sent back to the server. Due to the bandwidth limitation, the system could not relay all messages on the CAN bus and, therefore, only filtered out important ones. However, the paper did touch on the concerns of data integrity and confidentiality because an Internet connection was used by employing encrypted TCP connection along with RSA-based authentication mechanism.

Capturing all CAN data, particularly at high speeds, was a common problem that impacted both referenced CAN monitoring projects above. This is even harder to achieve when cybersecurity measures and wireless connection are implemented, because processing power and transmission bandwidth are limited, respectively.

E. Approach

Due to the complexity and high cost of integrating a new embedded system into the existed heavy truck network components, the best approach to collect CAN data is to design and build an affordable standalone device that can be easily connected to the vehicle CAN bus. Because the device is standalone, the data should be stored on an external memory storage, such as SD cards for simple management. The device must be able to capture all the data because missing abnormal messages will defeat the purpose of the project. To do so, the device needs to have a direct connection to the vehicle instead of wireless, even though the wireless feature can be beneficial for other applications, such as transferring existing data to a computer. Data can grow enormously, and thus, a cloud platform may be useful to store and manage the logs from many different uploading devices. Using third-party servers accessed over the Internet poses a risk from a cybersecurity aspect. Moreover, data integrity and confidentiality are two important factors that need to be protected. The reasons for encrypting the data are that altered data is useless and some vehicle owners do not wish to publish their data due. As a result, security measures such as cryptographic algorithms are utilized to encrypt, sign, and verify the data.

F. Contribution

In addition to the large data pool for references, this thesis should contribute:

1. Detailed documentation regarding the design of the CAN logging device, such that some of its applications are available and can be applied to the vehicle model with the purpose of increasing cybersecurity posture.

2. Use cases of logged data in a digital forensic context.

3. Aggregated CAN data from many different trucks.

4. Experiences and skills learned through this project are very valuable for protecting heavy vehicles and, thus, protecting the public.

G. Organization of Thesis

The thesis is divided into six chapters:

- Chapter 1 provides a basic introduction to the project regarding the trucking industry background, objective of the project, motivation, literature review of related researches, an approach to achieve the objective, and the contribution.
- Chapter 2 provides the hardware design which lists the project requirements, system block diagram, different design alternatives for consideration, detailed component schematics showing how the electrical components are connected to each other, Printed Circuit Board (PCB) layout displaying the placement of those components on the device, Bill of Materials (BOM) listing all required components, how the device is manufactured and assembled, and results from functional tests.
- Chapter 3 provides the software design, which consists of the process overview indicating the interactions between the all system components, the two-part processes of provisioning and normal operation, and the example transcripts. The chapter also dives

into the embedded firmware of the device, the code of the local computer application, and the interface of the cloud services for each of the operation modes.

- Chapter 4 discusses the experiences of visiting different field locations for testing and data collection. The data is decoded for some important vehicle parameters for analysis.
- Chapter 5 introduces the device's different applications in the cybersecurity aspect to help retrieve and reverse-engineer Cummins ECU data for forensics purposes. An SAE technical paper was written and submitted on this subject, with the title of "Chip and Board Level Digital Forensics of Cummins Heavy Vehicle Event Data Recorders" [10].
- Chapter 6 concludes the thesis with restatement of abstract, contribution, and lists some future works for project improvement.

Chapter 2. Hardware Design

A. Requirements

To carry out the objective, the CAN logger device must securely capture all CAN data under both normal and abnormal operating conditions. Secondly, the data must be securely stored and organized for easy retrieval and decoding by the data owner. Lastly, the design and source code should be made available to the public. A list of requirements for fulfilling the desired goals follows. While some requirements have not been vetted against industry standards, they have worked for laboratory uses. The pinout requirements are depicted in Figure 2-1 and summarized below.

1. The logger must support multiple CAN channels following the J1939 Deutsch 9-pin connector standards. This configuration is as follows:
    a. CAN0: J1939 has CAN-H on pin C and CAN-L on pin D.
    b. CAN1: OEM Specific has CAN-H on pin H and CAN-L on pin J.
    c. CAN2: Pins F and G should be multiplexed to have CAN-H/J1708-H and CAN-L/J1708-L, respectively.

Pin	Value
A	Ground
B	+12V
C	CAN/J1939 Hi
D	CAN/J1939 Lo
E	CAN/J1939 Shield
F	J1708/J1587 Hi
G	J1708/J1587 Lo
H	OEM Specific
J	OEM Specific

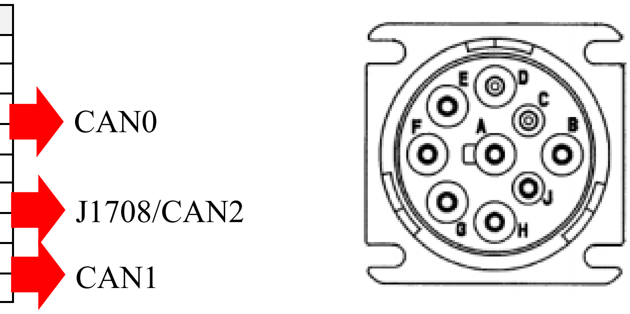


Figure 2-1. SAE standard 9-pin Deutsch connector for heavy truck [11]

Most vehicles have J1939 as the main CAN channel; however, many vehicles on the road still have the legacy J1708 network, which is an old serial communication protocol that is currently being replaced by J1939. Newer vehicles now also have CAN1 channel and for some PACCAR engines, J1708 has been replaced with CAN2. As a result, the design must have a multiplexing function to switch between J1708 and CAN2, depending on the vehicle. The connector is also known as the vehicle diagnostic port. A typical pin out for the connector is illustrated in Figure 2-1. Vehicles with 250kbps bus bitrate carry black connectors. New vehicles with 500kbps bus bitrate carry type-2 green connectors for easy indication. Figure 2-2 shows a type-1 to type-2 adapter cable.

Figure 2-2. SAE J1939 type-1 (black) to type-2 (green) adapter cable [12]

2. The logger needs to be inexpensive and easy to manufacture because a large number of devices is essential for efficiently collecting data from many different locations. The desired cost per device should not exceed $200.

3. The logger must be able to capture all CAN messages, even at 100% bus load. This ensures the device's reliable functionality to prevent losing any information that can be critical for data analysis.

4. In addition to the normal CAN messages, the logger must also capture error frames in order to help detect abnormal activity on the CAN bus.

5. The logger must use the vehicle battery line from the connector as a source for power to minimize cost associated with adding extra self-power components.

6. The logger must withstand power failure without losing current logging session. Power failures could occur if the device is disconnected from the port or if vehicle loses power from the battery or alternator.

7. The logger must handle typical voltages associated with vehicle system up to 24V. However, these transients may go up to 30V or more because there are load dumps and reversals associated with inductive loads and starters that create spikes. It is vital the device operation is sustainable and resilient in such conditions. Therefore, a maximum design system voltage of 36 V was chosen to mitigate the risk of system power failure that may occur. If the voltage exceeds the maximum specification, the device must also have an inexpensive way to protect critical components from permanent damage.

8. The logger must automatically detect different CAN bus speeds. Due to different CAN bitrates used on different vehicles, the device should be able to automatically detect the current bitrate on the bus. The most common ones on heavy trucks are 250 kbps and 500 kbps. Other bitrates that may be used are: 125 kbps, 666 kbps, and 1 Mbps. This feature

helps eliminate manual bitrate input from the user, and thus, making the operation quicker and more convenient.

9. The logger should have removable external storage for keeping the log data.
10. The logger must employ standard cryptographic implementations to protect data integrity and confidentiality. Asymmetric keys can be utilized for signing, verifying, and safely exchanging symmetric keys which are used for data encryption. Because the design and source code are going to be public, the objective is to achieve security through the use of open standards.
11. The backend storage system needs to enable secure and a scalable access to the data.
12. Users need a friendly and easy-to-navigate interface to upload and download files from the server. This application must be a secure gateway for the system to authenticate users and monitor their activities. Users should not have permission to directly access files stored on the server.

B. Design Alternatives

There are several CAN hardware devices on the market that can be used to log data for the project. For example, some common CAN analyzing tools are PEAK PCAN series, Vector CANlog, Intrepid ValueCAN, as shown in Figure 2-3. Diagnostic CAN tools, such as DG DPA5, Nexiq USB Link 2, Cummins Inline 7, also have features to capture data through their PC interface. Figure 2-4 shows examples of the diagnostic tools.

Figure 2-3. CAN analyzing tools: PEAK PCAN-USB [13] (left), Intrepid ValueCAN [14] (middle), and Vector CANlog [15] (right)

Figure 2-4. Diagnostic tools: DG DPA5 [16] (left), Nexiq USB-Link 2 [17] (middle), and Cummins Inline 7 [18] (right)

These tools, however, have other unnecessary features that significantly increase the cost, which ranges from at least a couple of hundreds to thousands of dollars. Moreover, these devices require a computer to interact with during operation, which can be inconvenient for logging purposes in some cases. They are closed source and as a result, it is hard for industry engineers to modify and develop functions for their needs. And more importantly, security measures, such as data authentication and encryption, have not been implemented by some.

The closest hardware design alternative, in the aspect of functionality, is the CSS CANedge2, as seen in Figure 2-5.

Figure 2-5. CSS CANedge2 [19]

The CANedge2 supports dual CAN channels, SD card storage with encrypted credentials, WiFi capability with secure HTTPS, and a cloud platform for data management. These features nearly satisfy the requirements of the CAN Logger 3. Nevertheless, the cost for the device, including SD card and cable adapter, of 539 EUR (~$589) is, again, the main factor that makes it unfeasible for the project.

Before this thesis was part of the project, different CAN Logger versions have been developed. The NMFTA CAN Logger, developed in 2017, was the first design. Figure 2-6 shows the image of a NMFTA CAN Logger. The information and source code of the device can be found on this GitHub repository [20].

Figure 2-6. NMFTA CAN Logger [21]

The device utilized the Teensy 3.2 with a 32-bit ARM Cortex-M4 K20 Sub-family processor, which has a speed of 72MHz and a built-in CAN controller for one CAN channel. A CAN controller and a SD card slot were added for a second CAN channel capability and data storage, respectively. A custom PCB was made for the Teensy 3.2 and the mentioned components were wrapped in a heat-shrink protective layer. Two LEDs were used to indicate different operational modes. A universal J1939 Deutsch 9-pin connector was used to connect the device to the vehicle network through the diagnostic port. The cost for each NMFTA CAN Logger is around $90, which was inexpensive comparing to others on the market. It was a good foundation for simply logging one CAN channel. However, the device did not meet the majority of the listed requirements, including the number of CAN channels, high voltage protection, data integrity and confidentiality protection, cloud storage, and user interface. Therefore, CAN

Logger 2 was developed, as seen in Figure 2-7. The device information can be found on GitHub at [22].

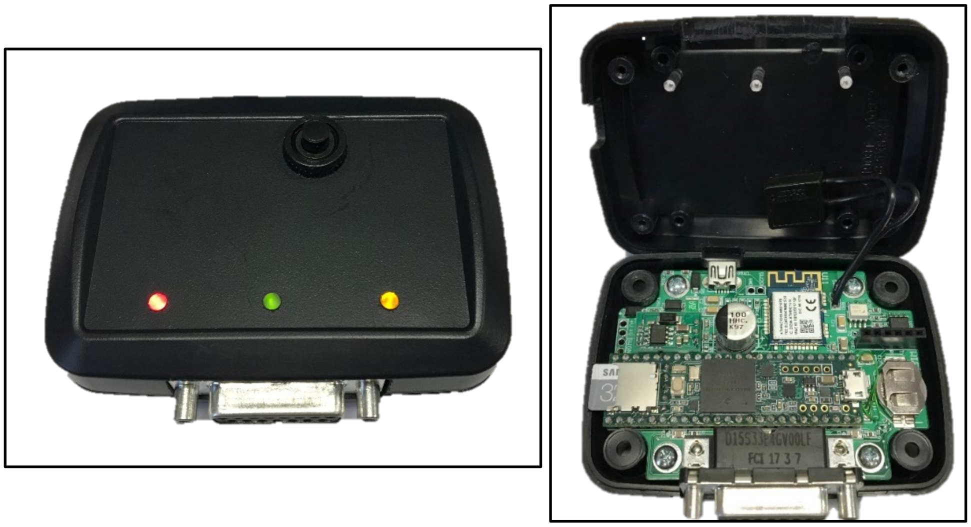


Figure 2-7. CAN Logger 2

The CAN Logger 2 utilized the Teensy 3.6 with a 32-bit ARM Cortex-M4F K66 sub-family processor. With a clock speed of 180MHz, the Teensy 3.6 is more than twice as fast as the teensy 3.2. A special component in the Teensy 3.6 is the Memory-Mapped Crypto Acceleration Unit (mmCAU) which can quickly perform cryptographic algorithms. Equipped with dual built-in CAN controllers, the CAN Logger 2 can support at least two CAN channels. A third CAN channel was added into the design using the MCP2515 CAN controller, which communicates with the Teensy 3.6 via Serial Peripheral Interface (SPI). A J1708 circuit was also added to the design using the SN75VD12 transceiver and the SN74AHCT inverter. Because the Deutsch 9-pin connector only supports up to three channels, the third CAN channel and the J1708 network was selected during operation by a programmable multiplexing switch ADG1634BCPZ. Moreover, Local Interconnect Network (LIN) with MCP2003A driver, Single-wire CAN (SWCAN) with NCV7356D1R2G chip, and WiFi capability with ATWINC1500 were added as optional features for future use. The device is protected from high voltage with a Transient Voltage Suppressor (TVS) and high current discharge with a Resettable Fuse (PTC) and reverse polarity with a Schottky diode. Three LEDs, instead of two, and a push button were built into the design to create more interactions for the user. The biggest difference was the Hardware Security Module (HSM) added for asymmetric cryptographic operations. The printed circuit board (PCB) is protected by a Bud HH-3642 enclosure. The total cost of the device is approximately $250. The CAN Logger 2 was able to satisfy all the stated hardware requirements; however, there were some details that needed to be improved during the development, and producing the device required significant specialized labor and customization, contributing to the higher device cost. Because of the high production costs, the CAN Logger 3 was developed as the next improved version from the CAN Logger 2, as seen in Figure 2-8. The details of the CAN Logger 3 hardware is available on GitHub [23].

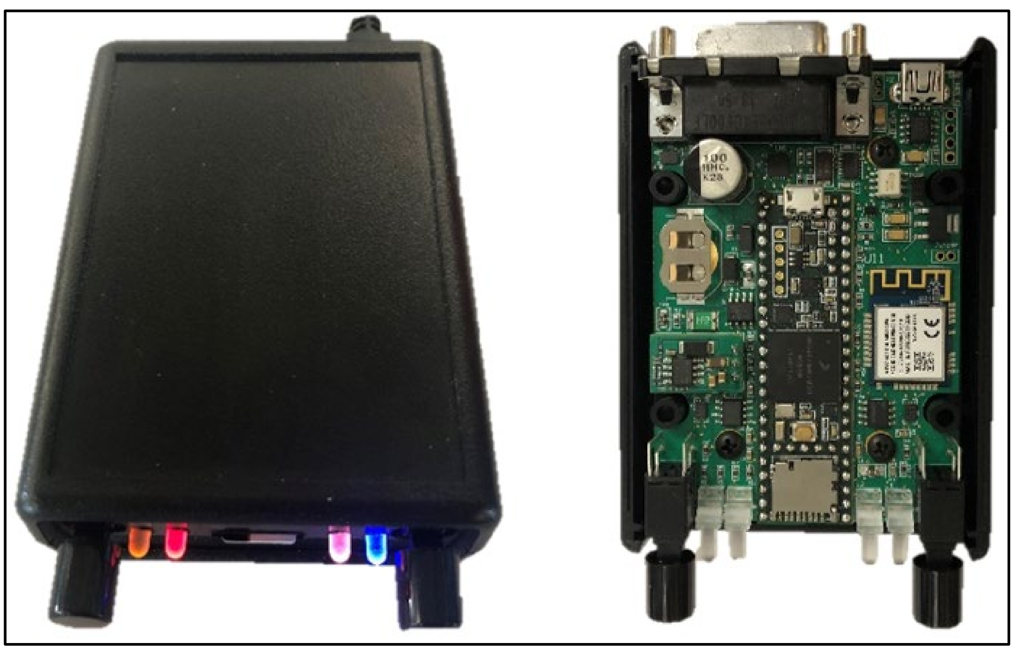


Figure 2-8. The first version of CAN Logger 3

The enclosure was changed to the BUD HP-3651-B for simplifying manufacturing procedures and thus, cheaper assembly cost. The entire PCB layout was redesigned. In addition, some minor upgrades have been made, such as the MCP2515 CAN controller was replaced with a new CAN controller that can accommodate flexible data rate CAN, the Microchip MCP2517FD. Two push buttons and four LEDs were used instead of one button and three LEDs, and a physical bridge for enabling/disabling WiFi feature was added. This brought total cost down to $220. However, the cost still has not met the $200 requirement. Instead of placing the entire teensy 3.6 board on the PCB, an approach of integrate its components in the design was pursued. The total cost per device was lowered to $180, which was the desired result. The latest CAN Logger 3 version, which is revision 3e as shown in Figure 2-9, is the current final product described in this thesis. Detailed schematics of the design will be explained in later section. The phrase “CAN Logger 3” mentioned from now on refers to the revision 3e.

Figure 2-9. This photograph of the CAN Logger 3, Rev 3e shows the single board solution using the K66 processor

C. Block Diagram

The CAN Logger 3 hardware design is illustrated through the block diagram in Figure 2-10, in which the components will be discussed in detail in the next section.

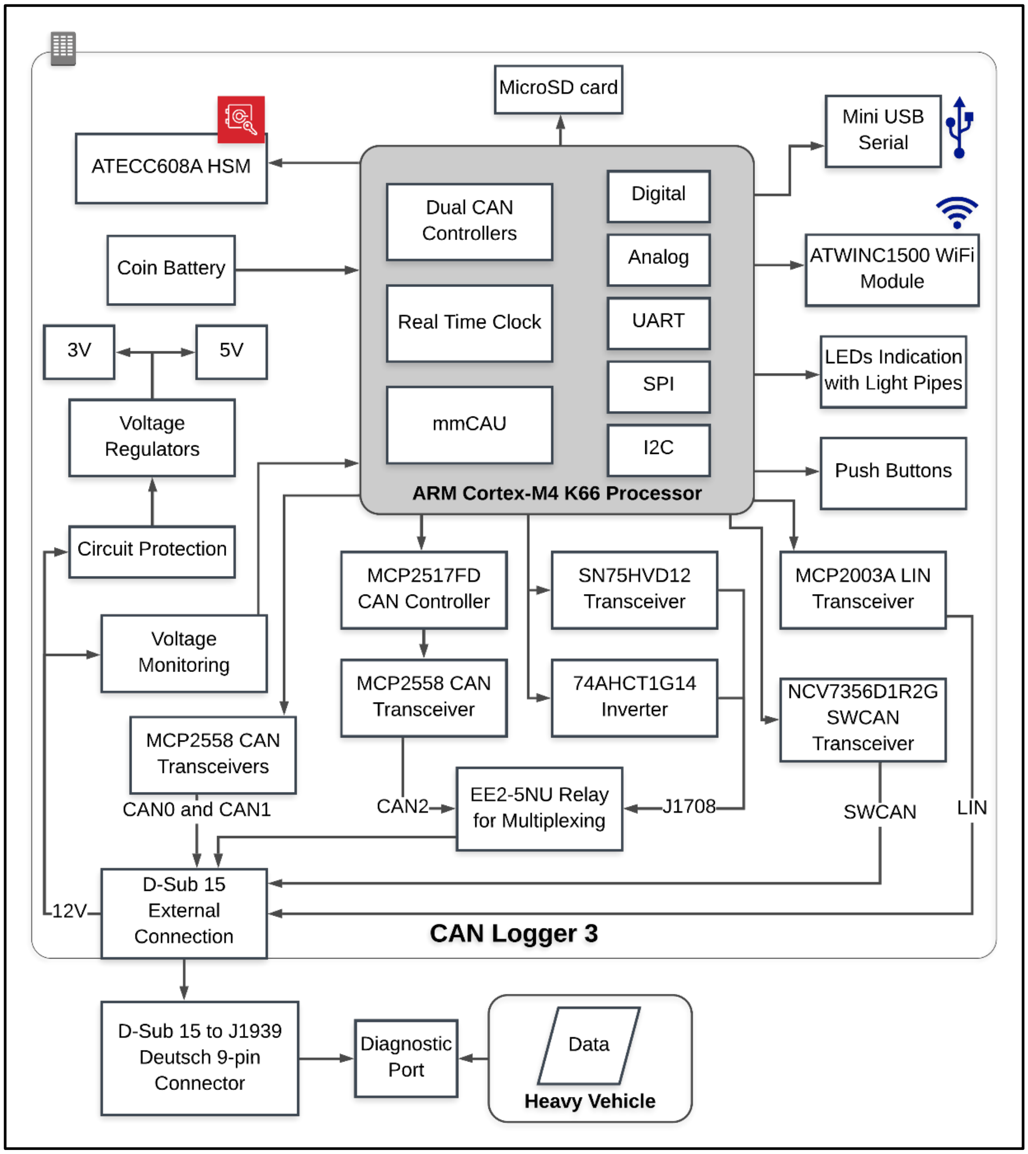


Figure 2-10. CAN Logger 3 hardware design block diagram

D. Detailed Schematics

Figure 2-11 to Figure 2-13 display the full detailed schematics of the CAN Logger 3 design.

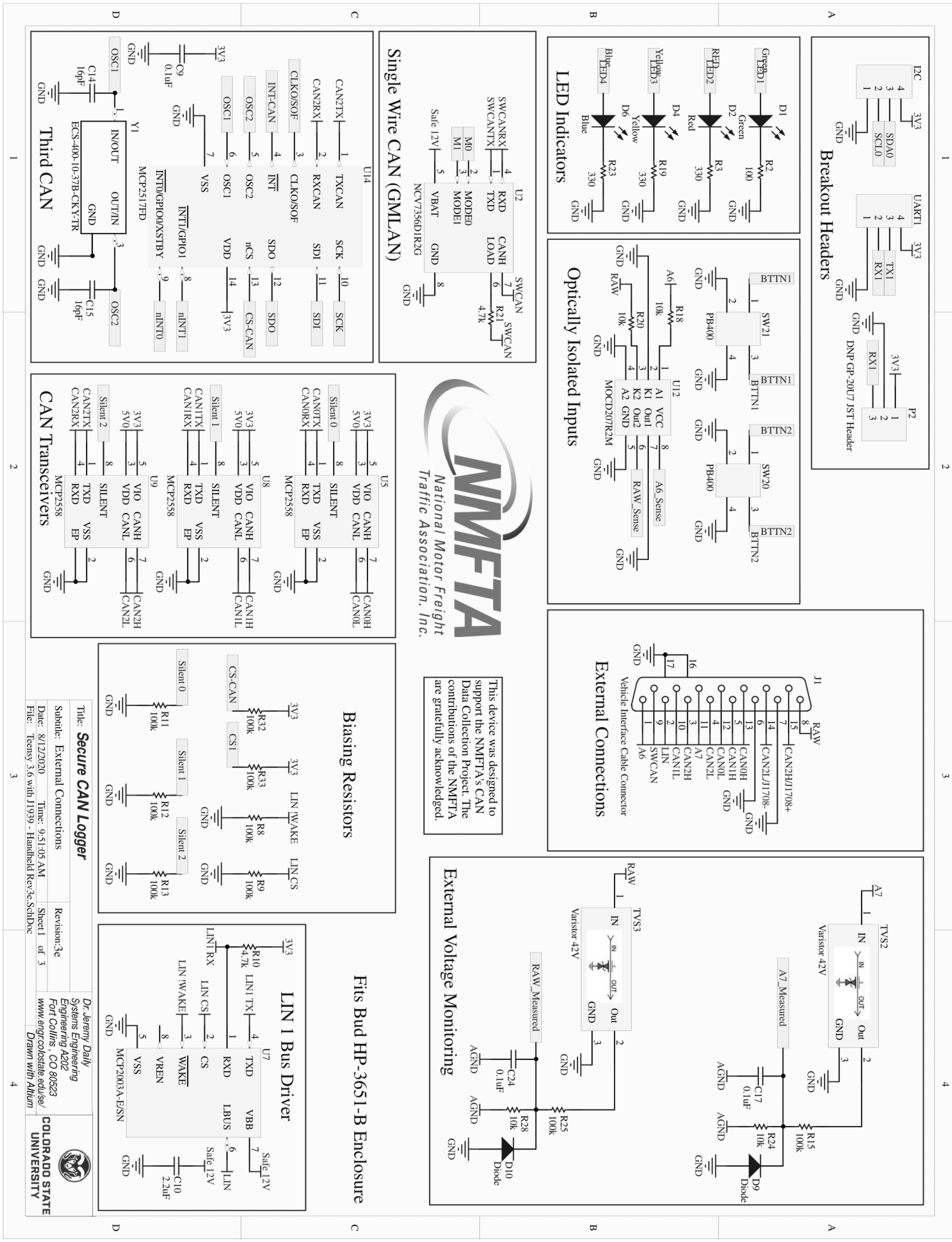


Figure 2-11. Page 1 of CAN Logger 3 schematics

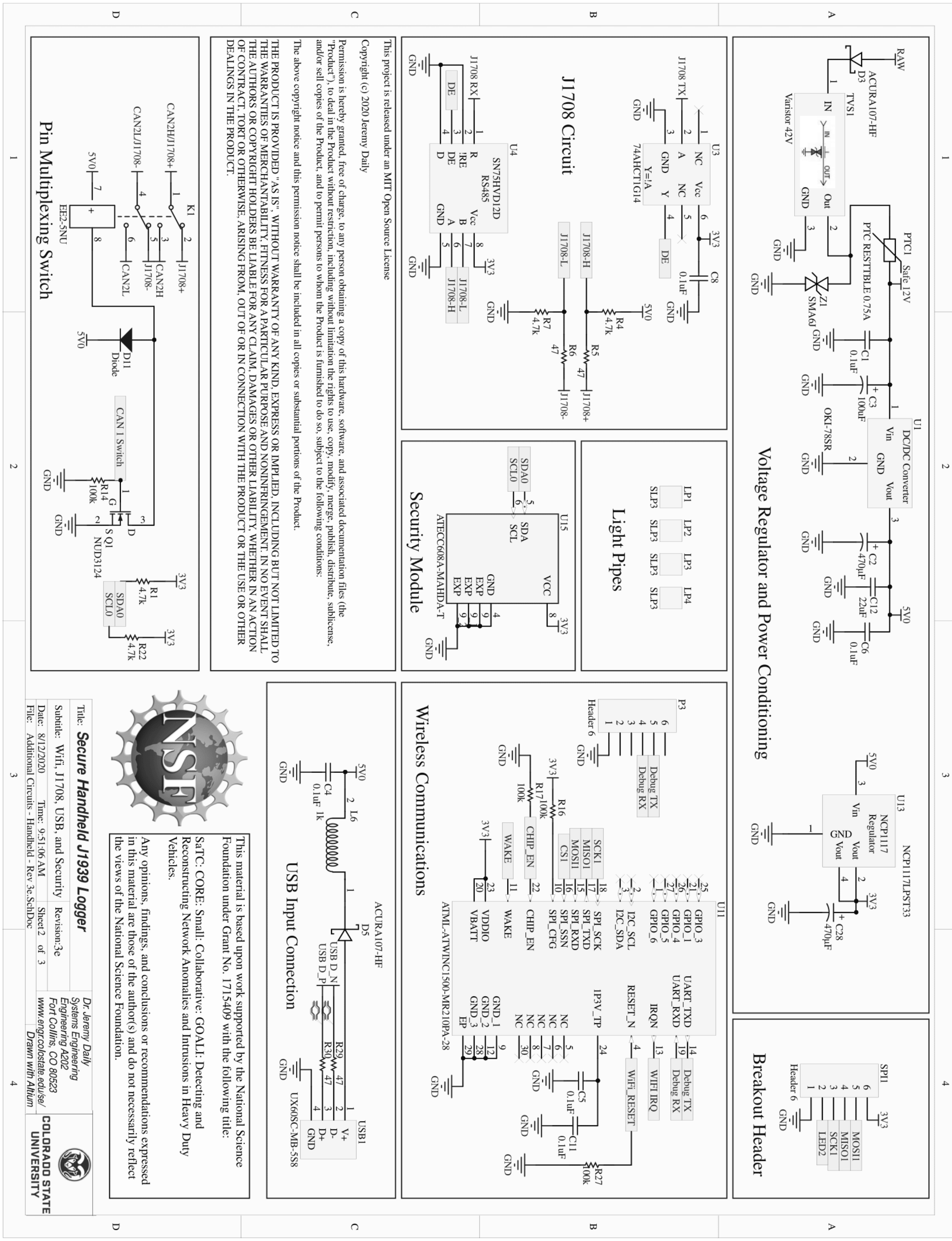


Figure 2-12. Page 2 of CAN Logger 3 schematics

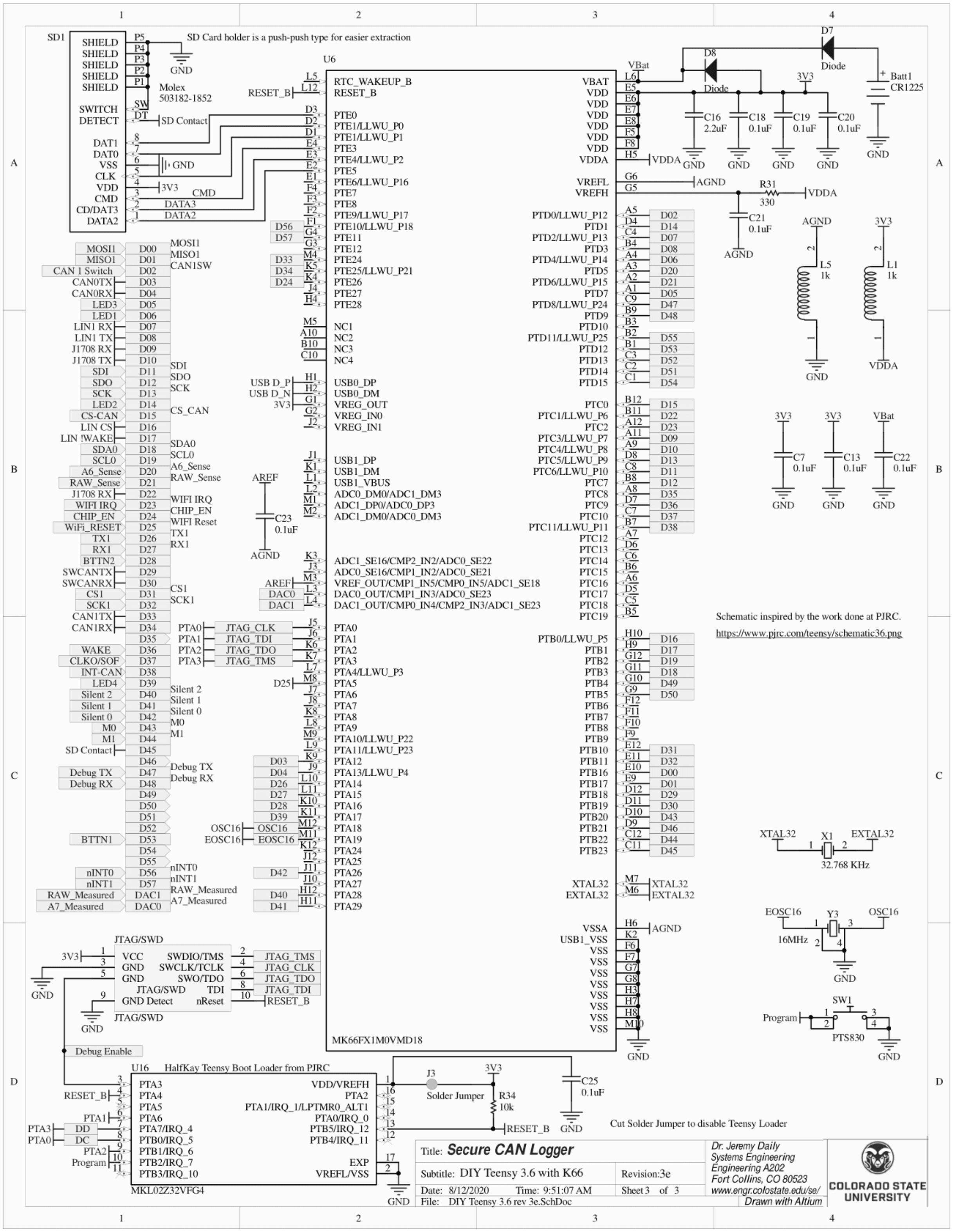


Figure 2-13. Page 3 of CAN Logger 3 schematics

Each section of the schematics is discussed as follow:

i. Teensy 3.6

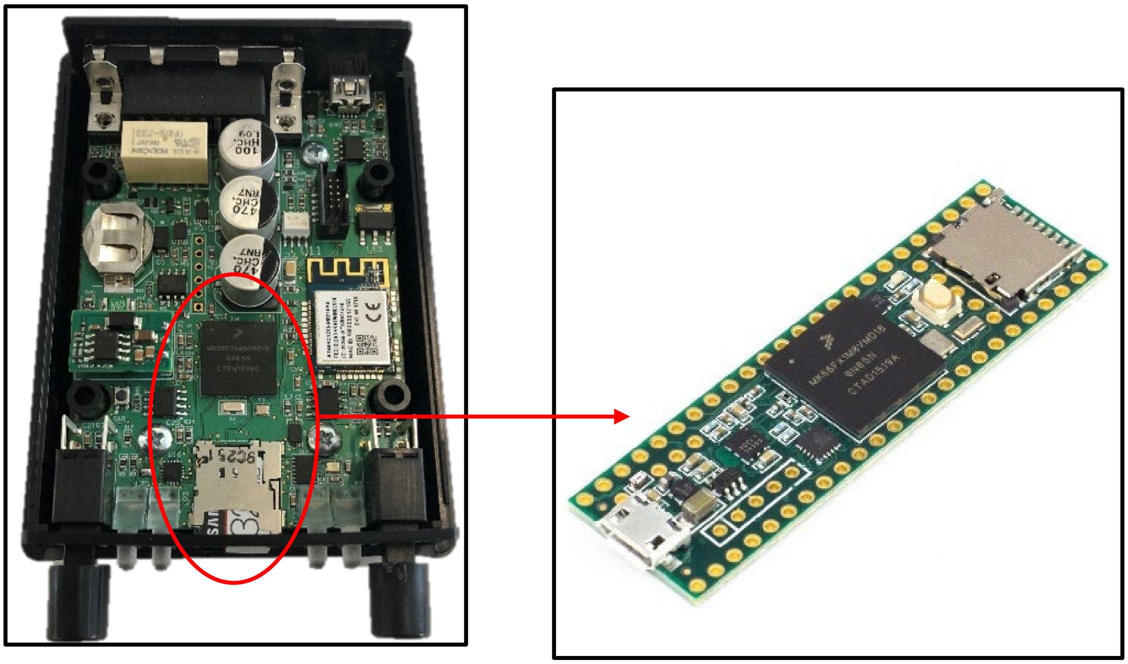


Figure 2-14. Integrated teensy 3.6 with ARM Cortex-M4 K66 processor on CAN Logger 3

The CAN Logger 3 is inspired by the Teensy 3.6, which contains the ARM Cortex-M4 K66 processor [24]. The Teensy 3.6 is an ARM32 based platform that works with the Arduino Integrated Development Environment (IDE). It is CAN compatible through the onboard FlexCAN controller peripherals, which are accessible using the FlexCAN library. The FlexCAN library can be found in [25]. To control costs, instead of using the original Teensy board, the necessary components of the Teensy 3.6 are integrated into the CAN Logger 3 circuit design with the same configuration as indicated on PJRC website [26]. A critical component from the Teensy 3.6 is the MKL02Z32VFK4 processor which comes pre-programmed from PJRC. It contains the bootloader that makes the K66 processor programmable with Arduino. There is a solder jumper connection (J3) that needs to be bridged to enable the bootloader, as seen in Figure

2-15. Some features that make the Teensy 3.6 an effective solution for a CAN logger include: dual CAN channels, real-time clock which is maintained by a 3V CR1225 coin cell battery, digital and analog input/output, SPI/UART/I2C communication, and an on-board SD card. The K66 processor operates with a clock speed of 180MHz, which is sufficiently fast to meet the design objectives. Moreover, the K66 also has an embedded mmCAU ColdFire coprocessor that is capable of implementing cryptographic algorithms such as AES-128, DES, 3DES, MD5, SHA-1, and SHA-256. Among those, AES-128 will be implemented to encrypt log data.

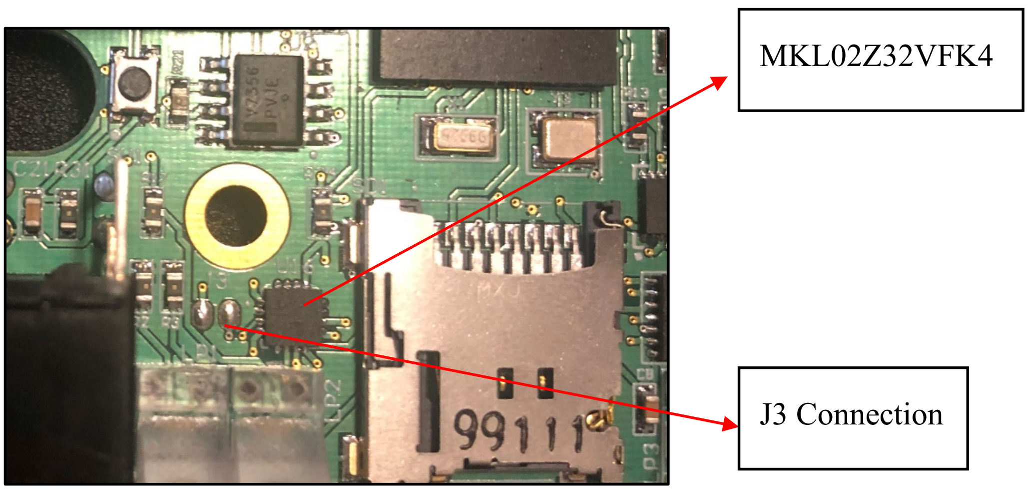


Figure 2-15. Teensy bootloader and J3 connection

ii. Third CAN channel

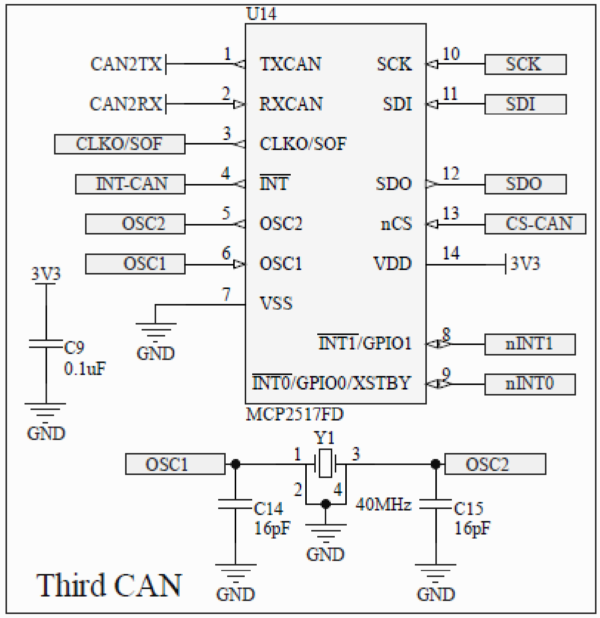


Figure 2-16. CAN Logger 3 schematics for MCP2517FD CAN controller

Because the Teensy 3.6 only has two built-in CAN controllers, the MCP2517FD [27] is added as an extra CAN controller for the third channel (CAN2), with the schematics shown in Figure 2-16. The chip communicates with the CAN Logger 3 processor via SPI with clock speed up to 20Mhz. A 40Mhz crystal clock is required for the transceiver with OSC1 and OSC2 connection, and two capacitors C14 and C15 are needed for proper oscillation [28]. A bypass capacitor, C9, is added to the 3.3V power supply for the controller to reduce high frequency noise [29]. The main reasons that the MCP2517FD is chosen over the previous MCP2515 on the CAN Logger 2 are that the MCP2517FD supports, besides the CAN2.0B, CAN FD as specified In ISO11898-1:2015 [30], two INT/GPIO, and start of frame (SOF) can be used as an interrupt. These features are optional but great to have for future use. The cost of the MCP2517FD is $2.31 in 2020.

iii. CAN Transceivers

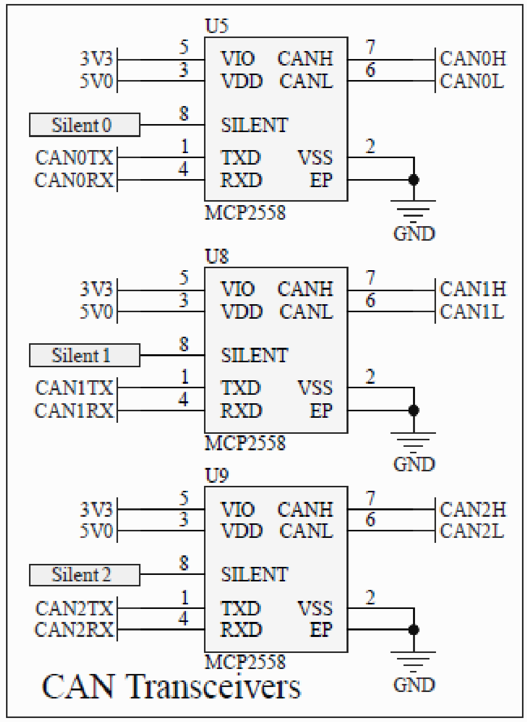


Figure 2-17. CAN Logger 3 schematics for MCP2558 CAN transceiver

The Microchip MCP2558 [31] was the chosen CAN transceiver for the three CAN channels, as shown in the CAN logger schematics in Figure 2-17. The MCP2558 meets the SAE J2962/2 “Communication Transceivers Qualification Requirements” and meets ISO-11898-1:2015 specifications [32]. The chip supports CAN FD with speeds up to 8Mbps. Besides the normal functions of a CAN transceiver, it also provides a silent mode which gives the CAN Logger 3 an ability to enable/disable its associated CAN channel transmission. Moreover, the low cost of $0.81 per chip is another reason for choosing the MCP2558. On the Teensy 3.6, pins 3 and 4 are used for the CAN0 transceiver, and pins 33 and 34 are used for the CAN1 transceiver. The CAN2 transceiver is connected to the MCP2517FD CAN controller via CAN2 TX and RX.

iv. J1708

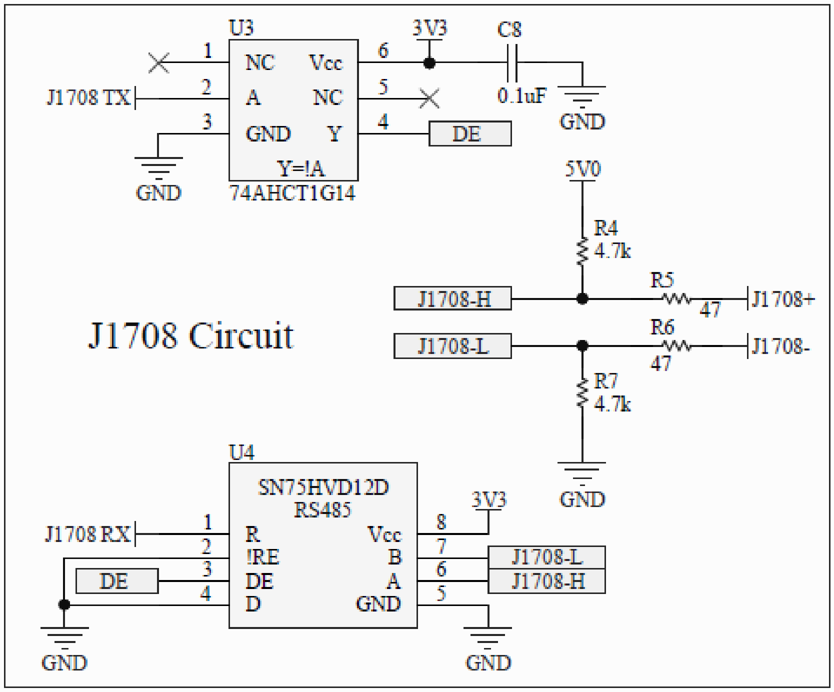


Figure 2-18. CAN Logger 3 schematics for J1708 circuit

The J1708 network consisted of the 74AHCT1G14 logic inverter [33] and the SN75HVD12D transceiver [34], as shown in Figure 2-18. This circuit is reused from the Smart Sensor Simulator 2 (SSS2) [35], which was adapted from the SAE J1708 standard [36]. The inverter provides a general-purpose logic with CMOS low power consumption and communicates with the processor using J1708TX. The transceiver is a combination of a 3-state differential line driver and differential input line receiver and communicates with the processor using J1708RX. Resistors R4, R5, R6, and R7 are added as shown, as recommended by the physical layer SAE J1708 standard. The costs for the inverter and the transceiver are $0.25 and $3.86, respectively. With this circuit, the CAN logger can communicate in the J1708 network via UART through J1708-L and J1708-H.

v. J1708/CAN2 Multiplexing

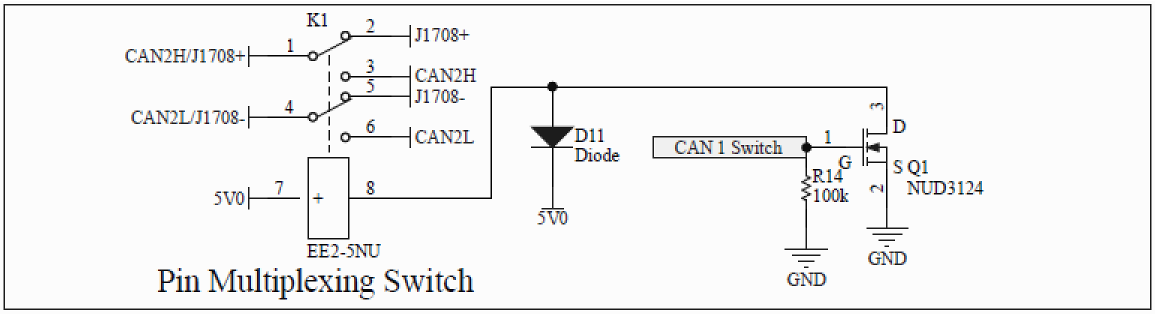


Figure 2-19. CAN Logger 3 schematics for multiplexing

CAN2 and J1708 are multiplexed due to conductor path limitations on the connector. The EE2-5NU relay [37] is used as a replacement of the ADG1634BCPZ switch [38], which is the version in the CAN Logger 2 and the original version CAN Logger 3. One of the reasons is that the switch uses the 12V line and could easily be damaged by transient overvoltage events. The relay costs $1.90 while the switch costs $6.23, which is 3 times more expensive. The relay is Double Pole Double Throw (DPDT) style, which means that the relay has two inputs and four outputs such that each input has two corresponding outputs that it connects to. This configuration is applicable because both CAN2 and J1708 use two wires for communication, as seen in Figure 2-19. The relay is also a non-latching type with a 5V single coil, in which when energized, the relay will change from J1708 to CAN2. The switch is controlled by the NUD3124 inductive load driver [39], which takes CAN 1 Switch from the processor digital pin as an input. When CAN 1 Switch is set high, the driver closes the connection and shorts pin 8 on the relay to ground, thus enabling the multiplexing. The R14 is placed at CAN1 as a pull-down resistor, which gives the driver input a known state during startup. The cost of the driver is $0.40 per unit. D11 is a flyback diode used to prevent voltage spikes from collapsing magnetic fields from the relay coil or from transients arising when the power supply is disconnected [40].

vi. Local Interconnect Network (LIN)

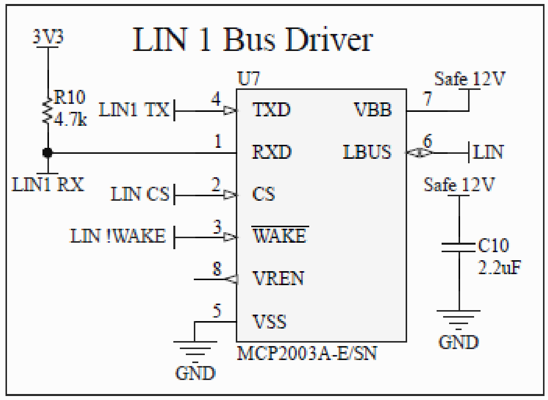


Figure 2-20. CAN Logger 3 schematics for LIN circuit

Similar to the J1708 circuit, the Local Interconnect Network (LIN) circuit is also reused from the SSS2, which is shown in Figure 2-20. MCP2003A [41] is the chosen LIN transceiver due to its cost effective of $0.82 per unit, SAE J2620 and LIN specifications compliance, overtemperature protection, and high immunity against electromagnetic, electrostatic discharge, and radio frequency disturbances. The transceiver runs on the 12V line, communicates with the processor through LIN TX and RX, and outputs a single LIN wire to the network for transmission. LIN CS has to be set high to enable the transceiver. As recommended by the datasheet, R10 is used as a pull-up resistor. C10 is also used as a bypass capacitor.

vii. Single-wire CAN

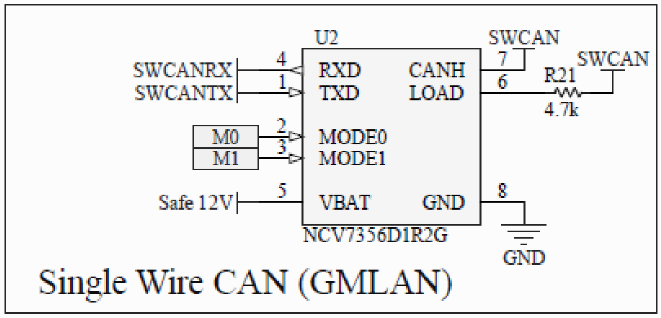


Figure 2-21. CAN Logger 3 schematics for SWCAN

Figure 2-21 shows the SWCAN schematics, which contains the NCV7356D1R2G SWCAN transceiver [42]. The chip is a physical layer device that is fully compatible with J2411 single wire CAN specifications and supports CAN 2.0 where highspeed application is not required. It is also cost effective with a unit price of $1.75. The transceiver is connected to CAN0 alterative on pin 29 and 30 of the processor, which has to be selected using the FlexCAN library to enable SWCAN feature. The transceiver operates with the 12V and communicates with the processor through SWCAN TX and RX. The SWCAN transmission is from the CANH and LOAD connection on the transceiver. A resistor of more than 600 Ohm is needed between the SWCAN line and LOAD, according to the datasheet. Therefore, the R21 with a resistance of 4.7k, which is commonly used, is placed as shown. Different operational modes of the transceiver, which are defined in the datasheet, can be selected using M0 and M1 digital outputs from the processor.

viii. Hardware security module

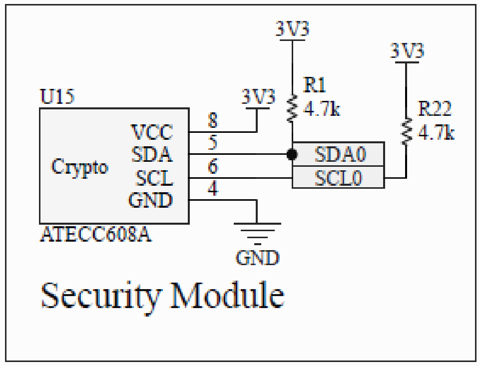


Figure 2-22. CAN Logger 3 schematics for ATECC608A HSM

The Microchip ATECC608A hardware security module is the key component for the security aspect of the logging process. General information about the module can be found in [43] and its schematics is shown in Figure 2-22. Because the module datasheet is not public at the time of this writing, a non-disclosure agreement has been signed to have access to the datasheet for fully utilizing the module functionality. The hardware uses 3.3V for power and communicates with the processor using I2C communication with SDA0 and SCL0. R1 and R22 resistors are placed on those two lines as recommended by common I2C circuit [44]. The cryptographic module is designed with hardware-based key storage that protects up to 16 keys. Once the keys or confidential data are stored and locked in the ATECC608A memory, the information cannot be easily read and can only be used internally by the hardware functions. This is a great feature for cybersecurity where there is a need to keep secrets in a safe space and not expose them to the external environments where they can be sniffed or exploited with methods such as middle-person attacks. Moreover, the ATECC608 supports cryptographic algorithms including:

- AES-128 encrypt/decrypt,
- Galois field multiply for generic authenticated encryption block cipher mode.
- SHA-256 & HMAC hash.
- 256-bit ECC following NIST standard with Elliptic-curve Digital Signature Algorithm (ECDSA) following FUPS186-3.
- Elliptic-curve Diffie-Hellman (ECDH) following FIPS SP800-56A standards.

For this project, the HSM AES-128, ECDH, and ECDSA functions will be implemented. The reason why ECC is preferably over other algorithms for asymmetric cryptography is that ECC can meet the same security standard with a much smaller key size, as shown in Table 2-1. A 160-bit ECC key is equivalent to a 1024-bit RSA and Diffie-Hellman, or a 256-but ECC key is equivalent to a 3072-bit RSA and Diffie-Hellman, and so on.

Table 2-1. NIST recommendation on key size for some algorithms [45]

Key Size(in bit)			
Symmetric	**Asymmetric**		
AES	**RSA**	**Diffie-Hellman**	**Elliptic Curve**
80	1024	1024	160
112	2048	2048	224
128	3072	3072	256
192	7680	7680	384
256	15360	15360	521

This means that ECC is much more powerful in terms of computing time and memory space. The speed test has been conducted on various embedded processors and the results in Table 2-2 shows the superior speed of ECC over RSA with the time combination of signing and verifying.

Table 2-2. Speed benchmark of crypto graphic standard algorithms on three different embedded processors [46].

Algorithm (speed/block)	**V850ES (32 MHz)**	**ARM7 (32 MHz)**	**Star12 (8 MHz)**
AES 128 bit	1 - 1.5 ms	1.5 - 2 ms	22 - 30 ms
ECC 192 bit Sign	100 - 150 ms	200 - 300 ms	3.5 - 4.5 s
ECC 192 bit Verify	350 - 500 ms	600 - 900 ms	10 - 13 s
RSA 1536 bit Sign CRT	5.5 - 7.5 s	10 - 13 s	100 - 140 s
RSA 1536 bit Verify (e=3)	35 - 50 ms	80 - 110 ms	0.7 - 1 s

With the cost of $0.75 per piece and the provided features, the ATEC608A HSM is the most suitable security module for the CAN logger device.

ix. WiFi module

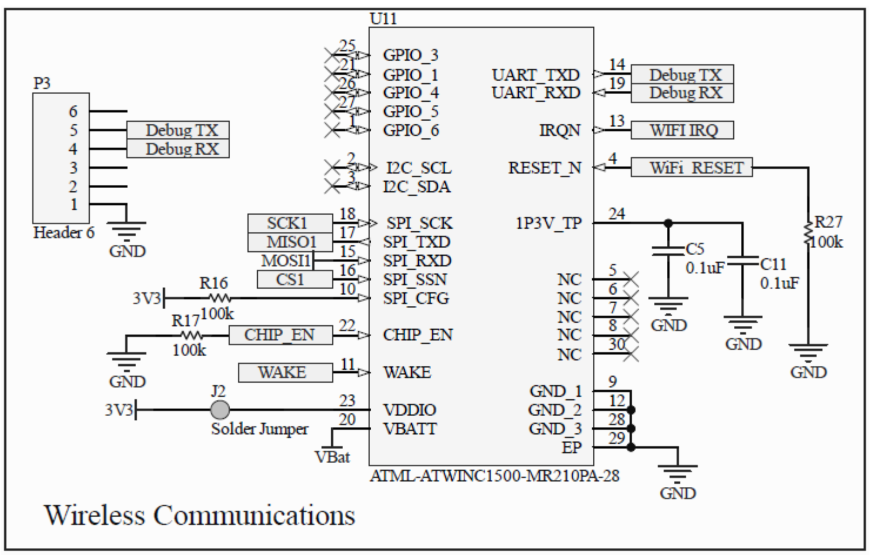


Figure 2-23. CAN Logger 3 schematics for ATWINC1500 WiFi module. The VBat net should be tied to 3.3V, which was fixed in a later hardware revision.

The CAN Logger 3 is equipped with the low power consumption ATWINC1500 WiFi module [47], which features IEEE 802.11 b/g/n and 2.4 GHz ISM band. The schematic for the module is illustrated in Figure 2-23. The C5 and C11 are decoupling capacitors as recommended by the datasheet. R16, R17, and R27 are pull-up/pull-down resistors. The module is powered with 3.3V and communicates with the processor using SPI, specifically SPI1 because SPI0 is taken by the MCP2517FD controller. P3 is a header used for easy debugging. The cost for the WiFi module is $7.84 per unit.

With this feature, the CAN Logger 3 can transfer log files wirelessly to the local computer before uploading or transfer the data straight to the server wirelessly, which is an option for the scope of the project. However, this also poses as an additional attack vector. To mitigate risk in specific applications, a physical switch is made in the design, as seen in Figure 2-24, such that users need to solder and bridge the J2 jumper to enable the WiFi module. Thus, if users do not wish to use the WiFi feature, they can physically disable the ATWINC1500 module maintaining peace of mind from the risk of wireless exploitation.

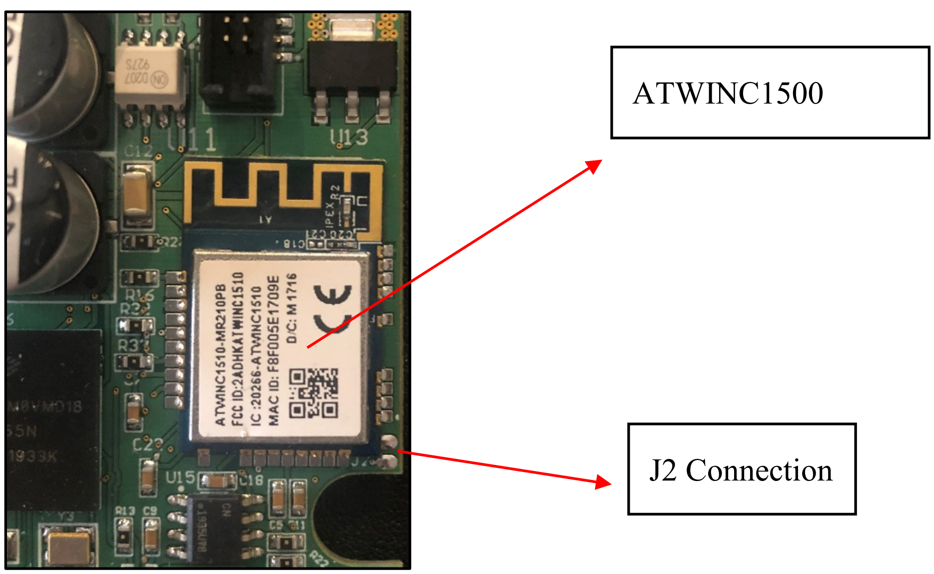


Figure 2-24. ATWINC1500 WiFi module and J2 connection

x. Voltage regulator and power protection

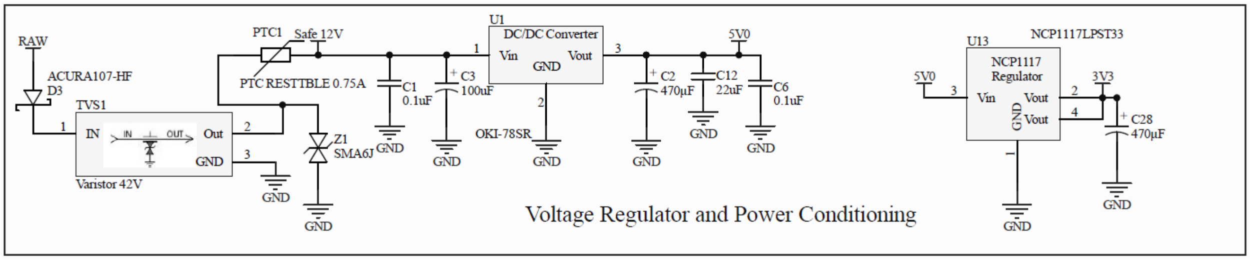


Figure 2-25. CAN Logger 3 schematics for voltage regulator and power protection

Figure 2-25 shows the voltage regulator and power protection schematics for the CAN Logger 3. To protect the device from high voltage spikes as well as EMI from the raw power supplied by the vehicle diagnostic port, the V2F118A400Y2EDP 42V transient voltage suppression (TVS) device [48] is used at TVS1. The specifications of the TVS are chosen based on experimenting with different voltages to achieve a maximum working voltage of 36V because this is approximately the cut-off for all other components that use the 12V line as well. How the testing was done will be discussed in the later section. The cost for the TVS is $0.60 per unit.

High current protection is done using the OZCG series PTC [49] at PTC1. When the temperature increases due to the excessive current passing through, the resistance of the PTC will increase as well to limit the current flow. When the current drops, the PTC will go back to normal state. The maximum working voltage is 33V and the hold current is 750 mA, which is the same as the TVS feedthrough current. The cost for each PTC is $0.21.

An ACURA107 Schottky diode [50] and a SMA6J series TVS diode [51] are used to protect the CAN Logger 3 against reverse polarity, as seen in D3 and Z1. The Schottky diode only allows current to flow one way; however, the voltage drops across it is smaller than the one from a regular diode. The TVS diode is bi-directional and helps protect sensitive electronic equipment from voltage transients induced by transient voltage events. The costs for the Schottky diode and the TVS diode are $0.39 and $0.53, respectively.

Two voltages regulators are used: the OKI-78SR [52] and the NCP1117 [53]. The OKI-78SR regulator takes in 12V (max 36V) input and outputs 5V while the NCP117 takes in 5V (max 18V) and outputs 3.3V. These two regulators are reused from the SSS2 design due to their robust and reliability as they have built-in protection against short circuit and high current, thermal, and noises. The costs for the OKI-78SR and the NCP1117 are $4.30 and $0.45, respectively.

C1, C6, and C12 capacitors are bypass resistors that help reduce high frequency AC noise present on the DC signal. C2, C3, and C28 are large capacitors used to keep the CAN Logger 3 functioning long enough to finalize the logging session at power loss. Similar to the TVS specification, the amount of capacitance is chosen based on experimenting, which will be validated in the testing section.

xi. Voltage monitoring

The external voltage monitoring feature is added mainly to help detect power loss on the raw 12V line, especially for slow voltage drop. The voltage is measured with an analog input, which only takes up to 3.3V before frying the processor. Because of that, a voltage divider is applied to convert the 12V to a lower one by using a pair of resistors. With the given maximum voltage of 36V for the raw line and the desire 3.3V output, the two resistors are determined using Ohms law [54], with the values of 100k and 10k. A TVS similar to the one used in the power protection schematics is used to protect the circuit. A flyback diode and a bypass capacitor are also added to prevent AC noise and voltage spikes. Similar voltage monitoring application also applies to A7 pin on the CAN Logger 3 external D-sub 15 connector for a future option, as needed. The schematics for both are illustrated in Figure 2-26.

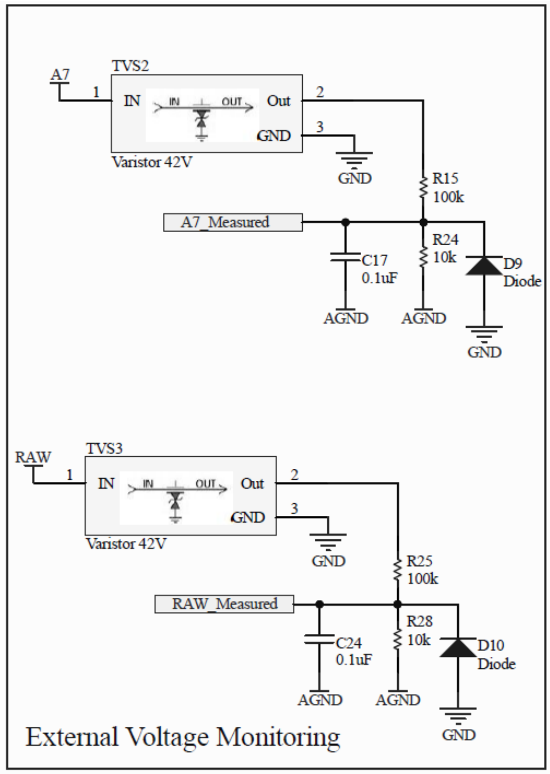


Figure 2-26. CAN Logger 3 schematics for external voltage monitoring

xii. Push button and optoisolator transistor output

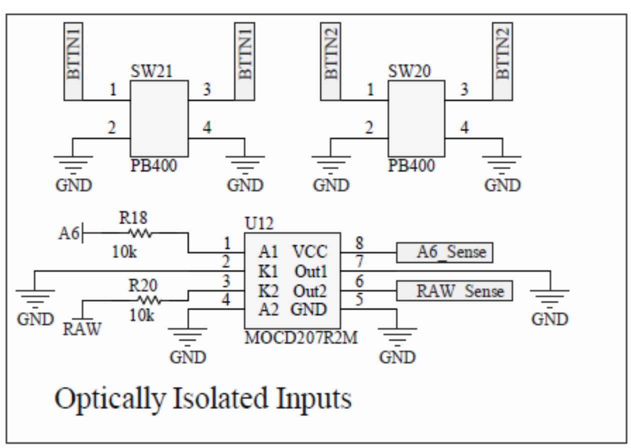


Figure 2-27. CAN Logger 3 schematics for push buttons and opto-isolator transistor output

Two PB400 push buttons [55] are added into the design for users to interact with the CAN Logger 3. Each button connects to a processor digital pin as a pull-up input. When the switch is closed, the pull-up input is connected to ground, which triggers an interrupt for a designated function. The buttons are Double Pole Single Throw (DPST) style, which means each of them has two inputs and two outputs, in which each input has one corresponding output. A Single Pole Single Throw (SPST) switch should be sufficient but because the design requires the button to point toward the end panel for easy fabrication, a right-angle configuration is needed. Thus, this push button is the most suitable one with a cost of $2.02 per unit.

Similarly, a MOCD208R2M opto-isolator transistor [56] is also added for a purpose of triggering an interrupt when from an external voltage. It consists of two infrared emitted diodes optically coupled to phototransistor detectors. Because of that, the transistor can detect sudden voltage drop from inputs precisely and quickly. In the design, the raw 12V line is one of the inputs for the transistor and its corresponding output is connected to a pull-up digital pin on the processor. During the power loss, the low voltage from the input will cause the transistor to ground the pull-up pin and hence, trigger an interrupt. Pin A6 on the D-Sub 15 connector is the other input for the transistor, which is currently saved for future option. The cost for this transistor is $0.98. Figure 2.27 shows the push buttons and the transistor schematics as described. R18 and R20 are used as pull-up resistors for the input voltage.

xiii. Mini USB

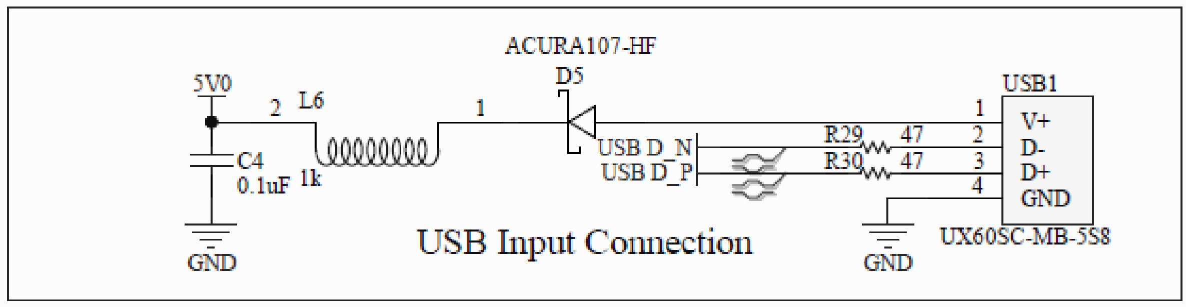


Figure 2-28. CAN Logger 3 schematics for mini USB connection

An USB connection is an essential component to interface with the processor for conveniently uploading or debugging firmware. A mini USB is one of the most commonly used types and thus, the UX60SC-MB-5S8 [57] connector was chosen for the design. The schematic for the USB connection is illustrated in Figure 2-28. There are four connections required on the USB: V+ (power), GND (ground), D+ (data+), D- (data-). D+ and D- are connected to USB D_P and D_N pins on the processor for data transmission, respectively. A 47-ohm resistor is placed between each data line to limit high current. V+ supplies 5V power, which has a Schottky diode D5 for reverse polarity protection, a 1k ferrite bead L6 (inductor) to flatten out voltage spikes from plugging and unplugging, and a bypass capacitor C4 to reduce AC noise that may be present. The cost of the mini USB connector is $1.05 per unit. Due to past experiences, the connector is mounted with two through-holes heavily filled with solder to prevent it from easily breaking off, as seen in Figure 2-29.

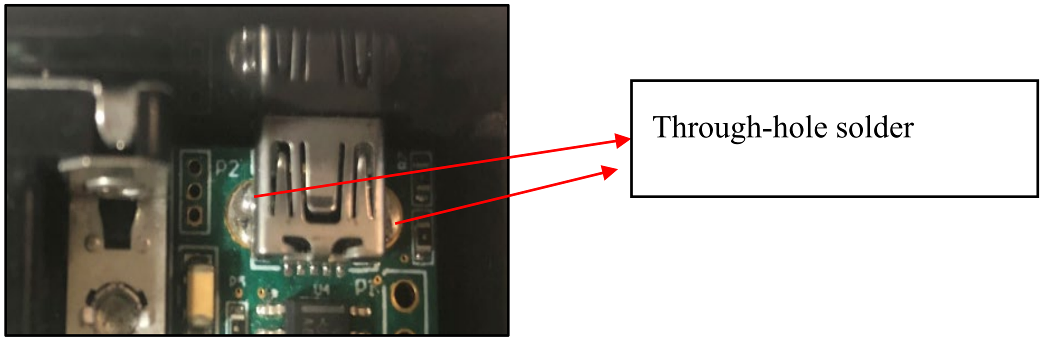


Figure 2-29. Mounting configuration for the mini USB connector

xiv. LEDs and light pipes

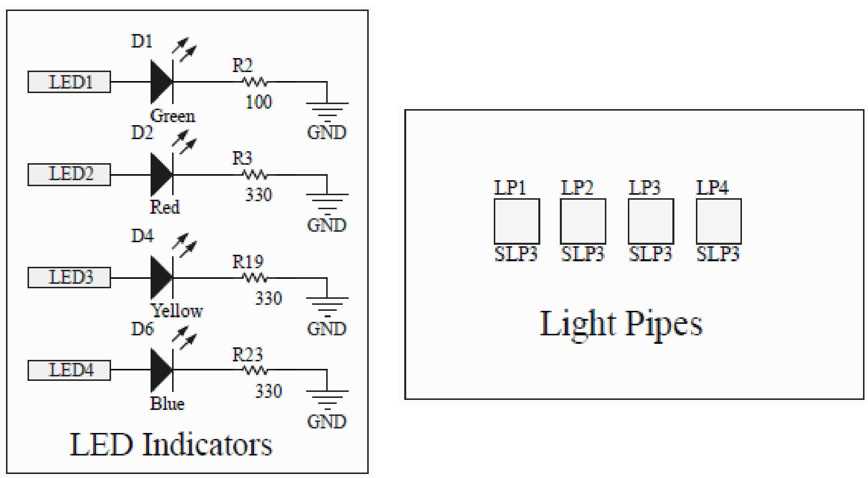


Figure 2-30. CAN Logger 3 schematics for LEDs and light pipes

Four LG R971 series LEDs [58] with color of red, green, yellow, and blue are used for operational indications, as shown in Figure 2-30. They are individually controlled by the processor with digital outputs. Originally, each LED had a 330 Ohms (Ω) resistor to limit current and control brightness. However, the green LED is too dim to be noticeable and therefore its resistor is adjusted to 100 Ω to make it significantly brighter. Each LED cost approximately $0.26.

Light pipes were needed to redirect the LED's light from the PCB to the end panel. The component had to fit well over the LED and required a right-angle configuration. Therefore, the SLP3 series light pipes [59] were selected and the cost per unit was $0.64.

xv. D-Sub 15 Connector

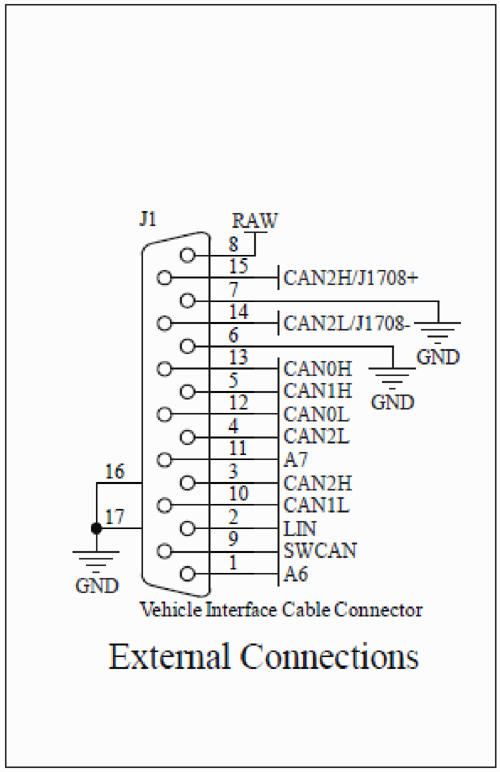


Figure 2-31. CAN Logger 3 schematics for D-Sub 15 connector

The D15S33E4GV00LF D-Sub 15 female connector [60] was chosen for the CAN Logger 3 main connector due to its known commonality, robust and reliability for major applications in the industry. The two threaded posts on each side help secure the cable connection under harsh operating conditions. Figure 2-31 shows the connector pinouts, which are consisted of the three CAN channels, LIN, SWCAN, raw 12V, ground, A6, and A7. A D-Sub 15 male connector to Deutsch 9-pin cable is used to connect the CAN Logger 3, as seen in Figure 2-

32. The pinouts of the Deutsch 9-pin side can be referred back to Figure 2-1. The D-Sub 15 connector and the D-Sub 15 to Deutsch 9-pin cable cost $2.30 and $11, respectively.

Figure 2-32. D-Sub 15 to Deutsch 9-pin cable

xvi. Bias resistors

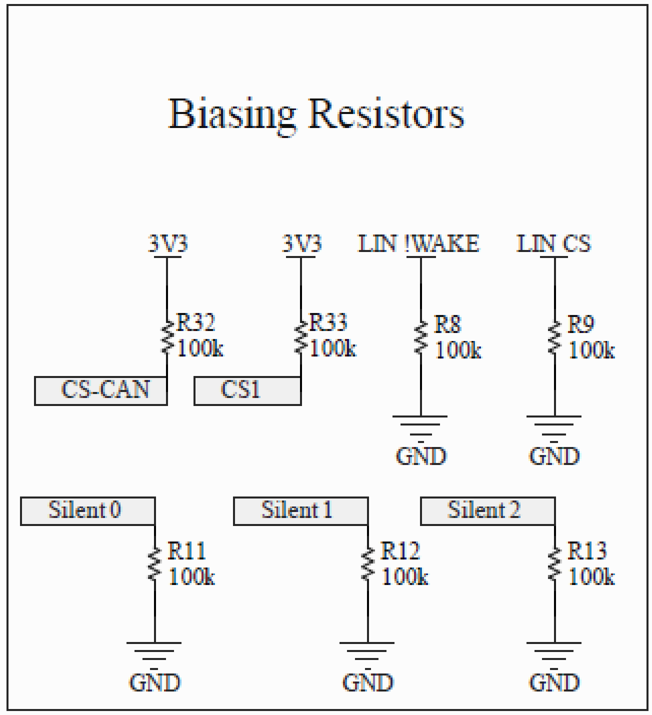


Figure 2-33. Schematics for biasing resistors

Figure 2-33 displays the rest of the biasing or pull-up/pull-down resistors that have not been included in previous schematics. There are four pull-up resistors for CS-CAN, CS1, LIN!WAKE, and LIN CS, and three pull-down resistors for Silent0, Silent1, and Silent2 pins. These resistors put the signal into known states if the processor is not driving them.

E. Printed Circuit Board (PCB) Layout

The CAN Logger 3 PCB was designed using Altium Designer software, as shown in Figure 2-34. Due to the limited board space and the complexity of the design, the PCB was built with four layers. With a dimension of 3.254” L by 2.229” W by 1.1” H, the handheld device is very compact and convenient for truck logging operation.

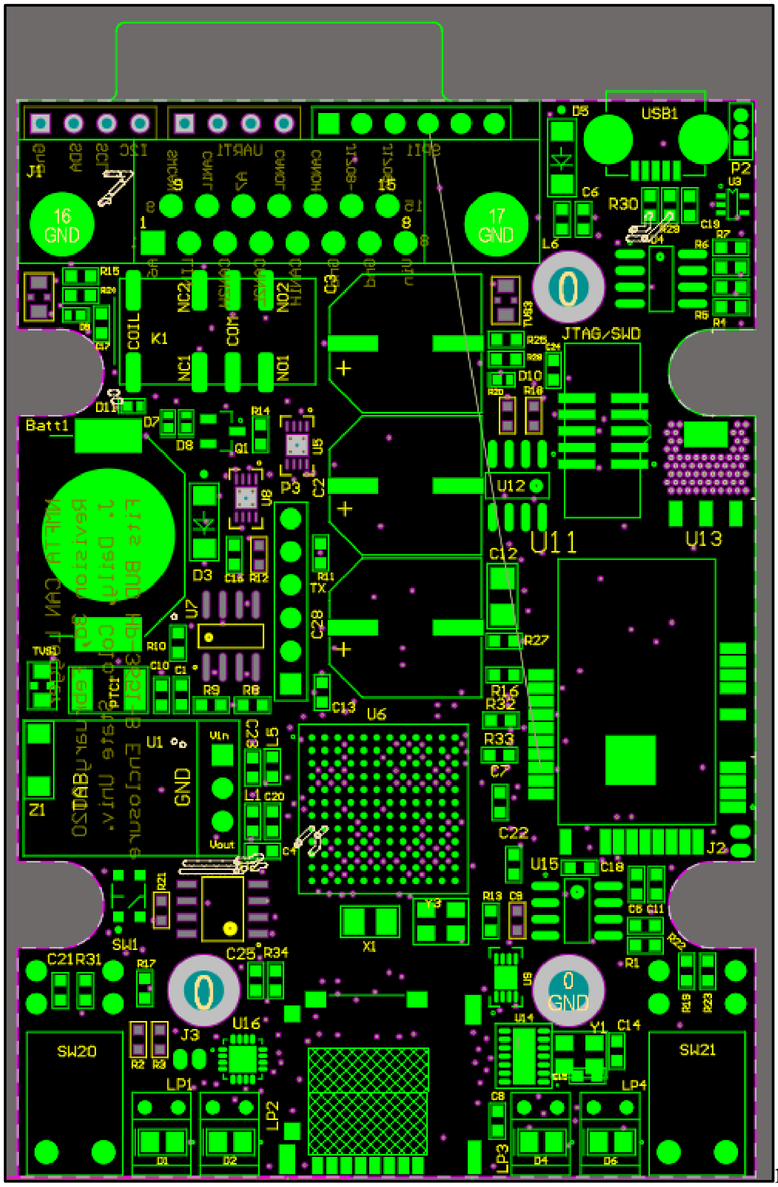


Figure 2-34. CAN Logger 3 PCB in Altium Designer

F. Bill of Materials

The complete Bill of Materials is shown in Table 2-3 for the printed circuit board.

Table 2-3. Bill of materials

Comment	Designator	Quantity	Supplier Part Number 1	Supplier Unit Price 1	Supplier 1
Yellow	D4	1	475-2560-1-ND	0.29	Digi-key
CR1225	Batt1	1	BAT-HLD-012-SMT-ND	0.29	Digi-key
0.1uF	C1, C4, C5, C6, C7, C8, C9, C11, C13, C17, C18, C19, C20, C21, C22, C23, C24, C25	18	399-6856-1-ND	0.101	Digi-key
2.2uF	C10, C16	2	587-3386-1-ND	0.12	Digi-key
22uF	C12	1	490-12451-1-ND	1.02	Digi-key
16pF	C14, C15	2	311-3964-1-ND	0.1	Digi-key
470µF	C2, C28	2	PCE3751CT-ND	0.81	Digi-key
100uF	C3	1	P19732CT-ND	0.81	Digi-key
Green	D1	1	475-1410-1-ND	0.27	Digi-key
Red	D2	1	475-1278-1-ND	0.29	Digi-key
ACURA107-HF	D3, D5	2	641-1884-1-ND	0.4	Digi-key
Blue	D6	1	516-1437-1-ND	0.95	Digi-key
Diode	D7, D8, D9, D10, D11	5	CCS15S30L3FCT-ND	0.38	Digi-key
Vehicle Interface Cable Connector	J1	1	609-1498-ND	2.3	Digi-key
JTAG/SWD	JTAG/SWD	1	1175-1735-ND	0.73	Digi-key
EE2-5SNU	K1	1	399-11056-5-ND	1.7	Digi-key
1k	L1, L5, L6	3	490-17350-1-ND	0.1	Digi-key
SLP3	LP1, LP2, LP3, LP4	4	492-2517-ND	0.64	Digi-key
PTC RESTTBLE 0.75A	PTC1	1	507-1765-1-ND	0.21	Digi-key
NUD3124	Q1	1	NUD3124LT1GOSCT-ND	0.41	Digi-key
4.7k	R1, R4, R7, R10, R21, R22	6	RHM4.7KAYCT-ND	0.14	Digi-key
10k	R18, R20, R24, R28, R34	5	RHM10.0KAYCT-ND	0.14	Digi-key
330	R3, R19, R23, R31	4	RHM330AYCT-ND	0.14	Digi-key
100	R2	1	RHM100AYCT-ND	0.14	Digi-key
47	R5, R6, R29, R30	4	RHM47AYCT-ND	0.14	Digi-key
100k	R8, R9, R11, R12, R13, R14, R15, R16, R17, R25, R27, R32, R33	13	RHM100KAYCT-ND	0.129	Digi-key
503182-1852	SD1	1	WM12834CT-ND	2.45	Digi-key
PTS830	SW1	1	CKN10587CT-ND	0.51	Digi-key
PB400	SW20, SW21	2	EG5548-ND	2.05	Digi-key
Varistor 42V	TVS1	1	478-2485-1-ND	0.62	Digi-key
Varistor 24V	TVS2, TVS3	2	478-2484-1-ND	0.79	Digi-key
OKI-78SR	U1	1	811-2692-ND	4.3	Digi-key
ATML-ATWINC1500-MR210PA-28	U11	1	ATWINC1510-MR210PB1140-ND	8.32	Digi-key
MOCD207R2M	U12	1	MOCD207R2MCT-ND	1.03	Digi-key
NCP1117LPST33	U13	1	NCP1117LPST33T3GOSCT-ND	0.46	Digi-key
MCP2517FD	U14	1	MCP2517FDT-H/JHACT-ND	2.31	Digi-key
ATECC608A	U15	1	ATECC608A-SSHDA-TCT-ND	0.75	Digi-key
MKL02Z32VFG4	U16	1	IC_MKL02Z32_QFN16	6.8	PJRC
NCV7356D1R2G	U2	1	NCV7356D1R2GOSCT-ND	1.8	Digi-key
74AHCT1G14	U3	1	74AHCT1G14SE-7DICT-ND	0.25	Digi-key
SN75HVD08DR	U4	1	296-37893-1-ND	3.97	Digi-key
MCP2558	U5, U8, U9	3	MCP2558FDT-H/MNYCT-ND	0.81	Digi-key
MK66FX1M0VMD18 (preprogrammed)	U6	1	568-13335-ND	17.65	Digi-key
MCP2003A-E/SN	U7	1	MCP2003A-E/SN-ND	0.82	Digi-key
UX60SC-MB-5S8	USB1	1	H11589CT-ND	1.05	Digi-key
32.768 KHz	X1	1	XC2292CT-ND	0.59	Digi-key
40MHz	Y1	1	XC3069CT-ND	0.5	Digi-key
16MHz	Y3	1	XC2866CT-ND	0.69	Digi-key
SMA6J	Z1	1	SMA6J24CA-TPMSCT-ND	0.41	Digi-key

The cost for each device, including manufacturing and final assembly, is approximately $180.

G. Assembly and Manufacturing

With the engineering of the system completed, the last step was shifting to manufacturing and assembly. Manufacturing the device included constructing the CAN Logger 3 PCBs as well as cutting enclosures to house and protect those boards.

The two chosen PCB manufacturers were Electronic Manufacturing Solutions Inc. (EMS) and Colorado PCB Assembly, who helped assemble all the electronic components on the PCBs using Surface-mount Technology (SMT). Because direct communication with those production specialists is essential for an affordable and reliable product, a visit to their facility has taken place to explore and understand the process of manufacturing PCBs with SMT. Figure 2-35 shows a picture of the visit at EMS's main facility.

Figure 2-35. Research team visiting Electronic Manufacturing Solutions Inc. in Arkansas

With full-turn-key assembly services, the Gerber files, NC Drill File, bill of materials, and pick-n-place files are supplied to the manufacturer and assembled devices are returned. An example of a completed PCBs order is shown in Figure 2-36.

Figure 2-36. Completed CAN Logger 3 PCBs

Because only PCB build operations were subcontracted, the rest of the manufacturing tasks - cutting the enclosure end panels - was completed with in-house laser cutter equipment. The sketches were made with manual measurements using a caliper and drawn on SolidWorks software, as shown in Figure 2-37. Figure 2-38 shows an example of the laser cutter being used to cut enclosures for the CAN Logger 3.

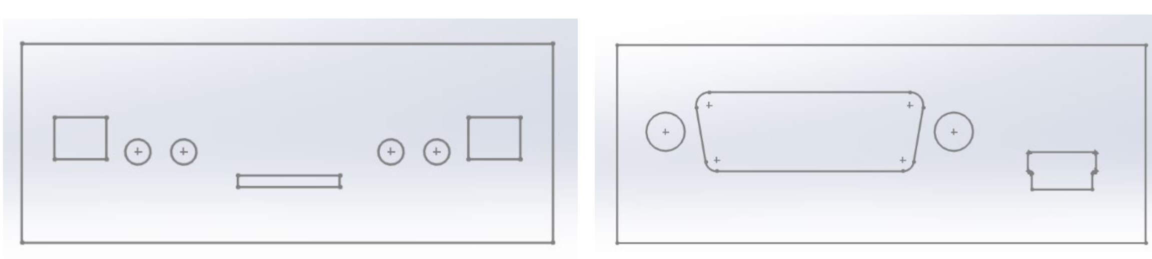


Figure 2-37. SolidWorks drawing of CAN Logger 3 enclosure end panels with SD card and buttons side (left) and D-SUB 15 cable and mini USB side (right)

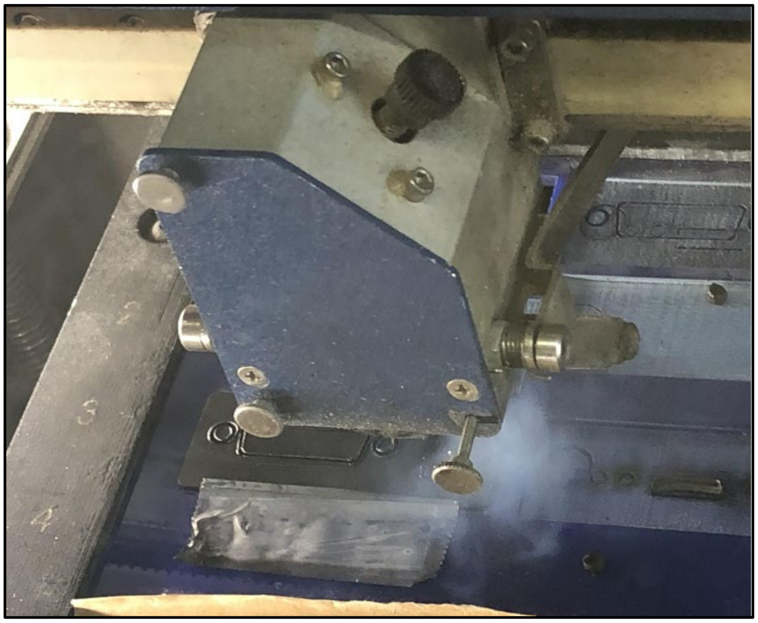


Figure 2-38. Laser cutter manufacturing for the CAN Logger 3 enclosure

Assembly refers to the process of taking all the individual components built by specialist manufacturers and putting them together to make a functioning product. Examples of the tasks involved include assembling PCB with their enclosures, labelling, and configuring firmware. Figure 2-39 shows a batch of assembled CAN Logger 3 devices after production.

Figure 2-39. CAN monitoring devices in production

H. Checklist

Before the CAN loggers were sent to the customers, they must go through a comprehensive checklist, which is illustrated below.

Complete this checklist before shipping a CAN Logger 3 to a Customer

☐ Customer Name: ____ ☐ Date: __________

☐ Remove SD card, connect logger to USB Serial and examine startup messages.

☐ With SD card removed, the red LED flashes.

☐ Logger time from USB Serial is within 1 minute of actual PC time.

☐ Logger and PC time zone: MDT (GMT-0700) or MST (GMT-0600) or Other: _________

☐ Red LED stops flashing after inserting an SD card.

☐ Enter the serial command for the ID (ID CSUXX) where XX is the logger number located on the enclosure.

☐ Device responds with Device ID: ___________________

☐ Enter the serial command to reset the count: COUNT 0.

☐ Device responds with Set current file to 000.

☐ Unplug and replug the USB Serial and observe solid green LED.

☐ Logger Number Printed on the enclosure: _________________

☐ The filename prefix matches the number printed on the enclosure.

☐ Connect Logger to live CAN bus. Observe Green and Yellow LED flickering.

☐ Record the ATECC608 SN:________________________________

☐ Record first digits of the IV:________________ If zeros, then no encryption.

☐ Press the left button (near green). Observe red LED slow flash.

☐ Double click left button (near green). Observe a new file was created.

☐ Previous file showed SIZE, BIN-SHA, TXT-SHA, and SIG.

☐ Note filename from Serial console.

☐ Disconnect USB Power first, then disconnect 12V Power.

☐ Remove SD Card from Logger, connect to computer.

☐ Open last file in hex editor (HxD) and calculate SHA-256:____________________

☐ Eject SD Card, reinsert to Logger, connect USB Serial.

☐ Previous file meta data shows BIN-SHA matching calculated SHA.

☐ Format SD card (FORMAT). Confirm with LS A being empty.

☐ Reset Counter to zero (COUNT 0).

☐ Logger Device ID and Serial Number match on

https://systemscyber.github.io/CAN-Logger-3/loggers.html

I. Functional Tests and Results

There are some crucial functions that the CAN Logger 3 has to properly perform to successfully fulfill the operational and performance requirements. This section discusses a series of comprehensive testing to ensure such functionality of the CAN Logger 3. The test scripts can be found in the GitHub repository at [23].

i. CAN0 and CAN1 test

The most important function of the CAN Logger 3 is to read and write CAN messages on the CAN0 and CAN1 channels. Two CAN loggers with the test script [61] and an SSS2 acting as a node with terminating resistors were connected to create a CAN bus for testing. The script from the test code is displayed below:

```
void setup() {
  //Set baudrate
  Can1.begin(BAUDRATE250K);
  Can0.begin(BAUDRATE250K);
}

void loop() {
// put your main code here, to run repeatedly:
if (Can0.available()) {
    Can0.read(rxmsg0);
    printFrame(rxmsg0,0,RXCount0++);//Print received frame
      }
if (Can1.available()) {
    Can1.read(rxmsg0);
    printFrame(rxmsg0,0,RXCount0++);//Print received frame
      }

//Write the message on CAN channel 0
      Can0.write(txmsg0);
//Write the message on CAN channel 1
      Can1.write(txmsg0);
}
```

In the setup function, both channels were initiated at 250kbps bitrate. In the loop function, the CAN Logger 3 would read any available message while writing a random frame on both CAN channels. Figure 2-40 displays the results from one of the CAN loggers 3 serial monitor window. The fact that there were messages shown up on both channels means the CAN Logger 3 was able to successfully write and read CAN messages on CAN0 and CAN1, and thus, passed the test.

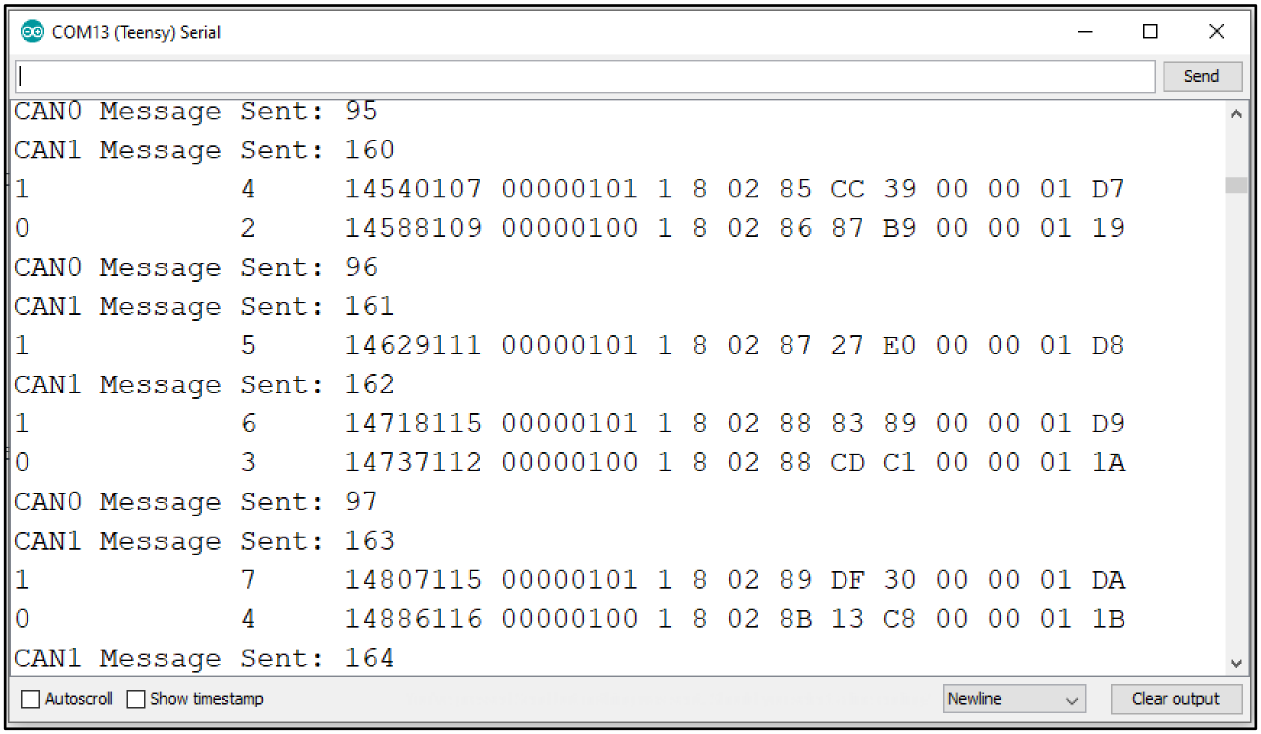


Figure 2-40. Results from one of the CAN loggers 3 serial monitor during the CAN0 and CAN1 testing

ii. Autobaud test

Autobaud or auto bitrate detection is an important feature of the CAN Logger 3. It is implemented by taking advantage of the Received Error Counter (REC), which can be found in the modified FlexCAN library at [25]. The process of how the autobaud feature works is indicated in Figure 2-41 below.

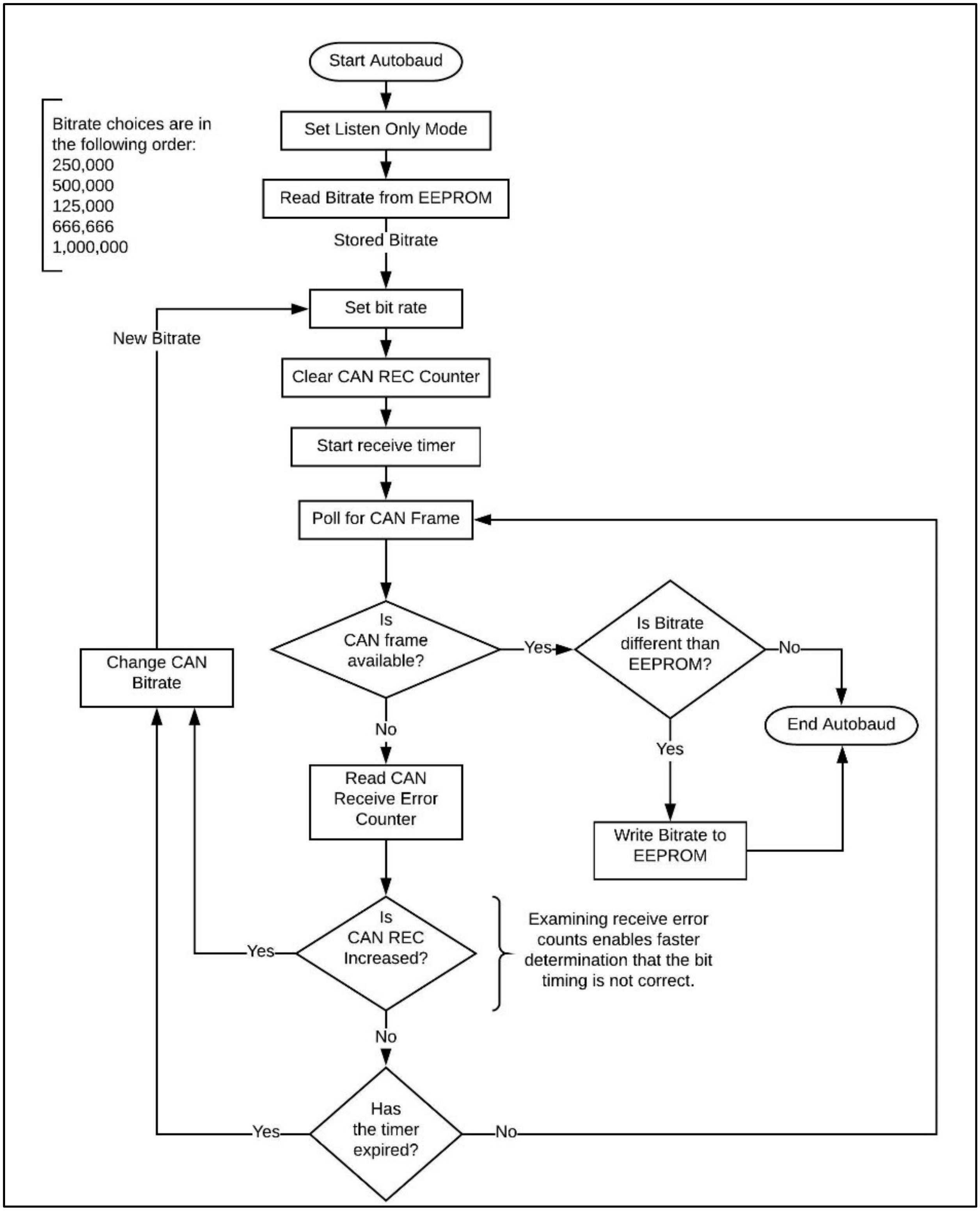


Figure 2-41. Autobaud feature diagram

The Autobaud process was designed based on the recommendations in J1939-16 Automatic Baud Rate Detection Process [62]. When the device is connected to the heavy vehicle network, the device will start Autobaud immediately and set to listen only mode. A sequence of bitrate choices is iterated through, which are 250,000, 500,000, 125,000, 666,666, and 1,000,000 bit/s. The device first starts with the initial bitrate read from EEPROM. The CAN REC and receive timer are reset to 0. The device then polls for any available CAN frame that it can detect with the current bitrate. There is a 150 milliseconds timeout for the receive timer. During that duration, if the device detects CAN frames, then it is on the correct bitrate. The device will update EEPROM with the correct bitrate setting if the previous one is different and end Autobaud feature.

However, if there is no CAN frame detected, the device will proceed to read the CAN REC. If the number increases, that means the device is on wrong bitrate setting and it will change the CAN bitrate to the next one on the list and repeat the process until the right one is selected. If the CAN REC does not increase, this could mean that there is no actual CAN message on the network. If the timer has not expired, the device will go back and continue to poll for CAN frame. Otherwise, the device will change the CAN bitrate. The sequence is repeated until CAN messages are available on the network.

To test the Autobaud feature, a setup of two networks with two different bitrates of 250kbps and 500kbps was made. The device was first plugged in to the network with 250kbps bitrate and starts logging. After that, it was plugged into the second network with 500kbps bitrate. The data was then examined, as seen in Figure 2-42. The metadata of the two log files show the bitrate on CAN0, which are 250kbps on one file and 500kbps on the other. In addition, the fact that the two corresponding log files were successfully created means the Autobaud feature passed the test.

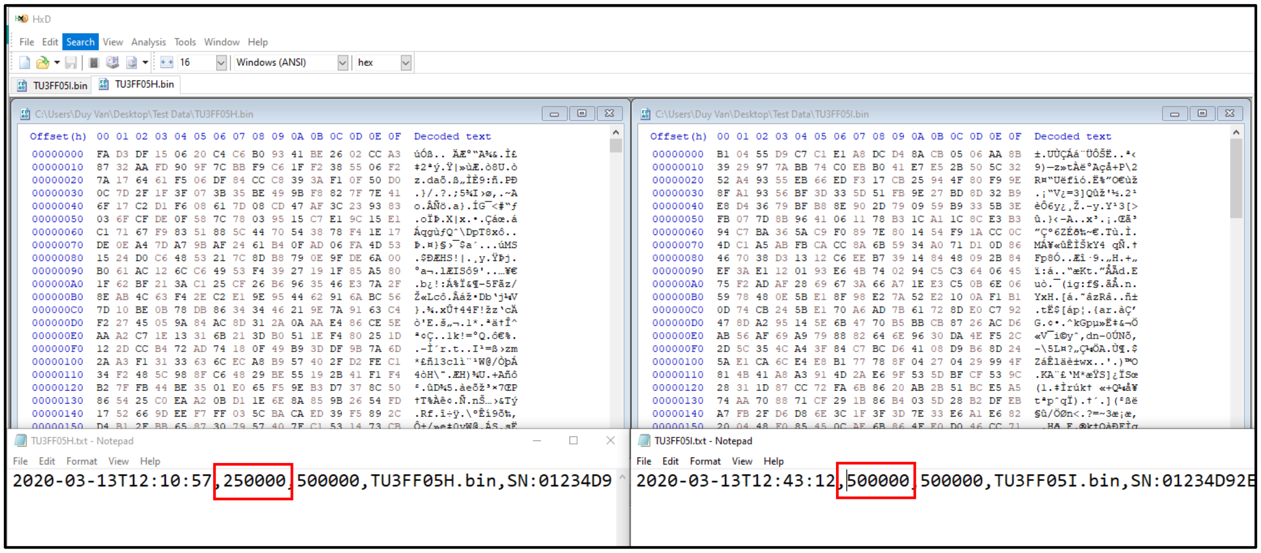


Figure 2-42. Autobaud test results from 250kbps (left) and 500kbps (right) bitrate networks

iii. AES-128 test

The log files are encrypted using the mmCAU with AES-128 algorithm. There are many AES encryption modes that can be implemented. Encryption using Electronic Code Book (ECB) mode is the first generation of AES and the most basic form of block cipher encryption. It breaks up the input data into many 16-byte blocks and encrypts them individually using its AES session key. Thus, data of any size can be used as input and will be padded to the size that is divisible by 16 if necessary. However, the disadvantage of this mode is that it lacks diffusion. If identical 16-byte blocks are encrypted in ECB, the results are also identical. As a result, this can expose data patterns and does not provide true confidentiality. As a matter of fact, a study on ciphertext entropy has proved that encryption using ECB mode is not suitable for image or text files that have repeated identical data [63]. This is crucial because some CAN frames are periodic, meaning that the same data are sent within the same constant interval. Thus, encrypting CAN data using ECB mode is vulnerable. AES in the cipher block chaining (CBC) mode is used to overcome this problem where an initialization vector (IV) or so-called salt, which is an arbitrary number that is only used once, is XORed with the first block, and the cipher result is then XORed with the next block and so on. Therefore, each cipher block depends on all the previous ones, which scrambles the patterns and creates diffusion. Figure 2-43 and Figure 2-44 illustrate ECB and CBC modes for AES encryption processes, respectively.

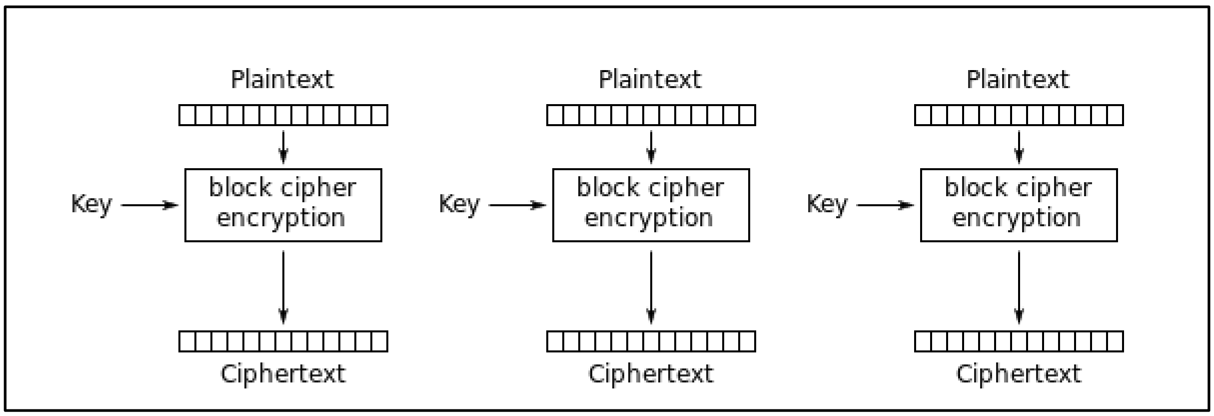


Figure 2-43. Electronic Codebook mode encryption [64]

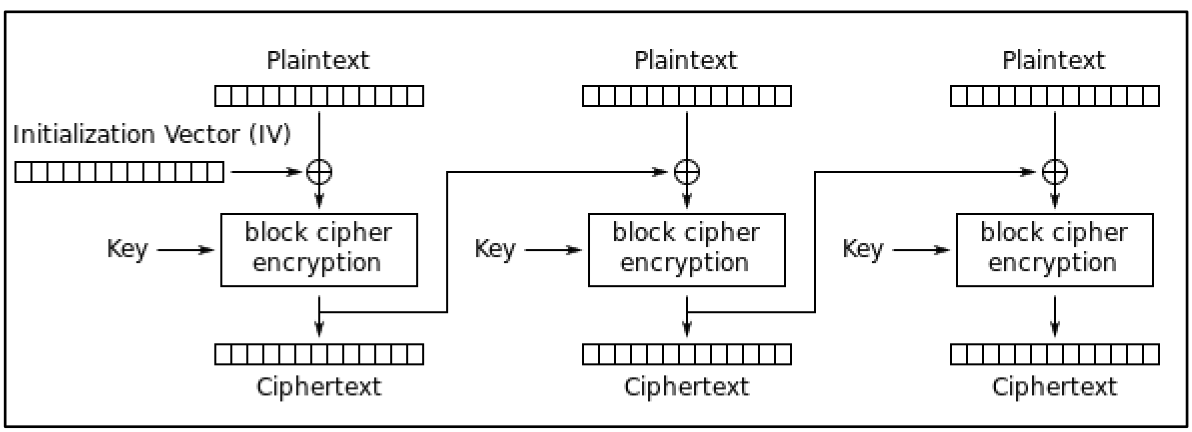


Figure 2-44. Cipher Block Chaining mode encryption [65]

A good graphical demonstration showing the differences between the ECB and CBC mode is the famous ECB Penguin [66], as seen in Figure 2-45. Pixels of the image were encrypted using ECB and the result clearly shows the input pattern. On the other hand, the encrypted image from CBC mode resulted in pseudo-randomness.

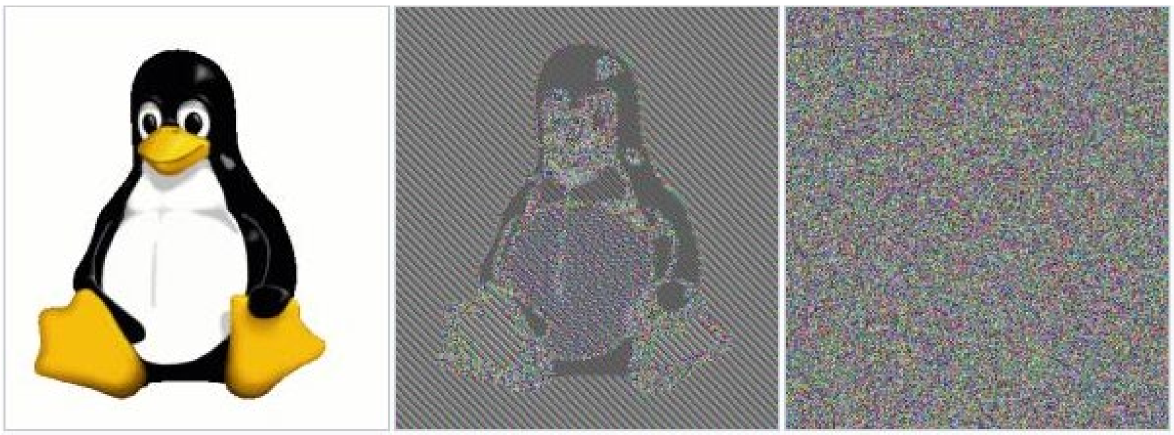


Figure 2-45. Graphical demonstration between original image (left) and image encrypted using ECB (middle) and image encrypted using CBC (right) [66]

The mmCAU uses the cryptolibAESSHA library [67] to implement its AES capability, with an Arduino interface published by Paul Stoffregen [68]. The AES-128 CBC encryption and decryption were tested against NIST test vectors [69]. The main functions from the test code [70] is displayed below:

```
mmcau_aes_set_key(aeskey,128,keysched);//Set key schedule

//Encryption
memcpy(out,init_vector,16); //Load IV
aes_cbc_encrypt(data_to_encrypt,cipher_text);

//Print out the first block of the cipher (16 bytes)
Serial.print("\nBlock 1 Cipher Text: ");
for (i=0;i<block_size;i++){
    sprintf(str,"%02X",cipher_text[i]);
    Serial.print(str);
  }
//Print out test vector to compare
Serial.print("\nTest Vector Block 1: 7649abac8119b246cee98e9b12e9197d");

//Decryption
memcpy(iv,init_vector,16);; //Load IV
aes_cbc_decrypt(clear_text,data_to_decrypt);

//Print out the first block of the cipher (16 bytes)
Serial.print("\nBlock 1 Clear Text: ");
for (i=0;i<block_size;i++){
    sprintf(str,"%02X",clear_text[i]);
    Serial.print(str);
  }
//Print out test vector to compare
Serial.print("\nTest Vector Block 1: 6bc1bee22e409f96e93d7e117393172a");
```

Initially, the AES key schedule was set and the IV was loaded in order to run encryption and decryption correctly. The *aes_cbc_encryption* and *aes_cbc_decrypt* functions have been added because the cryptolibAESSHA library only supports AES ECB. The AES-128 CBC encryption and decryption were tested against the NIST test vectors and the outputs, which were displayed in blocks of 16-byte, were compared to the expected values, as seen in Figure 2-46. The results show exact match, which means the test passed.

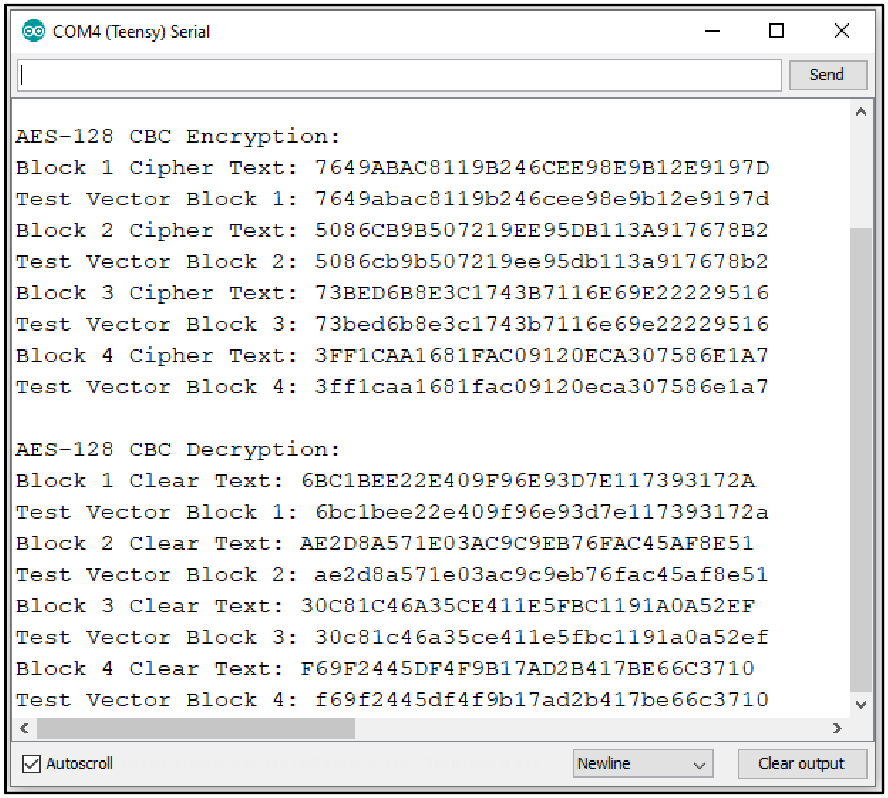


Figure 2-46. Test results for AES-128 CBC mode encryption and decryption

iv. Logging speed test

This test explored the actual AES encryption speed of the mmCAU and verified that the CAN Logger 3 was able to log data at full bus load. An Arduino script [71] was written to measure the rate of mmCAU encryption, which can be seen below:

```
//16-byte block encryption
t = micros();
cau_aes_encrypt (in, keysched, AES_128_NROUNDS, cipher_text);
t = micros() - t;
sprintf(str, "aes %d bytes %u us  KBs  ", sizeof(in), t);
Serial.print(str);
Serial.println(1000.*sizeof(in) / t);

//512-byte block encryption
t = micros();
aes_cbc_encrypt(data,cipher_text);
t = micros() - t;
sprintf(str, "aes cbc encrypt %d bytes %u us, KBs ", sizeof(data), t);
Serial.print(str);
Serial.println(1000.*sizeof(data) / t);
```

The script measured the time the mmCAU took to encrypt a 16-byte block using ECB and a 512-byte block using the CBC added function, in microseconds. The 512-byte size was chosen for this time testing because that is the buffer size the CAN Logger 3 adopts in its software design, which will be discussed further in Chapter 3. The results, as illustrated in Figure 2-47, show that encrypting 16-bytes took about 2 microseconds, which is equivalent to 8 Mbyte/second. However, encrypting a 512-byte took 80 microseconds, which is equivalent to 6.4 Mbyte/sec. The loss in speed was expected because CBC mode required more computing power than ECB.

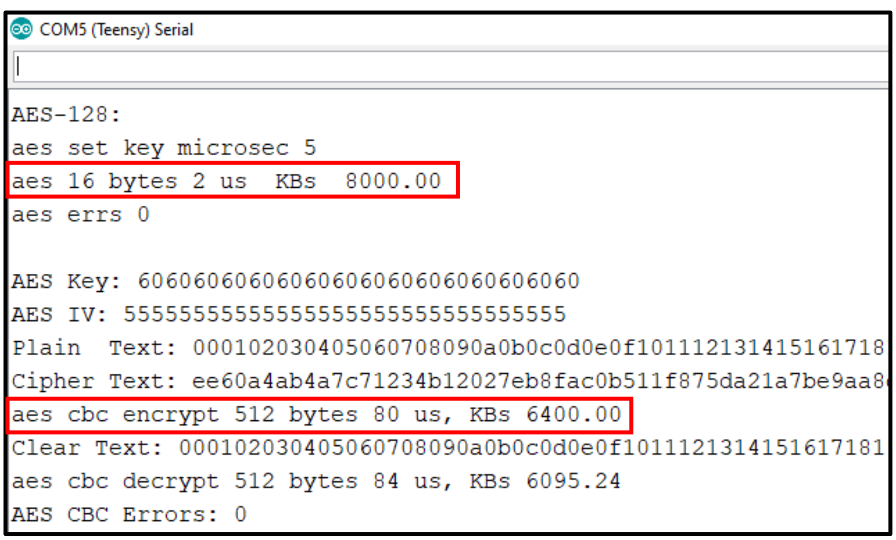


Figure 2-47. AES-128 encryption speed

However, to actually prove that the CAN Logger 3 has sufficient logging speed while implementing AES-128 CBC encryption, the device was tested at full bus load against a TruckCape with BeagleBone Black platform, which is a known CAN device that is capable of logging CAN messages at such speeds. The test started with 250kbps bus because it is the most common bitrate. At the beginning of the assessment, CAN messages were injected on both CAN0 and CAN1 as fast as possible using a Teensy 3.6 board. The CAN messages have a fixed ID of 0, and a data field starting at 0 and increasing by 1 for every message sent by the microprocessor. Both the TruckCape and the CAN Logger 3 devices then started logging the session. The CAN Logger 3 encrypted log data was decrypted afterward for comparison. The network speed was also measured using the TruckCape to ensure that bus load was indeed at 100% as intended, as shown in Figure 2-48. The 101% reading was due to bit stuffing and small percent errors. Another indicator showing that the bus was at full speed was that all the original ECM occurring normally were oppressed and only the test messages were present on the bus.

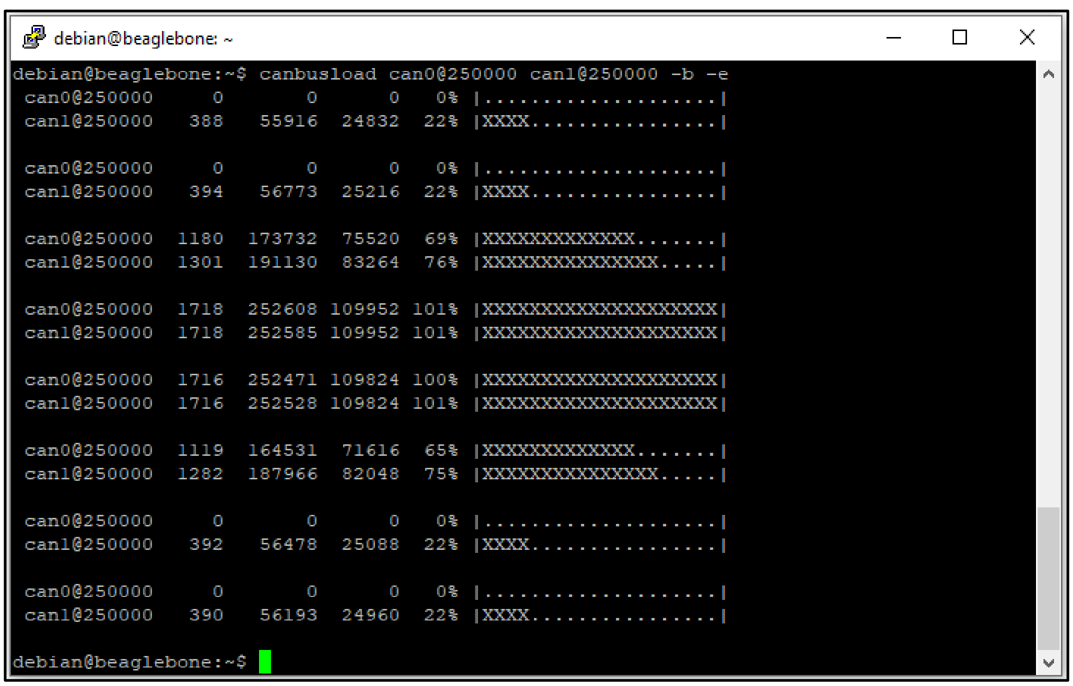
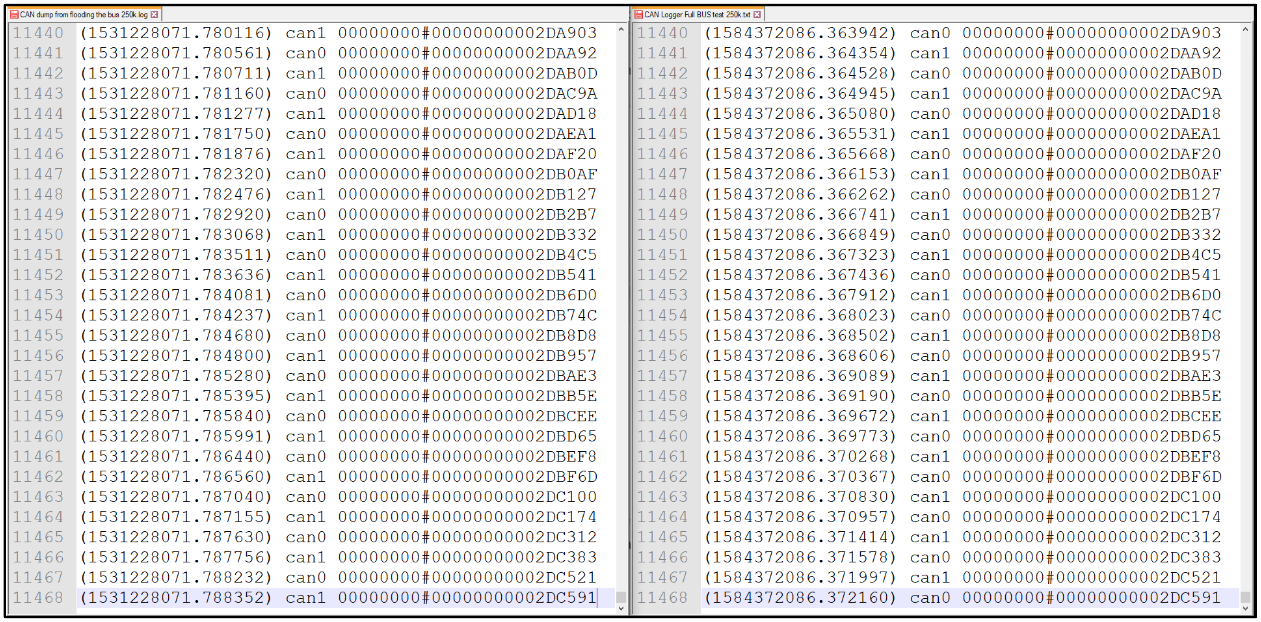


Figure 2-48. Bus load measurement by the TruckCape device



```
11440 (1531228071.780116) can1 00000000#00000000002DA903
11441 (1531228071.780561) can0 00000000#00000000002DAA92
11442 (1531228071.780711) can1 00000000#00000000002DAB0D
11443 (1531228071.781160) can0 00000000#00000000002DAC9A
11444 (1531228071.781277) can1 00000000#00000000002DAD18
11445 (1531228071.781750) can0 00000000#00000000002DAEA1
11446 (1531228071.781876) can1 00000000#00000000002DAF20
11447 (1531228071.782320) can0 00000000#00000000002DB0AF
11448 (1531228071.782476) can1 00000000#00000000002DB127
11449 (1531228071.782920) can0 00000000#00000000002DB2B7
11450 (1531228071.783068) can1 00000000#00000000002DB332
11451 (1531228071.783511) can0 00000000#00000000002DB4C5
11452 (1531228071.783636) can1 00000000#00000000002DB541
11453 (1531228071.784081) can0 00000000#00000000002DB6D0
11454 (1531228071.784237) can1 00000000#00000000002DB74C
11455 (1531228071.784680) can0 00000000#00000000002DB8D8
11456 (1531228071.784800) can1 00000000#00000000002DB957
11457 (1531228071.785280) can0 00000000#00000000002DBAE3
11458 (1531228071.785395) can1 00000000#00000000002DBB5E
11459 (1531228071.785840) can0 00000000#00000000002DBCEE
11460 (1531228071.785991) can1 00000000#00000000002DBD65
11461 (1531228071.786440) can0 00000000#00000000002DBEF8
11462 (1531228071.786560) can1 00000000#00000000002DBF6D
11463 (1531228071.787040) can0 00000000#00000000002DC100
11464 (1531228071.787155) can1 00000000#00000000002DC174
11465 (1531228071.787630) can0 00000000#00000000002DC312
11466 (1531228071.787756) can1 00000000#00000000002DC383
11467 (1531228071.788232) can0 00000000#00000000002DC521
11468 (1531228071.788352) can1 00000000#00000000002DC591
```

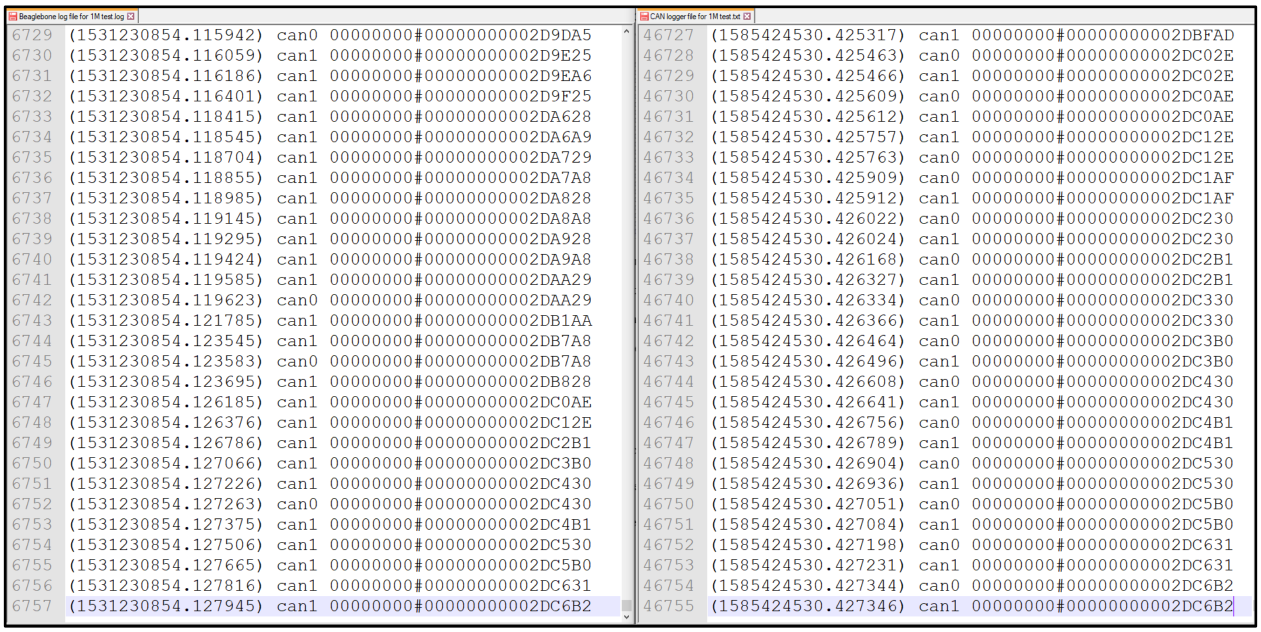
```
11440 (1584372086.363942) can0 00000000#00000000002DA903
11441 (1584372086.364354) can1 00000000#00000000002DAA92
11442 (1584372086.364528) can0 00000000#00000000002DAB0D
11443 (1584372086.364945) can1 00000000#00000000002DAC9A
11444 (1584372086.365080) can0 00000000#00000000002DAD18
11445 (1584372086.365531) can1 00000000#00000000002DAEA1
11446 (1584372086.365668) can0 00000000#00000000002DAF20
11447 (1584372086.366153) can1 00000000#00000000002DB0AF
11448 (1584372086.366262) can0 00000000#00000000002DB127
11449 (1584372086.366741) can1 00000000#00000000002DB2B7
11450 (1584372086.366849) can0 00000000#00000000002DB332
11451 (1584372086.367323) can1 00000000#00000000002DB4C5
11452 (1584372086.367436) can0 00000000#00000000002DB541
11453 (1584372086.367912) can1 00000000#00000000002DB6D0
11454 (1584372086.368023) can0 00000000#00000000002DB74C
11455 (1584372086.368502) can1 00000000#00000000002DB8D8
11456 (1584372086.368606) can0 00000000#00000000002DB957
11457 (1584372086.369089) can1 00000000#00000000002DBAE3
11458 (1584372086.369190) can0 00000000#00000000002DBB5E
11459 (1584372086.369672) can1 00000000#00000000002DBCEE
11460 (1584372086.369773) can0 00000000#00000000002DBD65
11461 (1584372086.370268) can1 00000000#00000000002DBEF8
11462 (1584372086.370367) can0 00000000#00000000002DBF6D
11463 (1584372086.370830) can1 00000000#00000000002DC100
11464 (1584372086.370957) can0 00000000#00000000002DC174
11465 (1584372086.371414) can1 00000000#00000000002DC312
11466 (1584372086.371578) can0 00000000#00000000002DC383
11467 (1584372086.371997) can1 00000000#00000000002DC521
11468 (1584372086.372160) can0 00000000#00000000002DC591
```

Figure 2-49. Full bus load logging result at 250kbps

for TruckCape (left) and CAN Logger 3 (right)

Figure 2-49 shows the log data of the two devices with message intervals of 600 µs on each channel at 250kbps. Given that a 29-bit CAN frame has a size of 126 bits plus bit stuffing (insertion of one opposite bit if five similar consecutive bits are sent), an average interval of 600 µs seemed appropriate in this case. In a 250kbps bus, a maximum of 250,000 bits can be transferred per second on the network and therefore, CAN messages will be lost or dropped on the bus if CAN nodes send data at a faster rate. As a result, it made sense that the data field in the captured CAN messages did not increase by one for every message under the full bus load test, which was desired for this experiment. The two files have the same number of messages (11,468 frames) and data contents, along with the results from the bus load measurement , the corresponding message interval, and the extreme log speed with encryption, it was safe to conclude that the CAN Logger 3 is capable of logging messages with AES encryption at 250kbps even at 100% bus load.

To really push the CAN Logger 3 limit, the same experiment was done again at the 1Mbps CAN bitrate because the message interval will be shorter. Figure 2-50 shows the log data of the two devices with message intervals of 150 µs, 4 times shorter than the one from 250kbps bus. The results show that the TruckCape captured 6,757 messages while the CAN Logger 3 captured 46,755 messages. This indicates that the TruckCape was not able to keep up with the speed and dropped messages, as seen in its log where can0 messages were missing in some areas.



```
Beaglebone log file for 1M test.log
6729 (1531230854.115942) can0 00000000#00000000002D9DA5
6730 (1531230854.116059) can1 00000000#00000000002D9E25
6731 (1531230854.116186) can1 00000000#00000000002D9EA6
6732 (1531230854.116401) can1 00000000#00000000002D9F25
6733 (1531230854.118415) can1 00000000#00000000002DA628
6734 (1531230854.118545) can1 00000000#00000000002DA6A9
6735 (1531230854.118704) can1 00000000#00000000002DA729
6736 (1531230854.118855) can1 00000000#00000000002DA7A8
6737 (1531230854.118985) can1 00000000#00000000002DA828
6738 (1531230854.119145) can1 00000000#00000000002DA8A8
6739 (1531230854.119295) can1 00000000#00000000002DA928
6740 (1531230854.119424) can1 00000000#00000000002DA9A8
6741 (1531230854.119585) can1 00000000#00000000002DAA29
6742 (1531230854.119623) can0 00000000#00000000002DAA29
6743 (1531230854.121785) can1 00000000#00000000002DB1AA
6744 (1531230854.123545) can1 00000000#00000000002DB7A8
6745 (1531230854.123583) can0 00000000#00000000002DB7A8
6746 (1531230854.123695) can1 00000000#00000000002DB828
6747 (1531230854.126185) can1 00000000#00000000002DC0AE
6748 (1531230854.126376) can1 00000000#00000000002DC12E
6749 (1531230854.126786) can1 00000000#00000000002DC2B1
6750 (1531230854.127066) can1 00000000#00000000002DC3B0
6751 (1531230854.127226) can1 00000000#00000000002DC430
6752 (1531230854.127263) can0 00000000#00000000002DC430
6753 (1531230854.127375) can1 00000000#00000000002DC4B1
6754 (1531230854.127506) can1 00000000#00000000002DC530
6755 (1531230854.127665) can1 00000000#00000000002DC5B0
6756 (1531230854.127816) can1 00000000#00000000002DC631
6757 (1531230854.127945) can1 00000000#00000000002DC6B2
```

```
CAN logger file for 1M test.txt
46727 (1585424530.425317) can1 00000000#00000000002DBFAD
46728 (1585424530.425463) can0 00000000#00000000002DC02E
46729 (1585424530.425466) can1 00000000#00000000002DC02E
46730 (1585424530.425609) can0 00000000#00000000002DC0AE
46731 (1585424530.425612) can1 00000000#00000000002DC0AE
46732 (1585424530.425757) can1 00000000#00000000002DC12E
46733 (1585424530.425763) can0 00000000#00000000002DC12E
46734 (1585424530.425909) can0 00000000#00000000002DC1AF
46735 (1585424530.425912) can1 00000000#00000000002DC1AF
46736 (1585424530.426022) can0 00000000#00000000002DC230
46737 (1585424530.426024) can1 00000000#00000000002DC230
46738 (1585424530.426168) can0 00000000#00000000002DC2B1
46739 (1585424530.426327) can1 00000000#00000000002DC2B1
46740 (1585424530.426334) can0 00000000#00000000002DC330
46741 (1585424530.426366) can1 00000000#00000000002DC330
46742 (1585424530.426464) can0 00000000#00000000002DC3B0
46743 (1585424530.426496) can1 00000000#00000000002DC3B0
46744 (1585424530.426608) can0 00000000#00000000002DC430
46745 (1585424530.426641) can1 00000000#00000000002DC430
46746 (1585424530.426756) can0 00000000#00000000002DC4B1
46747 (1585424530.426789) can1 00000000#00000000002DC4B1
46748 (1585424530.426904) can0 00000000#00000000002DC530
46749 (1585424530.426936) can1 00000000#00000000002DC530
46750 (1585424530.427051) can0 00000000#00000000002DC5B0
46751 (1585424530.427084) can1 00000000#00000000002DC5B0
46752 (1585424530.427198) can0 00000000#00000000002DC631
46753 (1585424530.427231) can1 00000000#00000000002DC631
46754 (1585424530.427344) can0 00000000#00000000002DC6B2
46755 (1585424530.427346) can1 00000000#00000000002DC6B2
```

Figure 2-50. Full bus load logging result at 1Mbps

for TruckCape (left) and CAN Logger 3 (right)

Figure 2-51 shows the graphical results from the two tests for better visualization. The horizontal axis represents the value, which has been converted from hexadecimal, in the data field from the captured CAN messages. The horizontal axis has been rescaled and only shows values ranging from 25,000 to 35,000 for better demonstration; however, it is not a full result. There are eight sets of data illustrated in the graph, created from CAN Logger 3 (CL3) and TruckCape (TC) logs at 250kbps and 1Mbps speed with CAN0 and CAN1 channels separated.

The gaps between the data point for each set were the dropped messages. In the 1Mbps bus, the gaps were smaller because the bus can transmit more data and hence, less CAN messages were dropped. Similar to the previous conclusions, the CAN Logger 3 and the TruckCape had the same performance at 250kbps, as seen with the same data pattern. On the other hand, the TruckCape failed to capture all messages on the bus at 1Mbps, as seen with large gaps with missing data points.

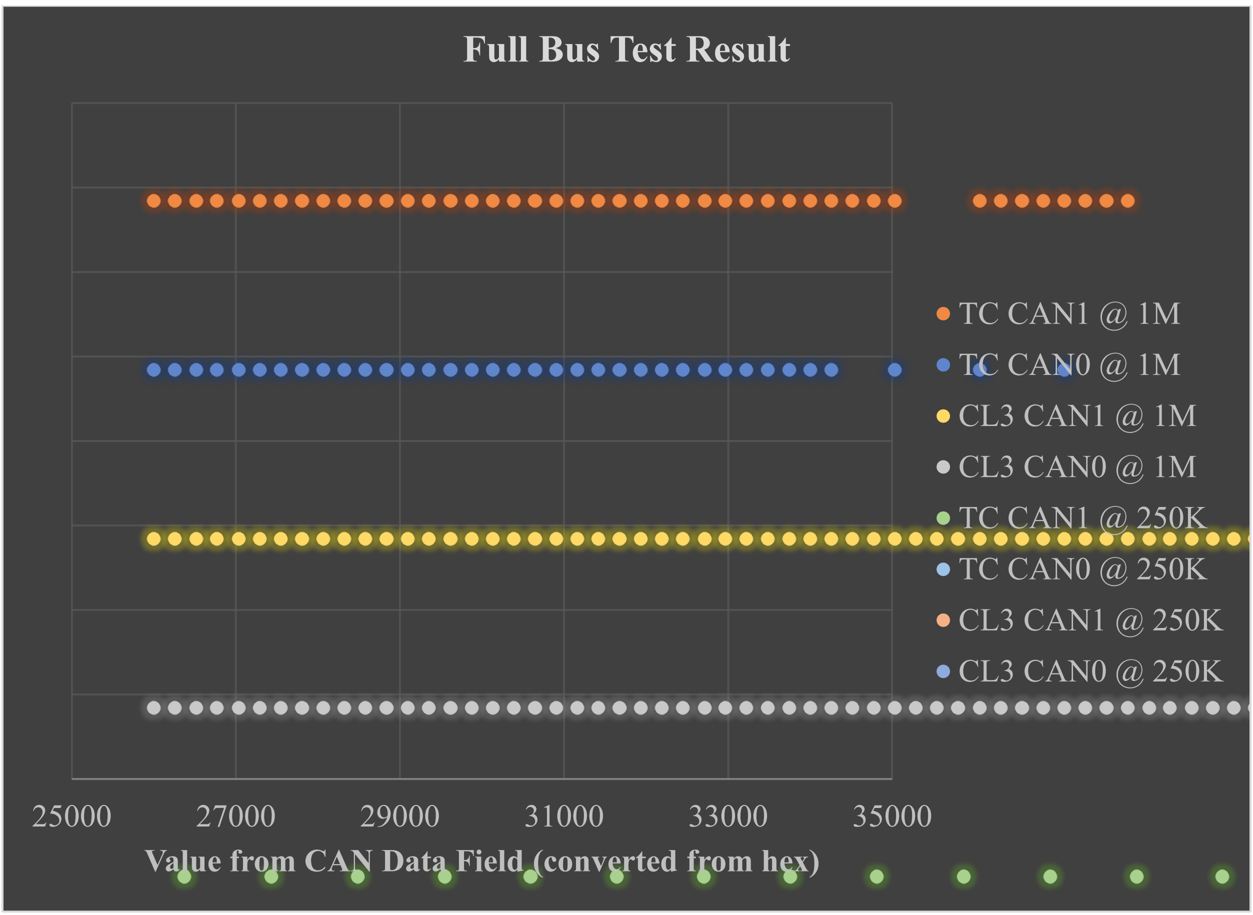


Figure 2-51. CAN Logger 3 and TruckCape full bus test results at 250kbps and 1Mbps

An addition test was conducted where the number of messages sent at full speed was reflected on the CAN Logger 3 captured data. The setup used was similar to the one from the previous test but the TruckCape device was no longer needed. 4,000 CAN messages with ID of 0x15555555 and data of all 0xAs were sent on both CAN channels at a fixed interval of 520.5 microseconds on a 250kpbs CAN bitrate. The message ID and data were selected as such to minimize bit stuffing as much as possible to really stress the bus because many more messages can be delivered over a period of time. The message interval was derived based on the one from the previous test at full bus load, and it was adjusted through many trials to achieve 100% bus load measurement, as seen in Figure 2-52. Again, the 101% reading was due to bit stuffing and small percent errors.

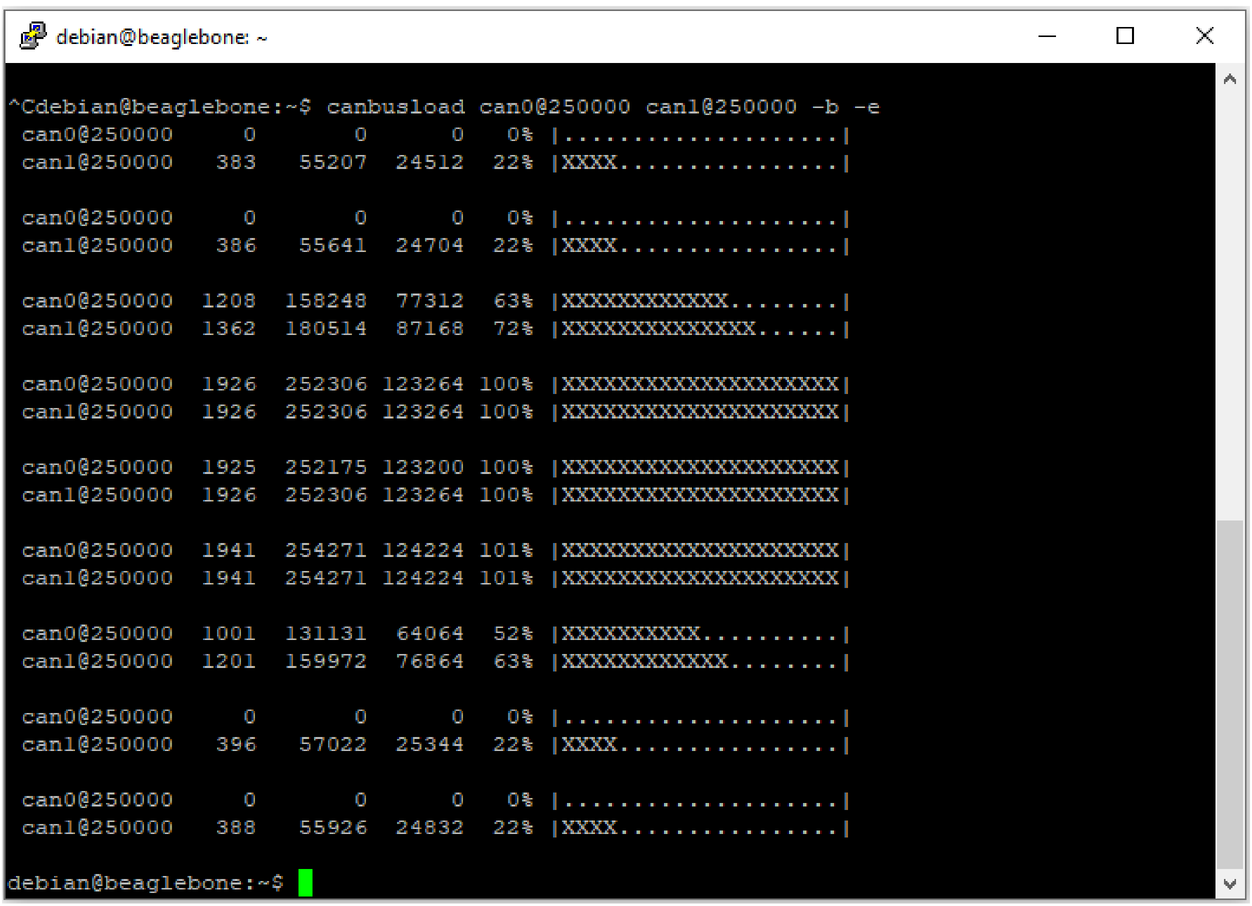
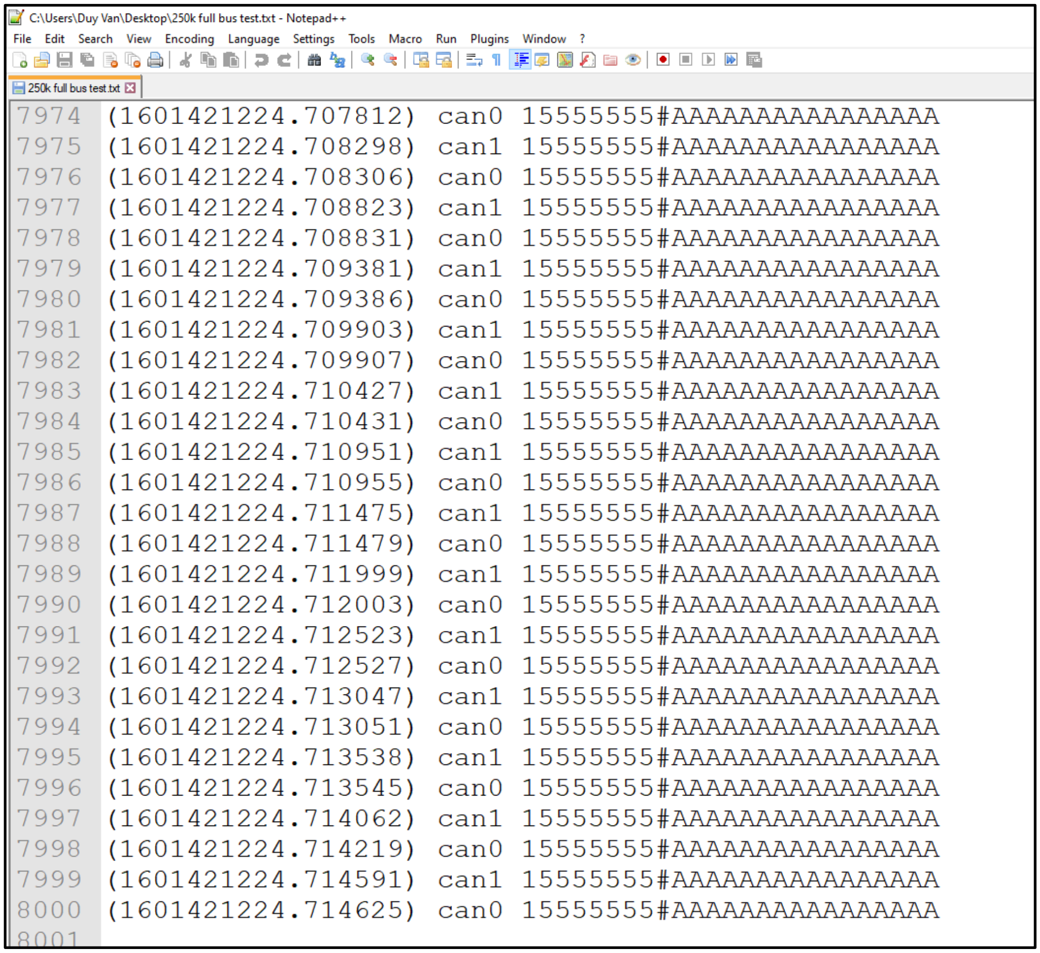


Figure 2-52. Bus load measurement by the TruckCape device for the full bus load test with fixed message interval

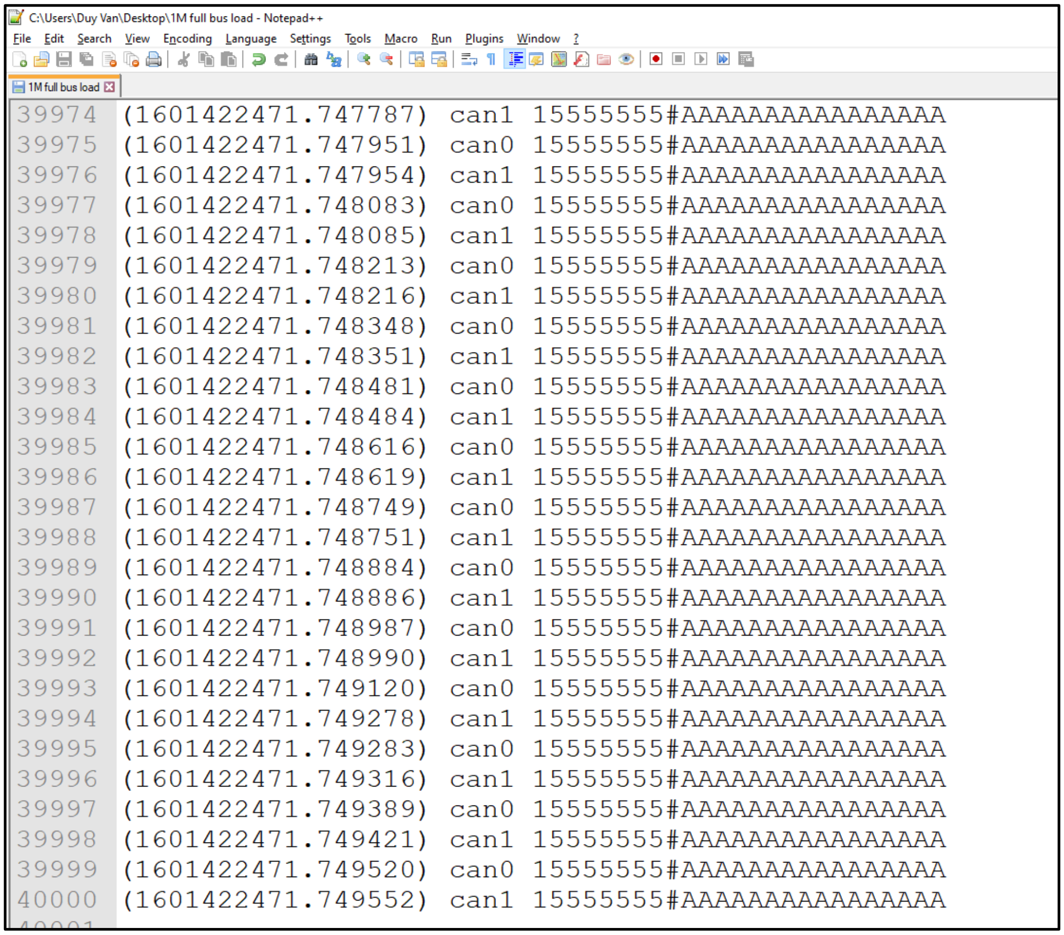



```
(1601421224.707812) can0 15555555#AAAAAAAAAAAAAAAA
(1601421224.708298) can1 15555555#AAAAAAAAAAAAAAAA
(1601421224.708306) can0 15555555#AAAAAAAAAAAAAAAA
(1601421224.708823) can1 15555555#AAAAAAAAAAAAAAAA
(1601421224.708831) can0 15555555#AAAAAAAAAAAAAAAA
(1601421224.709381) can1 15555555#AAAAAAAAAAAAAAAA
(1601421224.709386) can0 15555555#AAAAAAAAAAAAAAAA
(1601421224.709903) can1 15555555#AAAAAAAAAAAAAAAA
(1601421224.709907) can0 15555555#AAAAAAAAAAAAAAAA
(1601421224.710427) can1 15555555#AAAAAAAAAAAAAAAA
(1601421224.710431) can0 15555555#AAAAAAAAAAAAAAAA
(1601421224.710951) can1 15555555#AAAAAAAAAAAAAAAA
(1601421224.710955) can0 15555555#AAAAAAAAAAAAAAAA
(1601421224.711475) can1 15555555#AAAAAAAAAAAAAAAA
(1601421224.711479) can0 15555555#AAAAAAAAAAAAAAAA
(1601421224.711999) can1 15555555#AAAAAAAAAAAAAAAA
(1601421224.712003) can0 15555555#AAAAAAAAAAAAAAAA
(1601421224.712523) can1 15555555#AAAAAAAAAAAAAAAA
(1601421224.712527) can0 15555555#AAAAAAAAAAAAAAAA
(1601421224.713047) can1 15555555#AAAAAAAAAAAAAAAA
(1601421224.713051) can0 15555555#AAAAAAAAAAAAAAAA
(1601421224.713538) can1 15555555#AAAAAAAAAAAAAAAA
(1601421224.713545) can0 15555555#AAAAAAAAAAAAAAAA
(1601421224.714062) can1 15555555#AAAAAAAAAAAAAAAA
(1601421224.714219) can0 15555555#AAAAAAAAAAAAAAAA
(1601421224.714591) can1 15555555#AAAAAAAAAAAAAAAA
(1601421224.714625) can0 15555555#AAAAAAAAAAAAAAAA
```

Figure 2-53. Full bus load logging result at 250kbps for the full bus load test with fixed message interval

Figure 2-53 shows the decrypted log data from the CAN Logger 3 after the experiment. There were 8,000 messages captured, indicating that the CAN Logger 3 successfully captured all the messages sent on both channels at full speed.

Similarly, the same test was performed on a 1Mbps CAN bit rate. 20,000 messages were sent on both channels instead of 4,000. The message interval was selected to be 130.125 microseconds, which was 4 times faster than the one from the 250kbps bus. Figure 2-54 illustrates the captured messages from the experiment. There were 40,000 messages captured, indicating that the CAN Logger 3 successfully captured all the messages sent on both channels at full speed.


C:\Users\Duy Van\Desktop\1M full bus load - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

1M full bus load

```
(1601422471.747787) can1 15555555#AAAAAAAAAAAAAAAA
(1601422471.747951) can0 15555555#AAAAAAAAAAAAAAAA
(1601422471.747954) can1 15555555#AAAAAAAAAAAAAAAA
(1601422471.748083) can0 15555555#AAAAAAAAAAAAAAAA
(1601422471.748085) can1 15555555#AAAAAAAAAAAAAAAA
(1601422471.748213) can0 15555555#AAAAAAAAAAAAAAAA
(1601422471.748216) can1 15555555#AAAAAAAAAAAAAAAA
(1601422471.748348) can0 15555555#AAAAAAAAAAAAAAAA
(1601422471.748351) can1 15555555#AAAAAAAAAAAAAAAA
(1601422471.748481) can0 15555555#AAAAAAAAAAAAAAAA
(1601422471.748484) can1 15555555#AAAAAAAAAAAAAAAA
(1601422471.748616) can0 15555555#AAAAAAAAAAAAAAAA
(1601422471.748619) can1 15555555#AAAAAAAAAAAAAAAA
(1601422471.748749) can0 15555555#AAAAAAAAAAAAAAAA
(1601422471.748751) can1 15555555#AAAAAAAAAAAAAAAA
(1601422471.748884) can0 15555555#AAAAAAAAAAAAAAAA
(1601422471.748886) can1 15555555#AAAAAAAAAAAAAAAA
(1601422471.748987) can0 15555555#AAAAAAAAAAAAAAAA
(1601422471.748990) can1 15555555#AAAAAAAAAAAAAAAA
(1601422471.749120) can0 15555555#AAAAAAAAAAAAAAAA
(1601422471.749278) can1 15555555#AAAAAAAAAAAAAAAA
(1601422471.749283) can0 15555555#AAAAAAAAAAAAAAAA
(1601422471.749316) can1 15555555#AAAAAAAAAAAAAAAA
(1601422471.749389) can0 15555555#AAAAAAAAAAAAAAAA
(1601422471.749421) can1 15555555#AAAAAAAAAAAAAAAA
(1601422471.749520) can0 15555555#AAAAAAAAAAAAAAAA
(1601422471.749552) can1 15555555#AAAAAAAAAAAAAAAA
```

Figure 2-54. Full bus load logging result at 1Mbps for the full bus load test with fixed message interval

Based on the experiments, the CAN Logger 3 has met the desired performance requirement.

v. SHA-256 test

SHA-256 hashing is used for mapping data of arbitrary size to a unique fixed-size digest of 32 bytes for this algorithm. Any change to the data will result in a completely different hash digest. Thus, it is a good way to check if the data has been altered. The log file and some important information from the logging operation are SHA-256 hashed with the Teensy 3.6 Evaluation Board. The library used can be found at [72] and its function

was validated against NIST test vectors [73] and [74]. The functions from the test code [75] is are displayed below:

```
#include <sha256.h>

sha256Instance=new Sha256();
sha256Instance->update(text1, strlen((const char*)text1));
sha256Instance->final(hash);
```

After importing the SHA-256 library, a Sha256 instance was created. The update function took the data in to hash and updated the digest. The final function would complete and output the hash digest of all the combined input. Figure 2-55 shows the hash digest of NIST test vectors using the teensy library and their correct hashes. The values are identical, meaning that the SHA-256 library is valid.

Figure 2-55. Results from SHA-256 testing

vi. ATECC608A key configuration

The ATECC608A HSM main functions are to generate its own ECC key pairs and to load and securely store the server public key in its memory in order to perform ECC functions such as ECDH and ECDSA. The SparkFun_ATECCX08a_Arduino library that was used to interact with the HSM can be found at [76], which has been modified for the scope of the project. The functions for key generation and loading server public key from the test code [77] are displayed below:

```
#include <SparkFun_ATECCX08a_Arduino_Library.h>
#include <i2c_t3.h>

void setup() {
    Wire.begin(I2C_MASTER, 0x00, I2C_PINS_18_19, I2C_PULLUP_EXT, 100000);
    atecc.begin() == true;

    \\Configuration begin
    Serial.print("Write Config: \t");
    if (atecc.writeProvisionConfig() == true) Serial.println("Success!");
    else Serial.println("Failure.");

    Serial.print("Lock Config: \t");
    if (atecc.lockConfig() == true) Serial.println("Success!");
    else Serial.println("Failure.");

    Serial.print("Key Creation: \t");
    if (atecc.createNewKeyPair() == true) Serial.println("Success!");
    else Serial.println("Failure.");

    Serial.println("Configuration done.");
    Serial.println();

    \\Load public key and lock data
    Serial.println("Load server public key");
    Serial.print("Load Public Key: \t");
    if (atecc.loadPublicKey(server_public_key,false) == true){
    Serial.println("Success!");
    }
    else Serial.println("Failure.");

    Serial.print("Lock data and OTP zone: \t");
    if (atecc.lockDataAndOTP() == true) Serial.println("Success!");
    else Serial.println("Failure.");

  }
```

In the beginning of the script, the two libraries, SparkFun_ATECCX08a_Arduino and I2C, were imported and the ATECC608A was initialized. The configuration zone was written and locked with a specific structure based on the scope of the project. More detail on the configuration data can be found in the *writeProvisionConfig* function within the library file [78]. An ECC private key was able to be generated after that. An external public key was loaded and the data zone which contained those key slots were locked, meaning that any modification could no longer be made. Figure 2-56 shows the successful configuration from the test. The device was ready to perform other ECC algorithm tests.

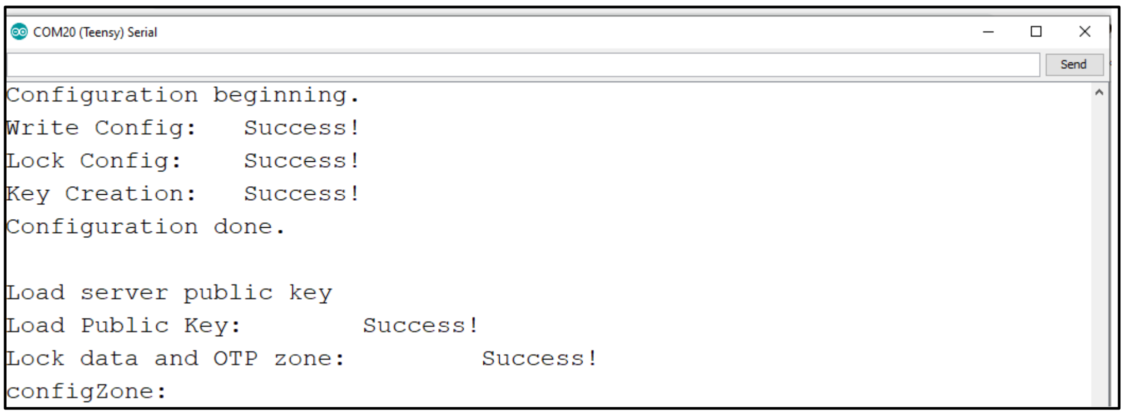


Figure 2-56. ATECC608A key configuration test results

vii. ECDH pre-master calculation test

ECDH pre-master calculation is a process of computing a 32-byte shared secret using the host's private key and the other party's public key. This test shows the concept of ECDH pre-master key exchange by showing the shared secret result from the server (Python) and the client (teensy), which simulates the asymmetric algorithm for secure communication between the CAN Logger 3 and the cloud services in the project. The Python and the teensy source code can be found on [79] and [80], respectively. The first step was to generate an ECC key pair for the teensy client, using the script from the key configuration script [77] without loading the server public key and locking the data because the server has not generated its key pair yet. Figure 2-57 displays the device public key.

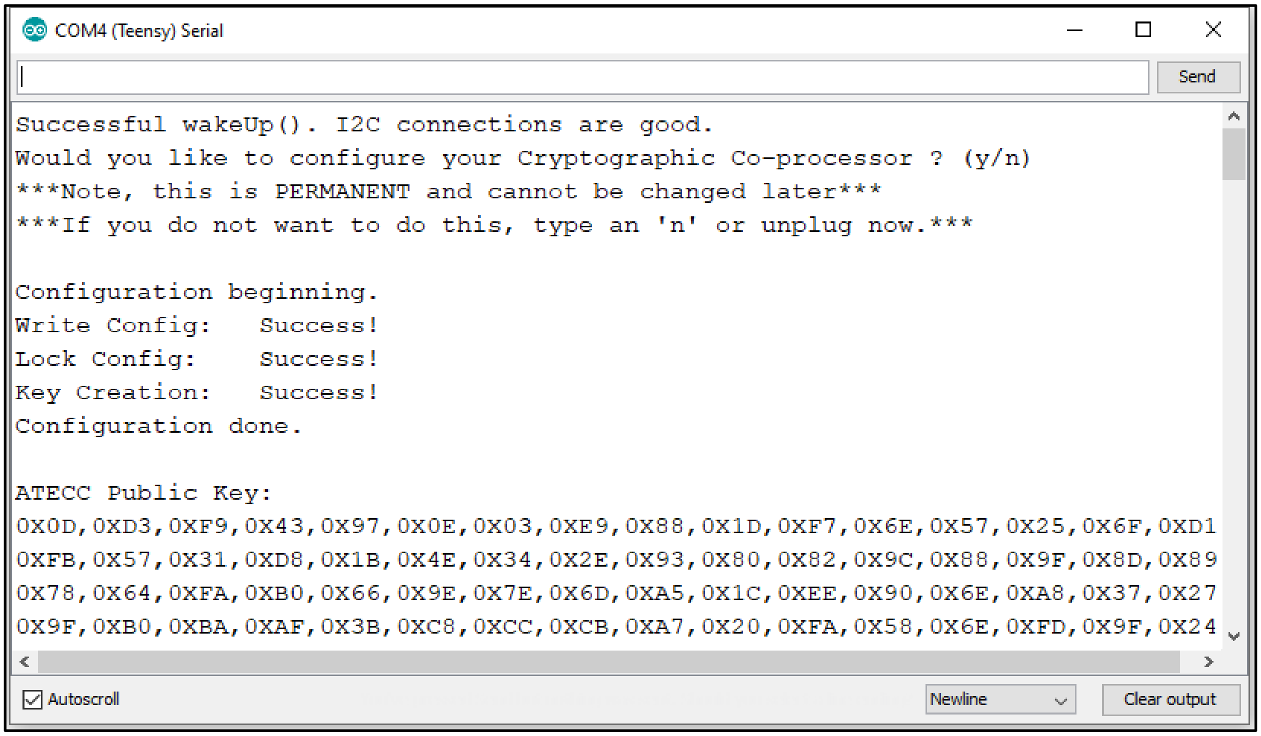


Figure 2-57. CAN Logger 3 device public key for ECDH testing

The client public key then was manually loaded into the server Python script, where the key was serialized into the right format before being loaded into the function. The server then generated an ECC key pair for itself and use its private key and the input client public key to calculate a shared secret. The code for the Python ECDH is shown below.

```
#generate server ECC key pair
server_private_key = ec.generate_private_key(ec.SECP256R1(),
default_backend())
server_public_key = (server_private_key.public_key())

PEM_public_key_first = '-----BEGIN PUBLIC KEY-----
\nMFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAE'
PEM_public_key_last = '\n-----END PUBLIC KEY-----\n'

#Input Teensy public key manually here:
Teeny_public_key_hex =
[0X0D,0XD3,0XF9,0X43,0X97,0X0E,0X03,0XE9,0X88,0X1D,0XF7,0X6E,0X57,0X25,0X6F
,0XD1,0XFB,0X57,0X31,0XD8,0X1B,0X4E,0X34,0X2E,0X93,0X80,0X82,0X9C,0X88,0X9F
,0X8D,0X89,0X78,0X64,0XFA,0XB0,0X66,0X9E,0X7E,0X6D,0XA5,0X1C,0XEE,0X90,0X6E
,0XA8,0X37,0X27,0X9F,0XB0,0XBA,0XAF,0X3B,0XC8,0XCC,0XCB,0XA7,0X20,0XFA,0X58
,0X6E,0XFD,0X9F,0X24]

#Finalize the teensy public key in serilized PEM format
public_key_teensy_string = PEM_public_key_first +
Teensy_PEM_public_key[:28]+'\n'+ Teensy_PEM_public_key[28:] +
PEM_public_key_last
serialized_public_teensy = bytes(public_key_teensy_string,'ascii')

#Load teensy public key
teensy_public_key =
serialization.load_pem_public_key(serialized_public_teensy,backend=default_
backend())

#Derive shared secret
shared_secret = server_private_key.exchange(ec.ECDH(),teensy_public_key)
print("Shared secret:",shared_secret.hex())
```

Figure 2-58 shows the server public key and the ECDH shared secret generated from the Python server script.

```
Server public Key is: 0X5d,0X9c,0Xc4,0Xcc,0Xee,0X18,0Xf0,0X38,0Xaf,0X35,0X28,0Xfa,0X
bb,0Xe8,0Xf2,0Xb1,0Xa7,0Xee,0Xe4,0Xd1,0X43,0X1c,0Xd4,0X4c,0Xa0,0X77,0Xa5,0X16,0Xf7,0
X8e,0X35,0X90,0X6e,0Xb4,0Xc6,0X9e,0X3c,0X06,0Xb3,0Xbd,0Xc6,0Xa4,0X6a,0X2d,0Xea,0X32,
0Xa2,0X3b,0X75,0Xb1,0Xe7,
0X6a,0X75,0Xc7,0X9f,0X2b,0X80,0X19,0Xb7,0Xb9,0Xfe,0X38,0Xbd,0Xba

Teensy Public Key: 0dd3f943970e03e9881df76e57256fd1fb5731d81b4e342e9380829c889f8d897
864fab0669e7e6da51cee906ea837279fb0baaf3bc8cccba720fa586efd9f24
Length: 64

Shared secret: 6c6ca37c963e5e2ad77bf93c37c96728363dd1b4baa314012ad2718888798627
[Finished in 0.4s]
```

Figure 2-58. Server public key and ECDH shared secret

After that, the server public key was manually loaded into the teensy script, where the teensy client used its private key and the server public key to calculate a shared secret. The code for the ATECC608A ECDH is shown below.

```
Void setup(){

      uint8_t server_public_key[64] =
{0X89,0X5e,0Xdf,0Xf7,0Xc5,0Xc2,0X96,0Xeb,0X97,0Xa1,0X71,0X98,0Xc2,0X53,0Xc1
,0X05,0Xf4,0Xe3,0Xda,0Xf6,0X29,0X64,0X71,0Xb2,0X15,0Xac,0X52,0X0e,0X0a,0X11
,0Xce,0X54,0Xa3,0Xec,0X91,0X0b,0Xa4,0Xe8,0X48,0X29,0Xec,0X69,0Xbe,0Xca,0Xc9
,0Xcf,0Xc8,0Xc4,0X32,0X8c,0Xec,0X5e,0X93,0X03,0X93,0Xac,0X10,0X5b,0X66,0X30
,0X49,0Xeb,0Xe4,0X87};

      Serial.print("Load Server Public Key: \t");
     if (atecc.loadPublicKey(server_public_key) == false){
     Serial.println("Failure.");
      }
    Serial.print("Lock Data-OTP: \t");
      if (atecc.lockDataAndOTP() == true) { Serial.println("Success!");
      }
    else Serial.println("Failure.");

      //Read stored public key for ECDH
       atecc.readPublicKey(true);
      //Let's calculate the shared secret!
      atecc.ECDH(atecc.storedPublicKey, ECDH_OUTPUT_IN_CLEAR,0x0000);
}
```

Figure 2-59 shows that the shared secret calculated from the client, which is the same one from the Python script. This demonstrates the Diffie-Hellman key exchange concept where public keys are exchanged, and the client and the server can use the other's public key to generate the same shared secret for further use in secure communication.

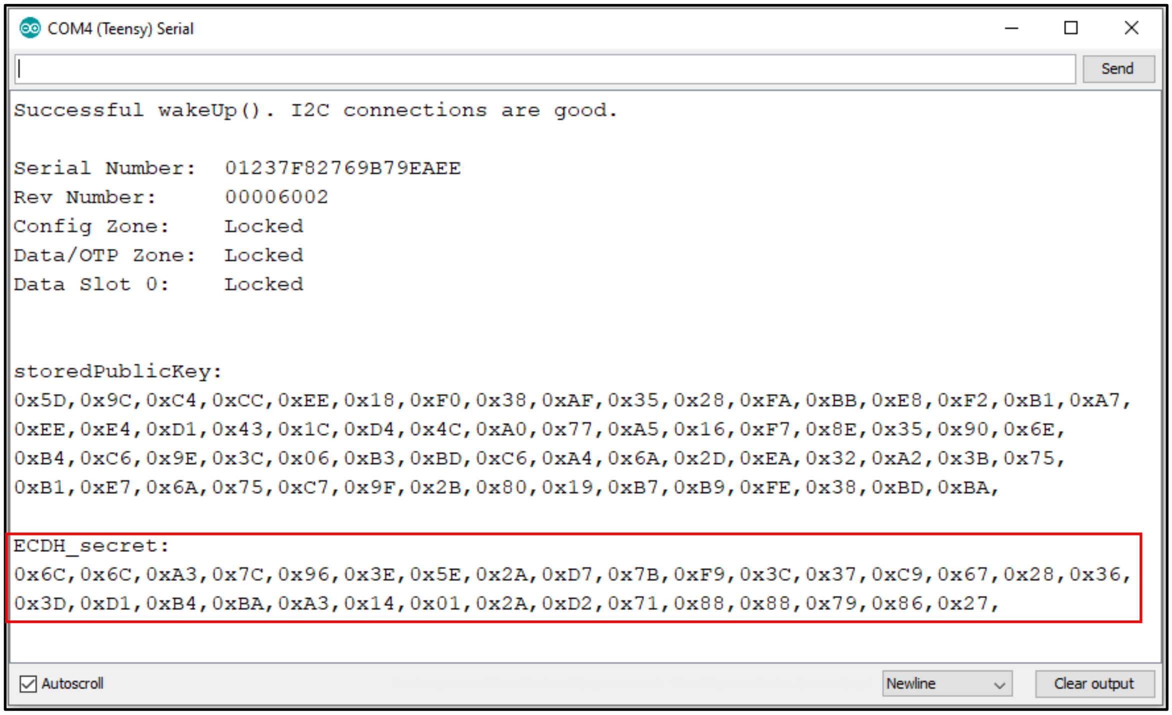


Figure 2-59. ECDH shared secret calculated from the teensy client

viii. ATECC608A AES-128

The ATECC608A also supports AES-128 ECB mode, which for this project, uses the ECDH shared secret as its key for encryption, thus provides data confidentiality. The AES-128 was tested to ensure its functionality. The test code is also part of the ECDH script that was mentioned above, which is displayed as shown:

```
//16-byte message
uint8_t message[16] = {
  0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0A, 0x0B,
0x0C, 0x0D, 0x0E, 0x0F
};

Void setup(){
//Read stored public key for ECDH
  atecc.readPublicKey(true);
  //Calculate the share secret and load in tempkey!
  atecc.ECDH(atecc.storedPublicKey, ECDH_OUTPUT_IN_TEMPKEY,0x0000);
//Encrypt data
  atecc.AES_ECB_encrypt(message);
}
```

After the teensy calculated the ECDH shared secret in the previous section, instead of outputting the value in clear text, the shared secret was loaded into the *TEMPKEY* where the ATECC608A would refer to that slot as the AES key. A random 16-byte message buffer was encrypted using the first 16 bytes of the ECDH shared secret, and the ciphertext was compared with the correct value from an online AES tool [81]. Figure 2-60 shows the ciphertext result from the ATECC608A AES-128 encryption, which is identical to the one calculated using the online tool shown in Figure 2-61. This indicates the function passed its test.

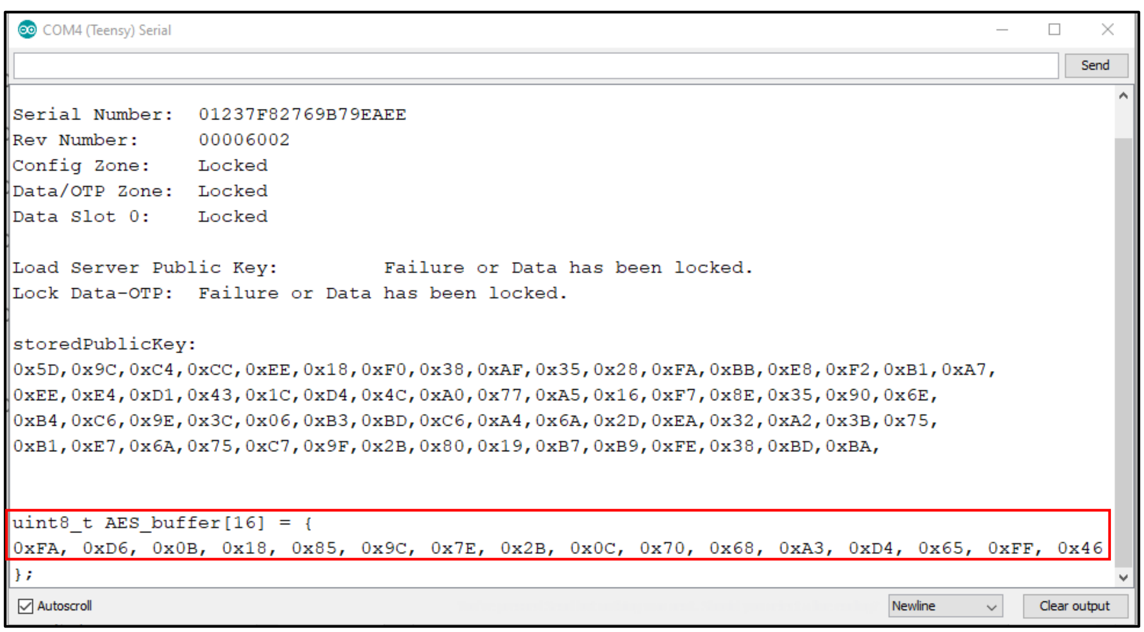


Figure 2-60. AES ciphertext calculated from ATECC608A and online tool

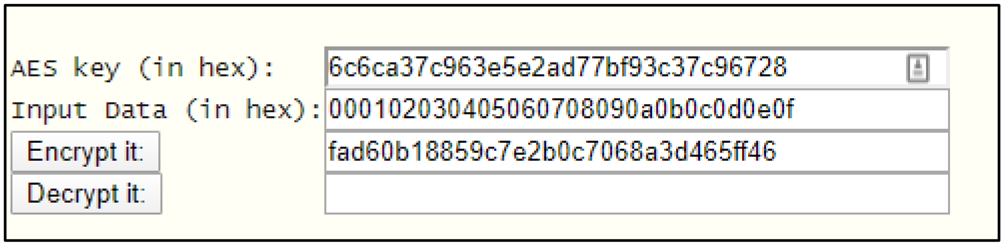


Figure 2-61. AES ciphertext calculated from an online tool

ix. ECDSA sign and verify test

Signing and signature verifying provide a recipient confidence in the received data that it was created by a known sender and the message has not been altered in transit. As a result, this application is used to authenticate and protect the integrity of critical information being sent from the CAN Logger 3 to the cloud services. This test demonstrates ATECC608A ECDSA signing and verifying functionality, and its compatibility with the Python server scripts because they are two platforms with different languages. In the first test, the teensy signed a random message and the Python server verified it and its signature. The test code for the Python server and the teensy client can be found in [82] and [83], respectively. The code for the teensy is shown below:

```
Teensy
//Message to be signed
uint8_t message[32] = {
  0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0A, 0x0B,
0x0C, 0x0D, 0x0E, 0x0F, 0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
0x18, 0x19, 0x1A, 0x1B, 0x1C, 0x1D, 0x1E, 0x1F
};
Void setup(){
   //hash message with SHA256
   sha256Instance = new Sha256();
   sha256Instance->update(message,sizeof(message));
   sha256Instance->final(hash_message);
   //Sign message with private key stored on slot 0
   atecc.createSignature(hash_message,0,true);
   printSignature();
}
```

```
Python
public_key_hex = [0x3B, 0x3C, 0x19, 0x35, 0x90, 0xDA, 0xE4, 0x6F, 0x64,
0x8C, 0x7E, 0x5E, 0x52, 0x82, 0xA0, 0x98,
0xA2, 0x5D, 0x7C, 0xC2, 0xDD, 0x3D, 0xA4, 0x8E, 0x18, 0xCF, 0x5E, 0xA1,
0x39, 0x73, 0x67, 0x6E,
0xDB, 0xD6, 0x25, 0xD2, 0xEC, 0x0E, 0xF7, 0x83, 0x4C, 0xC7, 0xD7, 0x5D,
0x5E, 0x02, 0x1D, 0x41,
0xCB, 0x25, 0xFD, 0x1A, 0x1E, 0xEA, 0x32, 0x6B, 0x61, 0xC6, 0xF4, 0xC1,
0xBC, 0xF2, 0x21, 0x01]

data_hex = [0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F,
0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, 0x18, 0x19, 0x1A, 0x1B,
0x1C, 0x1D, 0x1E, 0x1F]

signature_hex = [0x86, 0x2B, 0x67, 0x14, 0x1C, 0x06, 0xE7, 0x08, 0xF5,
0xFA, 0x1D, 0x17, 0x8E, 0x81, 0xF9, 0x79,
0x17, 0xBC, 0xBA, 0x85, 0xB4, 0x85, 0xAA, 0xBE, 0x1D, 0x1C, 0x2B, 0xCB,
0xE9, 0x43, 0x96, 0x3F,
0xB8, 0xFB, 0x75, 0x25, 0x3B, 0xF0, 0x0E, 0x0A, 0x76, 0x19, 0x58, 0x0F,
0xFA, 0x96, 0xB0, 0xCB,
0x68, 0xED, 0x44, 0x81, 0x9F, 0x7B, 0x91, 0x6F, 0x68, 0x31, 0x4D, 0xC2,
0x83, 0xEE, 0xF6, 0xE3
]

#Load Public Key
PEM_public_key = base64.b64encode(bytes(public_key_hex)).decode('ascii')
public_key_string = PEM_public_key_first + PEM_public_key[:28]+'\n'+
PEM_public_key[28:] + PEM_public_key_last
serialized_public_teensy = bytes(public_key_string,'ascii')
public_key =
serialization.load_pem_public_key(serialized_public_teensy,backend=default_
backend())

#Verify the signature
if public_key.verify(signature,data,ec.ECDSA(hashes.SHA256()))== None:
   print("Verify Signature Successfully!")
```

The teensy would need to go through key configuration step to generate an ECC private key along with its corresponding public key to prepare for ECDSA algorithm. After that, the message was hashed with SHA-256 and the digest was loaded into the ATECC608A to be signed with the device private key. A signature was generated and along with the device public key, were input in the Python scripts where the key was serialized to verify the message. Figure 2-62 shows that the Python script has successfully verified the message.

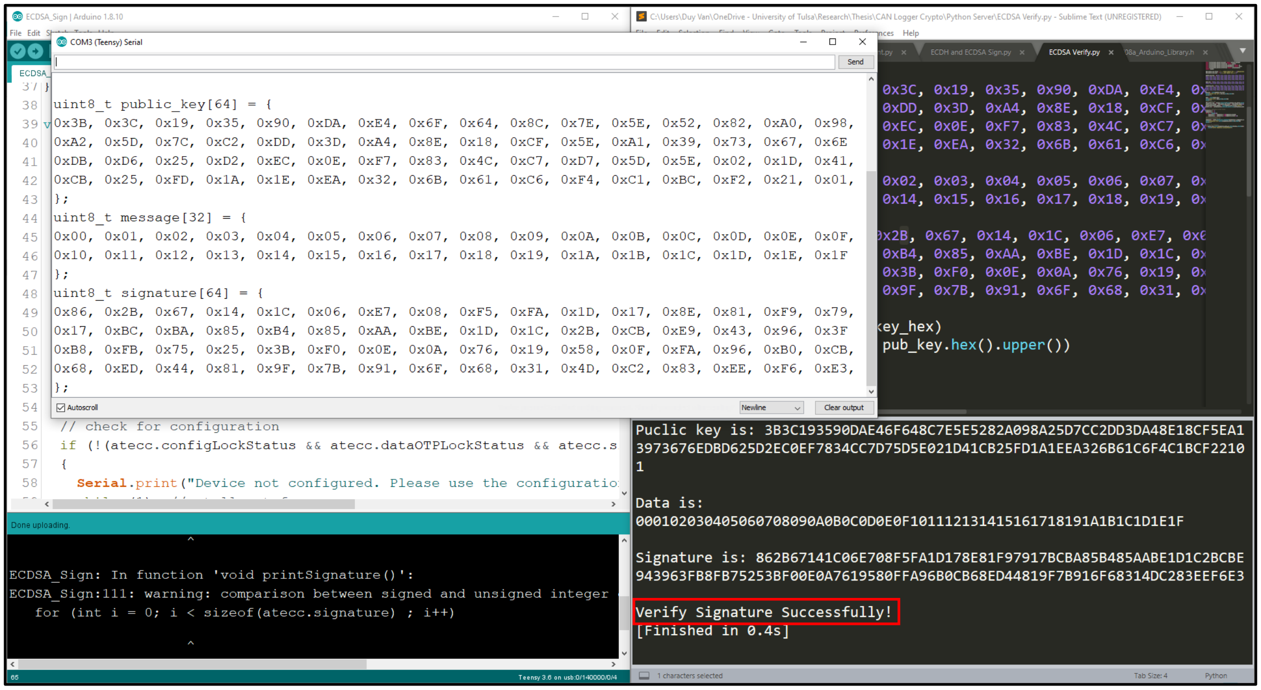


Figure 2-62. Results from ATECC608A signing and Python sever verifying

In the second test, the Python server signed a message and the device verified it and its signature. The test code for the Python server and the teensy client can be found in [79] and [84], respectively. The Python and teensy code is shown below:

Python

```
#generate server ECC key pair
server_private_key = ec.generate_private_key(ec.SECP256R1(),
default_backend())
server_public_key = (server_private_key.public_key())

#data to be signed
data_hex = [0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09,
0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F,
0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, 0x18, 0x19, 0x1A, 0x1B,
0x1C, 0x1D, 0x1E, 0x1F]
data = bytes(data_hex)

#Sign the message
signature1 = server_private_key.sign(data,ec.ECDSA(hashes.SHA256()))
```

```
Teensy
uint8_t publicKeyExternal[64] = {
0xD6,0x78,0xAB,0x2E,0x76,0x16,0xE0,0xF6,0x10,0x47,0x0F,0xB9,0x1C,0x4A,0x1A,
0x2D,0xAE,0xB8,0x1C,0x48,0xA2,0x8A,0xAE,0xB8,0x4E,0xC8,0x7B,0x88,0xEB,0xE8,
0x50,0xDE,0xCC,0xA2,
0x9B,0x6F,0x53,0x4A,0x21,0x58,0x06,0xF9,0xB8,0x92,0x43,0x01,0x5A,0x5C,0x67,
0x18,0xE6,0x51,0x61,0xA0,0xDA,0xBB,0x56,0xCE,0x56,0xFC,0x1B,0xD5,0xCB,0x49
};

uint8_t message[32] = {
0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0A, 0x0B,
0x0C, 0x0D, 0x0E, 0x0F,
0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, 0x18, 0x19, 0x1A, 0x1B,
0x1C, 0x1D, 0x1E, 0x1F
};

uint8_t signature[64] = {
0x72,0x1B,0xAA,0xC8,0x15,0x7C,0x68,0x14,0x01,0x3C,0x10,0x84,0x68,
0xF9,0xBA,0xDB,0x9C,0x26,0xAC,0xC0,0x84,0xE7,0xD3,0xEC,0x9B,0x66,
0x32,0x19,0x2C,0xB6,0xEC,0xA6,0xEA,0x24,0xF8,0xC8,0x04,0xD6,0x57,
0x6D,0x3E,0xDF,0xE2,0xF3,0x75,0xE0,0x04,0xD0,0x95,0x72,0x92,0xAA,
0xD0,0x65,0x43,0x1B,0x83,0xDD,0x31,0x35,0xAA,0xAF,0x84,0x00
};

Void setup(){
  //SHA256 hash mesasge
  sha256Instance = new Sha256();
  sha256Instance->update(message,sizeof(message));
  sha256Instance->final(hash_message);

  // Let's verirfy!
  if (atecc.verifySignature(hash_message, signature, publicKeyExternal)){
     Serial.println("Success! Signature Verified.");
  }
  else Serial.println("Verification failure.");
}
```

Initially, the server script generated an ECC key pair using the integrated library function. The same message in the first test was entered as a parameter in the signing function where the server private key was used. A signature was generated and it, along with the server public key, were input in the device teensy script where the ATECC608A loaded the data in its memory. The verifying function was performed, and the result shows a successful execution, as seen in Figure 2-63.

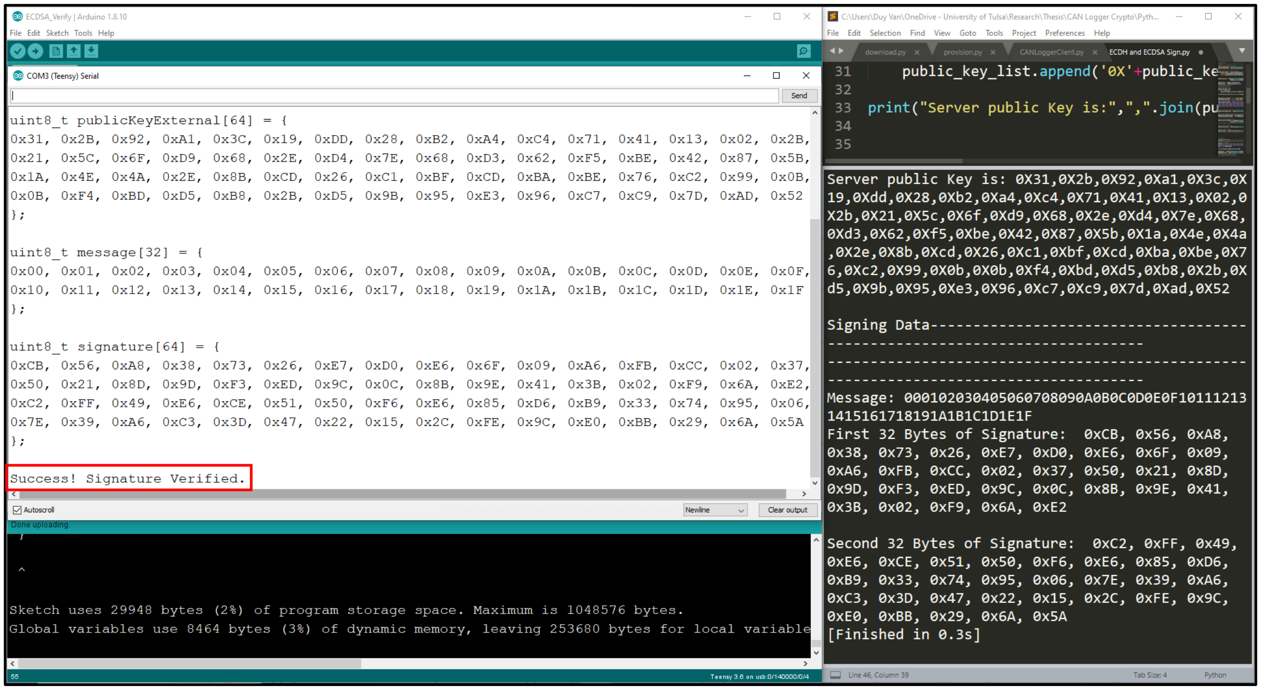


Figure 2-63. Results from Python sever signing and ATECC608A verifying the message.

x. CAN2 and multiplexing test

Because the MCP2517FD is a new CAN controller version, a different library was used to interact with the chip. The Arduino source code of the library can be found in [85], which was developed by Pierre Molinaro. However, the library needed to be explored and modified to send and receive CAN messages on the bus. Because CAN2 is not as popular in vehicles as CAN0 and CAN1, this application will be saved for future work. The purpose of this test was to ensure the MCP2517FD hardware has been wired correctly and can function with the right software. Even though CAN2 has not been able to work normally, the internal loopback mode can be activated for self-testing using the script [86] from the library. Moreover, the multiplexing feature is also tested in this script because it is required to enable CAN2 from default J1708. The test code is shown below:

```
#include <ACAN2517.h>

static const byte MCP2517_SCK = 13 ; // SCK input of MCP2517
static const byte MCP2517_SDI =  11 ; // SDI input of MCP2517
static const byte MCP2517_SDO =  12 ; // SDO output of MCP2517

static const byte MCP2517_CS  = 15 ; // CS input of MCP2517
static const byte MCP2517_INT = 38 ; // INT output of MCP2517

ACAN2517 can (MCP2517_CS, SPI, MCP2517_INT) ;

#define CAN_switch 2 //multiplexing pin

void setup () {
  pinMode(CAN_switch,OUTPUT);
  digitalWrite(CAN_switch, HIGH);

  SPI.setMOSI (MCP2517_SDI) ;
  SPI.setMISO (MCP2517_SDO) ;
  SPI.setSCK (MCP2517_SCK) ;
  SPI.begin () ;

  ACAN2517Settings settings (ACAN2517Settings::OSC_20MHz, 250 * 1000) ;
  // Select loopback mode
  settings.mRequestedMode = ACAN2517Settings::InternalLoopBack ;
}

void loop(){
  const bool ok = can.tryToSend (message) ;
  if (ok) gSentFrameCount += 1 ;
  if (can.receive (frame)) gReceivedCount += 1 ;
}
```

The library was imported in the beginning of the script. SPI pinouts such as SDI, SDO, CS, and INT were defined and set according the schematic design of the CAN Logger 3. Multiplexing CAN_switch pin was also defined and pulled high to enable CAN2. The library setting was chosen with 20MHz SPI speed, 250kbps bitrate, and *InternalLoopBack* mode. In the looping function, the CAN controller sent out a random message and increased the send counter by one if it was sent successfully. At the same time, if it received a message, the receive counter also increased by one. Figure 2-64 shows the results from the internal loopback test script. The MCP2517FD was successfully initialized and able to send and receive messages internally. This indicates that the hardware passed the test. However, full functionality will be investigated in the future.

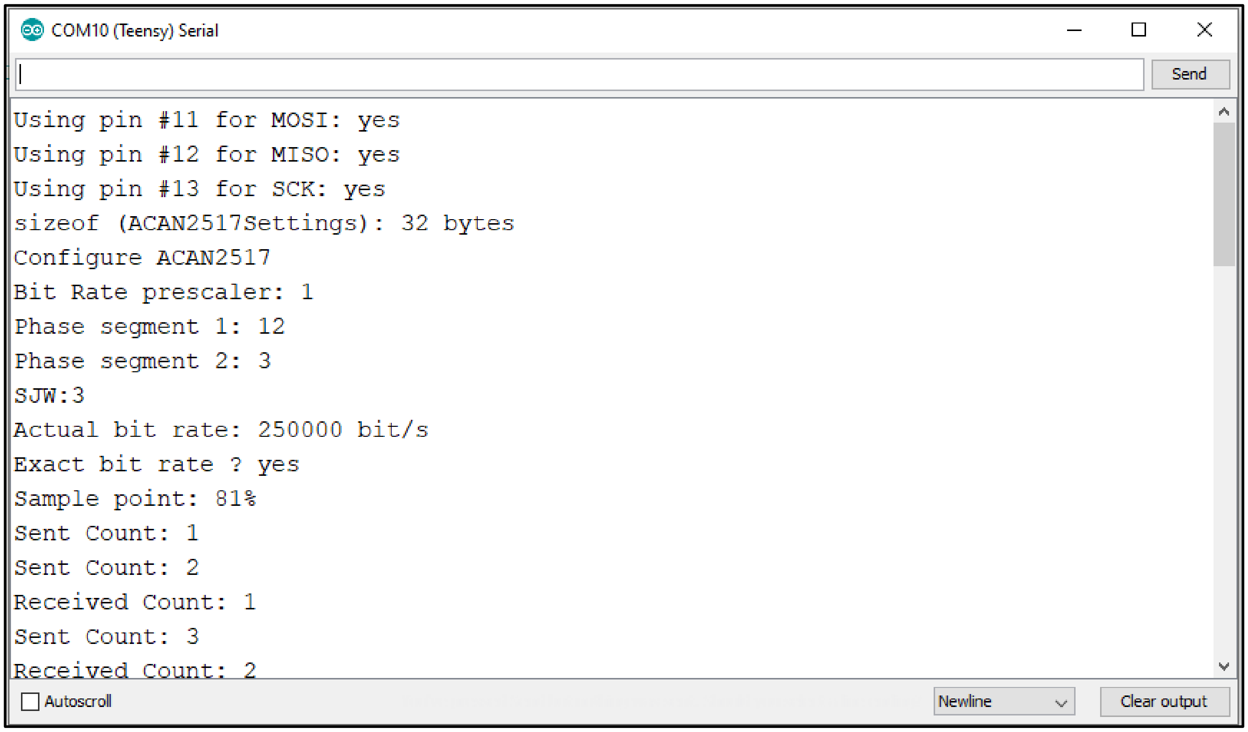


Figure 2-64. CAN2 internal loopback test result

xi. Single-Wire CAN test

The Single-Wire CAN is not in requirement and therefore, has not been tested.

xii. LIN test

The LIN feature is not in requirement and therefore, has not been tested.

xiii. J1708 test

J1708 feature was tested with a setup of two CAN loggers 3 connected to each other where one sent and the other received. The test script can be found here [87] and the code is shown below:

```
#define CAN_switch 2

void setup () {
  //Set CAN_switch low for J1708
  pinMode(CAN_switch,OUTPUT);
  digitalWrite(CAN_switch, LOW);
  Serial2.begin(9600); //Initialize Serial2
}
void loop() {
   int nBytes = Serial2.available(); //Read available messages
   if(nBytes > 0)
   {
       int nCount = Serial2.readBytes(sBuffer, nBytes);
       for(int nIndex = 0; nIndex < nCount; nIndex++)
       {
           Serial.print(sBuffer[nIndex], HEX); //Print messages in hex
       }
   }
    Serial2.write(message,4);//Write message
}
```

CAN_switch pin for multiplexing could optionally be set low, or left at its default value because J1708 was configured as the default network. In this script, both CAN loggers 3 sent out messages with data increasing by one through Serial2 for J1708 communication. At the same time, they also actively listened to available messages. Figure 2-65 shows the results from one of the devices.

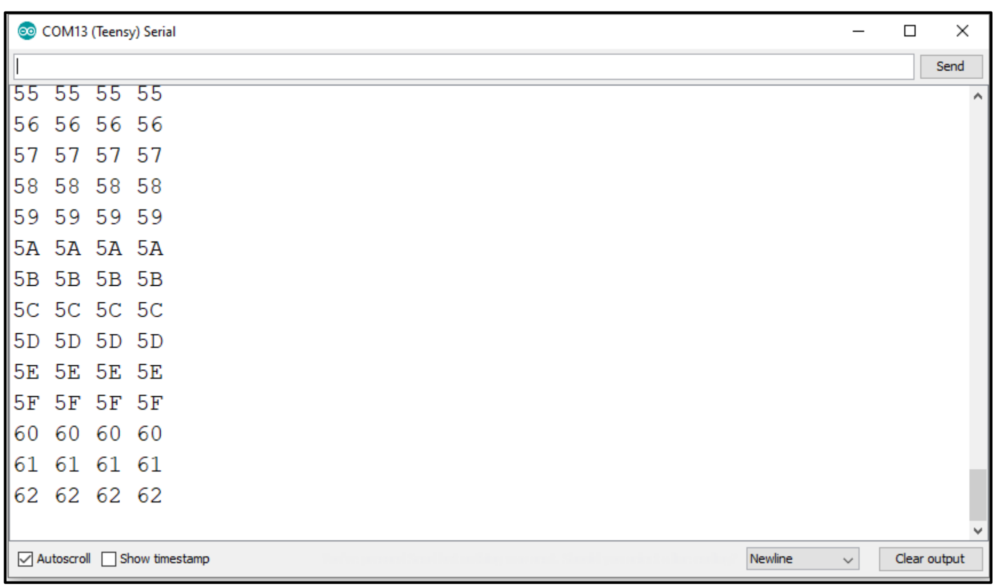


Figure 2-65. J1708 test results from one of the CAN loggers 3

The fact that there were messages shown on the serial monitor means that the CAN Logger 3 was able to send and receive J1708 messages successfully. The test shows that the J1708 circuit was designed correctly. However, full functionality will be investigated in the future.

xiv. Voltage monitoring test

The CAN Logger 3 voltage monitoring features, which include external analog voltage measurement of the raw 12V pin and of pin A7, and an optically isolated input of the raw 12V and of pin A6. These pins are located on the D-Sub 15; however, only the raw 12V line is connected to the vehicle through the Deutsch 9-pin connector. Pin A6 and A7 are saved for future use where extra interrupts are needed. They were all tested using this script [88], as shown in the code below:

```
//Define source pins
#define RAW_sense 21
#define A6_sense 20
int RAW_measure = A22;
int A7_measure = A21;

void setup() {
  // put your setup code here, to run once:
   pinMode(RAW_sense, INPUT_PULLUP);
   pinMode(A6_sense, INPUT_PULLUP);
}

void loop() {
  // put your main code here, to run repeatedly:
  Serial.print("RAW sense:");
  Serial.println(digitalRead(RAW_sense));
  Serial.print("RAW measure:");
  Serial.println(analogRead(RAW_measure));

  Serial.print("A6 sense:");
  Serial.println(digitalRead(A6_sense));
  Serial.print("A7 measure:");
  Serial.println(analogRead(A7_measure));
}
```

The pins were defined according to how they were assigned in the design. The optically isolated input pins were pulled high. In the loop function, the device printed out the converted analog values for the external voltage measurement and a digital value of 0 (12V) or 1 (0V) for the optically isolated input. Based on the experiment, the analog values for the raw 12V range from 0 to approximately 192 on an 0-12V scale, and 0 to approximately 211 on an 0-12V scale for pin A7. The digital reading for the raw 12V and pin A6 is 0 for 12V and 1 for 0V. Figure 2-66 reflects successful readings.

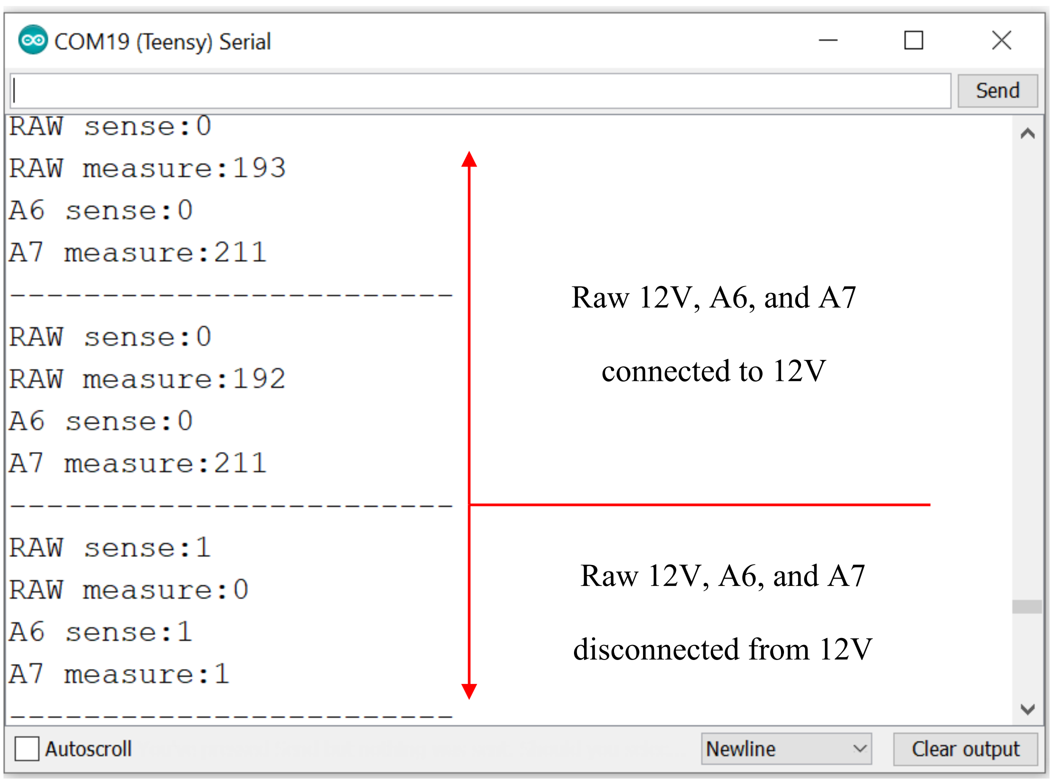


Figure 2-66. CAN Logger 3 voltage monitoring test results

xv. Power interruption test

In the event of a power interruption, the device needs to close the current session, and create and sign the associated metadata file to finish the logging process. The device's closing time was measured by modifying part of the main firmware [89] as shown below:

```
elapsedMicros micro_timer;

void close_binFile(){
   micro_timer = 0;
   binFile.close();
   Serial.println();
   Serial.print("Time to close bin file (us):");
   Serial.println(micro_timer);
   micro_timer = 0;

   write_final_meta_data();

   Serial.print("Total closing time to create metadata textfile (us):");
   Serial.println(micro_timer);
}
```

A timer variable in microseconds was set to 0 initially and printed after executing the *binFile.close()* and *write_final_meta_data()* functions to show long each of them took. The results are illustrated in Figure 2-67.

Figure 2-67. The time to close bin file and create metadata file

On average, it took approximately 5-9 milliseconds to close the binary log file and 130-140 milliseconds to create the metadata text file with its hash and signature and the total time required was about 150 milliseconds. A power loss event can interrupt this process and the data will not be completely recorded. Therefore, as mentioned before, large capacitors were added in the design to help power the teensy for a small amount of time to execute the closing functions during a power loss. However, the time for the metadata was still significantly large and the CAN Logger 3 might not be able to run for 150 milliseconds during a power loss event. As a result, the device's firmware was modified such that it would save important information of the current logging session into the processor's EEPROM memory before creating and hashing the metadata file. If the device failed to finish creating the metadata file due to power loss, it would redo the process during the next bootup. The time it took to write into EEPROM was measured. The code is shown:

```
void close_binFile(){
   micro_timer=0;

   //Write current file name to EEPROM
   EEPROM.put(EEPROM_FILE_ID_ADDR,current_file);

   //Write important metada of current file for backup
   EEPROM.put(EEPROM_filesize_hash_ADDR, filesize_hash_contents);

   Serial.print("Total closing time with backup data already stored in
EEPROM (us):");
   Serial.println(micro_timer);   write_final_meta_data();

   write_final_meta_data();
}
```

Figure 2-68. The time to close bin file and store backup data in EEPROM

Figure 2-68 shows the results with the added function. The device took similar time to close the binary log file and approximately 3-3.3 milliseconds to store the current session backup data to the EEPROM. Thus, with only 8-12 milliseconds , the device could have record everything safely. This was a huge improvement from the 150 milliseconds measurement.

There are two power interrupting events that can possibly occur during normal operation. The first one is when the CAN Logger 3 gets unplugged from the diagnostic port, which occurs regularly during operation. The second is during some unexpected and rare events (crashes, struck by lightning, etc.) , the vehicle losses power from the battery or the alternator and thus, the device losses power from the diagnostic port. The engine key on/off switch does not associate with these two events because the 12V on the diagnostic port typically remains on all the time. The main difference between the two power loss events is that the supply voltage drops from 12V to 0V immediately when the device is disconnected; during a loss of vehicle power the supply will decay over time due to existing capacitance in the vehicle's power network. Because the CAN Logger 3 is not isolated from the network, the raw 12V will exhibit the same decay as the vehicle power. Initially, the only voltage monitoring for the CAN Logger 3 was a binary output from the optically isolated input, which read false (0) if the supply voltage was non-zero, and true (1) if the supply voltage was zero. This posed a problem because the Teensy had insufficient warning that a loss of power had occurred, thereby jeopardizing the ability of the device to save and close the log file in order to prevent data loss. To mitigate this risk, an additional analog voltage monitoring system was added, as mentioned above, to the design that will trigger an interrupt when the supply voltage drops below 9V. The CAN logger was tested against those two events to ensure that no data was lost.

An oscilloscope was used to monitor the teensy Vin power pin along with the raw 12V (diagnostic port power from vehicle) input from the network and the safe 12V (diagnostic port power from vehicle after device's power protection). At the same time, the firmware was also modified to pull an LED high immediately after the device closed the binary file and wrote data to the EEPROM memory. The voltage of the LED was also monitored by the oscilloscope to determine whether the file closing occurred and how long it took. The results for the two power loss cases were graphed and analyzed as shown in Figure 2-69 and Figure 2-70.

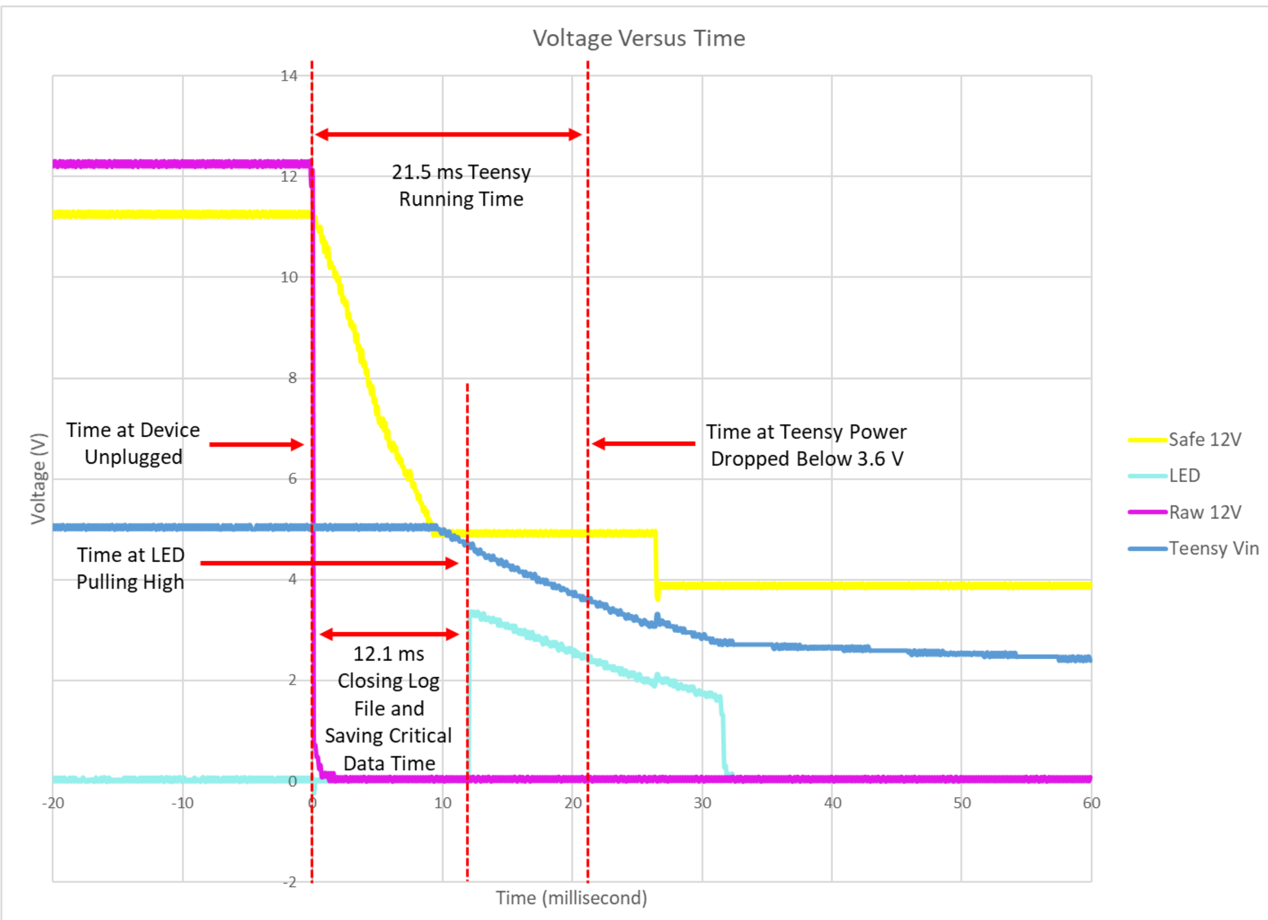


Figure 2-69. Analyzing the voltage dropped from unplugging the device

Figure 2-69 shows that when the device was unplugged, the raw 12V input immediately dropped below 1V, which was the first indication that the file closing function was triggered. The capacitors in the design still supplied power to maintain the teensy at a normal voltage level of above 3.6V for about 21.5 milliseconds while the device safe 12V gradually decayed. The safe 12V and the Teensy Vin eventually leveled out to 0V. The graph has been rescaled as shown for better visualization during this critical event. The LED turned on at 12.1 milliseconds, which indicated that the CAN Logger 3 had successfully closed the binary file and saved critical metadata information in the EEPROM. Therefore, no log data was lost when the power connection was interrupted.

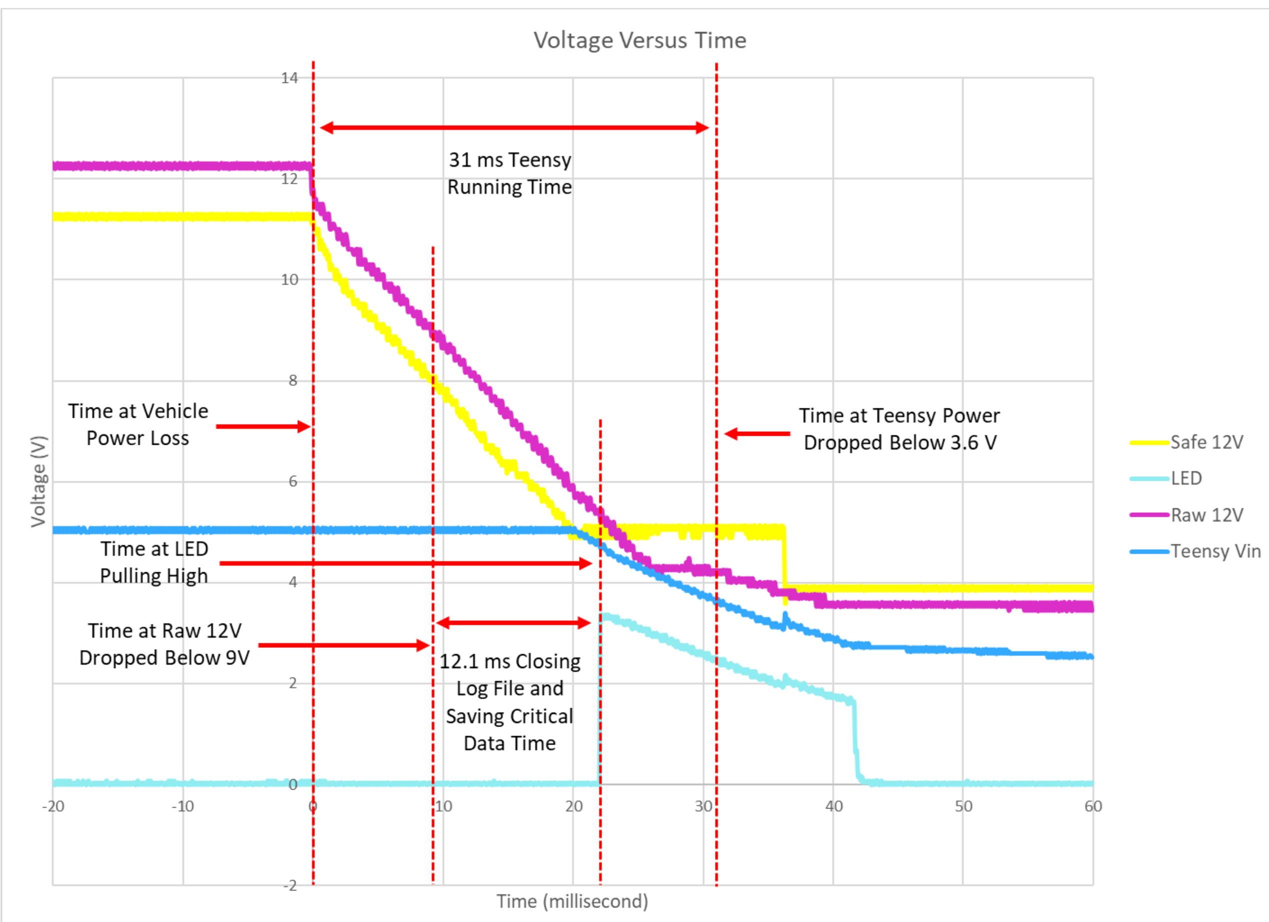


Figure 2-70. Analyzing the voltage dropped from vehicle power loss

Figure 2-70 shows that when the vehicle lost its power, the raw 12V input gradually decayed to less than 4V over the course of approximately 60 milliseconds . When the raw 12V dropped below 9V, the interrupt was triggered. The process to safely close the log file and save all data to the EPPROM took 12.1 milliseconds, which approximately the same as the previous test. The Teensy running time was extended from 21.5 milliseconds to 31 milliseconds due to the residual capacitance from the raw 12V. Again, the graph has been rescaled and the safe 12V, raw 12V, and the Teensy Vin eventually leveled out to 0V. The CAN Logger 3 has successfully and safely recorded the log file in this power loss event.

Both cases showed that the CAN Logger 3 passed the power interruption test. However, an important point from the graphs was that the teensy could not last more than 150 milliseconds to create and hash the metadata file. As a result, the added EEPROM function has solved the problem and the chosen capacitors in the design have provided sufficient running time to execute the function. These power loss tests were conducted in the lab settings. Starting an actual truck also draws current and drops the battery voltage. This test has not been conducted.

xvi. Destructive power test

The CAN Logger 3 was tested against two destructive power situations, which are high voltage and reverse polarity. For the first test, the raw 12V was increased up to 36V which was the intended maximum voltage for the design. The event of the voltage reaching 36V is unlikely, but is not impossible because voltage spike or vehicle struck by lightning can occur. The device raw 12V and ground were connected to a DC power generator where the output voltage was slowly increased from 0V to 36V. At the same time, the LED on the device was set high as an indication to determine whether the processor still functioned properly during the test. In the initial design, the TVS components used in the external voltage monitoring circuit were destroyed at approximately 24V. As a result, they were replaced with a different component that had a higher voltage rating. The test was conducted again, and the device failed at 36V, as anticipated.

For the second test, the raw 12V and ground wires were switched before connecting the CAN Logger 3. Through observation, the device repeatedly restarted itself, but no damage was observed. Both tests successfully demonstrated that the CAN Logger 3 has protection against high voltage up to 36V and reverse polarity.

xvii. LEDs and buttons test

The four LEDs and two push buttons on the CAN Logger 3 were tested for their functionality. The test code can be found here [90], which is displayed below:

```
//Define LED pins based on schematic
#define GREEN_LED_PIN 6
#define RED_LED_PIN 14
#define YELLOW_LED_PIN 5
#define BLUE_LED_PIN 39

//Define button pin, the button is soldered on SW21
#define button1 28
#define button2 53
//Create an on/off boolean for the button
bool buttonState1;
bool buttonState2;

void setup() {
  // put your setup code here, to run once:
  //Define LED pin mode
  pinMode(GREEN_LED_PIN,OUTPUT);
  pinMode(YELLOW_LED_PIN,OUTPUT);
  pinMode(RED_LED_PIN,OUTPUT);
  pinMode(BLUE_LED_PIN,OUTPUT);
  //Pull button high
  pinMode(button1,INPUT_PULLUP);
  pinMode(button2,INPUT_PULLUP);
}

void loop() {
  // put your main code here, to run repeatedly:
  //If button is pushed, the pin will pull low
  buttonState1= digitalRead(button1);
  buttonState2= digitalRead(button2);
  digitalWrite(GREEN_LED_PIN,buttonState1);
  digitalWrite(YELLOW_LED_PIN,buttonState1);
  digitalWrite(RED_LED_PIN,buttonState2);
  digitalWrite(BLUE_LED_PIN,buttonState2);

}
```

The pins for the four LEDs and the two push buttons were defined. Two Boolean variables with true/false state for the buttons were also defined. The LED pins were set to output mode while the button pins were set to pullup input. The Boolean variables were tied to the button digital value and the LED outputs were tied to the Boolean variables. When the test script was uploaded, the LEDs were always on until the buttons were pressed. This indicates that the LEDs and the buttons were wired correctly and functioned properly. Figure 2-71 shows the CAN Logger 3 with all four LEDs lit up.

Figure 2-71. LEDs test

xviii. WiFi test

Even though the WiFi function has not been implemented for the current operation, the ATWINC1500 WiFi module was tested to ensure proper functionality for future use. The library used for the module can be found here [91]. Because the module is connected to the processor SPI1, the library was changed to adapter for this design in WiFi101-master/src/bus_wrapper/source/nm_bus_wrapper_samd21.cpp, as seen:

```
53 #if !defined(WINC1501_SPI)
54 #define WINC1501_SPI SPI1 // Change here from SPI to SPI1
55 #endif
```

The first test was to check for firmware updates to make sure that the processor could correctly communicate with the module and it was up to date. Before running the test, the J2 connection was bridged to enable WiFi capability. The test script was used from the library example [92], as shown below:

```
#include <SPI.h>
#include <WiFi101.h>

//Define the pins for WiFi chip
#define WiFi_EN 24
#define WiFi_RST 25
#define WiFi_CS 31
#define WiFi_IRQ 23

void setup() {
//Initialize WiFi module
  WiFi.setPins(WiFi_CS,WiFi_IRQ,WiFi_RST);
  pinMode(WiFi_EN, OUTPUT);
  digitalWrite(WiFi_EN,HIGH);

// Print firmware version on the shield
  String fv = WiFi.firmwareVersion();
  String latestFv;
  Serial.print("Firmware version installed: ");
  Serial.println(fv);

  if (REV(GET_CHIPID()) >= REV_3A0) {
    // model B
    latestFv = WiFi_FIRMWARE_LATEST_MODEL_B;
  } else {
    // model A
    latestFv = WiFi_FIRMWARE_LATEST_MODEL_A;
  }

  // Print required firmware version
  Serial.print("Latest firmware version available : ");
  Serial.println(latestFv);
}
```

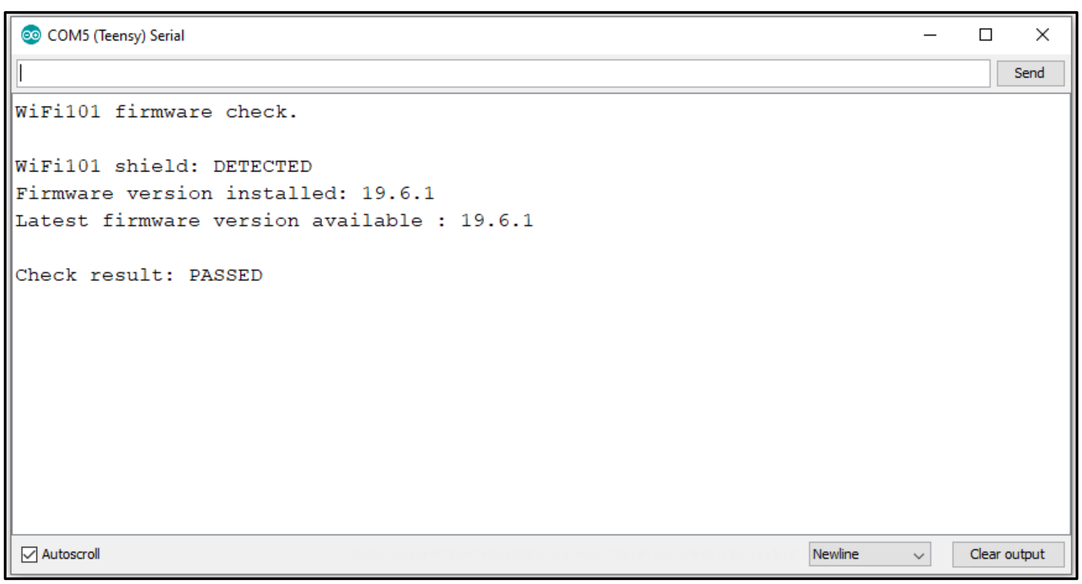


Figure 2-72. ATWINC1500 WiFi module firmware check

Because SPI1 was used, the pinouts for the communication were redefined according the design schematics: Enable (EN) – 24, Reset (RST) – 25, Chip Select (CS) – 31, and Interrupt Signal (IRQ) – 23. The Enable pin was set high for normal operation, according to the datasheet. The firmware was then retrieved and printed out on the Arduino console, as illustrated in Figure 2-72.

The next test was to examine how the ATWINC1500 module handled the speed of the CAN network and if logging CAN messages via WiFi was feasible for future reference. In this setup, the connection was established between the CAN Logger 3 and a local computer with Python application. The code for the CAN Logger 3 and the Python can be found at [93] and [94], respectively. The CAN Logger 3 test script is shown:

```
#include "arduino_secrets.h"
#include <FlexCAN.h>
#include <WiFi101.h>

char ssid[] = SECRET_SSID;
char pass[] = SECRET_PASS;

int port = 80;
WiFiServer server(port);

void setup(){
//Create open network.
status = WiFi.beginAP(ssid, pass);
}

void loop(){
WiFiClient client = server.available();
  if (client) {                // if you get a client
    if (Can0.available()) {
       Can0.read(rxmsg);
       load_buffer();
    }
    if (Can1.available()) {
       Can1.read(rxmsg);
       load_buffer();
    }
  }
}
```

And the Python code is:

```
import socket
import sys


#setup tcp client for CAN data transfer
SERVER_IP = "192.168.1.1" #insert IP address of server here
SERVER_PORT = 80

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

try:
    sock.connect((SERVER_IP, SERVER_PORT))
except OSError:
    print("Could not connect TCP Socket. Make sure SERVER_IP is correct.")
    sys.exit()

with open (LOG_FILE_NAME, 'w') as file:
    while True:
        data = sock.recv(Buffer_size)
```

The CAN Logger 3 acted as a host which broadcasted its WPA2 access point with the SSID and password stored in the arduino_secrets.h, and the port was set at 80. When the computer successfully connected to the wireless network with the correct password, the CAN Logger 3 started reading any available message from the bus, packed those data to the WiFi frame, and sent them over. When the computer received the data, it unpacked and saved them as a log. The test results are shown in Figure 2-73.

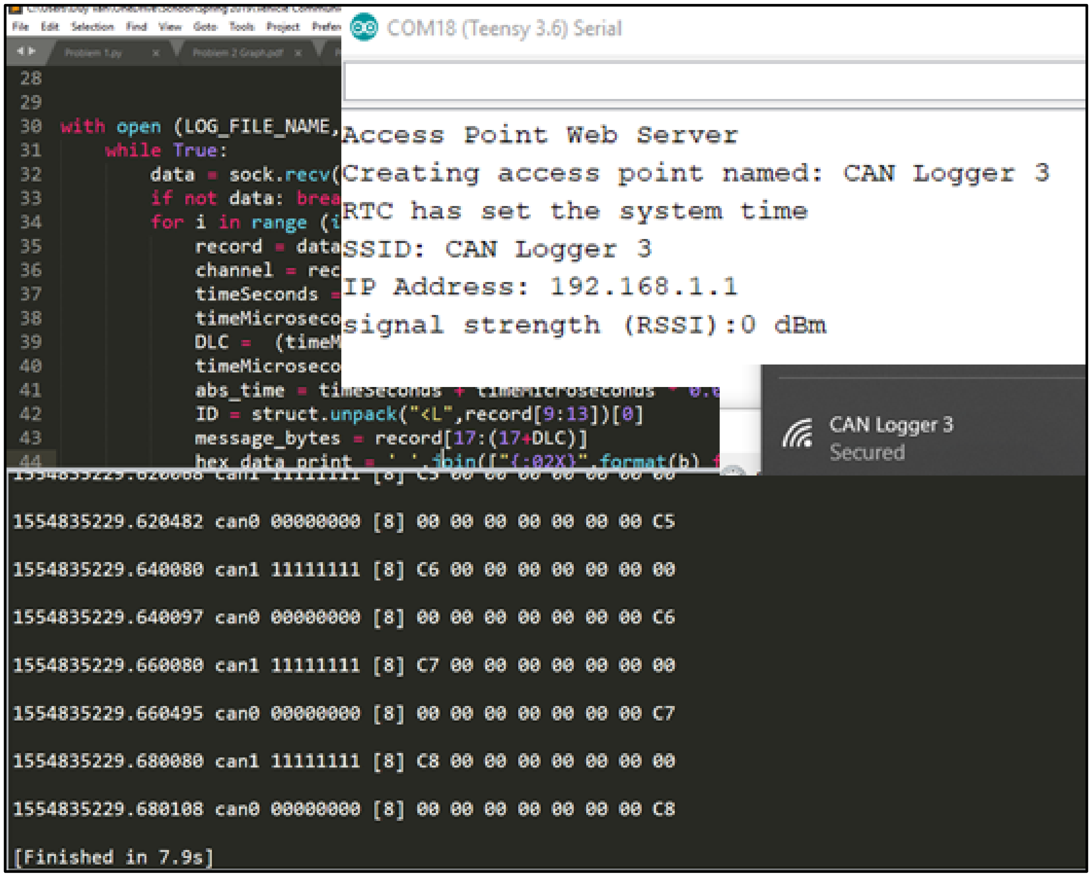


Figure 2-73. CAN logging via WiFi

However, the challenge discovered in this test was that there are missing messages from the log, indicating that the ATWINC1500C module could not keep up with the speed, even at normal bus load and without encryption. Therefore, the idea of using the CAN Logger 3 to stream live data via WiFi has not been further explored. On the other hand, without the speed constraint, the CAN Logger 3 can still utilize the WiFi feature to send the log files stored from the SD to the local computer. This is, in fact, in the scope of the project and can be used as an option for data uploading process in the future.

xix. Real-time clock test

The real-time clock provides an accurate timestamp of when the logging session starts, which is important for user and forensics reference. This function is based on the elapsed timer within the processor and is powered by the 3V coin cell battery. Information of the integrated time library from the teensyduino can be found here [95]. The code for time function used in the main logging firmware [89] is shown:

```
#include <TimeLib.h> // be able to keep realtime.

void setup(){
//Setup timing services
  setSyncProvider(getTeensy3Time);
  if (timeStatus()!= timeSet) {
    Serial.println("Unable to sync with the RTC");
  } else {
    Serial.println("RTC has set the system time");
  }
  setSyncInterval(1);

  sprintf(timeString,"%04d-%02d-%02d
%02d:%02d:%02d.%06d",year(),month(),day(),hour(),minute(),second(),uint32_t
(microsecondsPerSecond));
  Serial.println(timeString);
}
```

After the TimLib.h library was imported, the device synchronized its time with the clock of the computer where the firmware was uploaded. The 3V coin cell battery would keep the time running when the device was unplugged. Figure 2-74 shows the synchronized real time clock from the logging firmware.

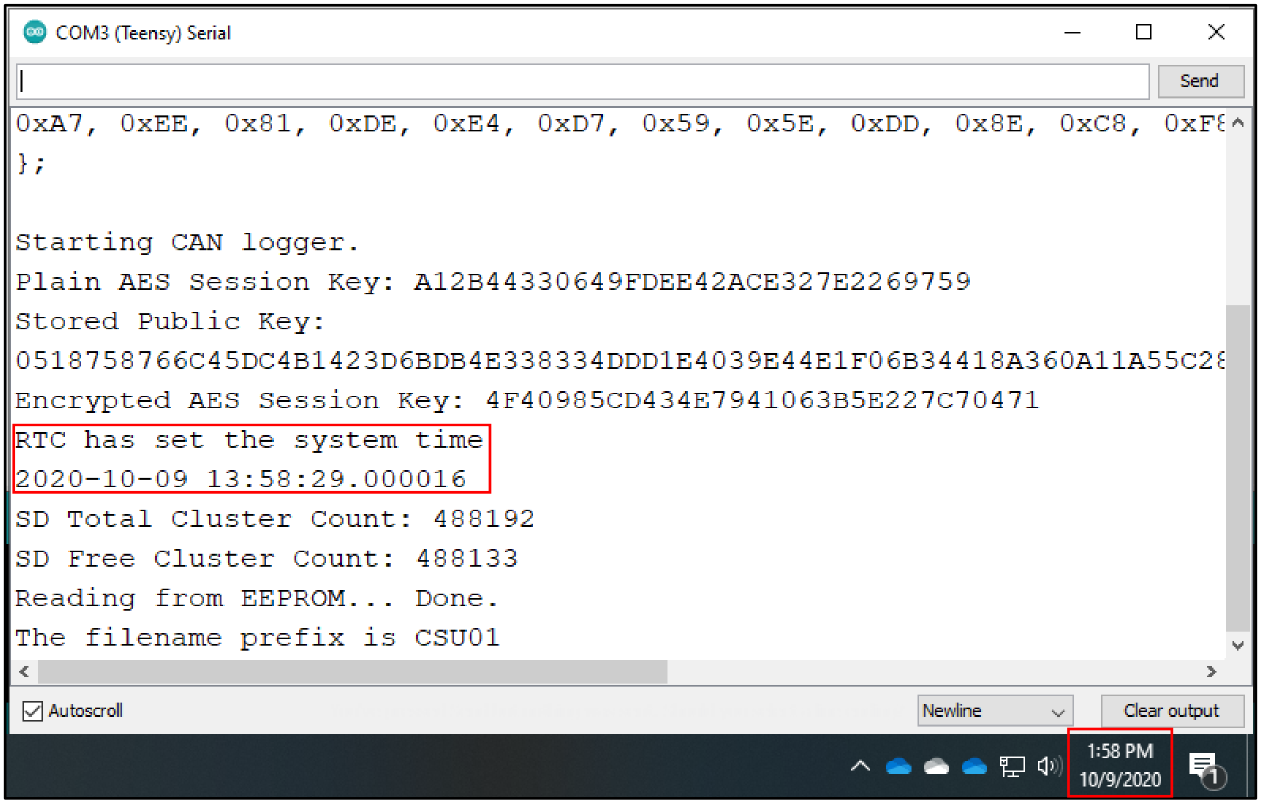


Figure 2-74. Setting real time clock from the logging firmware

To test the real time clock, the device was plugged into a CAN bus for logging two sessions with 7 days apart. The timestamps of those log files were compared with each other and with the actual time. They were accurate and therefore, the real time clock has worked as intended.

As mentioned above, the current configuration has the real-time clock of the CAN Logger 3 tied to the clock of the computer where the main firmware is uploaded, which can be limited and inconvenient. Therefore, setting time to UTC would be better.

xx. Error frame test

When errors occur during operation, the device should be able to capture error frames for troubleshooting. It is an important factor in the data pool for anomaly or intrusion detection. The FlexCAN library has been modified to capture error frames, which can be seen in the *FlexCAN::error_isr()* function in FlexCAN.cpp file [96]. The error.h file [97] from socketCAN is utilized to define the types of error. The device can capture six different errors:

1. A CAN frame is not acknowledged
2. Bit stuffing violation
3. The fixed-form bit field contains at least one illegal bit, causing format error
4. CAN frame Cyclic Redundancy Check (CRC) error has been detected
5. Unable to send dominant bit, which causes bit0 error
6. Unable to send recessive bit, which causes bit1 error

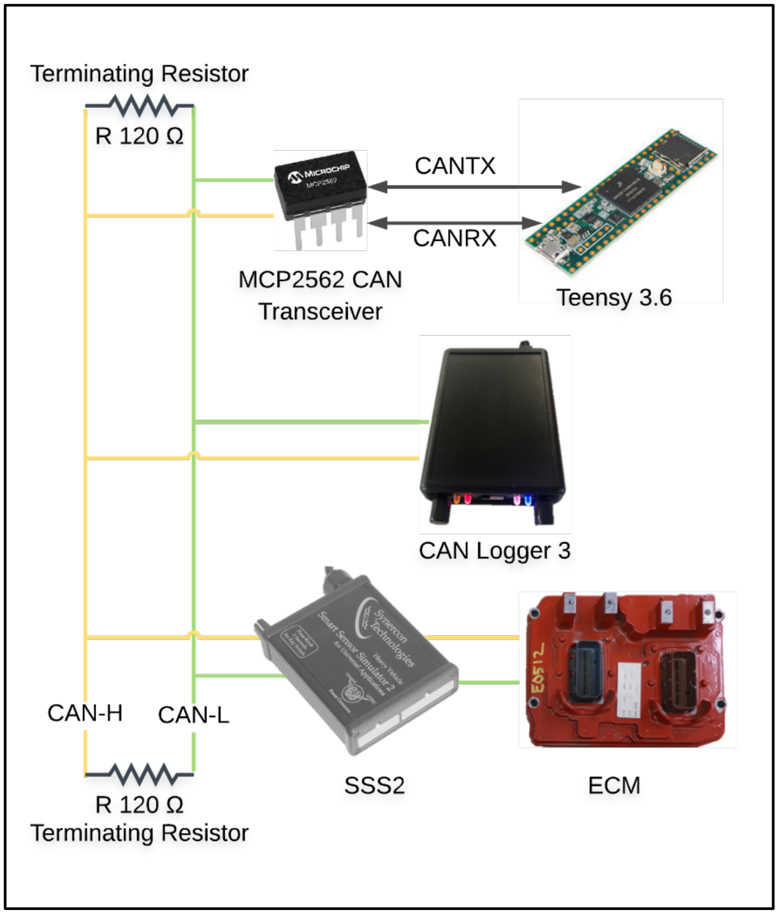


Figure 2-75. Bit stuffing error injection setup

To test the error frame capability, a setup was made with the purpose of injecting bit stuffing errors, as seen in Figure 2-75 above. An SSS2 and an ECM were used to simulate a truck network with two 120 Ω terminating resistors, one at either end. A CAN Logger 3 was connected to the CAN bus for data collection. A Teensy 3.6 and an MCP2562 CAN transceiver were also added as a node for bit stuffing error injection using the following code [98]:

```
#define rx_pin 4
#define tx_pin 3
elapsedMicros counter;

void setup() {
  // put your setup code here, to run once:
Serial.begin(9600);
pinMode(rx_pin,INPUT);
pinMode(tx_pin,OUTPUT);

}

void loop() {
  // put your main code here, to run repeatedly:

  if(digitalRead(rx_pin) == 0) {
   counter = 0;
   while (counter< 17) digitalWrite(tx_pin,LOW);
   digitalWrite(tx_pin,HIGH);
   delay(100);
  }
}
```

Pins 3 and 4 on the teensy 3.6 were attached to the CANTX and CANRX, respectively. Both of these lines operated at 3.3V. When there were CAN messages on the bus, the MCP2562 received the data and converted the CANRX signal to the Teensy 3.6. Bit stuffing is asserting a bit of opposite polarity after five consecutive bits of the same polarity. As a result, an interrupt was attached to the CANTX such that when the teensy 3.6 received the start of frame dominant bit signal on the CANRX, the CANTX would be set high for more than five-bit length to set a bit stuffing error. After many trials and errors, the right timing was determined to be 17 µs. Figure 2-76 illustrates the CANTX and CANRX signal using the Logic Analyzer Saleae during the error injection.

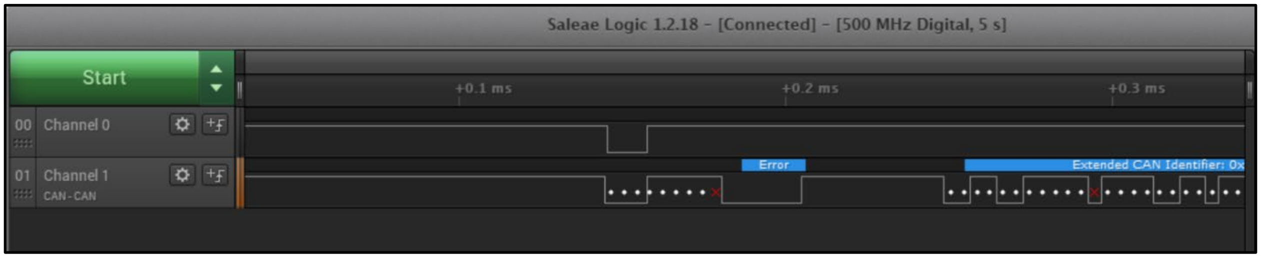
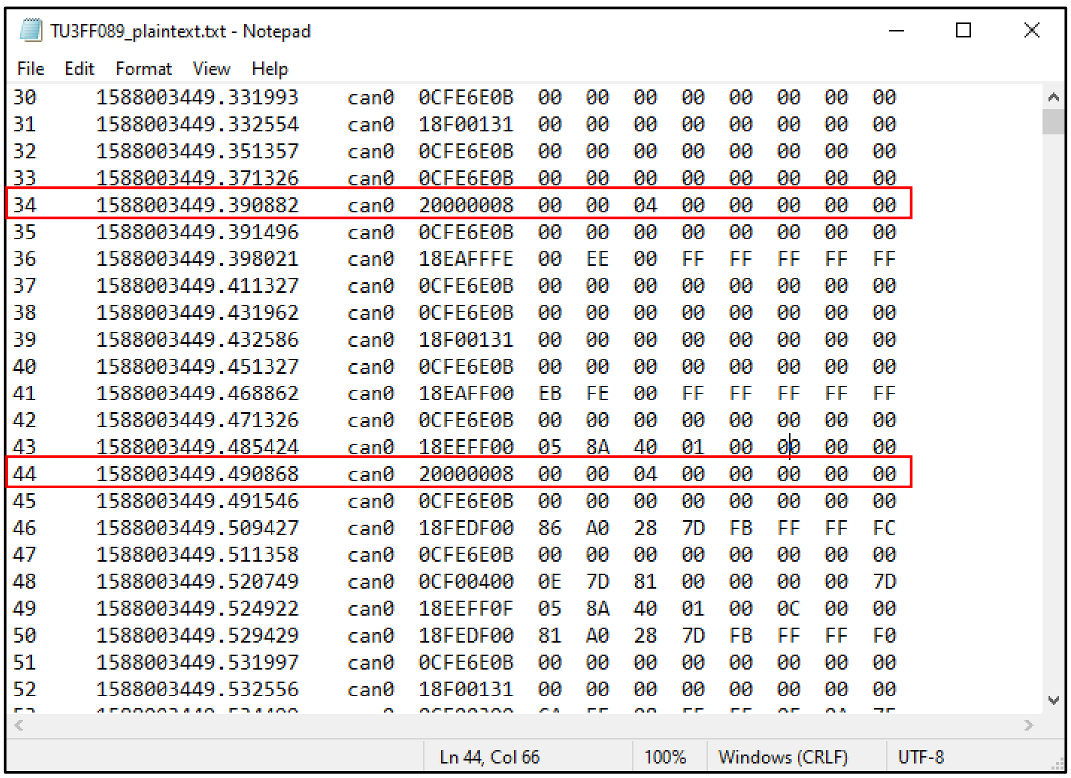


Figure 2-76. CAN0TX (channel 0) and CAN0RX (channel 1) signals

Figure 2-76 shows the exact setup as described above. When the first start of frame bit occurred, the CANTX (channel 0) was set for 17 µs , which successfully generated a bit stuffing error with 6 recessive bits in a row on CANRX (channel 1).



30	1588003449.331993	can0	0CFE6E0B	00	00	00	00	00	00	00	00
31	1588003449.332554	can0	18F00131	00	00	00	00	00	00	00	00
32	1588003449.351357	can0	0CFE6E0B	00	00	00	00	00	00	00	00
33	1588003449.371326	can0	0CFE6E0B	00	00	00	00	00	00	00	00
34	1588003449.390882	can0	20000008	00	00	04	00	00	00	00	00
35	1588003449.391496	can0	0CFE6E0B	00	00	00	00	00	00	00	00
36	1588003449.398021	can0	18EAFFFE	00	EE	00	FF	FF	FF	FF	FF
37	1588003449.411327	can0	0CFE6E0B	00	00	00	00	00	00	00	00
38	1588003449.431962	can0	0CFE6E0B	00	00	00	00	00	00	00	00
39	1588003449.432586	can0	18F00131	00	00	00	00	00	00	00	00
40	1588003449.451327	can0	0CFE6E0B	00	00	00	00	00	00	00	00
41	1588003449.468862	can0	18EAFF00	EB	FE	00	FF	FF	FF	FF	FF
42	1588003449.471326	can0	0CFE6E0B	00	00	00	00	00	00	00	00
43	1588003449.485424	can0	18EEFF00	05	8A	40	01	00	00	00	00
44	1588003449.490868	can0	20000008	00	00	04	00	00	00	00	00
45	1588003449.491546	can0	0CFE6E0B	00	00	00	00	00	00	00	00
46	1588003449.509427	can0	18FEDF00	86	A0	28	7D	FB	FF	FF	FC
47	1588003449.511358	can0	0CFE6E0B	00	00	00	00	00	00	00	00
48	1588003449.520749	can0	0CF00400	0E	7D	81	00	00	00	00	7D
49	1588003449.524922	can0	18EEFF0F	05	8A	40	01	00	0C	00	00
50	1588003449.529429	can0	18FEDF00	81	A0	28	7D	FB	FF	FF	F0
51	1588003449.531997	can0	0CFE6E0B	00	00	00	00	00	00	00	00
52	1588003449.532556	can0	18F00131	00	00	00	00	00	00	00	00

Figure 2-77. Error frames captured in CAN log file

Figure 2-77 shows the CAN data from the CAN logger after the test. Error frames with bit stuffing error type, were successfully captured, as shown in the figure with messages that contained ID of 0x20000008 and hex 0x04 on the third data byte. The error.h [97] and FlexCAN.cpp [96] files in the FlexCAN library [25] were used to define the error frames with specific message ID and data for different CAN violations. The bit stuffing error is defined in the codes below.

Error.h code:

```
#define CAN_ERR_FLAG 0x20000000U /* error message frame */
#define CAN_ERR_PROT       0x00000008U/* protocol violations /data[2..3] */
#define CAN_ERR_PROT_STUFF       0x04 /* bit stuffing error */
```

FlexCAN.cpp code:

```
void FlexCAN::error_isr (void)
{
  uint32_t status = FLEXCANb_ESR1 (flexcanBase);
  if (report_errors){
    CAN_message_t msg;
    msg.id = CAN_ERR_FLAG; //Set this to show this is an error id
    msg.len = 8;
    msg.ext = 1;
    memset(&msg.buf, 0, 8);

// A bit stuffing error was detected.
    if (status & FLEXCAN_ESR_STF_ERR) {
      msg.id |= CAN_ERR_PROT; /* protocol violations / data[2..3] */
      msg.buf[2] |= CAN_ERR_PROT_STUFF;  /* bit stuffing error */
    }
```

When there is an error occurred, the message ID is set to CAN_ERR_FLAG (0x20000000). This results in the 0x2 appeared in the most significant byte of the message ID. If the error is a bit stuffing violation, a bitwise OR is performed between the message ID and the CAN_ERR_PROT (0x00000008). The third byte in the data field is also set to CAN_ERR_PROT_STUFF (0x04). Therefore, the message ID is 0x20000008 and the third data byte is 0x04, as seen in the results from the test. In conclusion, the device passed the test.

xxi. Request message test

The CAN Logger 3 can send out request messages for On Request parameters defined in SAE J1939. These messages are defined by the Parameter Group Number (PGN) within the SAE J1939 standard [3], specifically the SAE J1939-71 vehicle application layer [99]. The request message PGNs used for this project are listed below:

```
uint16_t request_pgn[NUM_REQUESTS] = {
  65261, // Cruise Control/Vehicle Speed Setup
  65214, // Electronic Engine Controller 4
  65259, // Component Identification
  65242, // Software Identification
  65244, // Idle Operation
  65260, // Vehicle Identification
  65255, // Vehicle Hours
  65253, // Engine Hours, Revolutions
  65257, // Fuel Consumption (Liquid)
  65256, // Vehicle Direction/Speed
  65254, // Time/Date
  65211, // Trip Fan Information
  65210, // Trip Distance Information
  65209, // Trip Fuel Information (Liquid)
  65207, // Engine Speed/Load Factor Information
  65206, // Trip Vehicle Speed/Cruise Distance Information
  65205, // Trip Shutdown Information
  65204, // Trip Time Information 1
  65200, // Trip Time Information 2
  65250, // Transmission Configuration
  65203, // Fuel Information (Liquid)
  65201, // ECU History
  65168, // Engine Torque History
  64981, // Electronic Engine Controller 5
  64978, // ECU Performance
  64965, // ECU Identification Information
  65165  // Vehicle Electrical Power #2
};
```

Among those PGNs, the component and vehicle identification are the most important ones, which are 65259 (0x00FEEB) and 65260 (0x00FEEC), respectively. The code to send the request messages within the main firmware [89] are shown below:

```
if (send_requests){
    if (send_passes < NUM_REQUEST_PASSES){
      if (send_request_timer > REQUEST_TIMING){
        send_request_timer = 0;
        txmsg.len = 3;
        txmsg.id = 0x18EAFFF9;
        txmsg.buf[0] = (request_pgn[request_index] & 0x0000FF);
        txmsg.buf[1] = (request_pgn[request_index] & 0x00FF00) >> 8 ;
        txmsg.buf[2] = (request_pgn[request_index] & 0xFF0000) >> 16;

        //These are in reverse byte order.
        send_Can0_message(txmsg);
        if (RXCount1 > 0) send_Can1_message(txmsg);

        request_index++;
        if (request_index >= NUM_REQUESTS) {
          request_index = 0;
          send_passes++;
          //shuffle
          //Serial.println("Shuffling Requests");
          for (int i = 0; i < NUM_REQUESTS; i++) {
            int j = random(i, NUM_REQUESTS);
            auto temp = request_pgn[i];
            request_pgn[i] = request_pgn[j];
            request_pgn[j] = temp;
          }
        }
      }
    }
}
```

The request messages were sent individually with the ID of 0x18EAFFF9 and the data field consisted of the corresponding 3-byte PGN in the list as described previously. In the message ID, byte 0xEA represented Request PGN and byte 0xF9 represented the diagnostic tool source address. The PGNs were shuffled before being sent in the least significant byte order (little-endian).

To test this feature, the CAN Logger 3 was attached to a CAN network to log the data while sending request messages to the ECM. The log data was parsed to check if the ECM responded to the requests with valid information. Figure 2-78 and Figure 2-79 shows the request messages and the corresponding responses from the ECM for component and vehicle identification information, respectively.

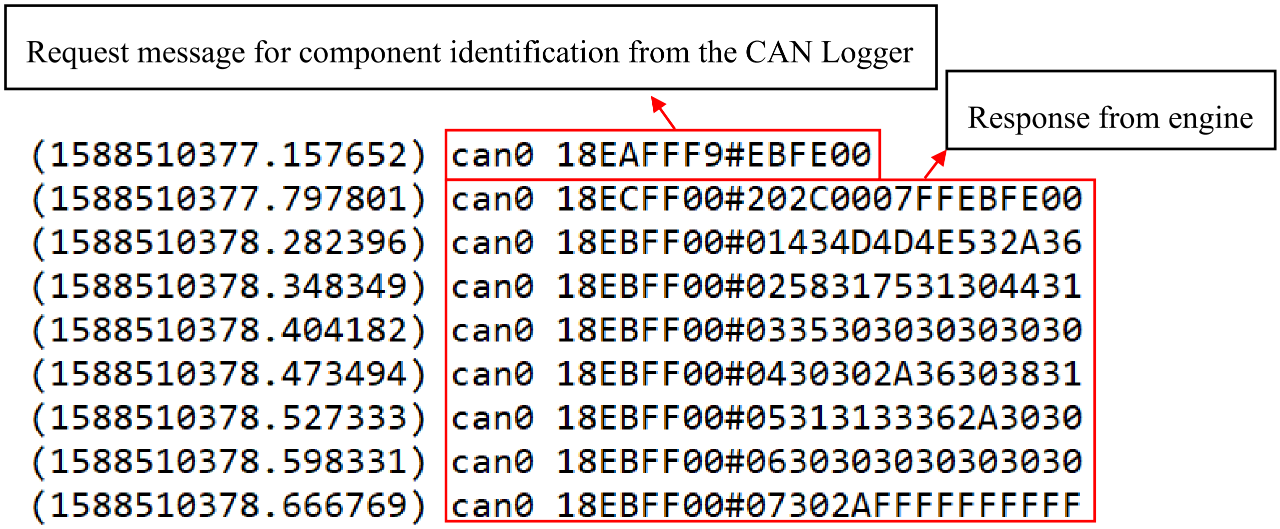


Figure 2-78. Request message and its responses for component identification

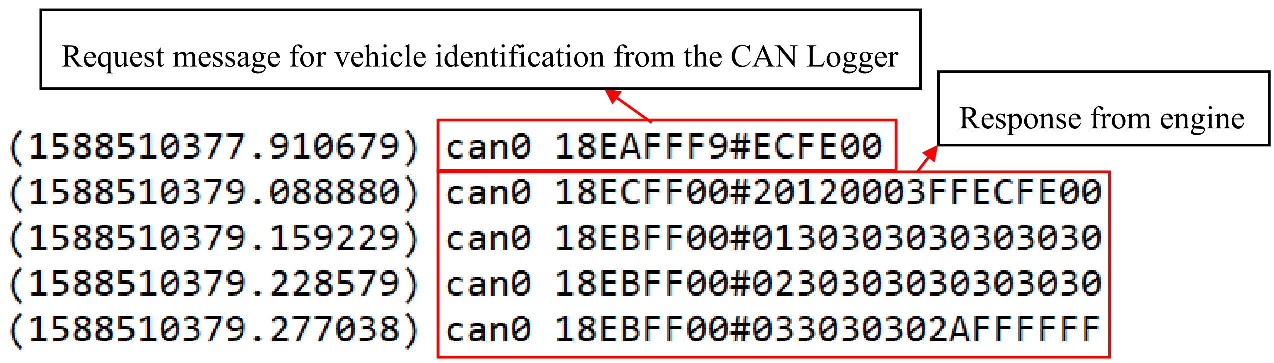


Figure 2-79. Request message and its responses for vehicle identification

In Figure 2-79, the first message was the request asking for the component identification with the PGN of 0x00FEEB (65259). The PGN in the CAN frame was in least significant byte order (EBFE00), as stated in the J1939-21 Data Link Layer [2]. The next CAN message with the ID of 0x18ECFF00 was a response from the engine saying it would response to that PGN request by sending seven messages containing the information using the transport layer protocol with the ID of 0x18EBFF00. Within the data field of those seven messages, the first byte indicated the index and the last seven bytes were the actual data. By combining all those together, the full response in hex was:

```
434D4D4E532A36583175313044313530303030303030302A36303831313133362A303030303
030303030302A
```

Similarly, the full message for the vehicle identification from Figure 2-79 was:

```
30303030303030303030303030303030302A
```

Converting to ASCII would give:

```
CMMNS*6X1u10D1500000000*60811136*0000000000* - component identification
00000000000000000* - vehicle identification (VIN)
```

To confirm if the received responses were valid, a DG DPA5 RP1210 device was used to retrieve the information, as shown in Figure 2-80.

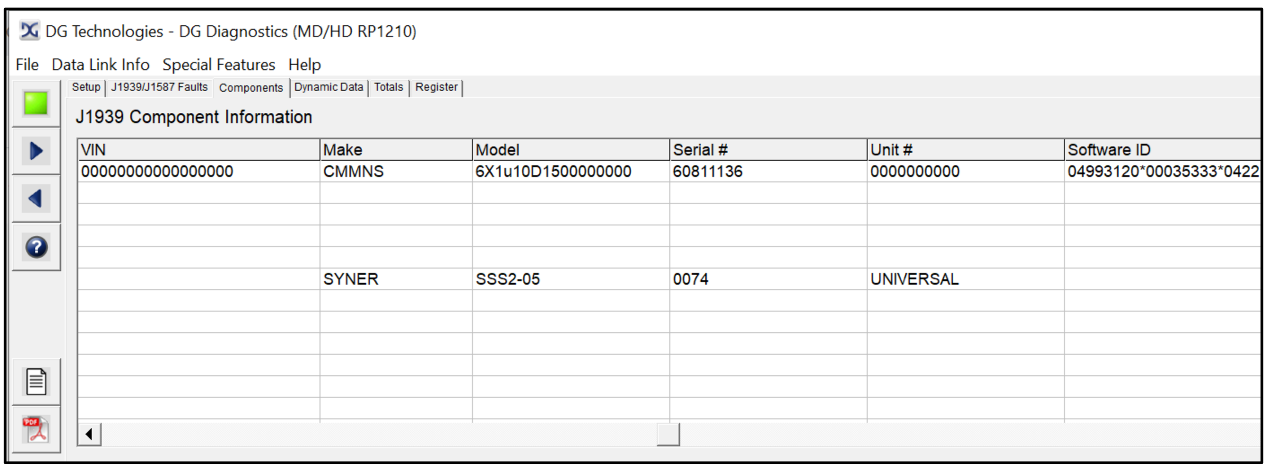


Figure 2-80. Component and vehicle identification information using DG Diagnostics tool

The component and vehicle identification information retrieved from the CAN Logger 3 matched the one from DG DPA5, which means that the CAN Logger 3 passed the test.

A. Process Overview

The CAN logging operation includes two main processes: provisioning and normal operation. Both are required to communicate with the Amazon Web Services (AWS), which was chosen as the third-party cloud services provider for this project. The interface between the CAN Logger 3 and the AWS services is done via a local computer running a Python application. The CAN logger devices communicate with the local computer through local serial USB. On the other hand, the connection between the computer and the AWS cloud is through the Internet with secure TLS using the Python requests module.

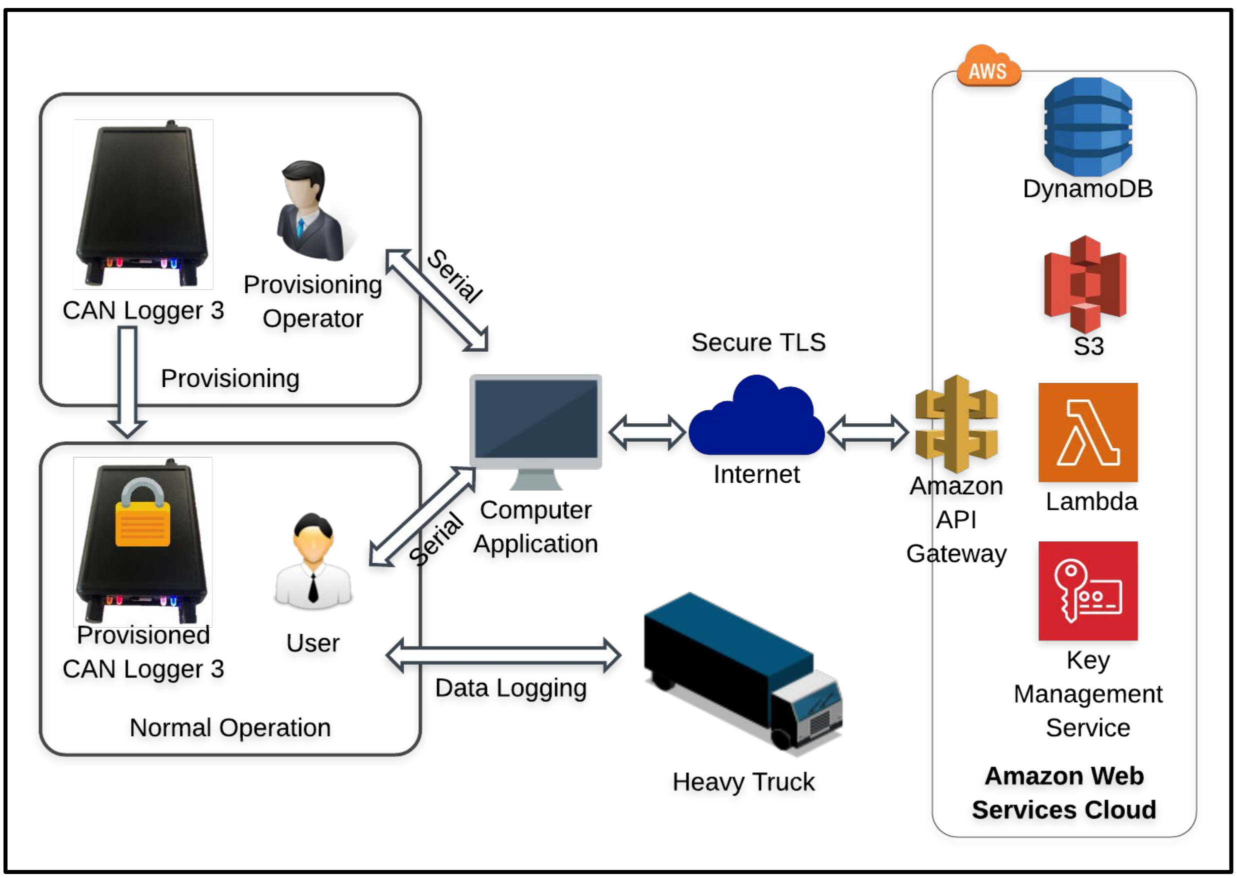


Figure 3-1. CAN Logger 3 software design overview

The provisioning process must happen first to configure the new device before it can be delivered to clients and function properly as intended. With the provisioned CAN logger, clients can use it as a standalone device to log data from heavy trucks securely with encryption. The encrypted log files will temporarily be stored on the device until uploaded to the AWS server for secure storage and data management. The process overview is depicted in Figure 3-1.

In order to achieve the security and privacy of this model, the following factors are assumed to be uncompromised:

1. The local computer with Python application
2. The provisioning operator
3. The Internet connection with secure TLS
4. The AWS third-party
5. The owner of the CAN logger

The local computer and the provisioning operator are parts of the device's manufacturing process. Preventing these two factors from being compromised is not in the scope of the CAN logging project but in the security of the local facility itself. As a result, these two factors are assumed to be safe in this project.

Transferring sensitive data via the Internet can be risky. However, by following the industry-standard using TLS, the connection via the Internet should be protected. Therefore, it is safe to assume that the communication between the Python application and the AWS is secure in this project.

Using a third-party cloud is a debatable subject because the data owners put all their trust and resources into the hand of a different company. However, this is common in the business world, where one relies on the services of data storage and the protection from others. On the other hand, some prefer to spend more resources to develop their own data management structure because the data may be too valuable to be stored elsewhere. The decision whether to use a third-party service depends on the needs of the data owner. Amazon is a big company with a favorable reputation, and its AWS provides a data management system with high security on its end at a much lower cost than building one. Therefore, AWS is trusted to be used in this project, and their security is assumed to not be easily compromised.

Lastly, the owner of the CAN logger is the only person who possesses and operates the device post-delivery. It is their responsibility to keep their device safe from unauthorized physical access. Any device that is in a wrong hand can be broken; it's only a matter of time because there is no such system that is 100% secure. For this project, the CAN logger owner is assumed to always have possession of the device and operate it correctly without any harmful intention. However, a well-designed system should make it extremely difficult for hackers to attack. It should take a lot of time and money to penetrate the system, and thus, the obstacles should discourage hackers from trying, or at least give the system administrator more time to detect and eliminate any threat. And if one device is compromised, it will not compromise all devices and the overall system should still function properly. The CAN logger was designed to follow this principle.

B. Provisioning

i. Provisioning function

The provisioning process is a one-time public key exchange between the CAN logger device and the server hosted through AWS before the devices get delivered to the users. A provisioning operator or system administrator will serve as a connecting role to implement and monitor the process between the device and the server. The initial provisioning's primary purposes are to acquire the device's identification for the server database and to exchange public keys to establish the same shared secret for secure communication using asymmetric cryptography. The steps in key exchange provision are depicted in Figure 3-2 and Table 3.1 below.

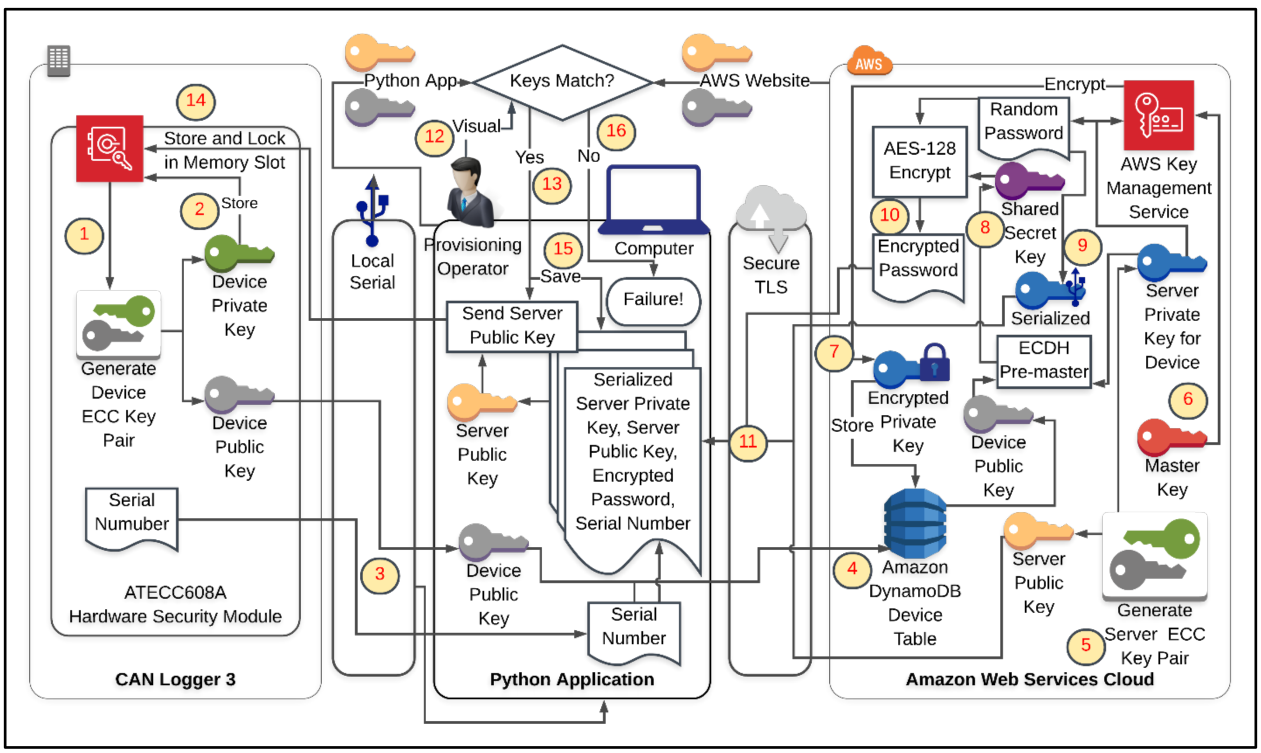


Figure 3-2. Key exchange provisioning process diagram

Table 3.1. Key exchange provisioning process

Process	System	Description
1	Embedded Firmware	Starting from the device, the ATECC608A hardware security module first generates an ECC key pair – the device private key and public key.
2	Embedded Firmware	The device private key is locked in the memory slot and cannot be changed or read.
3	Local Computer	The device's public key along with the ATECC608A ID are first sent to a Python application on a local computer controlled by the provisioning operator. The connection here is through local serial (mini USB cable).
4	Local Computer	The Python application then forwards the device public key and the HSM ID to AWS through the internet with secure TLS protocol.
5	AWS Cloud	Once the server receives the data from the Python application, it will use the lambda function to generate its ECC key pair specifically for this CAN logger.
6	AWS Cloud	The server private key is encrypted in AWS Key Management Service (KMS) using its master key (unique key managed by AWS).
7	AWS Cloud	The encrypted server private key is then stored and tied to the device ID in the AWS DynamoDB database.
8	AWS Cloud	The shared secret key is derived with ECDH pre-master with the device public key and the server private key.
9	AWS Cloud	The server private key is serialized and encrypted with a randomly generated 16-byte password for back up purpose.
10	AWS Cloud	The password is encrypted using AES-128 ECB mode because it is only 16 bytes. The AES encryption key used is the shared secret derived from ECDH.
11	AWS Cloud	The server public key, server serialized encrypted private key, and encrypted password are sent back to the Python application using the same secure TLS communication.
12	Local Computer	The provisioning operator will then perform a visual key comparation between the device and server public keys obtained from the Python application to the ones visible on AWS website. This makes sure that the server and the device both have the other's authentic public key in case the communication between the Python application and the AWS server is compromised.

13	Local Computer	Once the provisioning operator confirms the key match, the server public key will be sent to the CAN Logger 3 through local serial.
14	Embedded Firmware	The server public key is stored and locked in the ATECC608A memory key slot for future function implementation.
15	Local Computer	The provisioning operator can also save the serialized server private key, encrypted password, and the corresponding serial number to a JSON file, which is a physical backup that the administrators keep. However, to use the server private key, it needs to be loaded with the corresponding password, which can be decrypted as described in the next section.
16	Local Computer	If the key-check fails, the application will shows an error message.

ii. Get server private key password function

During the provisioning process, the random password generated for the server private key is sent to the local computer Python application. The key is converted to an ascii-armored PEM form, which is known as the serialized private key. The operator or administrator has an option to decrypt the password and use it to retrieve the serialized server private key stored in the JSON physical backup file. The process is illustrated in Figure 3-3 and Table 3-2.

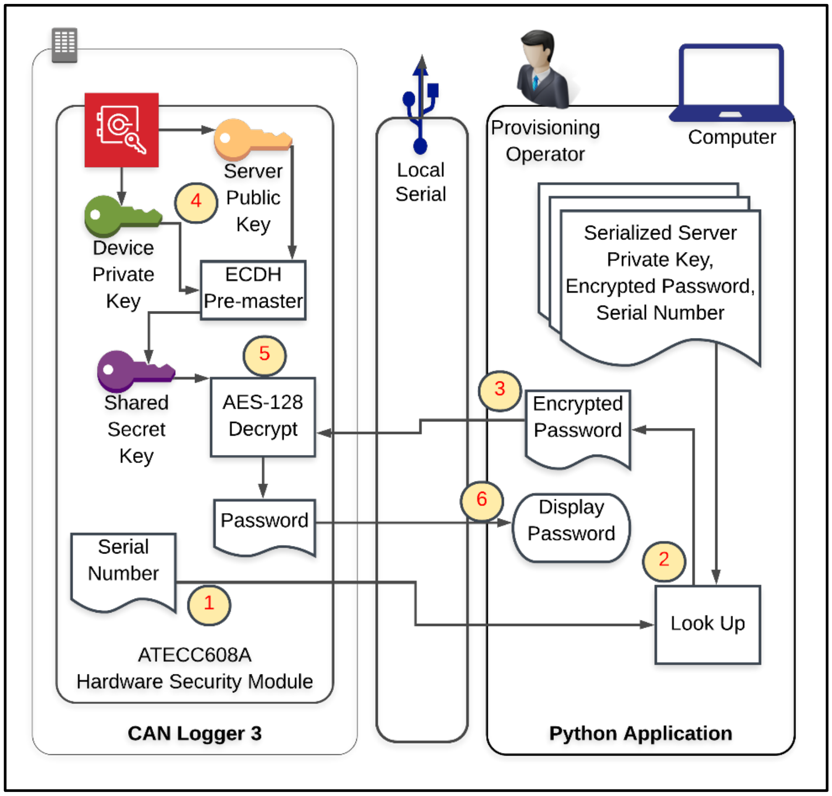


Figure 3-3. Get serialized server private key password function diagram

Table 3.2. Get serialized server private key password function process

Process	System	Description
1	Embedded Firmware	The CAN logger initially sends its serial number to the Python application for identification.
2	Local Computer	The local computer Python application loads the backup JSON and looks up the encrypted password from the corresponding serial number from the file.
3	Local Computer	The encrypted password is sent to the CAN logger device via local serial.
4	Embedded Firmware	The shared secret key is derived from ECDH pre-master algorithm with the stored device private key and the server public key.
5	Embedded Firmware	The encrypted password is decrypted using the shared secret key.
6	Local Computer	The decrypted password is sent back to the local computer application where it is displayed for the operator or administrator.

C. Normal Operation

i. Logging

After the key exchange provision has been completed, the device is ready to be used for logging sessions. In this process, the log data will be encrypted in real-time and stored locally on the SD card. It is then hashed and its digest along with the session metadata are signed before being sent to the server. These steps ensure that the contents of the files are not exposed or modified in storage and while being transmitted to the server through the Internet. Signing the logs verify that the server receives authentic data from the correct sender. The logging and file uploading process is depicted in Figure 3-4.

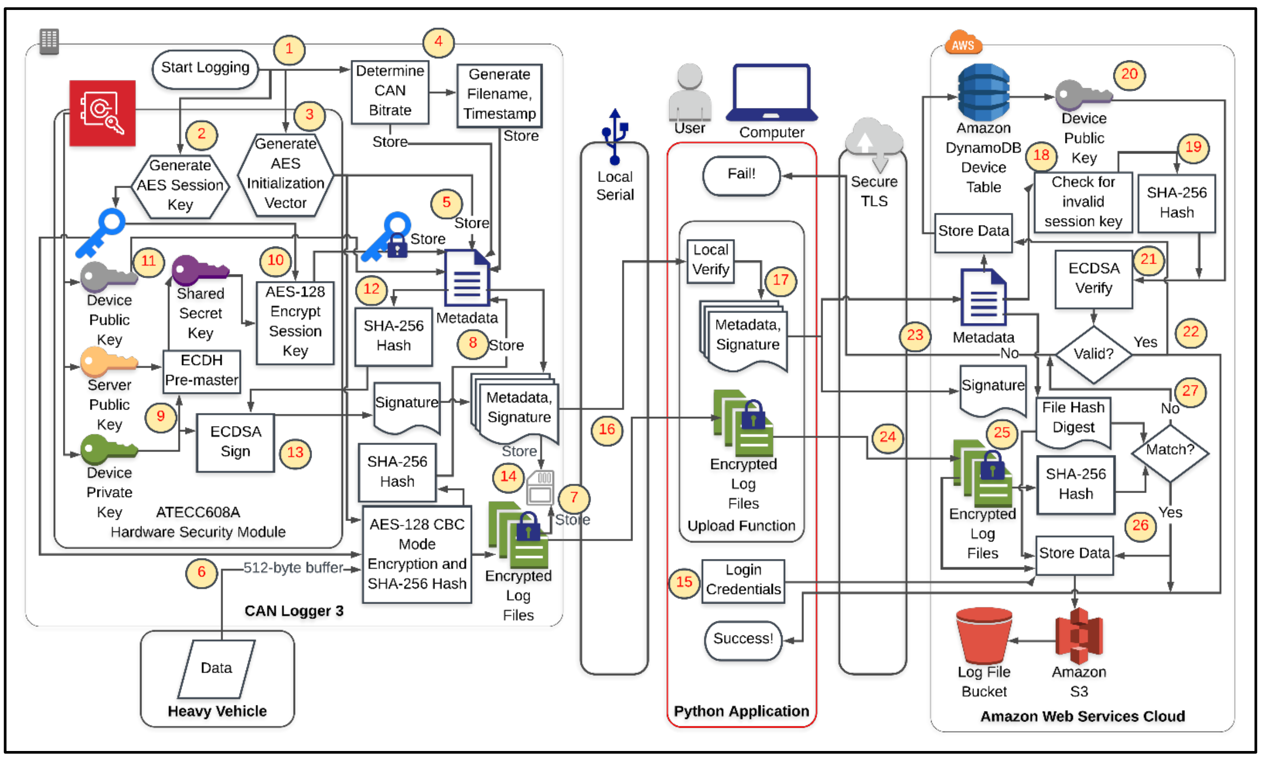


Figure 3-4. Logging and uploading files process diagram

Table 3-3. Logging and uploading files process

Process	System	Description
1	Embedded Firmware	When logging session starts, the ATECC608A HSM generates a 32-byte random number.
2	Embedded Firmware	The first 16 bytes of the 32-byte number is designated for the AES key of this logging session.
3	Embedded Firmware	The last 16 bytes of the 32-byte number is designated for the initialization vector (IV) for the AES CBC mode.
4	Embedded Firmware	The CAN logger initially determines the CAN bus bitrate with autobaud, and generates a metadata text file with the same name as the log file, which contains the timestamp and bitrate, to be stored on the SD card.
5	Embedded Firmware	The AES IV is appended to the metadata file.
6	Embedded Firmware	The CAN logger collects heavy vehicle data in 512-byte buffer. The first 508 bytes are actual data and the last 4 bytes are CRC-32 checksum for error detection. During the logging, the buffer is encrypted by the mmCAU and written to the binary file. When this buffer is full, the teensy processor SHA-256 hashes and updates the hash with previous buffers, if any. The buffer is reset, and the process repeats until the logging stops. A new log file is started when the current logging session reaches 1Gb of data.
7	Embedded Firmware	After the logging session finishes, the encrypted log file is stored in the SD card. This file has the same name as the metadata file and is in binary format.
8	Embedded Firmware	The SHA-256 hash of the encrypted log file is appended to the metadata file.
9	Embedded Firmware	The shared secret key is derived from ECDH pre-master algorithm using the device private key and the server public key stored in the ATECC608A HSM.
10	Embedded Firmware	The 16-byte AES session key is encrypted with AES-128 ECB using the shared secret key. The encrypted key is then appended to the metadata file.
11	Embedded Firmware	The device public key stored in the ATECC608A HSM is appended to the metadata file for later local verification.
12	Embedded Firmware	The metadata file is hashed using SHA-256.
13	Embedded Firmware	The metadata file hash digest is signed with ECDSA using the device private key.

14	Embedded Firmware	The metadata text file appended with its signature is stored in the SD card.
15	Local Computer	Before uploading the file to AWS, the user must log in with their credentials to identify themselves and establish secure connection. Their credentials will be tied to the uploading session later. The login process follows the AWS API authentication, which will be explained in detail later.
16	Local Computer	Through local serial, the device connects to the application which extracts the metadata file with its signature and the encrypted log file.
17	Local Computer	The metadata file signature is verified using the device public key stored in the metadata file. This process mainly checks the metadata file for error that may occur during logging operation or transmission to the computer application. However, it does not guarantee the file's integrity because the device public key used for verification is stored in the data to be verified itself and thus, the key is not reliable. Malicious users can replace the key with their own public key and resign the metadata file. A true integrity check will be performed on the AWS side. After the metadata is successfully verified, the metadata and its signature are sent to AWS via the Internet with secure TLS.
18	AWS Cloud	Once the server receives the metadata, it first checks for invalid session key, such as key containing all 0xFF or 0x00 that could occur when the logger failed to encrypt the AES session key.
19	AWS Cloud	The metadata file is hashed with SHA-256. The hash digest will be used for ECDSA verification.
20	AWS Cloud	The device public key is retrieved from AWS DynamoDB database using the device serial number from the metadata. The device public key here is from the provisioning process and thus, it is reliable to be used in ECDSA verification.
21	AWS Cloud	The metadata file is verified with ECDSA using the metadata file hash, its signature, and the device public key.
22	AWS Cloud	If the metadata verification is successful, AWS sends a response back to the local computer application with a message that the metadata verification has passed.
23	AWS Cloud	If the metadata verification fails, AWS sends a response back to the local computer application with a message that the metadata verification has not passed and the metadata may have been compromised.

24	Local Computer	When the local computer application receives the message that the metadata has been verified successfully, the application starts sending the encrypted log file to AWS.
25	AWS Cloud	When AWS receives the encrypted log file, the server hashes the file with SHA-256 and the hash digest is compared with the one from the metadata file.
26	AWS Cloud	If the hashes match, the encrypted log file with its corresponding hash and user credentials are stored in Amazon S3 Bucket. AWS also sends a response back to the local computer application with a message that the encrypted log file has been uploaded successfully.
27	AWS Cloud	If the hashes do not match, AWS also sends a response back to the local computer application with a message that the encrypted log file has not been uploaded because the file has been compromised.

Data structure

The data structure of how the 512-byte buffer is collected is illustrated in Figure 3-5. This format is from CAN logger version 2 and has not been changed for version 3. The EEPROM memory map for the autobaud feature (Chapter 2, part H.ii.) is also explained in the same figure.

SD Card Memory Block. 512 Bytes are stored at a time in the following format

Bytes	0	1	2	3	4 through 478	479	480	481	482	483	484	485	486	487	488	489	490
Data	C	A	N	2	Nineteen (19) CAN Frames	RXCount0				RXCount1				RXCount2			
Hex	43	41	4E	32	SEE CAN FRAME STRUCTURE	MSB			LSB	MSB			LSB	MSB			LSB
Notes	Characters					uint32_t				uint32_t				uint32_t			

491	492	493	494	495	496	497	498	499	500	501	502	503	504	505	506	507	508	509	510	511
Can0	Can1	Can2	Can0	Can1	Can2	T	U	2	_	_	N1	N2	N3	Write Time			CRC32			
uint8_t	uint8_t	uint8_t	uint8_t	uint8_t	uint8_t	54	55	32			ASCII Encoded			MSB		LSB	MSB			LSB
Receive Error Counts			Transmit Error Counts			Version			Logger Number		File Number			Microseconds for SDCard			Calculated from bytes 0 through 507			

CAN Frame Structure

Bytes	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
Data	Channel	Timestamp				System				CAN Identifier				DLC	Microseconds per			B0	B1	B2	B3	B4	B5	B6	B7
Hex	0 | 1 | 2	LSB			MSB	LSB			MSB	LSB			MSB	8	LSB		MSB	01	02	03	04	05	06	07	08
Notes	Corresponds to Can0, Can1, or Can2	Number of seconds from the epoch (1970)				The system microsecond counter when the CAN registers were read.				CAN ID with the Error Flags and Extended Flag, like Socket CAN				Data Length Code	Fractional seconds per tick of the Timestamp			Message Data Bytes padded with x0FF if not used.							

EEPROM Memory Map

0x00	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Data	Bitrate	Bitrate	Bitrate	RES	2	_	_	null	N1	N2	N3	null	T	U	null
Hex					32			0x00	ASCII Encoded						0x00
Notes	Can0 Bitrate	Can1 Bitrate	Can2 Bitrate		Logger Identifier of 2 uppercase letters				File ID. Each digit can be 0-9 or A-Z for a total of 36*3 = 46,656 files.				Brand Name of Logger (i.e. "TU") to start each filename.		

CAN_message_t

```
uint32_t id;          // can identifier
uint32_t micros;      // system microseconds
uint32_t rxcount;     // number of received messages
uint16_t timestamp;   // FlexCAN time when message arrived
struct {
  uint8_t extended:1; // identifier is extended (29-bit)
  uint8_t remote:  1; // remote transmission request packet type
  uint8_t overrun: 1; // message overrun
  uint8_t reserved:5;
} flags;
uint8_t len;          // length of data
uint8_t buf[8];       // data bytes
```

Figure 3-5. Data structures for 512-byte buffer, CAN frame, and autobaud EEPROM

ii. Get Key and/or Decrypt Log File Function While Connecting to Logger

After the user has successfully uploaded the metadata and the encrypted log file, they can retrieve the file AES session key in plaintext, which is then used to decrypt the file locally. As shown in Figure 3-6, this process can only be performed where the local computer Python application still has the uploaded metadata file and the encrypted log file in its memory, either from the uploading process or the files have to be extracted again from the same connected CAN logger.

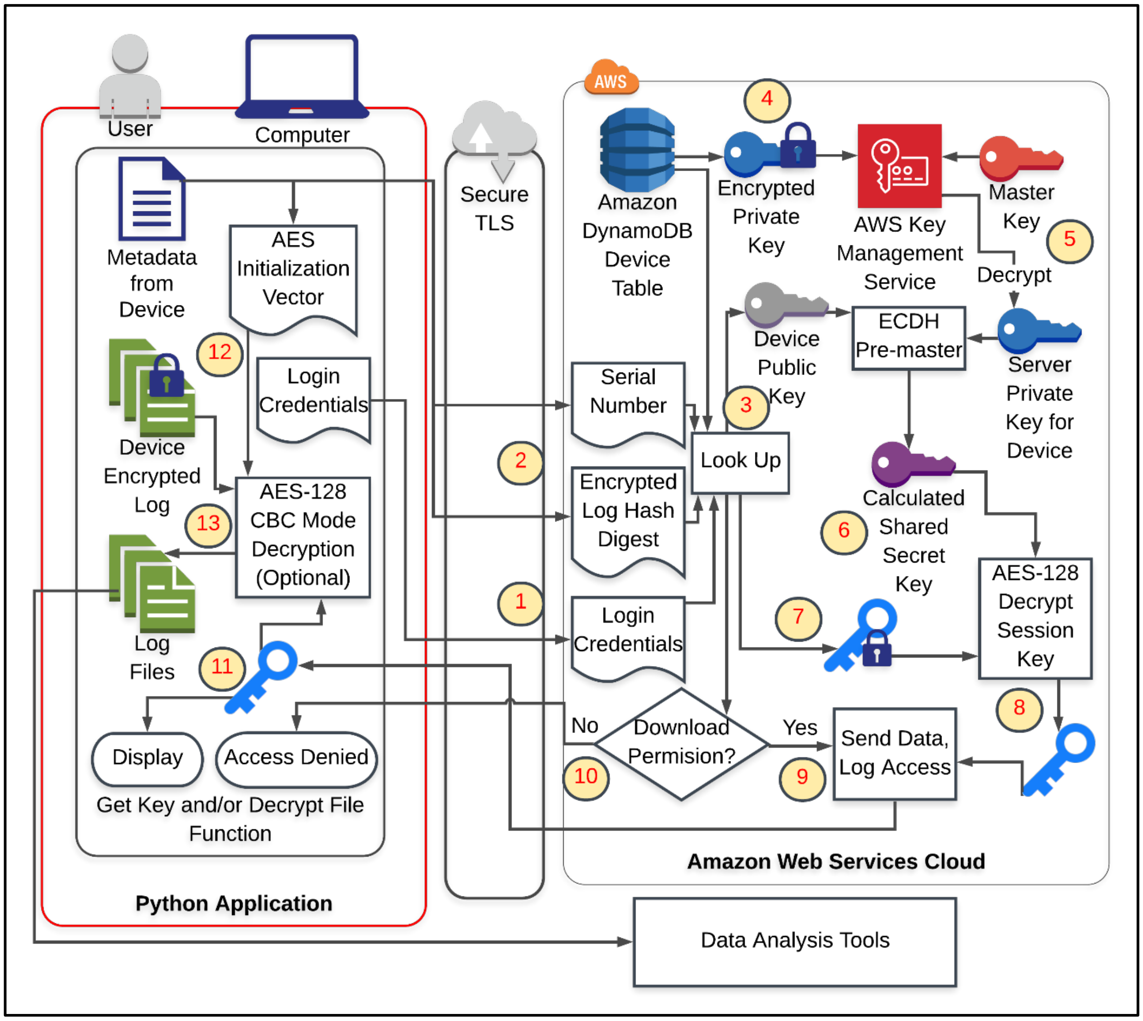


Figure 3-6. Get file AES session key and/or decrypt the log file diagram.

Table 3-4. Get file AES session key and/or decrypt the log file process

Process	System	Description
1	Local Computer	The user has to login to identify themselves if they have not done so. The login credentials are sent to AWS server via the Internet with secure TLS as a JSON web token (JWT).
2	Local Computer	The device serial number and the encrypted log file SHA-256 hash digest are sent to AWS server via the Internet with secure TLS.
3	AWS Cloud	The server uses the serial number and the log file hash digest to identify the corresponding file from the database and look up its associated device public key.
4	AWS Cloud	The server uses the serial number and the log file hash digest to identify the corresponding file from the database and look up the server encrypted private key for this CAN logger.
5	AWS Cloud	The AWS KMS uses the master key associated with the user account to decrypt the server private key
6	AWS Cloud	The shared secret key is derived from ECDH pre-master algorithm using the server private key and the device public key obtained earlier.
7	AWS Cloud	The server uses the serial number and the log file hash digest to identify the corresponding file from the database and look up the encrypted AES session key associated with the log file.
8	AWS Cloud	The 16-byte AES session key is decrypted with AES-128 ECB mode using the shared secret key.
9	AWS Cloud	The server uses the login credentials to determine if the user has permission to download the file. Only data owner, administrator, and users granted access by the owner have permission to download. If the user has permission, the AES session key in plaintext is sent back to the local computer Python application via the internet with secure TLS.
10	AWS Cloud	If the user does not have permission, the server responses with a message indicating that the user does not have permission to download the file.
11	Local Computer	The local computer Python application displays the AES session key in plaintext for the user.
12	Local Computer	If user wants to decrypt the log file with the key, the Python application first looks up the AES IV for that file.

13	Local Computer	The log file is decrypted with AES-128 CBC using the AES session key and the IV obtained earlier. The decrypted log file then can be extracted for data analysis.

iii. List and Download File Function While Connecting to the Server

Once the user has successfully uploaded all the desired encrypted log files and metadata files to the server, they can also download those files from the server through the local computer Python application without having the device connected. This process has two parts: connect to the server to list all the files that the user has permission to view and download, and download the chosen file. Figure 3-7 shows the stated process.

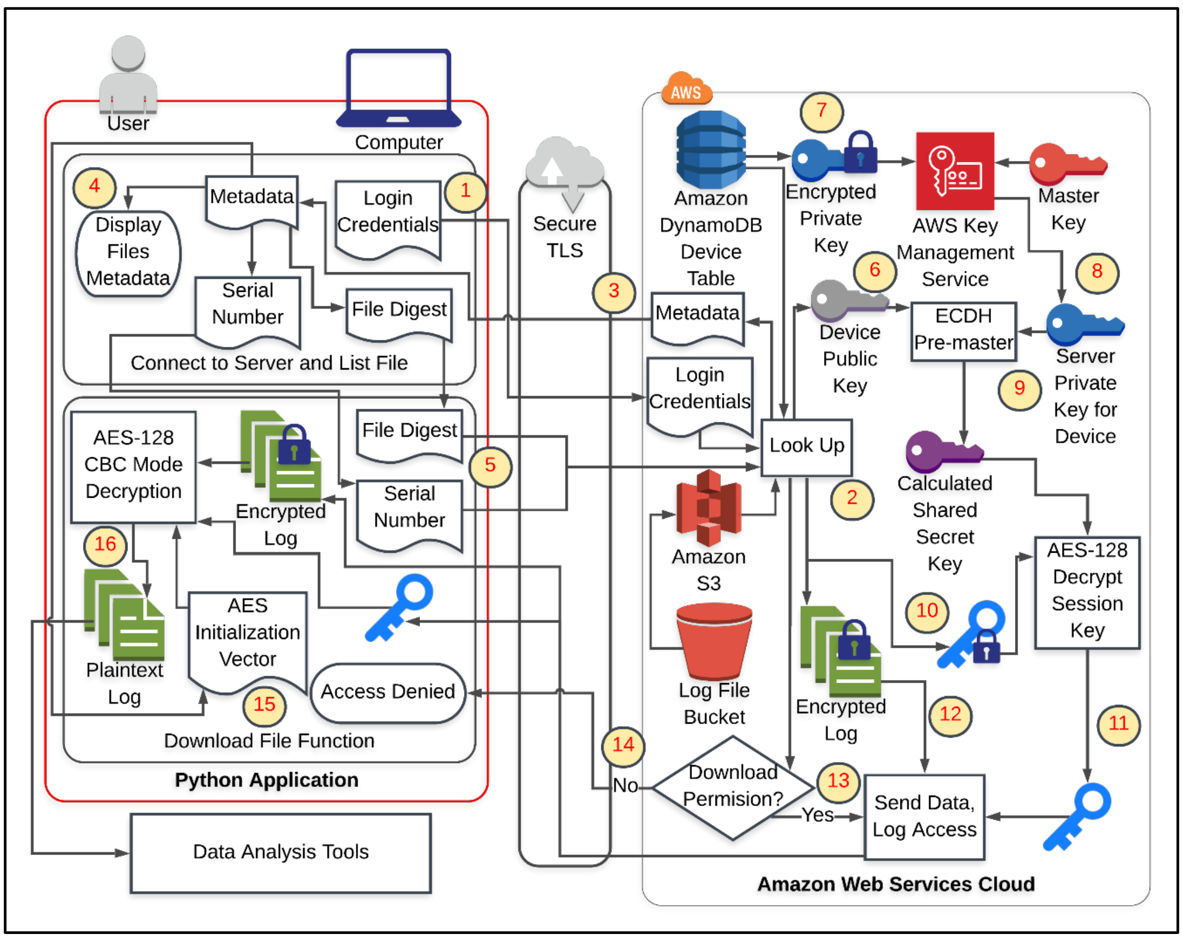


Figure 3-7. List and download file while connecting to the AWS server diagram

Table 3-5. List and Download File While Connecting to the AWS Server Process

Process	System	Description
1	Local Computer	The user has to login to identify themselves if they have not done so. The login credentials are sent to AWS server via the Internet with secure TLS.
2	AWS Cloud	The server uses the login credentials to look up all the files that the user has access to in the AWS DyanomoDB database.
3	AWS Cloud	The list of all those files along with their metadata are sent back to the local computer where they will be loaded into the Python application memory.
4	Local Computer	The list of all the files obtained from the server is displayed on the application.
5	Local Computer	The user can choose a specific file from the list to download. The device serial number and hash digest for that file are sent back to the server via the Internet with secure TLS.
6	AWS Cloud	The server uses the serial number and the encrypted log file hash digest to identify the corresponding file from the database and look up its associated device public key.
7	AWS Cloud	The server uses the serial number and the log file hash digest to identify the corresponding file from the database and look up the server encrypted private key for this CAN logger.
8	AWS Cloud	The AWS KMS uses the master key associated with the user account to decrypt the server private key.
9	AWS Cloud	The shared secret key is derived from ECDH pre-master algorithm using the server private key and the device public key obtained earlier.
10	AWS Cloud	The server uses the serial number and the log file hash digest to identify the corresponding file from the database and look up the encrypted AES session key associated with the log file.
11	AWS Cloud	The 16-byte AES session key is decrypted with AES-128 ECB mode using the shared secret key.
12	AWS Cloud	The log file hash digest to look up the encrypted log file binary from AWS S3 Bucket.
13	AWS Cloud	The server uses the login credentials to determine if the user has permission to download the file. If the user has permission, the AES session key in plaintext and the

		encrypted log file are sent back to the local computer Python application via the internet with secure TLS.
14	AWS Cloud	If use does not have permission, the server responses with a message indicating that the user does not have permission to download the file.
15	Local Computer	The AES IV for the encrypted file is retrieved from the metadata file obtained earlier.
16	Local Computer	The log file is decrypted with AES-128 CBC using the AES session key and the IV obtained earlier. The decrypted log file then can be extracted for data analysis.

D. Example Transcripts

i. CAN Logger 3

Serial Commands

Typically, the CAN Logger 3 works as a standalone device with only logging and sending request messages. However, it can also be operated with a computer where the user can run other different device functions with built-in serial commands, as shown in Figure 3-8.

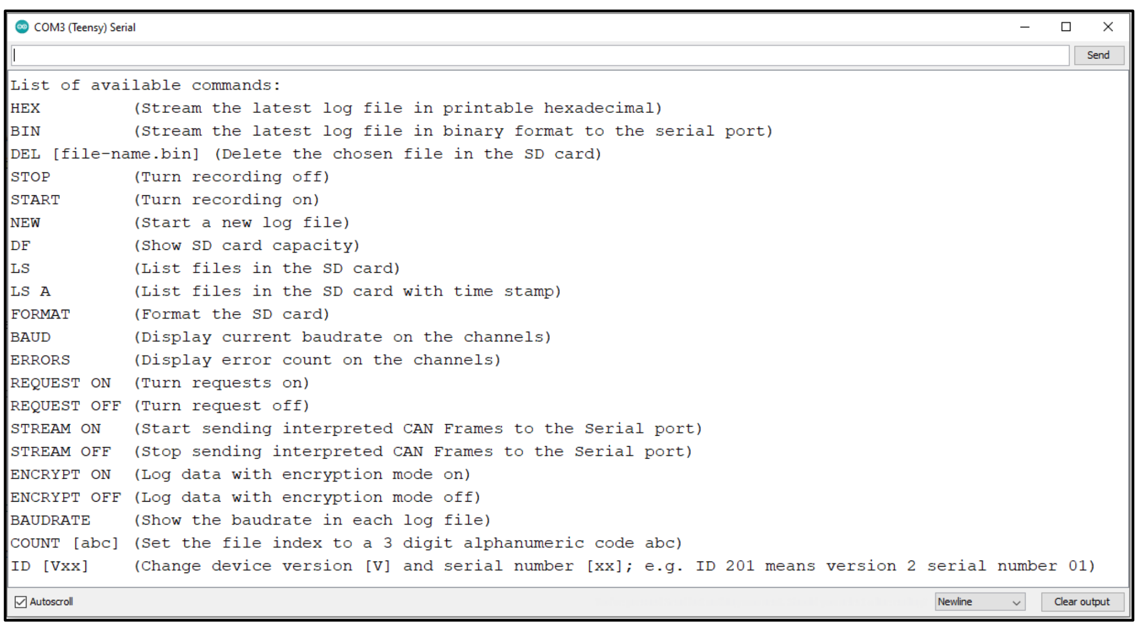


Figure 3-8. Built-in serial commands for the CAN Logger 3

These serial commands help the user to access the full functionality of the CAN Logger 3 right on the spot, such as managing files on the SD card, starting and stopping logging sessions, configuring the device identification, etc. Figure 3-9 shows an example of the STREAM ON serial command, which is commonly used during operation for CAN bus observation. This command streams CAN frames as seen on the bus to the serial monitor; however, it is an add-on feature and does not affect the main logging functionality. The columns from left to right in the Figure 3-9 are CAN channel, number of messages received, time in microseconds since the device began running, CAN frame ID, CAN frame extended ID specifier (0 is not extended ID and 1 is extended ID), a number of byte in CAN frame data field, and CAN frame data field.

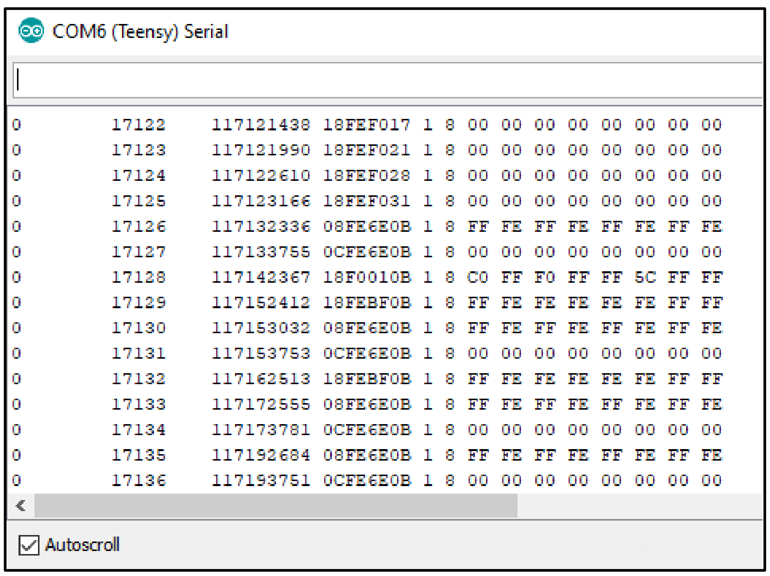


Figure 3-9. Streaming CAN frames on serial monitor

Operation with encrypted logging

By default, the CAN Logger 3 always encrypts heavy truck data during logging sessions, even when it operates as a standalone device or with a computer. When the device powers on, it goes through a routine setup before being able to start logging, such as checking the I2C

ATECC608A HSM connection, generating random AES session key and IV, setting RTC, etc. The green LED on the device should turn on after the setup finishes successfully. If the SD card is missing, the red LED on the device will flash until the card is inserted. Figure 3-10 displays the device information for user reference and debugging on the Arduino serial monitor after the initial setup.

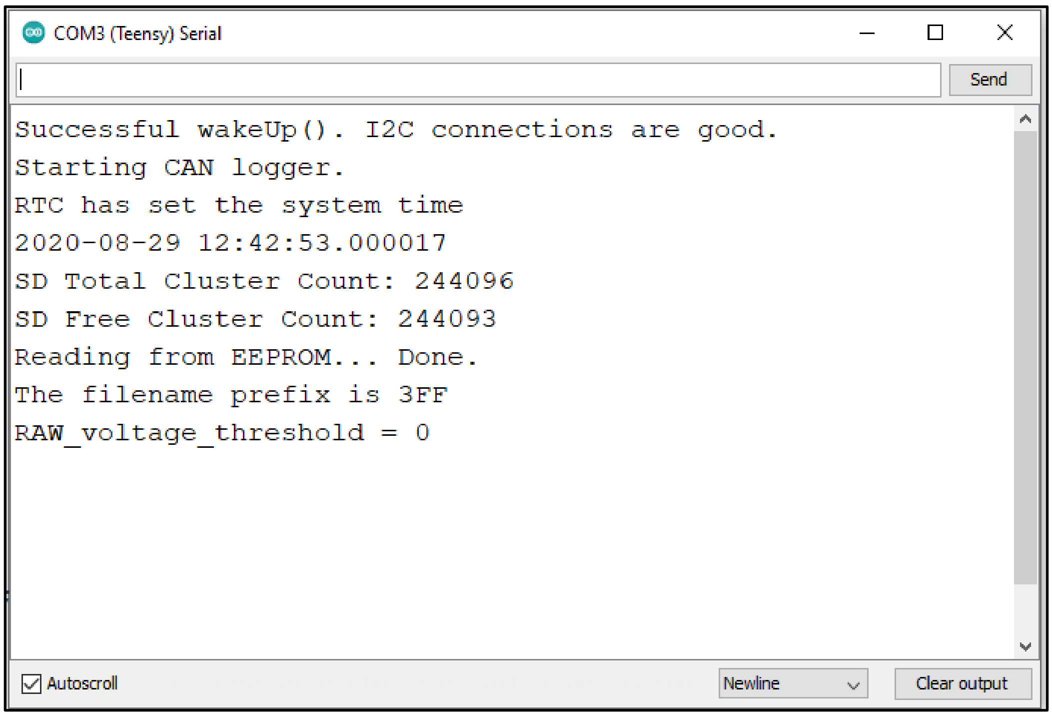


Figure 3-10. CAN logger startup information on Arduino serial monitor

The user can connect the CAN Logger 3 to the vehicle CAN bus via the diagnostic port to start the logging session. If there are messages present on the network, the device will automatically log the data to the SD card in the encrypted format, indicated by the green and yellow LEDs toggling. Figure 3-11 shows an example of an encrypted log file in hexadecimal with decoded text in ASCII, where the data looks random with no correlation or meaning.

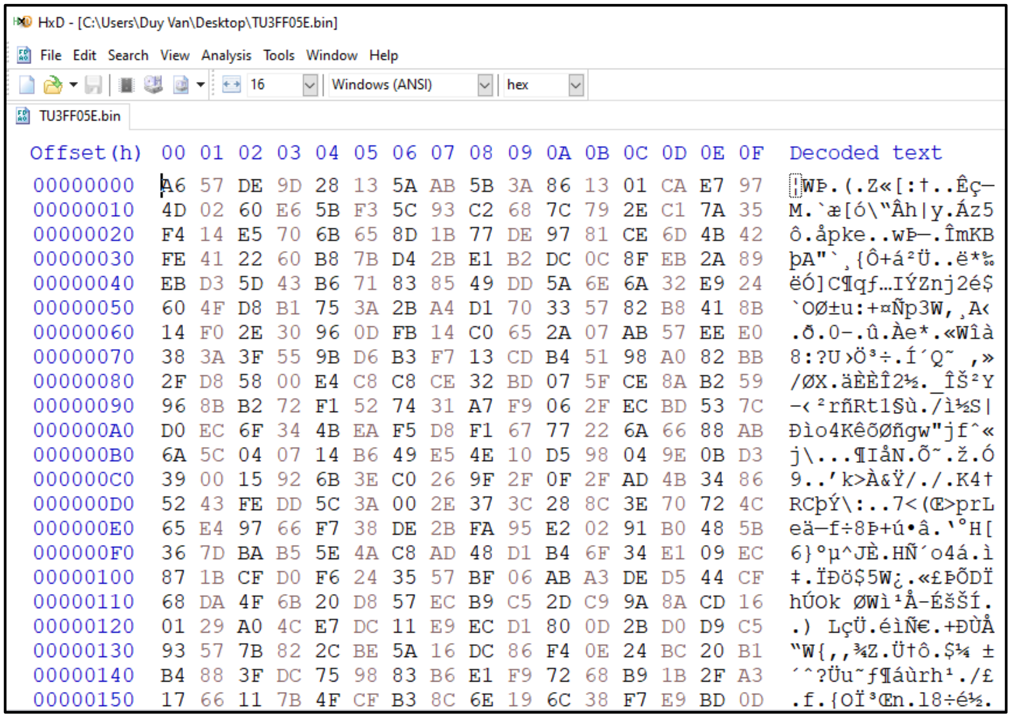


Figure 3-11. Encrypted log file in hex format

With the CAN logger as a standalone device, the user can double click the left button (the panel facing the user) to start a new log file. A single click on the same button will trigger the device to send request messages for vehicle identification. To stop the current logging session, the user can simply unplug the device from the diagnostic port or start a new log file. The user can also do all those steps with serial commands if the device is operated by a computer. The encrypted log files saved in the SD card can then be uploaded to the AWS server via the Python client application, which will be explained in the next section.

Operation with non-encrypted logging

There is an option to switch the device to non-encrypted logging mode for situations where the user only wants to have the plain log files right away for convenience without having to go through the uploading and decrypting process through the AWS server. To switch to non-encrypted logging mode, the user has to run the *ENCRYPT OFF* command on the serial monitor. The data will then be collected as is and written to the SD card. Figure 3-12 shows an example of a log file in plaintext where the data follows the 512-byte structure as described in the previous section (Chapter 3, part C.i. Data structure).

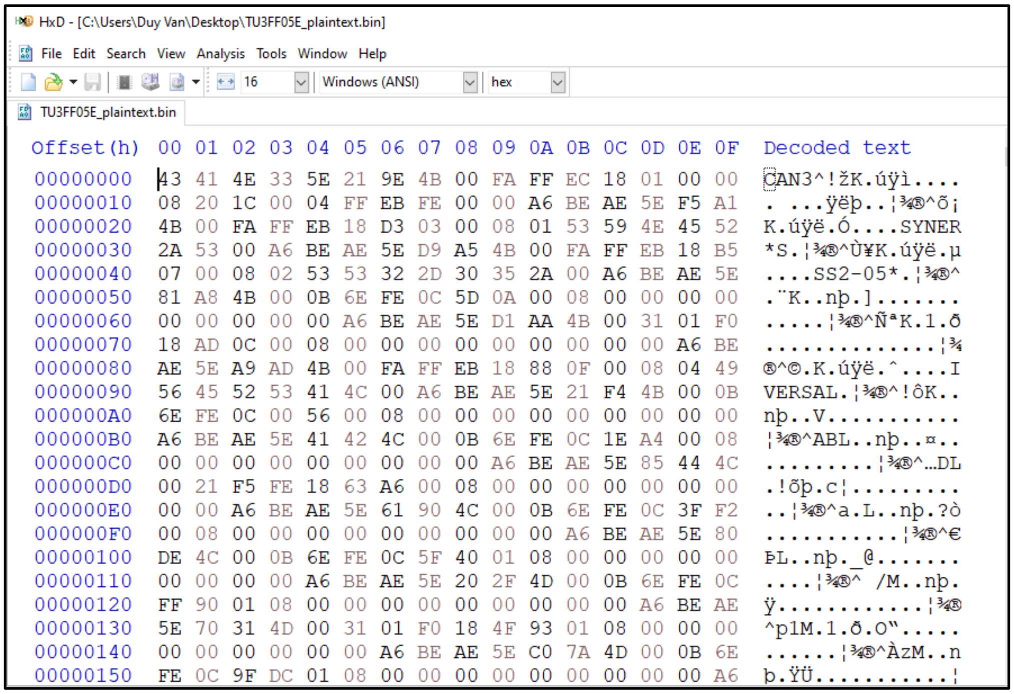


Figure 3-12. Non-encrypted log file in hex format

ii. Python Client Application

The CAN logger overview processes described above are implemented through a Python interface, which is controlled by the user. The source code can be found in [100]. The GUI is shown in Figure 3-13, and its functions are described in Figure 3-14. All functions are tested to show how they should respond and to ensure that the program works correctly, as anticipated in this section.

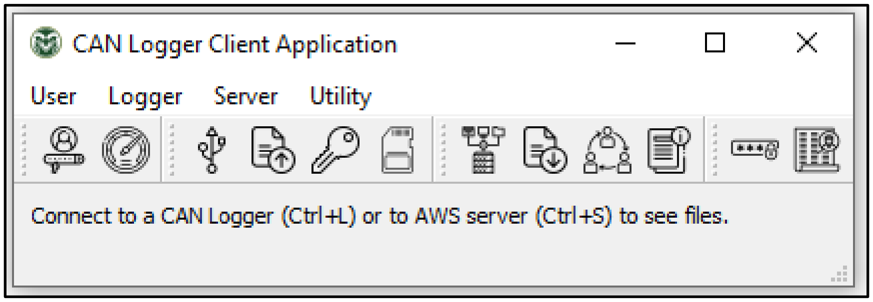


Figure 3-13. Local computer Python client application interface

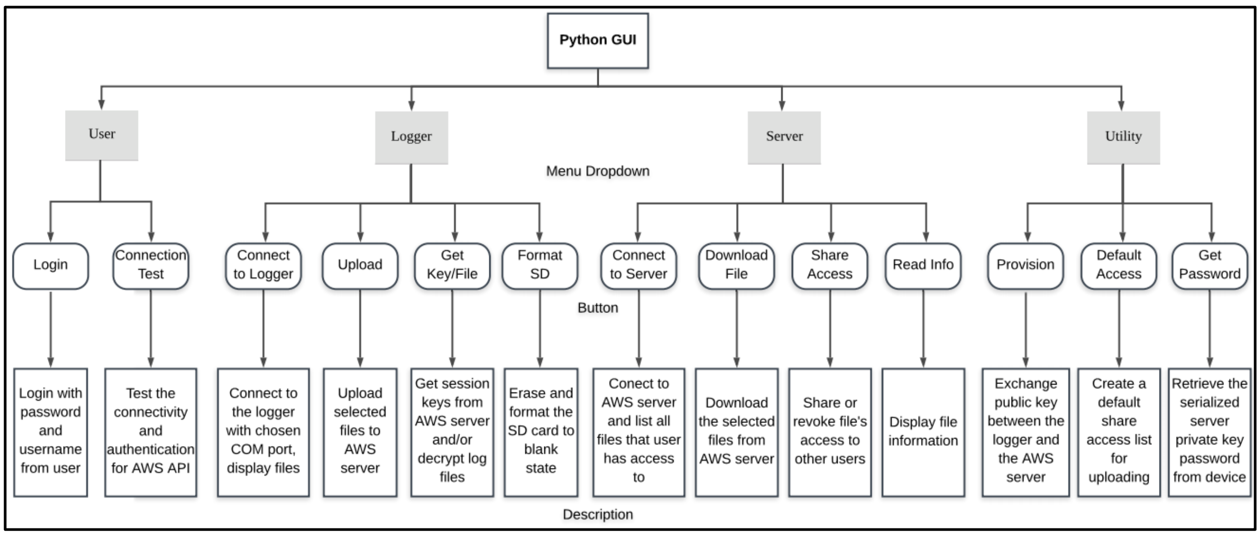


Figure 3-14. Description of the features offered within the application

User – Login

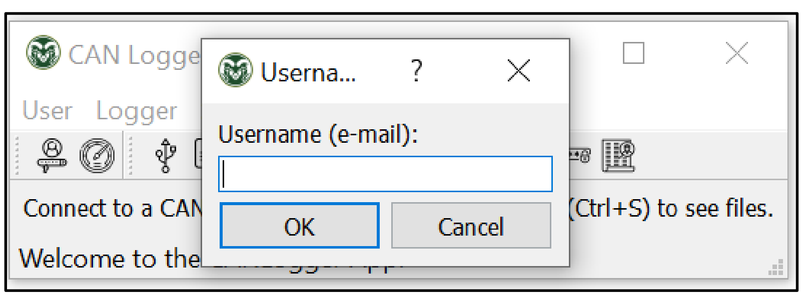


Figure 3-15. Login function interface

When the user first runs the application, it will automatically ask for a username and password with a dialog box, as seen in Figure 3-15. The account can be created on the CAN logging project website on [101], as seen in Figure 3-16. The user will be asked to verify their email during the registration to ensure the system is not flooded with invalid emails. After the user has successfully registered an account, the AWS User Pool will be updated with the registered account and ready for the authentication process in the Python application.

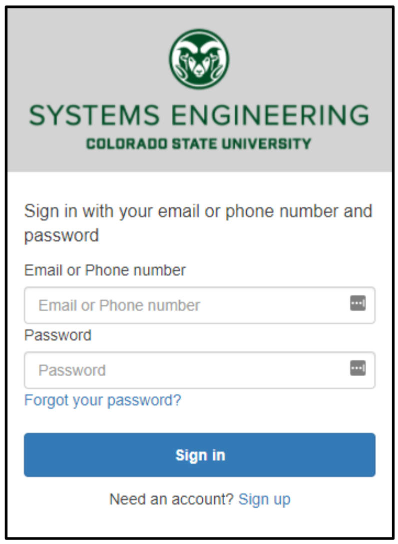


Figure 3-16. CAN logging project website

When the user enters their username and password, these credentials are submitted to the AWS User Pool, which returning the access token, ID token, and refresh token to the user. These tokens are used to authenticate against AWS API Gateway, and only when the authentication is successful, all the functions on the server as described in section B and C can be executed using Lambda. The tokens have a one-hour timeout, and therefore, login is required again after they expire. This login procedure follows the AWS guideline and recommendation [102], as seen in Figure 3-17.

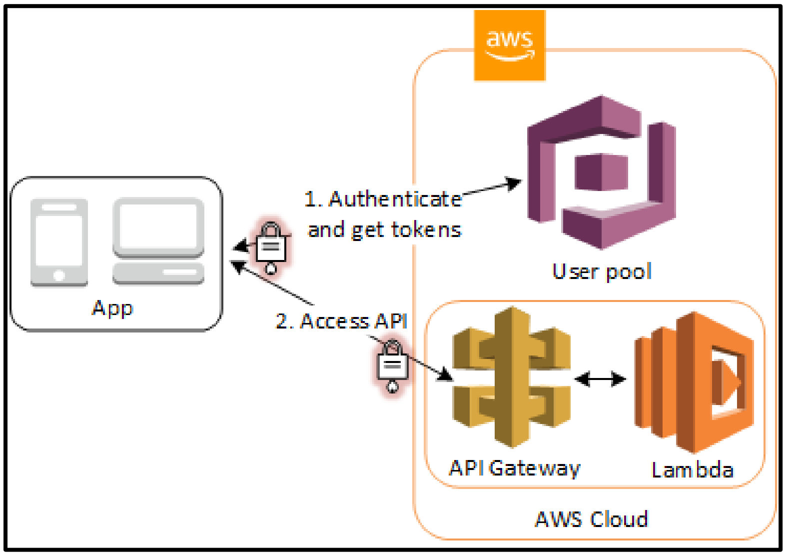


Figure 3-17. Access resources with API Gateway and Lambda with a User Pool diagram

The login process can also be executed manually by selecting *login* function under the User dropdown menu.

User – Connection Test

A quick way to check the communication between the application and the server is to do a *connection test* function under the User dropdown menu. If everything works as expected, the server will successfully respond with a code of 200, as shown in Figure 3-18. Otherwise, users will need to contact the administrators to resolve the problem.

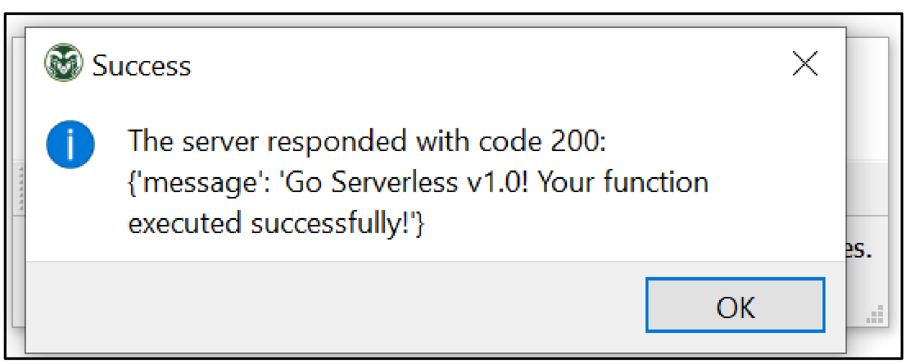


Figure 3-18. Connection test dialog

Logger – Connect to Logger

Once the clients have logged in, they can proceed to upload their files to AWS server for secure storage. After selecting the *connect to logger* function under the Logger dropdown menu, the application will retrieve and display all the log files along with their metadata from the SD card, as shown in Figure 3-19. Clients can inspect the data and proceed to upload function.

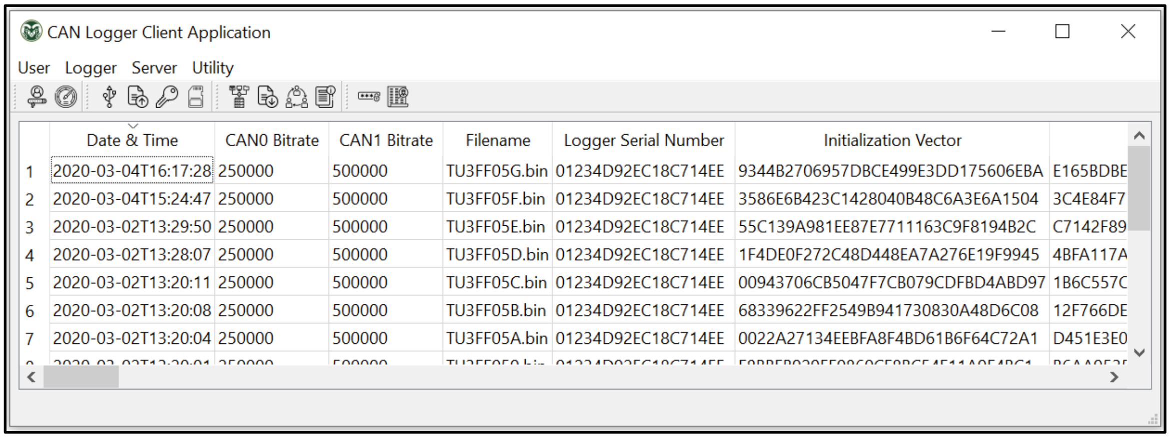


Figure 3-19. Client application connects to device and displays files on SD card

Logger – Upload File

The user can upload a chosen file by clicking on the row that contains the corresponding information on the application and selecting the *upload* function. A form for user input regarding log file information such as name, company, make, model, year, and a note will pop up, as shown in Figure 3-20. This tag helps clients to identify and distinguish different log files. At the end of the form, the user also has an option to add a default access list, which is created using the utility share access function. This step makes it convenient in case the user has to add access to multiple files with the same email list.

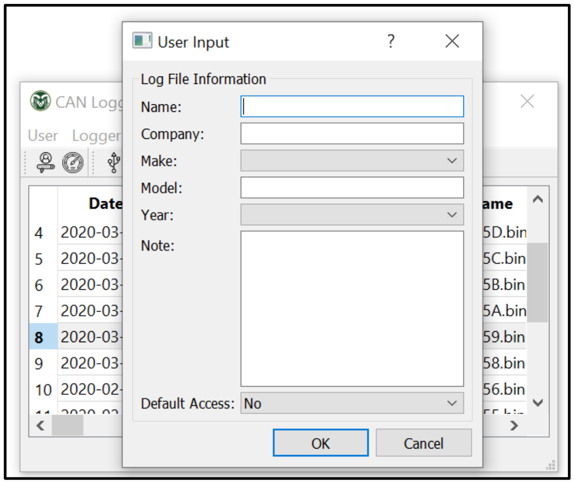


Figure 3-20. User input form layout for tagging log file before upload to server

After the user input is filled and submitted, the application will receive back a status code of 200, which means that the file is successfully uploaded to the server, and the file status will change to "verified".

However, if the server returns an error with a status code of 400, there are some possible reasons that cause failure in the uploading process:

- "Email not verified" – the login email has not been registered and needs to be verified.
- "Serial not found" – the logger has not been provisioned and/or its data has not been correctly stored in the database.
- "Public key from metadata does not match the one from server" – the public key stored in the metadata does not match the public key stored in the server, which means that the data has been compromised or the log file is not uploaded from its original device.
- "Metadata failed to verify" – the metadata file fails to verify its integrity, which means that the data has been compromised.

- “File is rejected due to invalid encrypted session key” – the encrypted session key is all 0x00 or 0xFF, which is invalid.
- “Hash Digest already exists or data is missing” – the file is already uploaded or missing data.
- “Log file cannot be found in s3 Bucket” – there is an error uploading the file to s3 storage, and the server could not locate the file.
- “Log file hash does not match” – the hash of the log file uploaded to AWS S3 storage does not match the expected hash from the metadata, which means that the content of the log file in S3 has been compromised.

Logger – Get Key/Decrypt File

After the file is uploaded successfully to the server, the users can obtain the AES session key from the server to decrypt the file. With the application still displays all the available log files from the connected device, clients can select the desired file and choose *get key/decrypt file* function under the Logger dropdown menu. The server will respond with the AES session key if successful, as seen in Figure 3-21. The user can copy the key for themselves or use it in the application to decrypt the file. If clients choose to decrypt the file with the application, it will decrypt then save the file locally, as shown in Figure 3-22. The user now has the decrypted version of the log file for further usage.

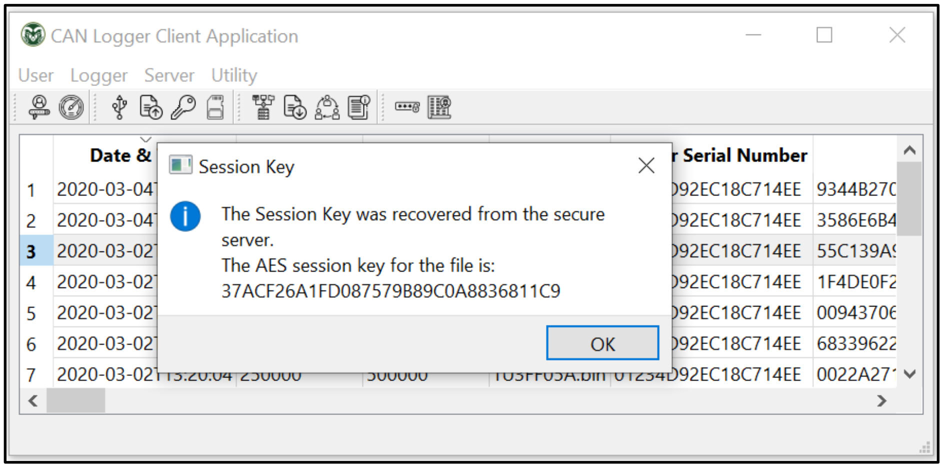


Figure 3-21. An AES session key in plaintext is retrieved from the server using *get key/decrypt file* function

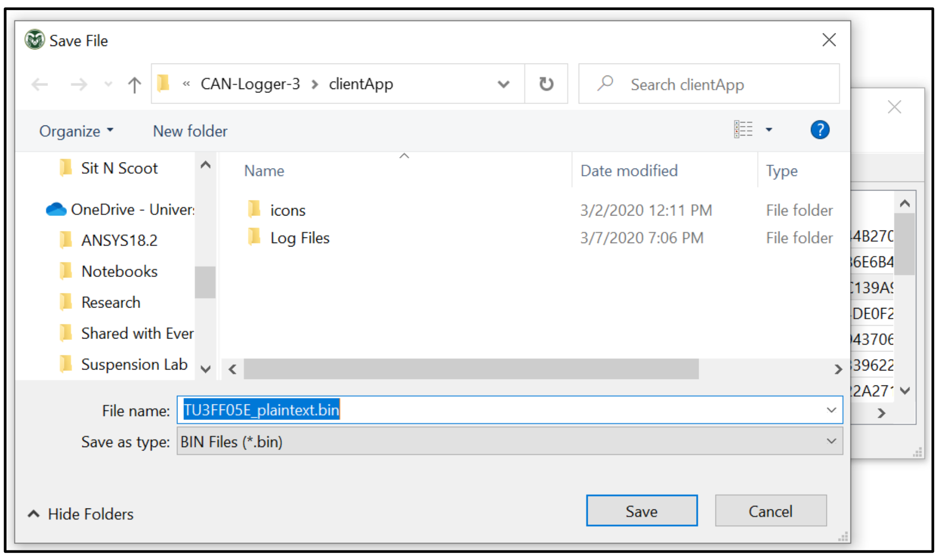


Figure 3-22. Saving the decrypted log file to local computer

Figure 3-23 shows the decrypted content of the downloaded file from the server using the *get key/decrypt file* function. Because the log file follows the 512-byte data structure, the data is displayed with 512 hex bytes per line for easy observation. The content has meaning decoded text, which means the file has been correctly decrypted.

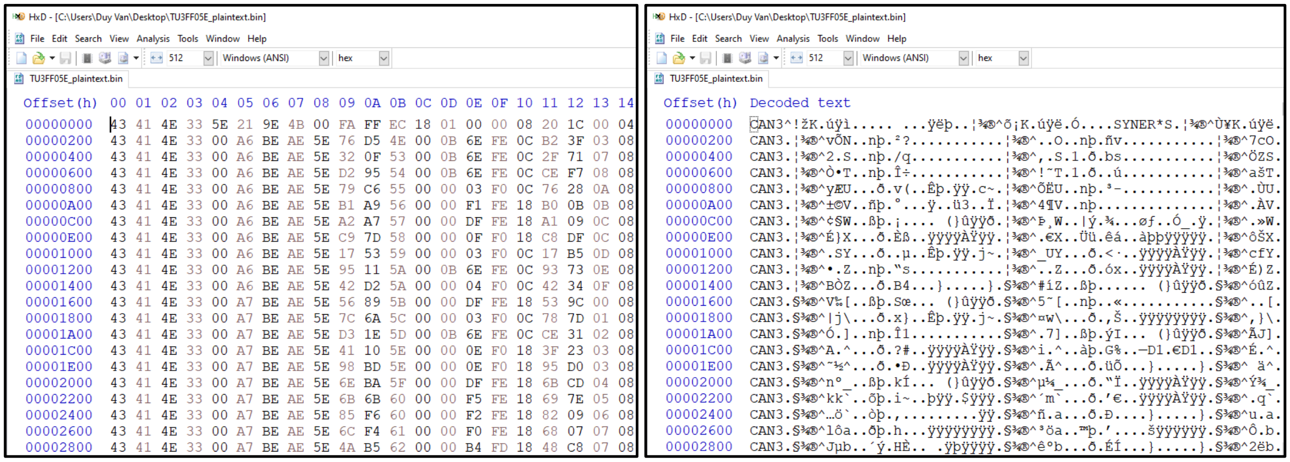


Figure 3-23. A decrypted log file downloaded from the server

in hexadecimal (left) with ASCII decoding (right)

Logger – Format SD Card

The user has an option to format the SD card while the device is being connected to the local computer Python application. This can be done by selected the *format SD* function under the Logger dropdown menu. A window will pop up to confirm, as shown in Figure 3-24. The application will return a message indicating the SD card has been successfully formatted.

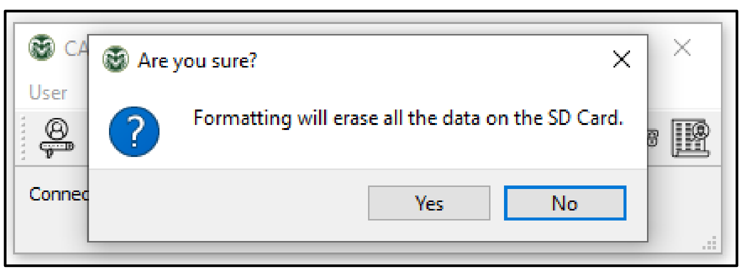


Figure 3-24. Formatting SD card to blank state

Server – Connect to Server and List Files

The user can select the *connect to server* function under the Server dropdown menu to view all the uploaded files or files that clients have shared access to, as shown in Figure 3-25.

This function does not require the user to have their device connected because the server looks up the all the files that have been uploaded by the user or files that the user has been granted access to from the database and return their metadata information. The application will then display all the data for the user's view.

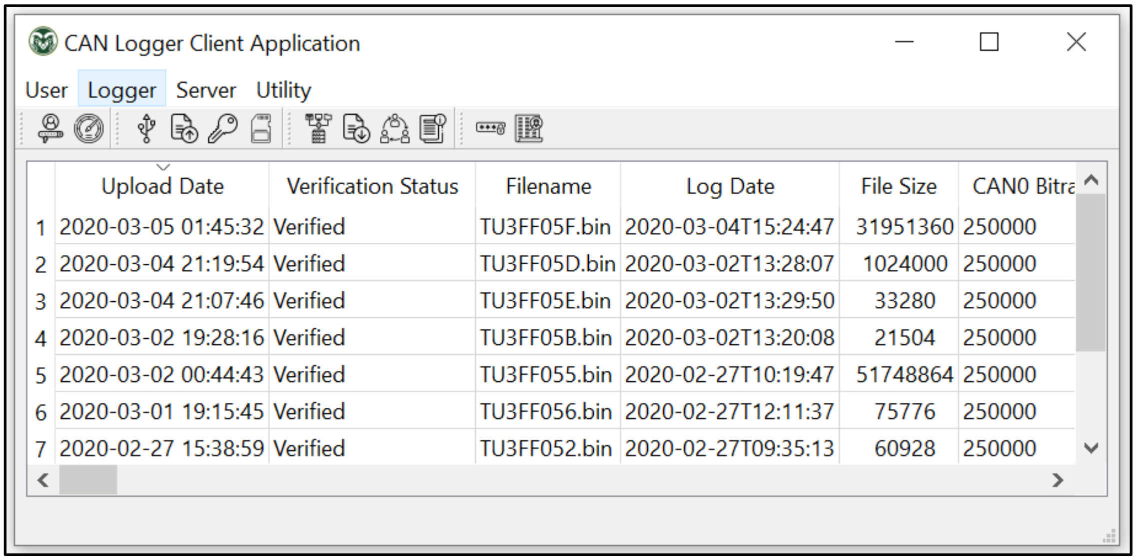


Figure 3-25. Connecting to server and listing all files

If the server returns an error with a status code of 400, there are some possible reasons that cause failure in retrieving the data:

- "Email not verified" – the login email has not been registered and needs to be verified.
- "Unable to retrieve table item" – there is an error while scanning the database.

Server – Download File

With the list of available log files displayed on the application, the user can click on the desired file and select the *download file* function under the Server dropdown menu. If there is no error occurs, the server will return the encrypted binary of the request log file. The application will ask the user if they want to save the encrypted or plaintext version of the log file, as seen in

Figure 3-26. If the user chooses to save the encrypted version, the application will decrypt the file locally.

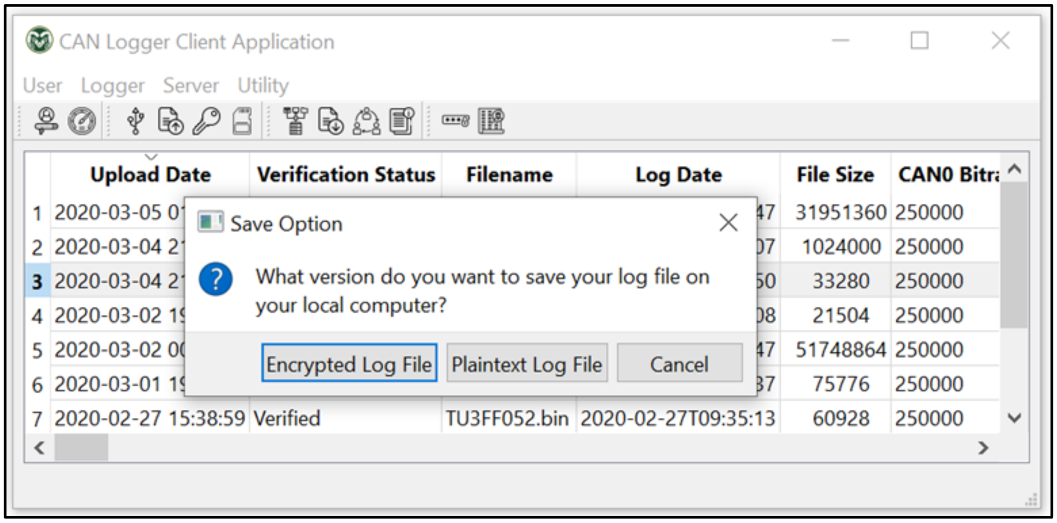


Figure 3-26. Downloading a log file from server to local computer

If the server returns an error with a status code of 400, there are some possible reasons that cause failure in retrieving the data:

- "Missing required parameters" – some data sent from the application are missing.
- "Email not verified" – the login email has not been registered and needs to be verified.
- "Unable to retrieve serial number from table" – there is an error while scanning the database.
- "Data Key is Not Available" – there is an error while decrypting server private key, which means that the logger that the requested file belongs to has not been provisioned and/or its data has not been correctly stored in the database.
- "File Meta data not available. Please upload file!" – the file metadata has not been uploaded successfully to the database.

- "You do not have permission to download this file" – the file does not belong to clients or they don’t have share access.
- “Log file cannot be found in s3 Bucket” – the requested file is not in the AWS s3 storage, which means the log file was not successfully uploaded before.

Server – Share Access

With all the files listed on the application after connecting to the server, the user has an option to share or revoke access of a file to other users. This can be done by clicking on a desired log file on the application and selecting the *share access* function under the Server dropdown menu. A window will pop up, asking if the user wants to share or revoke access. Depending on the needs, the user will choose appropriately and enter the user email they want to share or revoke. The function only takes in one email input at a time, which is shown in Figure 3-27. The server will return a status code of 200 upon successful execution, with a message of “{email input} has been added to / revoked from the access list.”

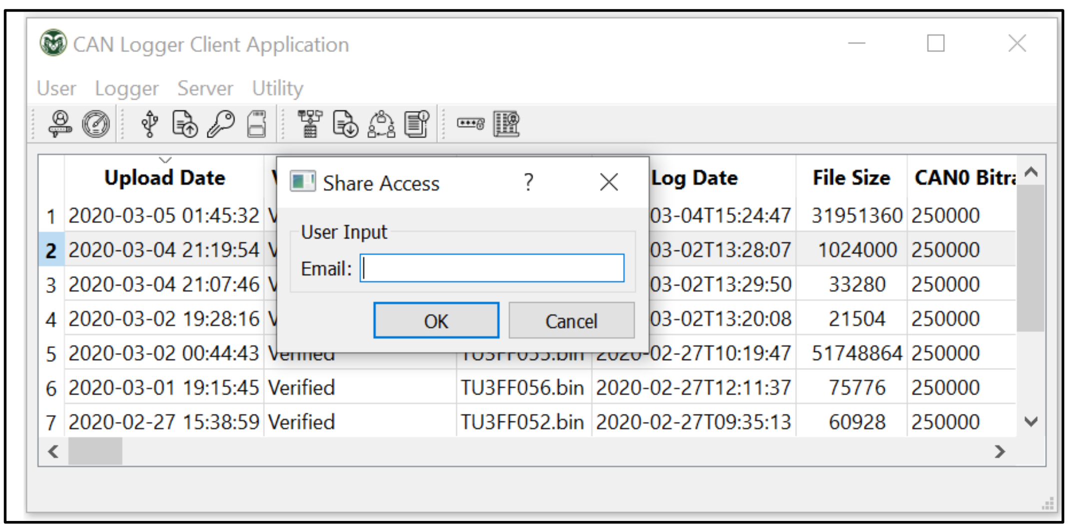


Figure 3-27. User input for file access sharing or revoking

If the server returns an error with a status code of 400, there are some possible reasons that cause failure in editing the access list:

- “Missing required parameters” – some data sent from the application are missing.
- “Email not verified” – the login email has not been registered and needs to be verified.
- “File digest not found” – the server could not locate the file, or the file has not been uploaded correctly.
- “You do not have permission to share or revoke access to the selected file” – only the owner of the file can share or revoke access.
- “There is no {email input} in access list to revoke access” – the email to revoke does not exist in the access list.

Server – Read File Info

If the user wants to view the information of a file quickly, they can either double click on the desired file or highlight the file and select the *read file info* function under the Server dropdown menu. This function displays all parameters of the file, such as uploader, share access list, name, company, make, model, year, note, and download log, as seen in Figure 3-28.

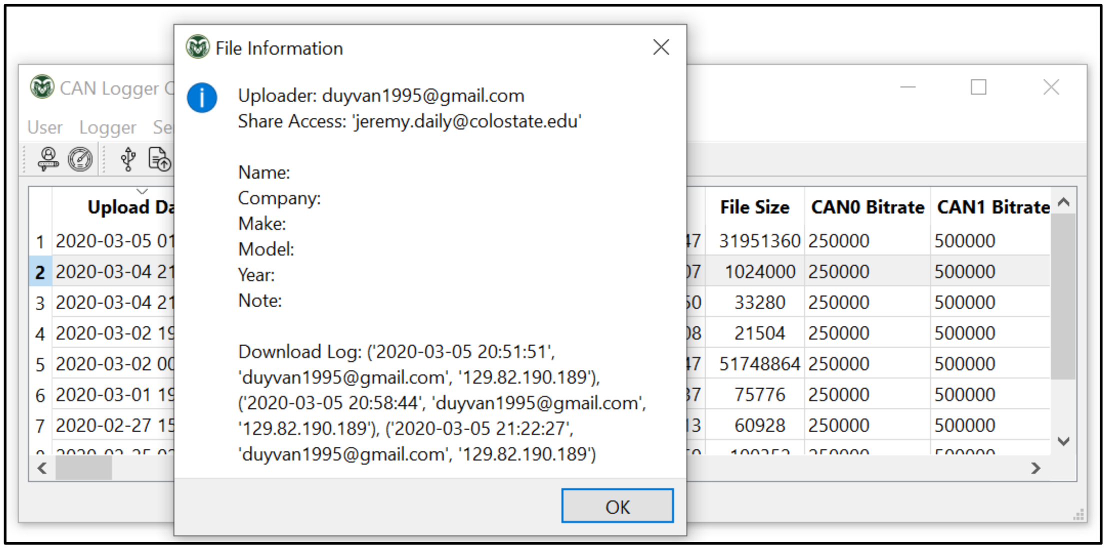


Figure 3-28. Reading file information

Utility – Provision

This function is only used during the provisioning process by the operator or administrator to exchange the public keys between the logger and the server. The device must have the provisioning firmware [103] before proceeding further. The operator will need to have the device connected to the application and select the *provision* function under the Utility dropdown menu. After the server successfully creates its ECC key pair and obtains the device public key, the server public key will be sent back to the application. At this time, the operator will perform a visual key confirmation and compare the server public key and device public key from the application to those from the server on the AWS database website. For easier visualization, the public keys are hashed, and only the first 10 bytes are displayed, as seen in Figure 3-29. This process ensures that the server and the device have each other's correct public key during the key exchange process.

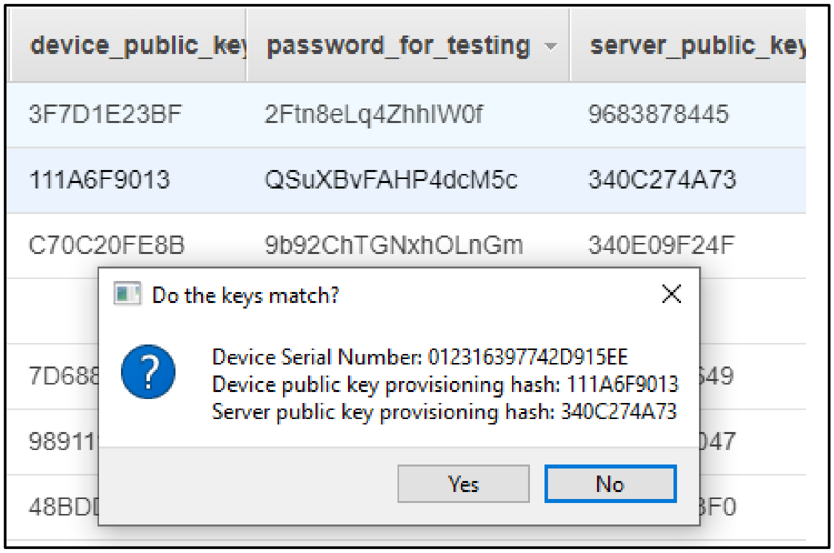


Figure 3-29. Visual key confirmation between the client application and AWS server

Once the operator confirms that the keys match, the application will send the server public key to the device where it will be stored and locked correctly. If keys do not match, then there could be an error during transmission, or the communication has been compromised. The operator will have to delete the device information in the AWS database and restart the provisioning process.

In addition, after the device is successfully provisioned, the application will update a security backup list with the serialized server private key and the encrypted password associated with the provisioned device, as seen in Figure 3-30. This list is made as a physical backup to calculate the ECDH shared secret for decrypting the AES session key locally in case the AWS server is offline, as mentioned in the previous section (Chapter 3, part B.i.). However, the security backup list cannot be used until the encrypted password for the server private key is decrypted. The *get password* function in the below section will need to be executed to obtain the password in plaintext.

Figure 3-30. Security backup list example

Utility – Get Password

This function decrypts the password that is used to load the serialized server private key into the Python program for ECDH pre-master calculation. With the current firmware from the provisioning process, the operator can execute the *get password* function under the Utility dropdown menu. The application will require the operator to select the security backup list, and the password in plaintext will be returned upon successful execution, as shown in Figure 3-31. The password in plaintext can then be recorded into a different list. Due to the important confidentiality, the security backup list and the password list will be stored in a flash drive and mailed to the administrators.

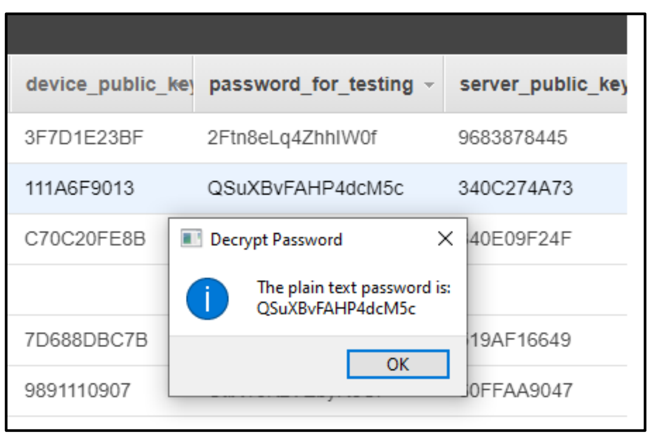


Figure 3-31. Decrypting password for the encrypted server private key

Utility – Default Access

As mentioned above, there is an option to add a default access list in the upload function. The list can be made using the *default access* function under the Utility dropdown menu. A dialog will appear for clients to input all the emails that they want to share access to, as seen in Figure 3-32. Clients have to input one email with no comma or space per line. A CSV file with the input contents will be made and saved on the local computer. When clients decide to use the default access list while uploading a file, they can select this CSV file for convenience and time-saving.

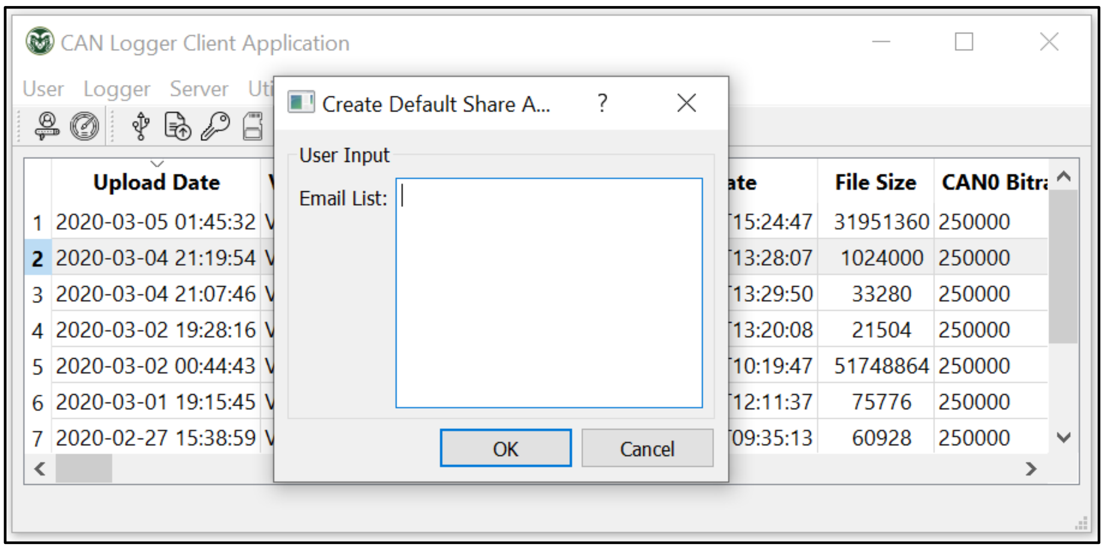


Figure 3-32. Creating default access list for file uploading process

iii. Cloud Backend

The server backend can be accessed through the AWS management console website. There are some important AWS services used for this project: DynamoDB, Simple Storage Service (S3), Identity and Access Management (IAM), CloudWatch, Cognito, and KMS. Clients won't have permission to view or edit because these services are only for administrative use, except for S3, where users can manage their uploaded data. The structure of the CAN logger project is designed in a way that the sensitive information regarding the CAN loggers and their associated log files within DyanmoDB and S3 are encrypted. If AWS is exposed, the confidentiality of the loggers and the log files are still protected.

DynamoDB

DynamoDB is a no-SQL database, which contains two tables that store important data for the CAN logging project. The CANLoggers table contains the device's unique information that is registered after the provisioning process, as seen in Figure 3-33. The table contains the following parameters:

- "id" – device serial number.
- "device_public_key" – the public key of the registered logger.
- "email" – the operator's account used during the provisioning process.
- "encrypted_data_key" – encrypted customer master key.
- "encrypted_server_pem_key" – encrypted server private key.
- "sourceIp" – the IP address of the provisioning operator.
- "device_public_key_hash" – the first 10 bytes of the device public key SHA-256 hash.
- "server_public_key_hash" – the first 10 bytes of the server public key SHA-256 hash.

These parameters are usually updated or deleted during the provisioning process if an error occurs or when the device has been revoked.

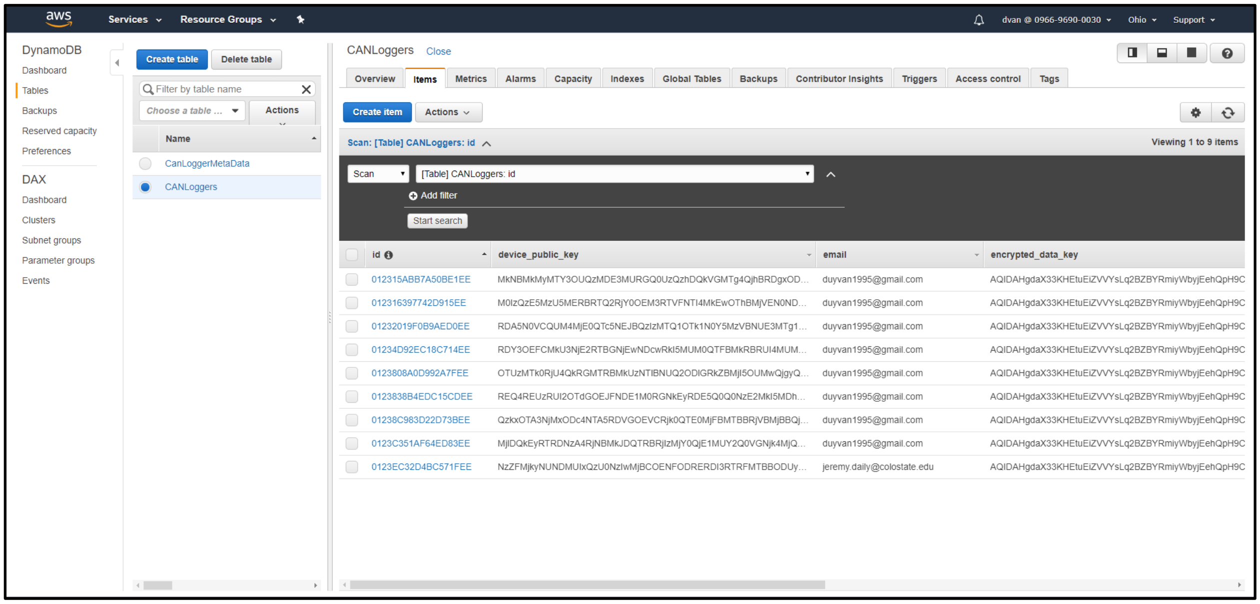


Figure 3-33. AWS DynamoDB CANLoggers database

The second table is CanLoggerMetaData, which contains metadata of uploaded log files, as shown in Figure 3-34. The table contains the following parameters:

- “digest” – the SHA-256 hash of the log file.
- “CAN0/CAN1” – CAN bitrate of the log file.
- “access_list” – the emails who have shared access to the file.
- “datetime” – the time when the log file was created.
- “download_log” – the log consisted of the time, and the user’s IP address and email when the file is accessed and downloaded.
- “filename” – the name of the file.
- “filesize” – the size of the file.
- “init_vect” – the AES initialization vector.

- “meta_data” – the user input note from the file uploading.
- “serial_num” – the serial number of the device.
- “session_key” – the encrypted AES session key.
- “signature” – the signature of the metadata text file.
- “text_sha_digest” – the SHA-256 hash digest of the metadata text file.
- “upload_date” – the date when the file was uploaded.
- “uploader” – the email of the uploader.
- “verify_status” – the integrity status of the log file.

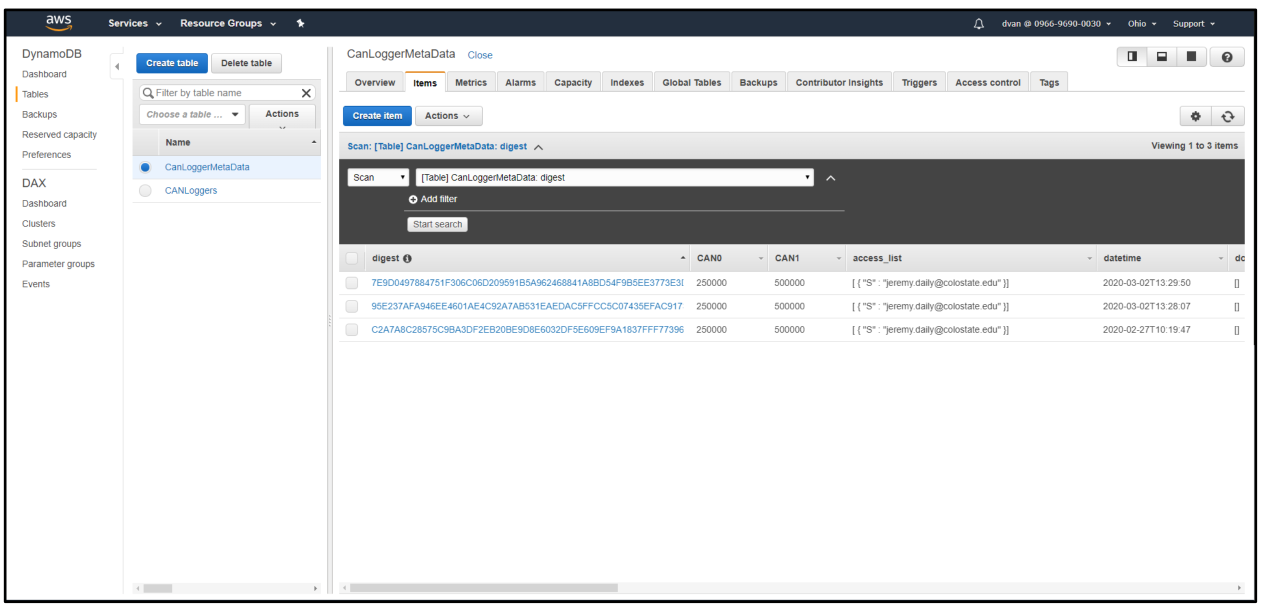


Figure 3-34. AWS DynamoDB CanLoggerMetaData database

S3 Bucket

S3 is used to store all the log files is called can-log-files bucket, as shown in Figure 3-35. On the website, the following parameters can be seen:

- “Name” – the SHA-256 digest of the file.
- “Last Modified” – the time when the file was uploaded or last modified.
- “Size” – the size of the log file.
- “Storage class” – the class of the storage.

Administrators and users can view and edit these files, such as changing parameter values or delete files as needed. Currently, users can access all uploaded files, including files from other users. For future work, access control needs to be added so that users can only view or edit their own data or data with access permission from others. However, the Python client application and the CAN logger project website do have access control. Users can view their data on both options, but they can only download their log files through the client application using pre-signed URLs.

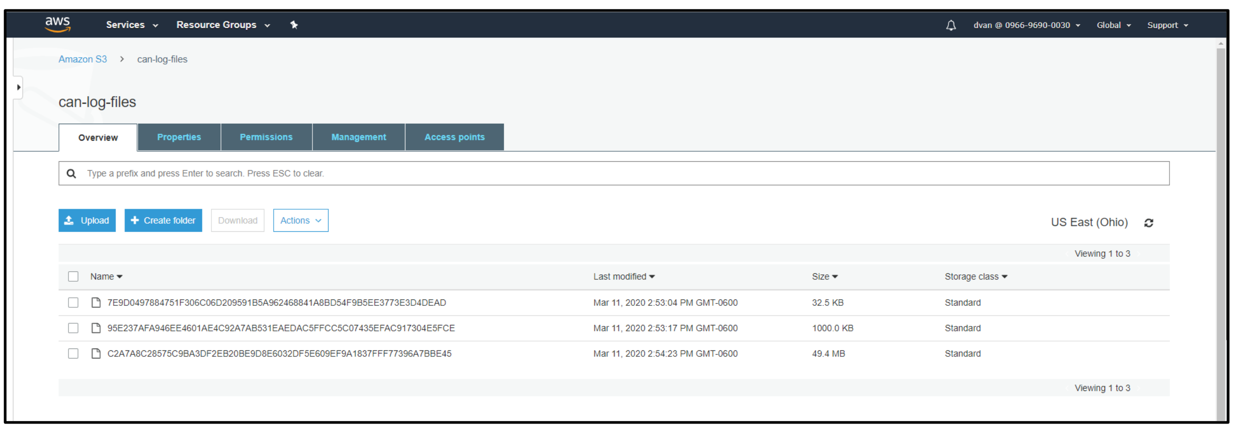


Figure 3-35. Encrypted log files stored on AWS S3

IAM

IAM helps manage user access to different AWS services and resources securely. Different users and groups can be established using IAM, where different permission levels can be specified for each of them. Therefore, IAM is the key used to separate administrators from normal users. Moreover, an additional layer of protection for the administrator accounts can be added using multi-factor authentication within IAM. Figure 3-36 illustrates a typical IAM main page.

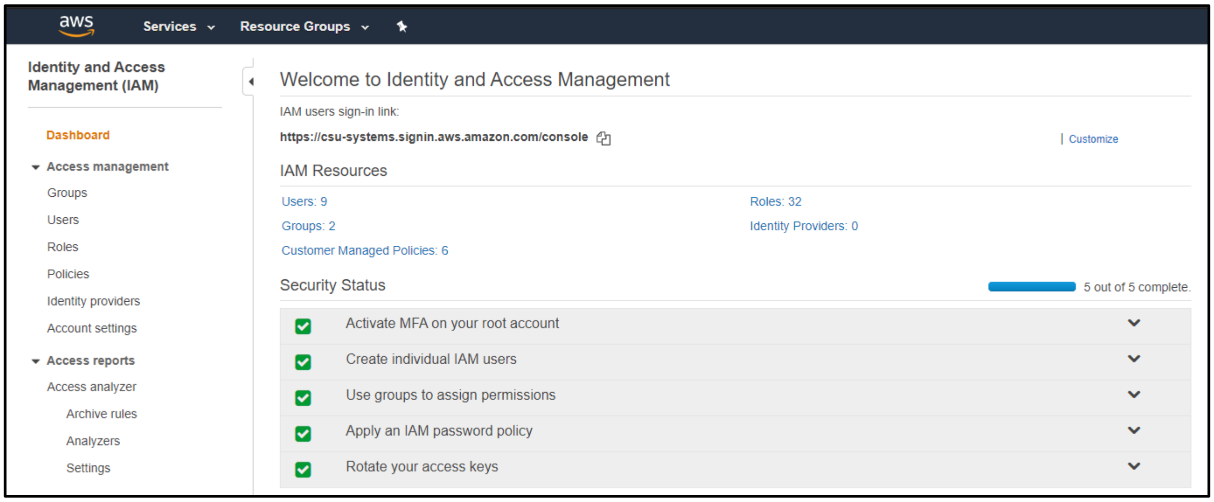


Figure 3-36. AWS IAM main page

Cognito

Cognito is used for identity management. An AWS User Pool provides user account registration and authenticates users. As described in the previous section (Chapter 3, part D.ii. User – Login), the account registration takes place on the CAN Logger project website. The AWS User Pool then updates its database with a successful registered account, as seen in Figure 3-37. The information in the User Pool is then used for login authentication where users get tokens to access API Gateway and Lambda.

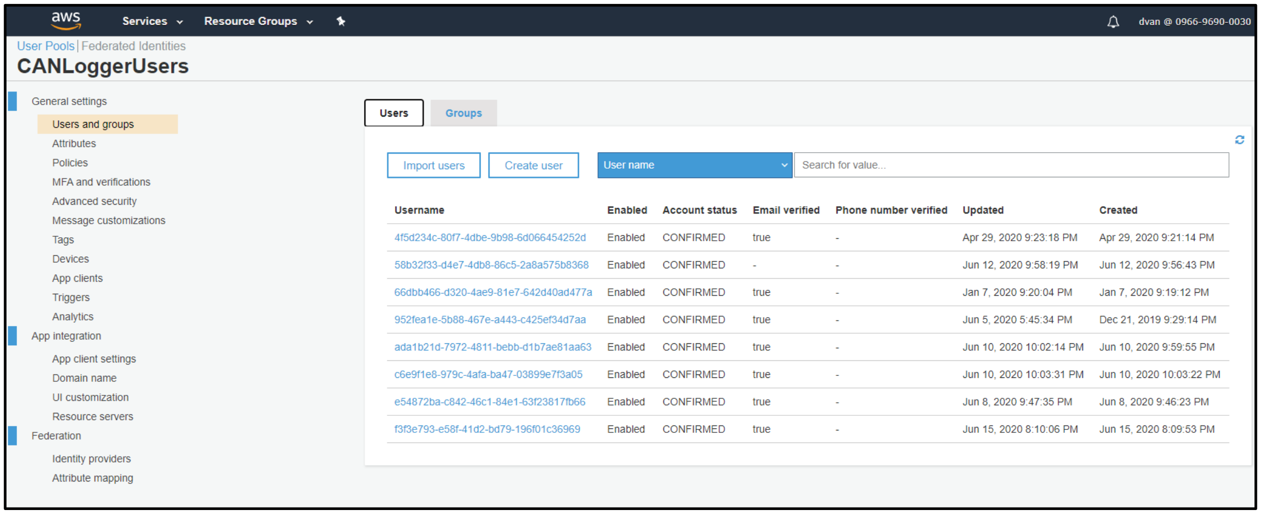


Figure 3-37. AWS Cognito User Pool main page

CloudWatch

CloudWatch is a powerful AWS monitoring and observability service. For the scope of the project, CloudWatch is mainly used for debugging code and logging all the operating functions as described in the client application section. The CloudWatch interface on the website can be seen in Figure 3-38. There are eight functions used that are shown:

- "auth" – used for getting AES session key from the server.
- "download" – used for downloading a file from server.
- "hello" – used for testing the connection.
- "list" – used for connecting to server and listing files.
- "provision" – used for the provisioning process.
- "share" – used for sharing or revoking access.
- "upload" – used for uploading files to the server.

- "verify_upload" – used for verifying the integrity of the log file after being uploaded to the server.

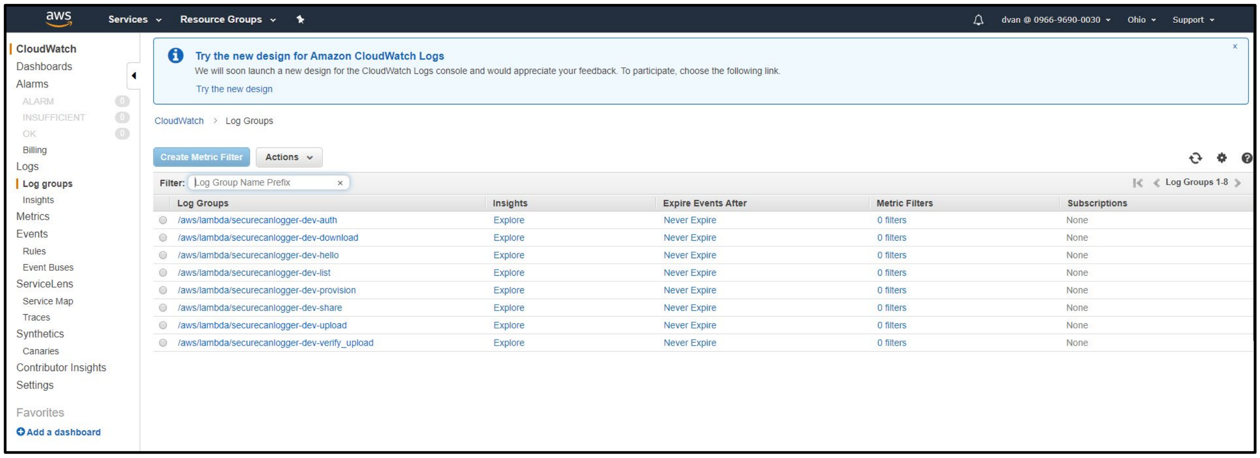


Figure 3-38. AWS CloudWatch main page

Key Management Service (KMS)

KMS is a service that securely creates and manages cryptographic keys across different AWS services and resources. These keys are stored in AWS hardware security module (HSM), where they are highly safeguarded and tamper-resistant. The CAN Logger project uses one KMS master key to secure secrets and private keys, as seen in Figure 3-2, 3-6, and 3-7 previously. The main page of the AWS KMS is illustrated in Figure 3-39.

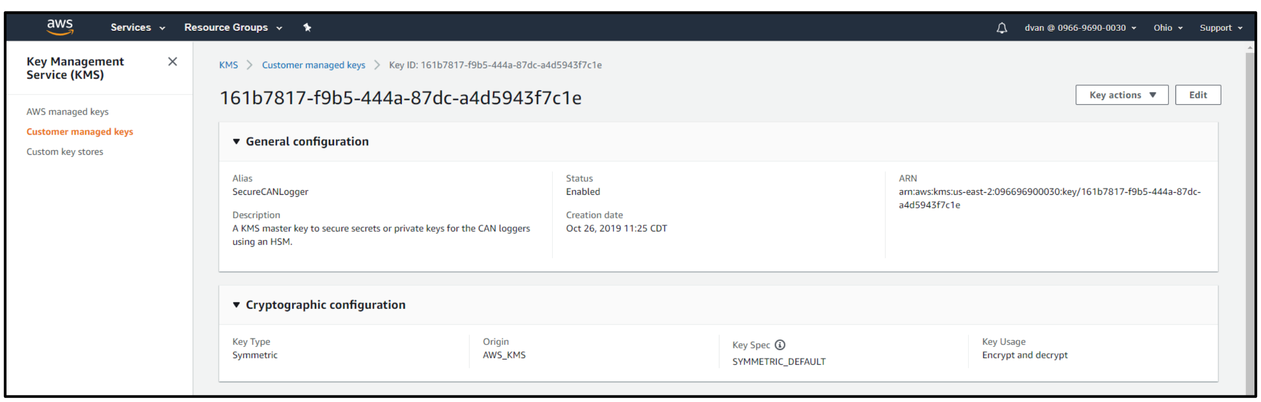


Figure 3-39. AWS KMS main page

E. Chapter Summary

Secure end-to-end communication between vehicles and their data management services is vital when confidentiality and integrity are important factors in the processes of data monitoring and collection. In a typical heavy truck model, OEMs are not required to design a built-in data monitoring and management system for the customers. However, due to the horizontal integration design, this can be done mostly by telematics companies or third-party devices that involve a cloud IoT platform. Secure end-to-end communication may or may not be implemented by these third-party service providers. However, if they do implement it, their process is likely to be proprietary and the customers have to trust their implementation.

This chapter describes, in detail, the CAN Logger 3 software design that provides a secure end-to-end data transmission between the vehicles to the AWS cloud platform with the Python client application as a user supporting interface. There is no one unique way to implement a secure end-to-end communication, but this project uses off-the-shelf products as well as industry recommended practices to carry out the task. The documentation and source codes of the CAN Logger 3 design are available to the public for references, and it has the following features:

- A low-cost hardware security module is used for secure key storage along with cryptographic implementations, including Diffie-Hellman key exchange, digital signature, and encryption. Its library can be found at [76].
- A public key exchange process between the CAN Logger 3 and the AWS cloud is performed during the provisioning process at production. The same shared secret key can be derived later from both parties for secure communication.

- Every truck logging session is encrypted using a randomly generated key, which is then encrypted using the shared secret key from the provisioning process. Thus, all the sensitive information is encrypted to protect data confidentiality before being stored on the local SD card.
- A client application interface is made for users to transfer their data from the CAN Logger 3 to the AWS server as well as to view and download uploaded files from the database. The communication between the device and the client application is through local serial, and the communication between the client application and AWS server is through the Internet with secure TLS using the Python requests module.
- Every truck logging session is hashed, and the hash digest along with the logging session metadata are signed using the device private key. The signature has to be successfully verified by the AWS server using the device public key obtained from the provisioning process before the log data is uploaded and stored on the server database. This step verifies that the data is from the correct sender and it has not been altered in any way, which is very important in cybersecurity measures as well as forensics purposes.
- User access control is implemented to ensure that only authorized users can access their data only or data that has been shared with them.
- Each device's vital information is backed up to a physical flash drive, which is kept by the administrators.

Chapter 4. Field Testing on Vehicle

A. Data Collection

One of the main purposes of this project is to actually collect heavy vehicle network data from operating trucks to generate large and beneficial resources for the trucking industry. As a result, CAN messages from operating heavy trucks have been gathered along the CAN loggers' development, starting by collecting data using the NMFTA CAN Logger in 2017. NMFTA has been playing a major role in supporting the project by providing data resources from different trucking companies. A batch of 100 NMFTA CAN Loggers and 25 CAN Logger 2 were built and shipped to NMFTA, where they were distributed to the volunteering companies for data collection. Because secure cloud storage was not in the design at this time, these companies had to ship the SD cards back to NMFTA, where the data was locally stored under a non-disclosure agreement with NMFTA. As of now, the NMFTA database has the following statistical information:

- Total number of captured messages: 11,035,396,328.
- Total size of all log files: 667.83 GB.
- Number of different trucks: 21.
- Number of CAN Loggers used: 54.
- Number of Companies involved: 11.
- Figure 4-1 shows a bar graph describing the number of messages per engine make.

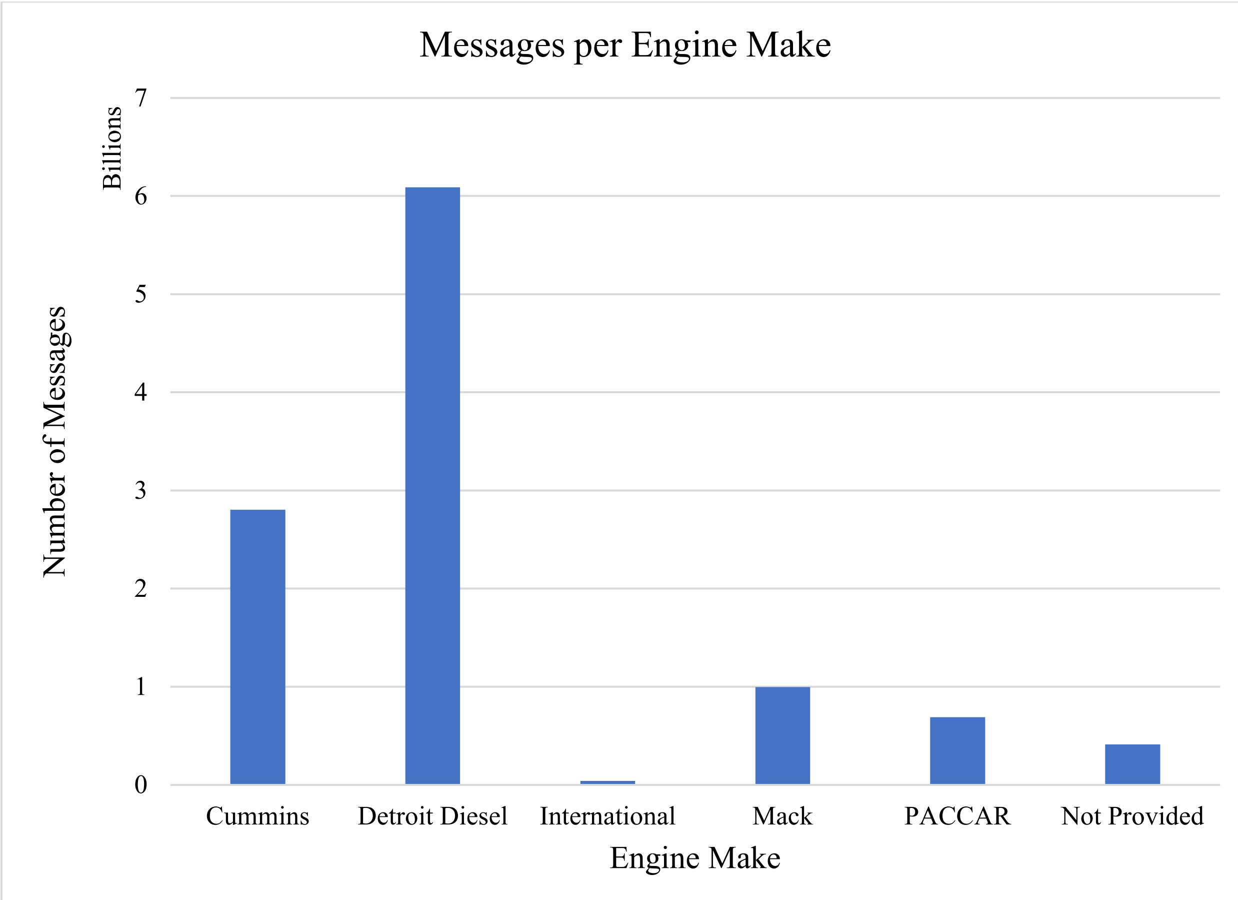


Figure 4-1. Number of messages per engine make

- Figure 4-2 shows a bar graph describing the number of messages per truck model year.

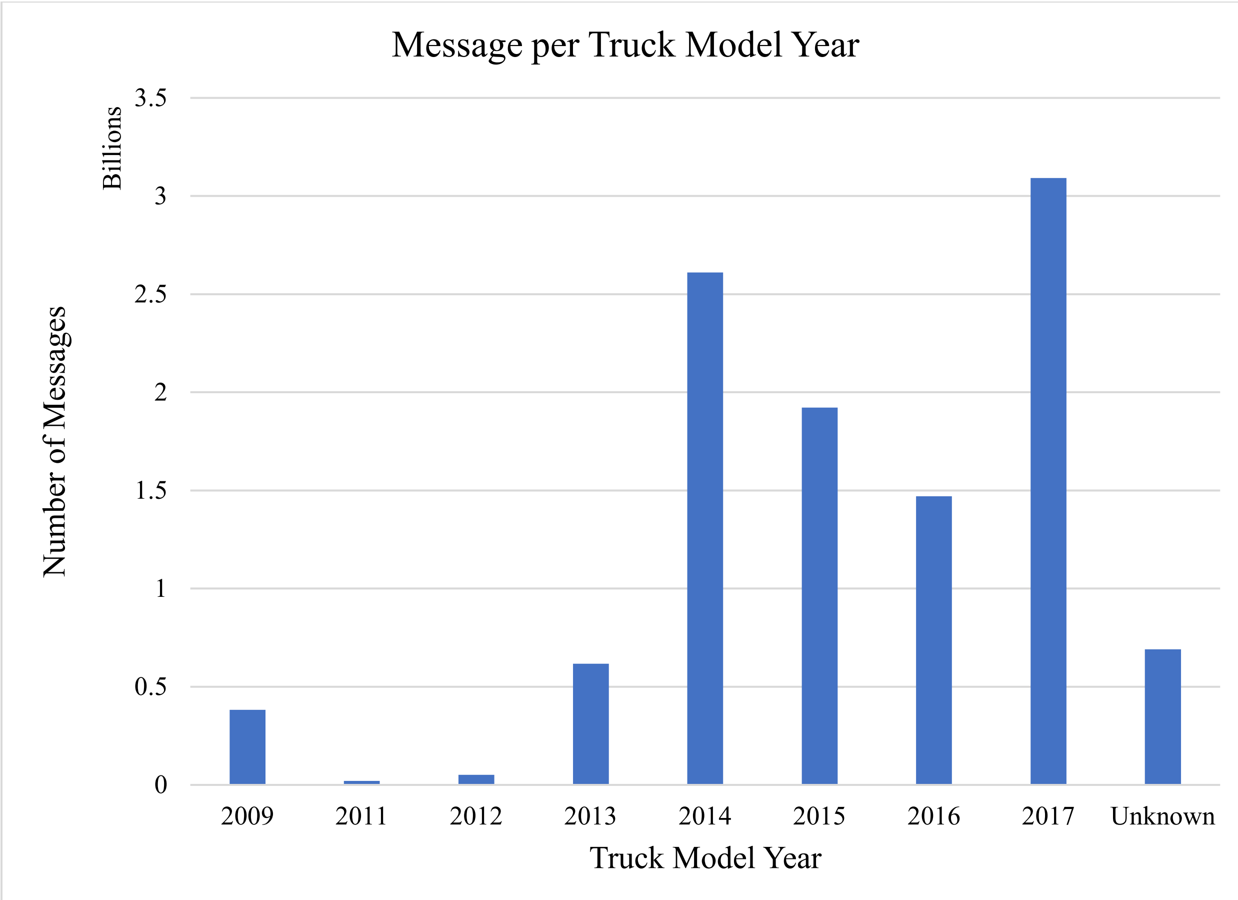


Figure 4-2. Number of messages per truck model year

In addition, the researchers have been making efforts to collect data when opportunities arise. In 2017, a 2007 Sterling truck was donated to the University of Tulsa, where the project was being conducted at the time. The Sterling was a great resource for data collection as well as device testing because instead of relying only on the truck in a box setup using an SSS2 as seen in Figure 2-75 (Chapter 2, part H.xxi.), the truck provided an actual operating platform which can be accessed anytime and modified as needed. Figure 4-3 shows the researching team working on the Sterling CAN network at the University of Tulsa facility.

The CAN logging project has moved to Colorado State University facility because the researchers conducting the project have transferred to this location from the University of Tulsa this past year. Even though the Sterling truck was no longer available, the Systems Engineering Department of Colorado State University obtained a 2014 Kenworth truck for research in heavy vehicle networks and cybersecurity. Figure 4-4 shows the 2014 Kenworth truck at Colorado State University facility.

Figure 4-3. Research team working on the CAN network of the 2007 Sterling truck

Figure 4-4. 2014 Kenworth truck of Systems Engineering department and Duy Van

The data pool for the CAN logging project needs log files from various types of trucks to create a large sample size. As a result, the data collected from the Sterling and the Kenworth was not adequate. To gain access to other trucks, the research group has been building good relationships with the local truck dealerships by visiting their sites to obtain truck parts for different research projects as well as attempting to repair their broken diagnostic tools. As a sign of friendship, the dealerships gave the team access to or even drove around their vehicles for data collection. Figure 4-5 shows a data logging session on a brand-new 2019 International truck at the Tulsa Summit Truck Group dealership in December 2018. Both the NMFTA CAN Logger and the CAN Logger 2 were used for data comparison.

Figure 4-5. Data collection on a 2019 International truck at the Summit Truck Group dealership in Tulsa

Moreover, the CAN Logger 3 has also played an important part in other's projects in terms of providing and analyzing the network data for references. One event was helping the 2019 senior project team from the University of Tulsa with capturing vehicle and engine speed for their tire pressure monitoring system test runs on a 2019 Freightliner. Figure 4-6 shows the mentioned event that took place March 2019 in Dallas, Texas. The collected log files were also used for the data pool.

Figure 4-6. Helping the University of Tulsa senior project team with capturing vehicle data

B. J1939 Decoding

After obtaining the log files in plaintext, either from logging with non-encrypted mode or decrypting the files using the Python client application, the data needs to be decoded into engineering units using SAE J1939. Furthermore, there were existing tools, like the Linux can-utils based on SocketCAN to inspire common storage formats for CAN data. The socketCAN candump log was chosen to be the universal format because it has been commonly known and widely used in the community. A GUI was developed to convert log files in plaintext captured from any CAN logger version to the candump format. The source code can be found here [104]. The GUI also displays all the transport layer protocol messages as well as verifies all the 512-byte block CRCs, which have been added in the CAN logger version 2 and 3 data structure, to ensure that the data did not have any error occurred during operation. This candump converted file can be generated by selecting the *save as candump format* function. Figure 4-7 shows the GUI interface after loading a CAN Logger 3 binary with CRC check. Figure 4-8 shows the same file saved in socketCAN candump format, which has the following structure from left to right: real-time clock, CAN channel, CAN ID, and data field.

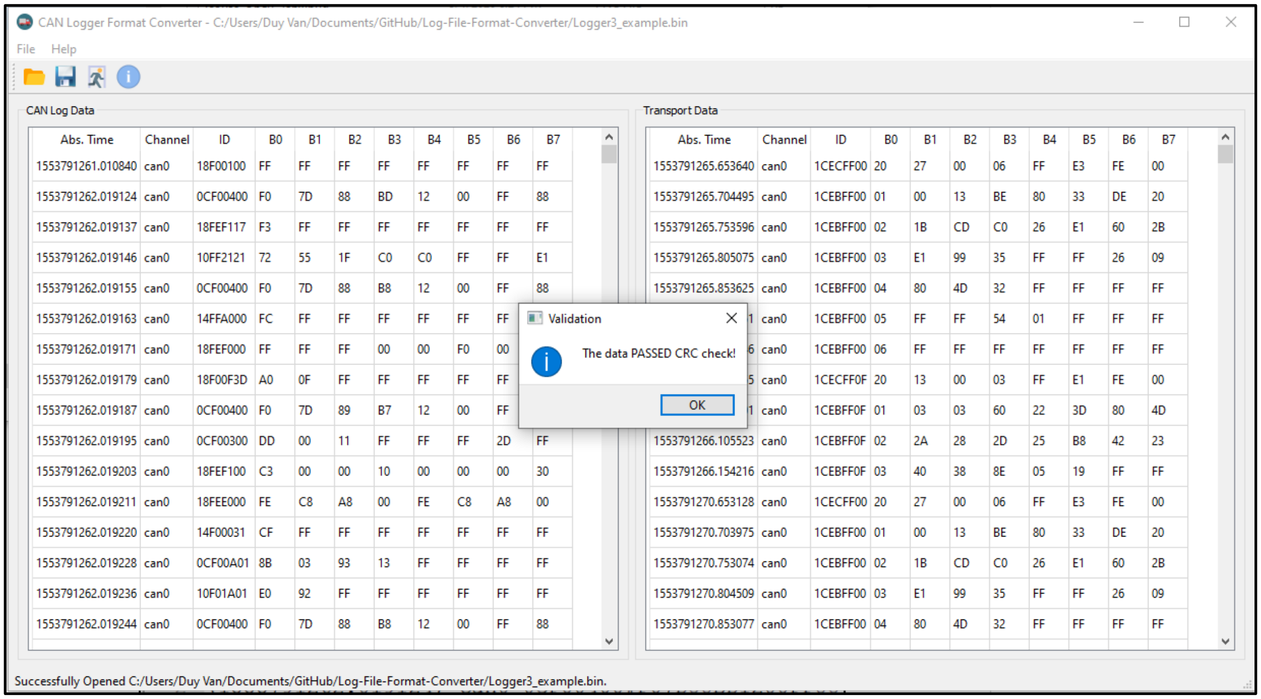


Figure 4-7. CAN logger format converter GUI

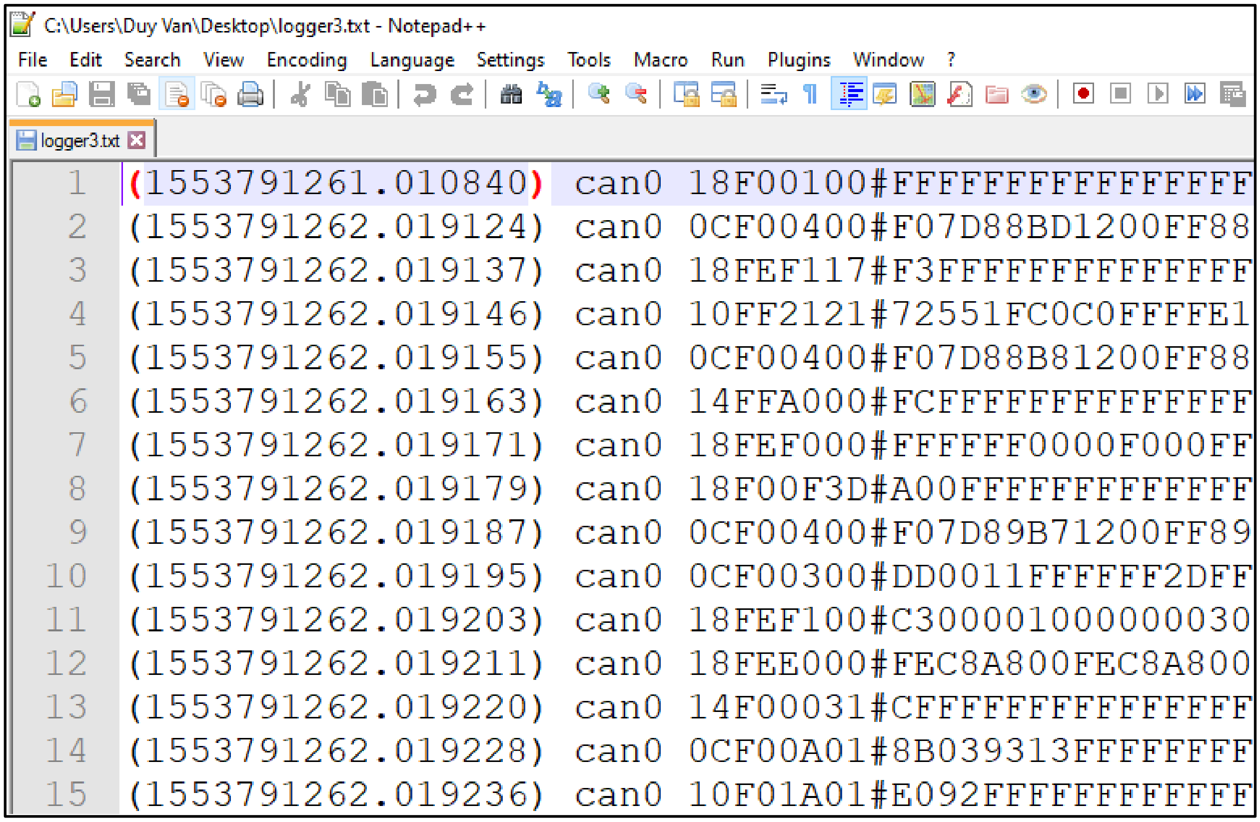


Figure 4-8. A CAN Logger 3 file converted to socketCAN candump format

On the other hand, the user can also save the file in format as shown in the GUI, where each parameter, including every byte in the data field, is illustrated per column. This format is easier to visualize and can be generated by selecting *save as text format* function. Figure 4-9 displays the log file in the text format.

C:\Users\Duy Van\Desktop\text_format.txt - Notepad++

	Abs. Time	Channel	ID	B0	B1	B2	B3	B4	B5	B6	B7
1	1553791261.010840	can0	18F00100	FF	FF	FF	FF	FF	FF	FF	FF
2	1553791262.019124	can0	0CF00400	F0	7D	88	BD	12	00	FF	88
3	1553791262.019137	can0	18FEF117	F3	FF	FF	FF	FF	FF	FF	FF
4	1553791262.019146	can0	10FF2121	72	55	1F	C0	C0	FF	FF	E1
5	1553791262.019155	can0	0CF00400	F0	7D	88	B8	12	00	FF	88
6	1553791262.019163	can0	14FFA000	FC	FF	FF	FF	FF	FF	FF	FF
7	1553791262.019171	can0	18FEF000	FF	FF	FF	00	00	F0	00	FF
8	1553791262.019179	can0	18F00F3D	A0	0F	FF	FF	FF	FF	FF	FF
9	1553791262.019187	can0	0CF00400	F0	7D	89	B7	12	00	FF	89
10	1553791262.019195	can0	0CF00300	DD	00	11	FF	FF	FF	2D	FF
11	1553791262.019203	can0	18FEF100	C3	00	00	10	00	00	00	30
12	1553791262.019211	can0	18FEE000	FE	C8	A8	00	FE	C8	A8	00
13	1553791262.019220	can0	14F00031	CF	FF	FF	FF	FF	FF	FF	FF
14	1553791262.019228	can0	0CF00A01	8B	03	93	13	FF	FF	FF	FF
15	1553791262.019236	can0	10F01A01	E0	92	FF	FF	FF	FF	FF	FF
16	1553791262.019244	can0	0CF00400	F0	7D	88	B8	12	00	FF	88
17	1553791262.019252	can0	18FEDF00	83	00	13	FF	7D	FF	FF	FF
18	1553791262.019260	can0	14FF3131	00	30	03	00	00	FF	FF	FF
19	1553791262.019269	can0	14F00131	FF	FF	FF	FF	00	FF	FF	FF
20	1553791262.020572	can0	0CF00400	F0	7D	89	B7	12	00	FF	89

Figure 4-9. A CAN Logger 3 file converted to text format as shown in the CAN logger format converter GUI

C. Data Interpretation

A CAN data analyzer GUI was developed to interpret the log files from any CAN logger version, as seen in Figure 4-10. This program parses through the data and reconstructs the information based on the J1939db JSON file [105] to human-readable and user-friendly display. The source code can be found here [106]. The main features of the CAN data analyzer program are:

- In the CAN ID Table, each unique CAN message ID is displayed per line along with their parameters such as Parameter Group Number (PGN), Suspect Parameter Number (SPN), Source Address (SA), Destination Address (DA), counts, period, and frequency.

- When the user clicks on a specific CAN ID in the CAN ID Table, the Data Table will display all the CAN messages with that ID.
- Transport layer protocol messages are displayed in the Transport Layer Message Table.
- The user can select a CAN ID with a specific PGN in the CAN ID Table to plot its associated SPN values, which can be looked up in the J1939-71 standard [99]. The column parameter in the Data Table can also be plotted by selecting the desired header.

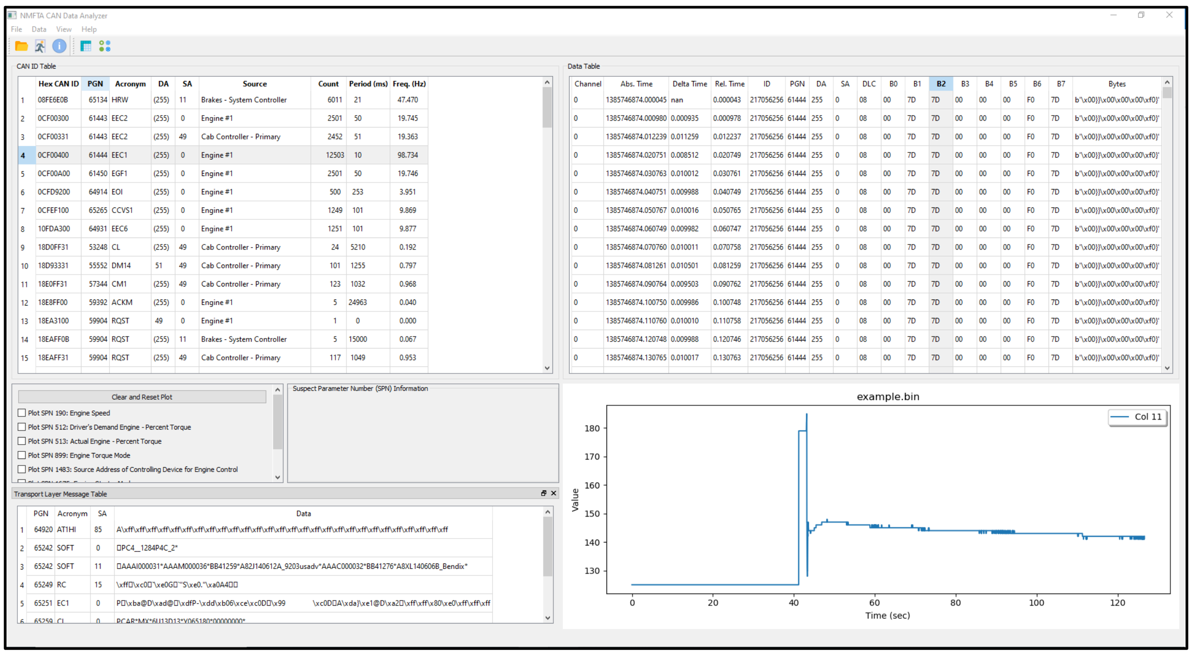


Figure 4-10. CAN data analyzer GUI

With the CAN logger format converter and CAN data analyzer tool, log files can be decoded and their information can be obtained. Taking the senior project event in Dallas, TX as an example, some basic properties of the data recorded during the testing are depicted in Table 4-1, through five different runs. The parameters describing in each run are total time in second, log file size in byte, number of total messages, average number of message per second, number of unique PGN, number of unique SA, top vehicle speed in mph, and top engine speed in RPM.

Table 4-1. Information of five truck runs from the Dallas trip

	Run 1	Run 2	Run 3	Run 4	Run 5
Total time (s)	801.2	392.5	457.75	1067.4	585.7
File size (bytes)	13,851,648	6,776,832	7,900,160	15,360,000	10,262,528
Total messages	514,028	251,486	293,172	570,002	380,838
Messages/second	641.6	640.7	640.5	534	650
Number of unique PGN	99	99	90	97	93
Number of unique SA	20	19	13	17	15
Top vehicle speed (mph)	63.96	47.4	48.16	49.17	45.9
Top engine speed (RPM)	1,635.25	1,551.375	1.612	1,645.5	1,517.9

Some conclusions were drawn from Table 4-1, including, but not limited to:

- During a normal drive, the truck generated approximately 640-650 messages per second on average. Run 4 had significantly smaller messages per second (534) because the truck was turned off, which produced fewer messages, for a portion of the logging session.
- The vehicle did not generate the same set of CAN messages on every run based on the fact that the numbers of unique PGN and SA were not the same.
- The second run had the highest top vehicle speed among the five.
- The vehicle reached a top engine speed of approximately 1500-1600 RPM.

The CAN ID table from the CAN data analyzer can give a better understanding of the messages within a log file. Table 4-2 displays the CAN ID table information of the messages captured during the second run.

Table 4-2. CAN ID table of the of the second run from the Dallas trip

	Hex CAN ID	PGN	Acronym	DA	SA	Source	Count	Period (ms)	Freq. (Hz)
1	0CF00400	61444	EEC1	(255)	0	Engine #1	39336	9	100.212
2	10FF2121	65313	PropB_21	(255)	33	Body Controller	8251	46	21.304
3	18F00F3D	61455	AT1OG1	(255)	61	Exhaust Emission Controller	7996	49	20.044
4	0CF00A01	61450	EGF1	(255)	1	Engine #2	7973	49	20.043
5	10F01A01	61466	TFAC	(255)	1	Engine #2	7973	49	20.043
6	0CF00300	61443	EEC2	(255)	0	Engine #1	7867	49	20.042
7	18FEDF00	65247	EEC3	(255)	0	Engine #1	7867	49	20.041
8	18FEF117	65265	CCVS1	(255)	23	Instrument Cluster #1	4159	99	10.017
9	18FD8C3D	64908	AT1GP	(255)	61	Exhaust Emission Controller	3998	99	10.021
10	18FDB23D	64946	AT1IMG	(255)	61	Exhaust Emission Controller	3998	99	10.021

The CAN ID table from Table 4-2 was made up of different combinations of the unique PGN (derived from Hex CAN ID) and SA. The data was sorted by the Count column, from largest to smallest. Due to the large number of unique PGN (99) and SA (20), only the first 10 combinations were displayed. From here, better insights can be made for this second run, such as all the decoded messages PGN acronym and SA for easy interpretation, the message that occurred the most frequently (or least) and its corresponding interval, message counts, etc.

Vehicle wheel speed (PGN 65265, SPN 84) and engine speed (PGN 61444, SPN 190) are the two parameters that are commonly analyzed to determine how the vehicle was operated throughout the trip. Generated by the CAN data analyzer GUI, Figure 4-11 and Figure 4-12 show the vehicle speed and engine speed in RPM over time plots from the data collected in the second run during the Dallas trip. The same plots can also be generated using the vehicle speed plotting script [107] and engine speed plotting script [108]. The difference between using these scripts and the CAN data analyzer plotting feature is that the scripts require the log file input in socketCAN candump format while the CAN data analyzer requires the log file input in raw format. These plots provided the senior project team at the University of Tulsa beneficial insights to help complete their project.

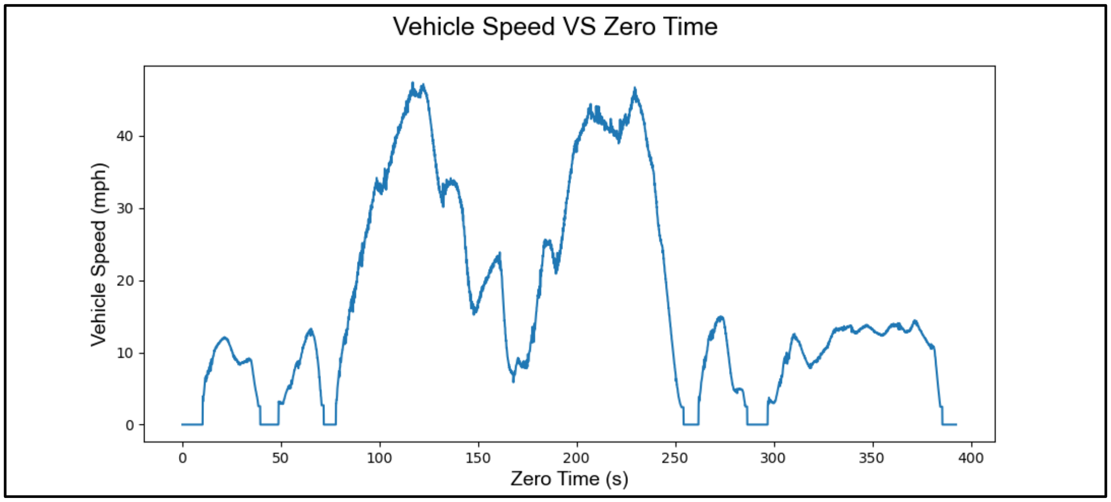


Figure 4-11. Vehicle speed plot from the second runs for the University of Tulsa senior project

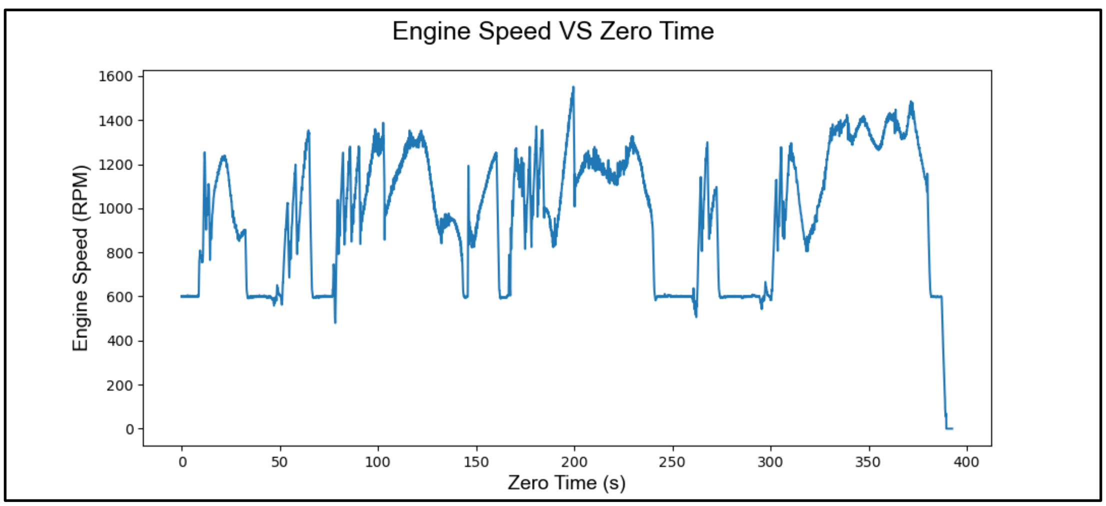


Figure 4-12. Engine speed plot from the second runs for the University of Tulsa senior project

Chapter 5. Chip Level Digital Forensics Application

This chapter discusses an application of using the CAN Logger 3 for researching heavy vehicle digital forensics. This study was conducted by Dr. Jeremy Daily and Duy Van, with the support from Matthew DiSogra from Delta |v| Forensic Engineering. The CAN logger was used for logging data and acting as a middle person device to extract critical information to complete this research, as described in section H of this chapter. The research was published by SAE International, with the title of "Chip and Board Level Digital Forensics of Cummins Heavy Vehicle Event Data Recorders," and the paper number is 2020-01-1326 [10]. This chapter follows the same structure as the SAE paper.

A. Abstract

Crashes involving Cummins powered heavy vehicles can damage the ECM containing heavy vehicle event data recorder (HVEDR) records. When ECMs are broken and data cannot be extracted using vehicle diagnostics tools, more invasive and low-level techniques are needed to forensically preserve and decode HVEDR data. A technique for extracting non-volatile memory contents using non-destructive board level techniques through the available in-circuit debugging port is presented. Additional chip level data extraction techniques can also provide access to the HVEDR data. Once the data is obtained and preserved in a forensically sound manner, the binary record is decoded to reveal typical HVDER data like engine speed, vehicle speed, accelerator pedal position, and other status data. The memory contents from the ECM can be written to a surrogate and decoded with traditional maintenance and diagnostic software. The research also shows the diagnostic trouble codes from the ECM are preserved. In other words, the digital forensic technique of extracting memory contents through the in-circuit debugging port does not

introduce any new fault codes. Cryptographic hashing of the forensic binary data provides a mechanism to verify the original digital forensic record. Finally, the decoding for the HVEDR binary record is presented so investigators can decode the forensic record without the need for a surrogate ECM. The techniques in this paper provide a new method for extracting data from heavy vehicle ECMs.

B. Introduction

Since approximately 2002, Cummins ECMs have had the capability to record the data for Sudden Deceleration Data Reports (SDDRs). When the measured wheel speed changes by an amount greater than some programmed threshold, a SDDR recording is triggered. A typical SDDR will contain a second-by-second history of vehicle speed. In 2009, Bortolin, et al., validated the speeds recorded in the SDDR against a V-BOX III GPS data logger establishing a basis for their use in accident reconstruction [113].

Engine control modules with HVEDR capabilities can be compromised in a crash. The location of a module in the engine compartment is more vulnerable than if the HVEDR is in the cab. For example, the Cummins ECM is located on the drivers' side of the engine, as shown in Figure 5-1. In frontal collisions, the region containing the ECM is prone to damage, which can render the ECM inoperable. Often, separation of the engine from the frame in a crash compromises the vehicle side connector of the ECM, as shown in Figure 5-2. The effects of power loss (as a result of mechanical damage) on the Cummins family of electronics were studied by Messerschmidt, et at. in 2010 [110]. Findings showed that depending on the mechanics of the power loss, SDDR data may or may not be retained by the ECM.

When ECM connectors or casings are damaged, often the internal components and integrated circuits (ICs) are in good working order. This means data bearing chips (e.g. EEPROM, flash memory, and microprocessors) can still contain data. In 2015, Daily, et al. validated such a method on the DDEC V family of ECMs [111]. The purpose of this chapter is to explain how to extract data from damaged Cummins CM870, CM871, and CM2350 electronic control modules. The techniques presented in this work should apply to other Cummins ECMs with the in-circuit debugging port enabled, but were not tested. These include the CM570, CM875, CM876, CM2150, and CM2250 that have been used to control on-highway Cummins engines. The difference in architecture between the CM870 series and CM2350 series presented in the paper should give the reader an appreciation for the process to accommodate different hardware architectures.

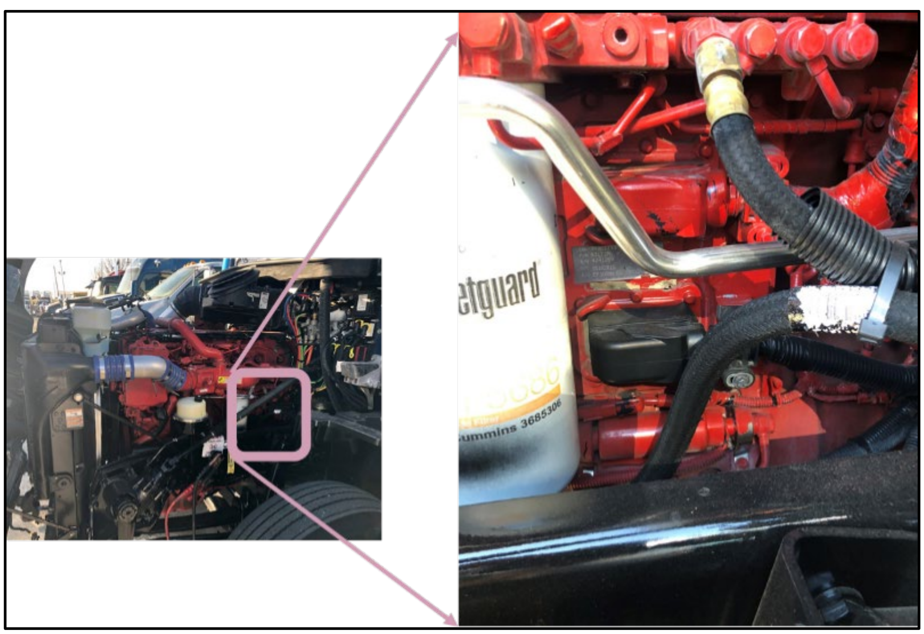


Figure 5-1. Typical location of the Cummins electronic control module on the drivers' side of the engine compartment

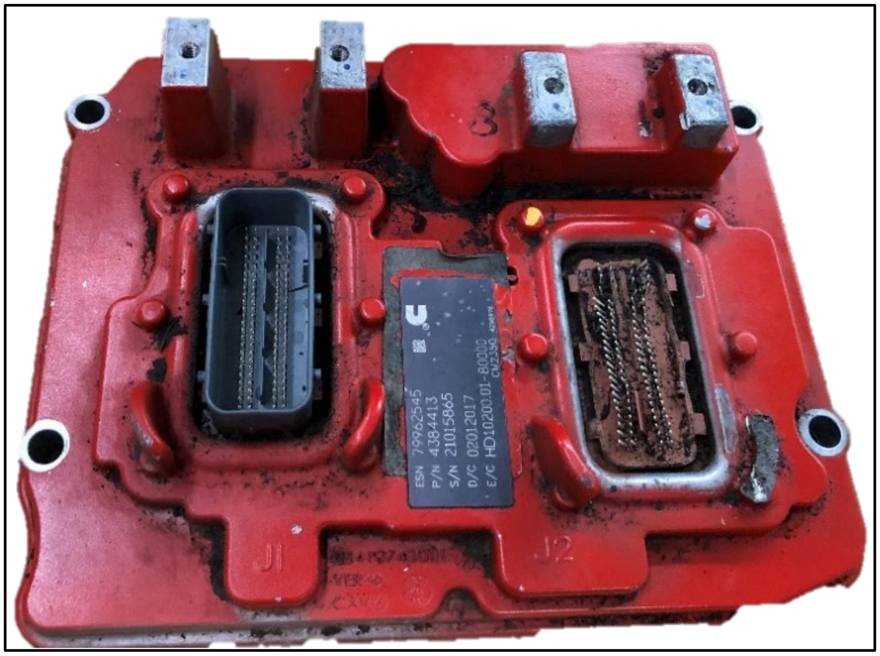


Figure 5-2. Cummins electronic control module with damage to the vehicle connector, which is consistent with a crash where the engine is dislodged from the frame

While the primary focus of this paper is recovery and interpretation of the SDDRs, Cummins ECMs also have the capability of storing data in the form of fault code snapshots. These snapshots can be triggered by electrical faults detected by the ECM and can include a record of vehicle speed at the time of occurrence. Because vehicle speed is recorded, fault code snapshots are also useful for the purposes of accident reconstruction. However, these snapshots can be overwritten by subsequent occurrences of the same fault. A secondary application of the methodologies outlined in this paper is preservation of fault code snapshots. Before an ECM is powered up for a network level acquisition, fault code data can first be preserved via board level Joint Test Action Group (JTAG) in-system programming port forensics. If the network level acquisition proves to introduce or overwrite fault code data, the process can be repeated by reprogramming the ECM with the originally imaged data. This process can be repeated indefinitely until the desirable results are achieved without risking data loss.

C. Procedure

A wide view of the process is shown in Figure 5-3 with the verbs in boxes acting on the data in the ECM under investigation. The first step in the process is to extract the data from the subject ECM. There are three ways to extract the data: 1) through the network, 2) through the in-circuit debugging port, 3) direct reading of the data bearing chips. Traditionally HVEDR is downloaded over the network. In this paper, we focus on extracting data using board and chip level techniques.

Once the complete binary image of the ECM is extracted, parts of it can be decoded to develop an understanding of its contents. A good candidate for decoding is the sudden deceleration data. Regardless of the decoding of the raw binary, the image can be written to an exemplar or surrogate ECM that does not have damage. A network-based download (e.g. using Cummins PowerSpec) of the data can then be performed. If there is a decoded record from the binary, these data can be compared. If there is no external decoding of the binary, the extracted binary image is cloned by writing the binary image to the exemplar.

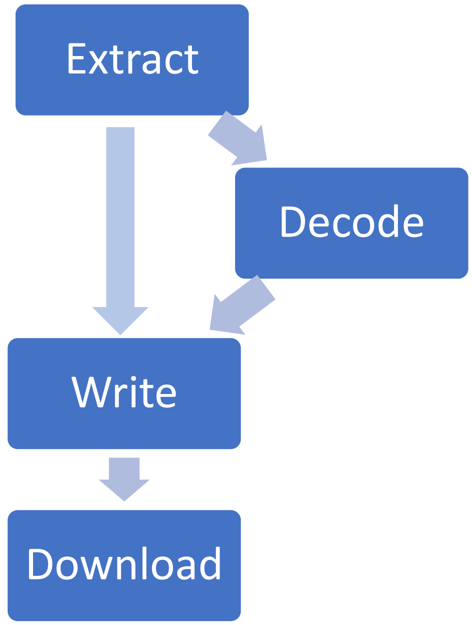


Figure 5-3. Simplified process diagram

D. Extracting Data

The first part of a digital forensics analysis is data acquisition. This this process has three levels: 1) Network Level, 2) Board Level, and 3) Chip Level. Network level acquisition of digital forensic data typically uses a vehicle diagnostic adapter (VDA) with some manufacturer specific software to download the data over the in-vehicle network. Most data extractions use this method when the ECM is intact. Network based extractions, while the most common, are outside the scope of this paper. Instead, we focus on board level and chip level forensics. There are three examples of these techniques in the following sections.

i. Chip Level Forensics

Often, broken electronic control modules still have intact circuits and ICs, commonly referred to as chips. The data bearing chips can be integrated into the main processors or built as additional memory devices, such as flash or EEPROM devices. In the case of the Cummins

CM870, the memory components are found in three distinct parts: 1) the flash memory, 2) EEPROM, and 3) processor memory. The flash memory location is shown in Figure 5-4 for a CM870. The CM871 uses a similar processor as the CM870 but does not have an EEPROM and the flash memory is a ball grid package. The flash memory for the CM871 is shown in Figure 5-5. An overview of the chipsets are shown in Table 5-1.

A strategy to recover or read the binary data in these memory bearing devices is to remove or detach them from the circuit board and insert them in the chip reading device. The photograph in Figure 5-7 shows the process of removing a flash memory chip from an engine control module using hot air. Care must be taken to keep the pins from the surface mount package straight and clean, since they will be read using a chip reader.

Table 5-1. Chipsets for the Cummins ECMs under study

ECM	**Processor**	**FLASH**	**EEPROM**
CM870	Freescale MPC555 BGA	Intel FLASH 28F800F3 TSOP	AT25128 8-pin SOIC
CM871	Freescale MPC565 BGA	AM29BDD160G 16Mbit Flash	Integrated
CM2350	NXP MPC5674 BGA	Integrated	Integrated

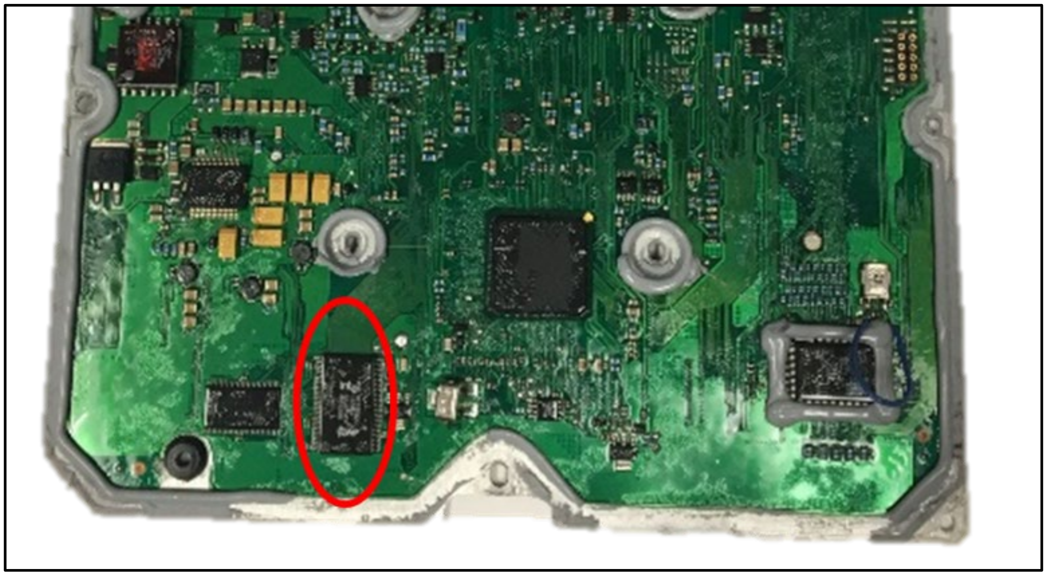


Figure 5-4. The location of the Flash memory in the Cummins CM870

Figure 5-5. The internal contents of a Cummins CM871 with the flash memory highlighted

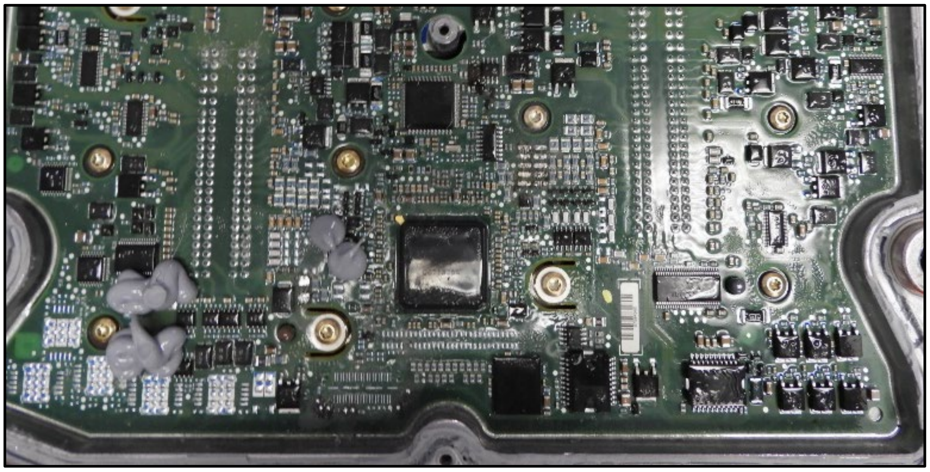


Figure 5-6. The inside of a Cummins CM2350. All data bearing devices are integrated into the main processor

A chip reader is a general-purpose machine with adapter sockets to accommodate different chip shapes and pin spacings. These readers are more appropriately called programmers as they are designed to write programs to the memory chips during end of line programming. In a forensic sense, the ability to write to a chip poses a risk of data spoliation, so caution must be taken to avoid using the erase and write functions of the chip programming utility.

Figure 5-7. Removing flash memory from an engine control module using a hot-air rework station

The Xeltek SuperPro 6000 programmer, shown in Figure 5-8, was used to read the Intel automotive flash memory found in many engine control modules (e.g. Cummins, Caterpillar, Detroit Diesel). The operation requires removing the chips from the printed circuit board, ensuring the pins are straight and free from excess solder, inserting the chip into the holder and attaching the holder to the device. A photograph of the Intel flash memory chip from the Cummins CM870 control module is shown in Figure 5-9.

Once connected, the software for the Xeltek programmer can image (read) the binary data from the chip in its entirety. This a bit for bit copy of the data on the flash memory, or an image, of the memory chip as it was while was in the ECM. The process of creating this copy is called imaging.

Figure 5-8. General purpose chip programmer

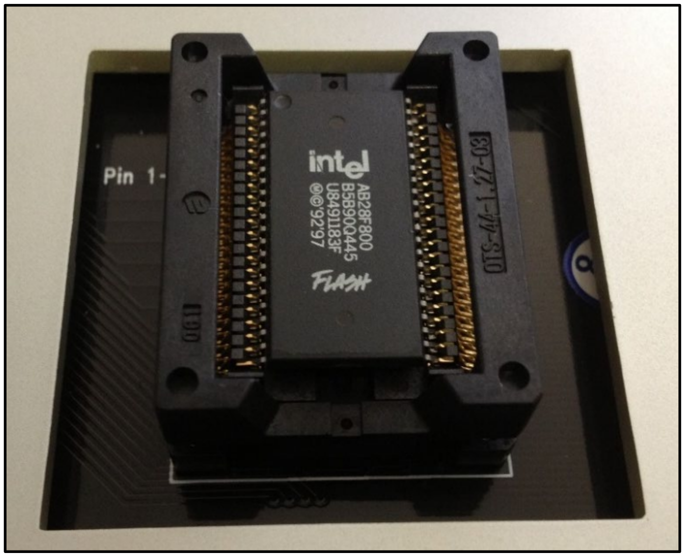


Figure 5-9. Intel flash memory used to hold the Cummins Sudden Deceleration data for the Cummins CM870 engine control module

Acquiring chip level forensic data is destructive to the ECM and requires a delicate process of lifting the chip from the board as shown in Figure 5-7. To read the chip contents without a chip reader, the data bearing memory chip needs to be transplanted onto a surrogate ECM and processed. However, these procedures are challenging, especially for Ball Grid Array (BGA) chips, like the one used in the Cummins CM2350. Fortunately, there is an alternative method that leaves the board and chip intact, which uses the Joint Test Action Group (JTAG) in-circuit programming port. As it turns out, this method applies to all the Cummins modules in this research.

ii. Board Level Forensics

If the ECM is opened to expose the board, the circuitry remains intact inside, and the JTAG port is enabled (by default from the factory), then an in-circuit programmer can be used to extract the memory contents for the data bearing chips. It is important to note the necessity of the

JTAG port being open. As OEMs harden their ECMs against cybersecurity attacks, these ports may be shut off in the future. For MY 2019 and older Cummins ECUs, the JTAG port is open. It is expected this technique may not work in models going forward due to the lockout of the JTAG port.

With the JTAG, data obtained can then be either directly decoded, or, the data can be reprogrammed into a surrogate ECM and then downloaded using traditional techniques. In this paper, the two tools used to image the memory contents through the JTAG port are explained and compared. This comparison demonstrates the techniques and data between the tools are consistent and reliable. The comparison of the tool output is critical in establishing a sound scientific and engineering basis for the process. In practice, only one tool is necessary. The two tools are the AlienTech KTAG, as shown in Figure 5-10 and the PEmicro Cyclone, shown in Figure 5-15.

Using the Alientech KTAG

The KTAG system, based out of Europe, is primarily marketed to the so-called tuner community who are enthusiasts interested in modifying the performance of their engines by manipulating the firmware binary codes on the ECM. There are different levels of the software service for the KTAG, one for developing and distributing a modified firmware, and the other for writing a firmware developed by someone else. Since a forensic examination is focused on reading, the master version (used for developing) of the KTAG hardware was used.

The second part of the KTAG system is the software. Alientech, the makers of the KTAG system, tries to accommodate many makes and models of microprocessors found in engine control modules. Their business model is to sell subscription services for the programming protocols for the different processors. Therefore, to read the data from the Cummins ECMs, which use the MPC5XX series processors, the Alientech protocol package for the MPC5XX series needed to be active.

This paid subscription for the KTAG software provided benefit for extracting the data through the JTAG port with clear instruction on setup. Visual guides in the computerized help system instructed the technician performing the forensic examination where to connect the different color cables, power connectors, and jumper wires. These instructions eliminated many hours of reverse engineering to properly connect to a JTAG port.

Figure 5-10. Alientech KTAG kit as pictured on the KTAG website

An example connection of the KTAG cables and jumper to the Cummins CM870 is shown in Figure 5-11. The CM870 is a module made using a FlexPCB that conforms to its metallic substrate. The malleable sides of the engine controller are screwed and sealed to a rigid frame in the core. To access the crystal (for the jumper wire) and the JTAG port, only the back side of the module needs to be opened. The backside is the side of the module opposite the connectors.

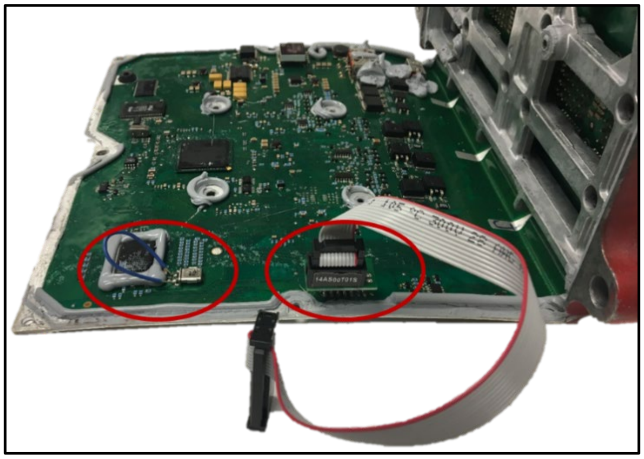


Figure 5-11. The inside of the Cummins CM870 with a bridge attached to the crystal and the JTAG port connected with a ribbon cable on the right

The front side, or connector side, accepts the power cables and key switch signals from the KTAG device. In situations where the connectors are broken, the connections may need to be made by finding an intact trace on the inside of the board. The key switch is controlled through software, so there is no visible switch with the KTAG system. The connections for the power cable and ignition switch from the KTAG are shown in Figure 5-12.

Figure 5-12. : Power cable with key switch shown connected to a Cummins CM870 ECM

Once physical connections are established and stable power is applied, the user selects the correct protocol and reads the data. The graphical interface for these options is shown in Figure 5-13. The data extraction process takes about 5 to 15 minutes, during which the software displays indicators of the memory sectors being read.

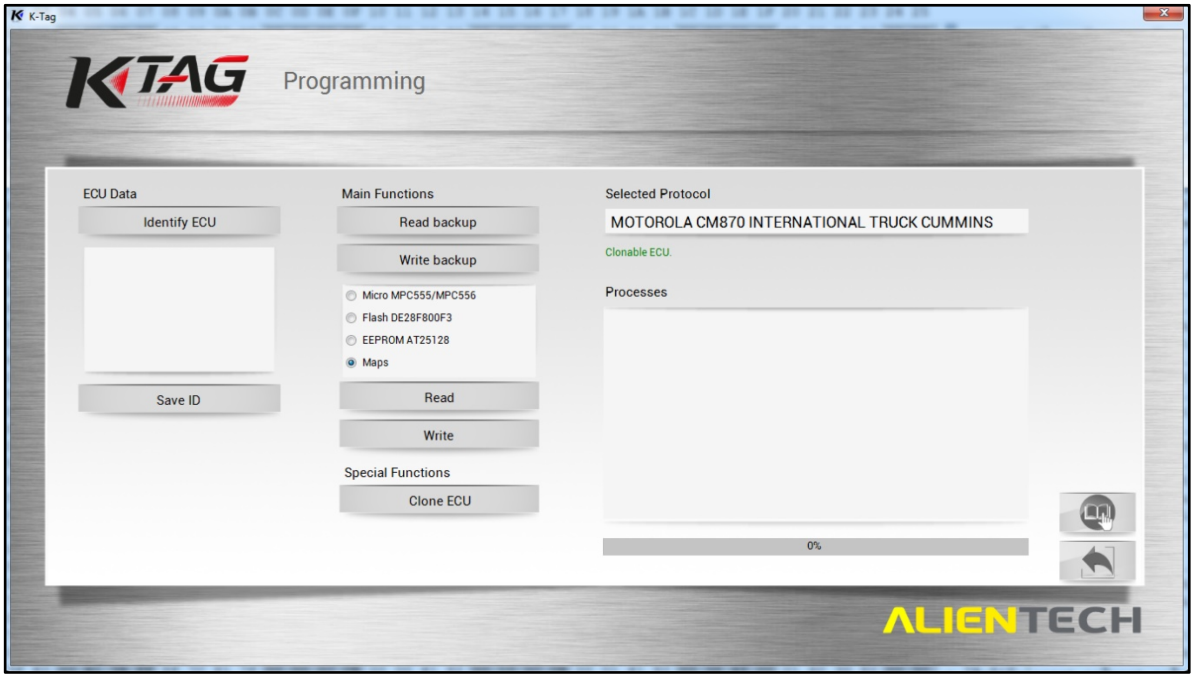


Figure 5-13. Selecting the correct ECM through the KTAG software and reading the memory contents through the JTAG port

Upon completion, the user can save the files. An option to save the files separately is offered and the user should select yes, as shown in 5-14. By saving the files separately, the raw binary images of the memory bearing chips are preserved as individual files. Whereas choosing to not save files separately results in the KTAG system transforming the data into their own proprietary format.

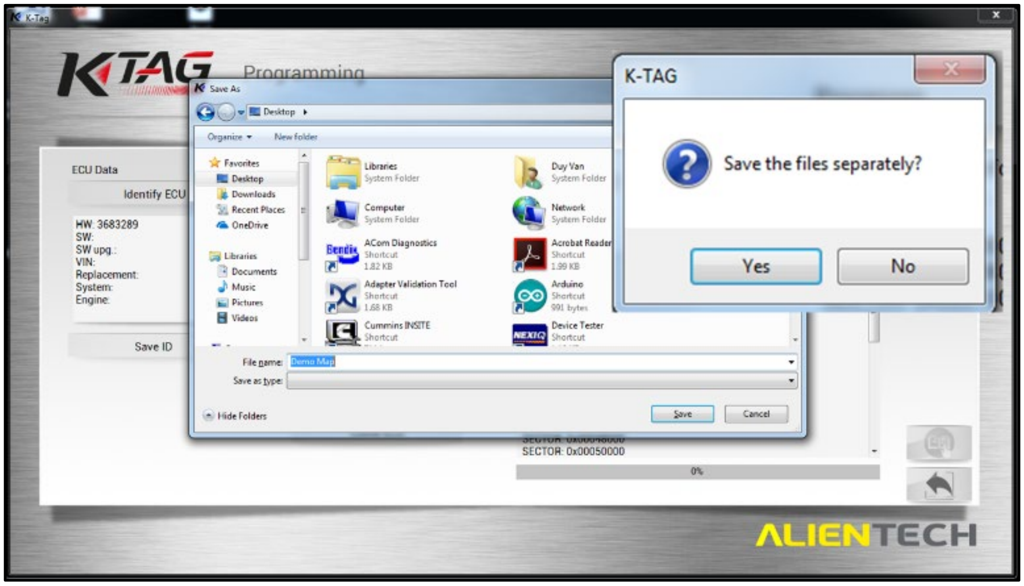


Figure 5-14. Saving the files after reading the memory contents using the K-TAG software

Upon collecting and saving the file, it is critical to rapidly convert the binary image into a forensic image. This quick action, before any other, ensures a defensible integrity check of the data anytime in the future. This is especially useful if there is a cybersecurity event, a crash event, or the potential for litigation. A discussion of digital forensic soundness ensues.

Using the PEmicro Cyclone

The PEmicro Cyclone is a general purpose in-circuit programmer/debugger shown Figure 5-15. It is a commercially available tool with applications to many microprocessors for any type of electronic device. It does not have any specific vehicle ECM related capabilities or functionality like the KTAG system. Instead, the forensic technician will have to have prior knowledge of how to connect the JTAG port to the PEmicro Cyclone. The Cyclone FX is a feature-full offering from PEmicro. Less expensive tools can accomplish the same task described in this section.

The JTAG ports in the CM870 and CM871 follow a standard layout, only headers need to be soldered to the pads on the board to connect a ribbon cable. The same pin headers can be used for both the KTAG and the PEmicro. However, the CM2350 printed circuit board does not have headers in a standard JTAG configuration. Therefore, the knowledge gained from the KTAG was used to determine the location of the pins for the CM2350.

Figure 5-15. Image of the PEmicro Cyclone in-circuit programmer and debugger

The KTAG user guide was specific to the KTAG color cables and did not give insight into the name of the connection. To determine pin locations, the microprocessor was removed from the CM2350 board and a continuity test was conducted between the pads suggested by the KTAG help file and the BGA pads on the PCB. Since the processor is the NXP MPC5674F and has a publicly available data sheet [114], the signals can be inferred by the name of the ball pads for the microprocessor. The results of reverse engineering traces in the PCB are shown in Figure 5-16.

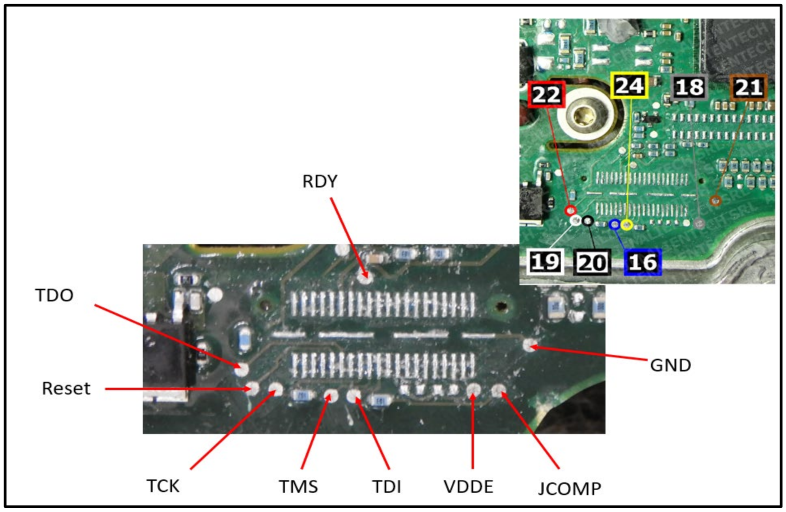


Figure 5-16. Determination of JTAG signal names and the KTAG color guide for the CM2350

After obtaining the necessary information about individual JTAG pinout, the CM2350 data can be extracted using the PEmicro Cyclone. Figure 5-16 illustrates individual JTAG pinout for PEmicro connection.

Using the PEmicro device requires the technician to have additional knowledge for the signal traces and components in the ECM. One must identify the family of the module microprocessor and identify the proper port in the PEmicro, as seen in Figure 5-17. This can be done by examining the part number imprinted on the top surface of the processor, and looking up the datasheet. The CM2350 module contains a MPC56XX series, which corresponds to port C on the PEmicro Cyclone. The manual for the PEmicro Cyclone shows connector port C to have the pinout shown in Figure 5-18.

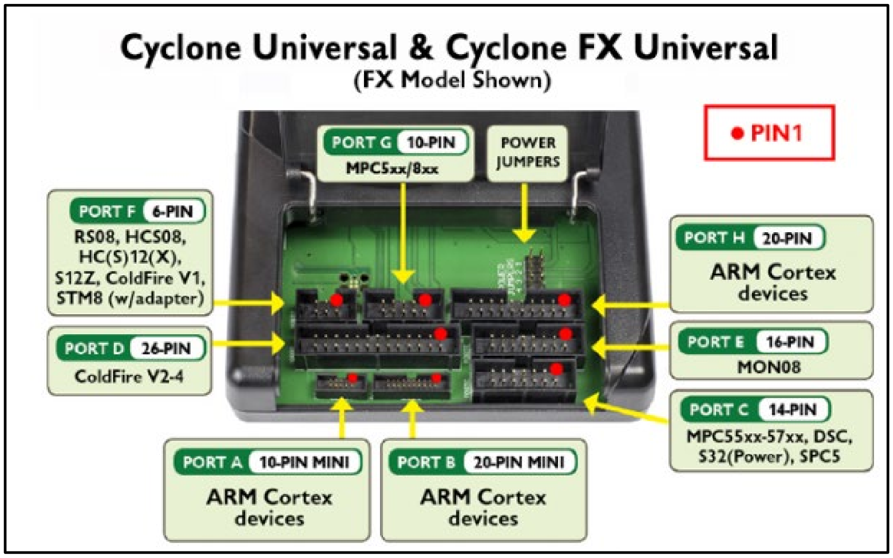


Figure 5-17. Supported microprocessor families by PEmicro Cyclone on different ports

PORT C: 14-Pin Debug Connector (MPC55xx-57xx, SPC5, DSC, S32 (Power))

The Cyclone provides a standard 14-pin 0.100-inch pitch dual row 0.025-inch square header for MPC55xx-57xx, DSC (MC56F8xxx), S32R, or STMicroelectronics' SPC5 targets. The location of the this header is indicated as PORT C in **Figure 3-5**.

MPC55xx-57xx, SPC5, or S32 (Power) Pinout

TDI	**1**	**2**	GND
TDO	**3**	**4**	GND
TCK	**5**	**6**	GND
NC	**7**	**8**	NC
RESET	**9**	**10**	TMS
VDDE7	**11**	**12**	GND
RDY	**13**	**14**	JCOMP

Figure 5-18. JTAG pinout for port C on PEmicro Cyclone

A setup assembly for CM2350 data extraction using the PEmicro Cyclone is shown in Figure 5-19. The PEmicro Cyclone is connected to the CM2350 module at JTAG pinout from the above information using a ribbon cable and grabber clips. A SSS2 is used to supply power for the CM2350 module for the extraction process [115].

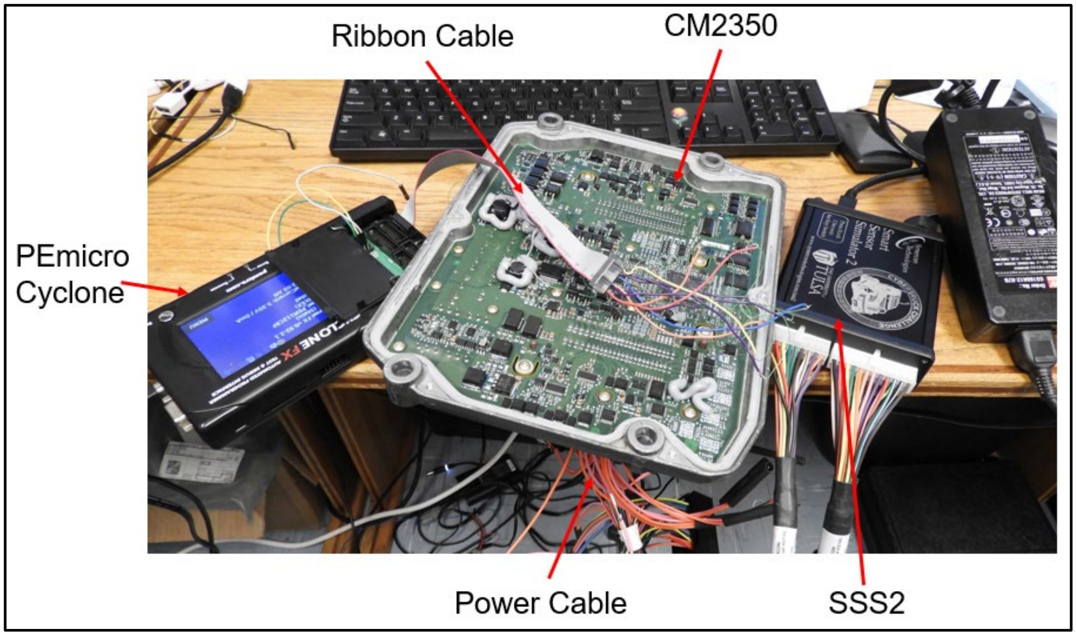


Figure 5-19. Setup for PEmicro data extraction on CM2350

After the physical setup is completed, the investigator would use the PEmicro software, which has both command line interfaces and graphical user interfaces. Each microprocessor family has its own programmer and for the CM2350 MPC5674; the correct software is progppcnexus_cyclone.exe [115]. Figure 5-20 shows the interface of the programmer on a Windows computer.

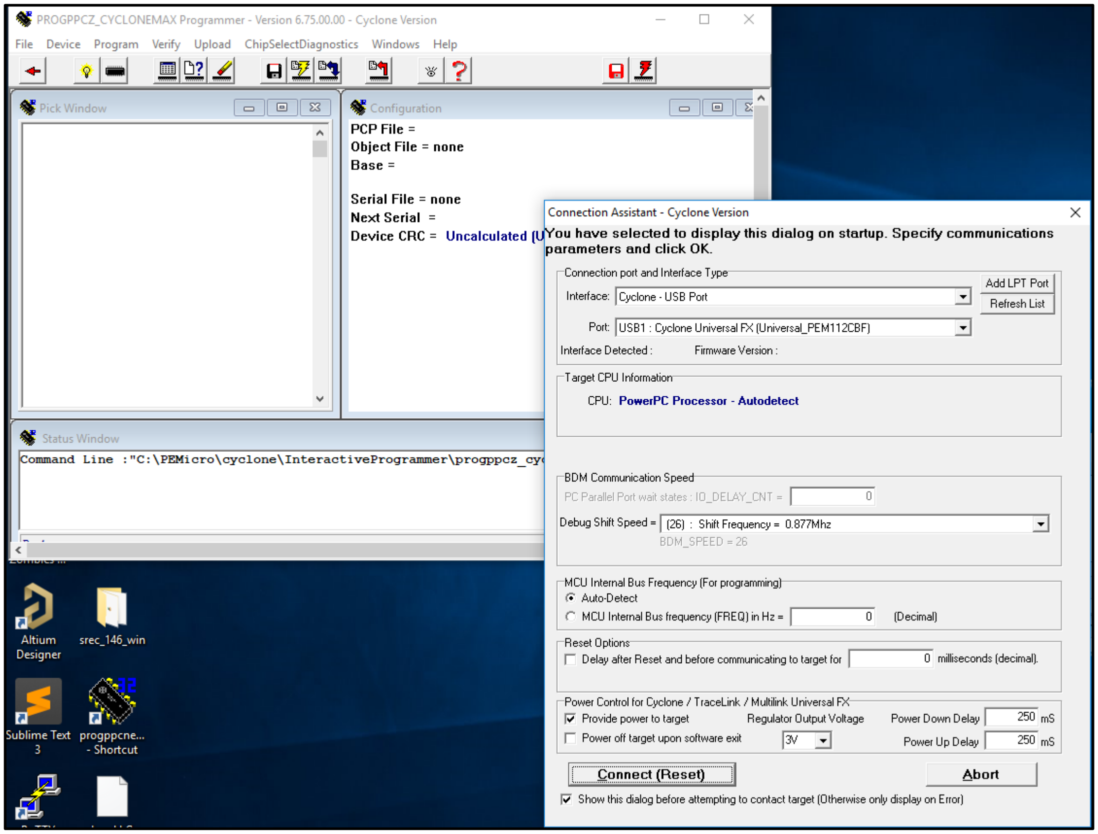


Figure 5-20. PEmicro interactive programmer

Once successfully connected to the CM2350, the PEmicro programmer will ask for an algorithm to run on the module. The algorithm is needed to identify which memory to be read, which is based on the exact part number of the microprocessor and its memory size. The algorithm for this CM2350 is NXP_MPC5674F_1x32x1024k.pcp, as shown in Figure 5-21 [115].

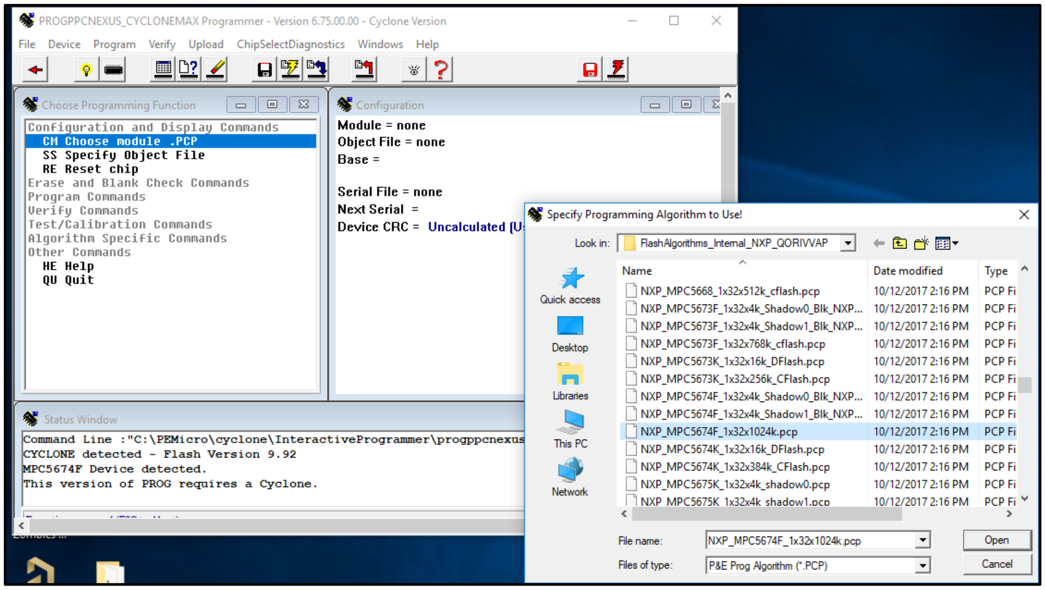


Figure 5-21. Choosing algorithm for data extraction

The memory data of the CM2350 can now be retrieved using the Upload Module function in the programmer, as seen in Figure 5-22. The programmer will ask for a name for the output file before extracting the data.

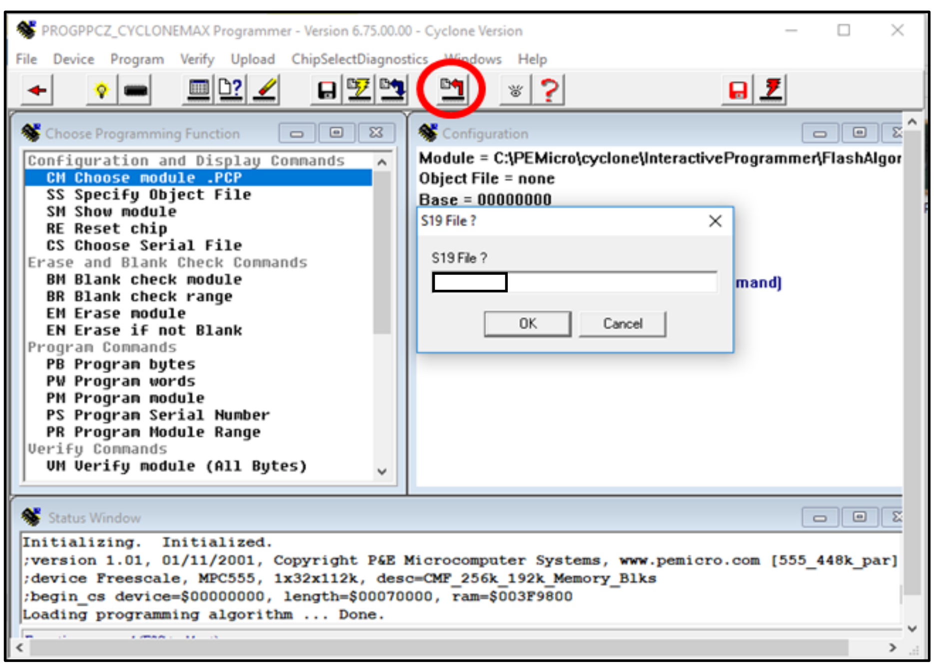


Figure 5-22. Extracting data in software using *upload module* function

The PEmicro programmer reads and outputs the memory content in an S19 file type. The S19 record is a format developed by Motorola and it is the only compatible file type for PEmicro programmer usage. Figure 5-23 shows the data extracted from the CM2350 using PEmicro Cyclone.

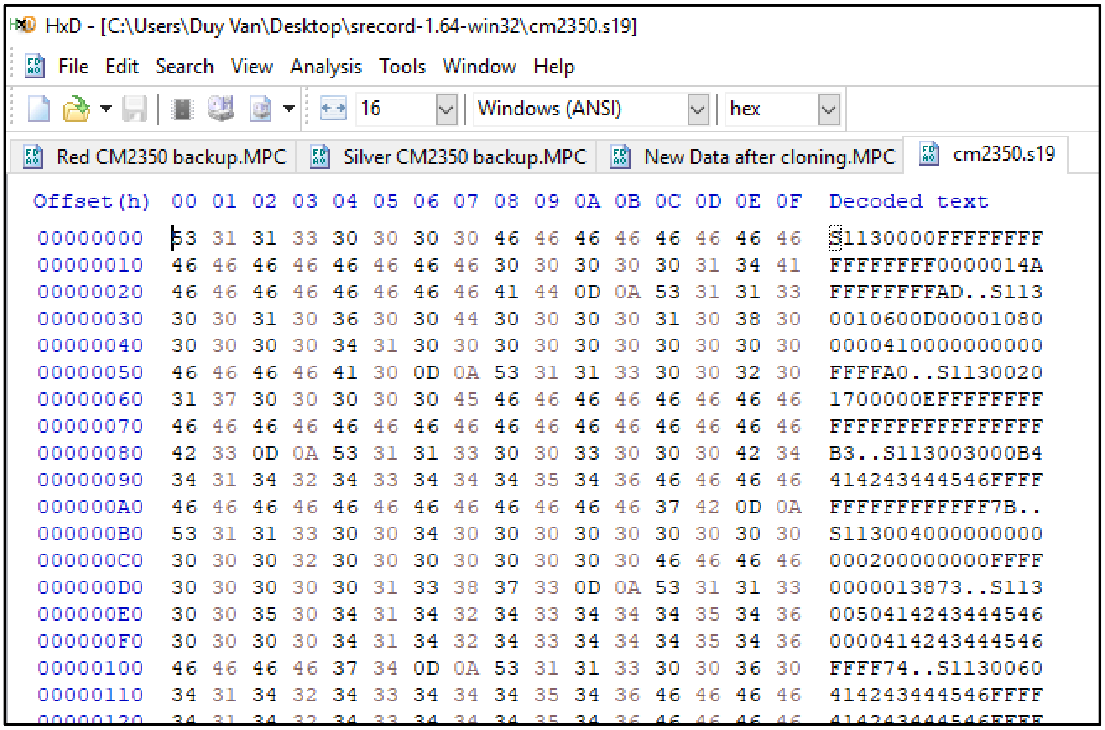


Figure 5-23. Data retrieved using PEmicro Cyclone in S19 file type shown in a hex editor

However, S19 records are not an easy format for data analysis. Thus, it is necessary to convert S19 records to raw binary using the `srec2bin.exe` program. Figure 5-24 shows the execution in the windows command prompt window where the S19 file from the CM2350 data extraction process is converted to BIN file.

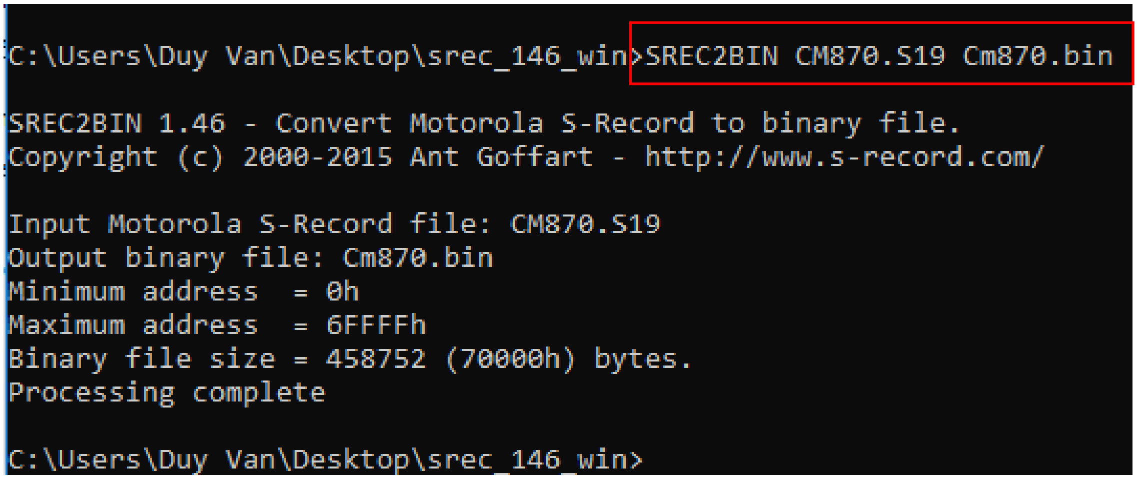


Figure 5-24. Execution command to convert S19 to BIN file

The binary data obtained using the PEmicro is then compared with the one obtained using the KTAG. The result is shown in Figure 5-25. Calculation of the SHA-256 digest on each record produces the same value, which indicates that the data extraction process using the PEmicro device or the KTAG are equivalent. Either method is appropriate and produces the same result for the flash memory of the CM2350.

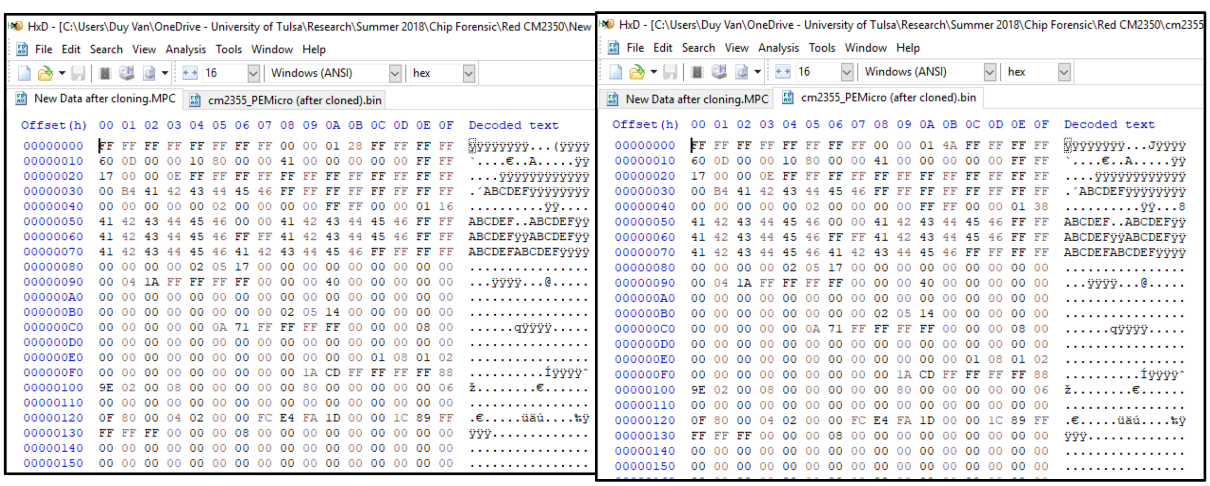


Figure 5-25. Comparison between binary data retrieved using KTAG (left) and PEmicro (right)

iii. Extraction Tool Comparison

The binary images from the KTAG came in three separate files for each ECU: 1) the microprocessor, 2) the EEPROM, 3) the flash memory. The PEmicro, however, would only produce the memory contents of the microprocessor. In the case of the CM2350, the flash is in the microprocessor, so the there is an opportunity to compare the binary image from the KTAG and the PEmicro. As expected, the binary images matched. This was determined by calculating the SHA-256 hash of each file and comparing the digests. Since the hashes match, the files are identical, bit by bit. The results of calculating the hash values for the files containing the binary images shows the tools provide the exact same results. This means only one tool is needed to perform the extraction.

It is important to note the PEmicro extraction does power on the module and some run-time counters will resume. Therefore, subsequent extractions will not have the same SHA-256 digest. The meaningful data, like the sudden deceleration records, do not change or have any differences that are dependent on the method or tool used for the binary data extraction. Even though these counters change, they will be reset back to the original image each time the data is reflashed from the original chip image.

iv. Forensic Soundness

The premise of creating a forensic image is to establish forensic soundness as originally described by McKemmish [112] and applied to HVEDRs by Johnson, et al. [113]. To summarize, forensic soundness has the following elements:

- **Meaning** is a term that denotes confidence in the interpretation of extracted evidence data
- **Error Detection** denotes processes for detecting or predicting errors in the forensic process

- **Transparency** means the forensic process is documented, known, and verifiable
- **Expertise** is required for investigators examining digital data
- **Tamper detection** involves processes to evaluate if data in the original record has been changed

The process for establishing a forensic image that has tamper detection is to calculate a cryptographic hash and associate it with the file. Ideally, the hash would be digitally signed to provide attestation for the hash and file. The signed hash should accompany the file in all transfers of information. With this digitally signed hash data and the original binary image, anyone can verify the digital signature and attest the investigator was the signatory and the file contents have not changed.

Cryptographic Hashing and a Digital Forensic Image

To create a forensic image of the chip, the memory contents read from the device needs to be hashed to provide a mathematical method to determine if the data remains unaltered. The strategy to achieve a forensic image is to compute the SHA-256 digest and include that hash digest with the original image. Since the SHA-256 algorithm is standard, it can be implemented many ways. For example, some free and open-source programs are available as add-ins for the file explorer in Windows' right-click menus.

Hashing the file is a manual step in the processes that needs to be performed immediately after the original image is acquired. The timeliness of the hash is critical to establish provable confidence of the authenticity of the data. Hash values should always be calculated before any analysis is performed. This reduces the likelihood of critique in manipulating the data to a specific end. If a forensic investigator has calculated the hash before any decoding, then any manipulation would be at random, since there would be no knowledge of the data. If a forensic investigator would prefer not to use the Python script, there are utilities available for Windows that compute cryptographic hash values. These "right-click" menu options can be discovered using an internet search engine.

The saved binary file is native machine code and does not have an application to open it. In this case, to view the raw binary (also known as hex codes), a hex editor is common. With the hex editor, the raw data can be viewed for analysis. Many hex editors, like HxD, have calculation tools. The example shown in Figure 5-26 displays a tool to calculate the SHA-256 hash digest. This digest should be archived before performing any analysis or subsequent data interpretation. An easy method to archive this hash is to send it, along with the original file, to a trusted e-mail account.

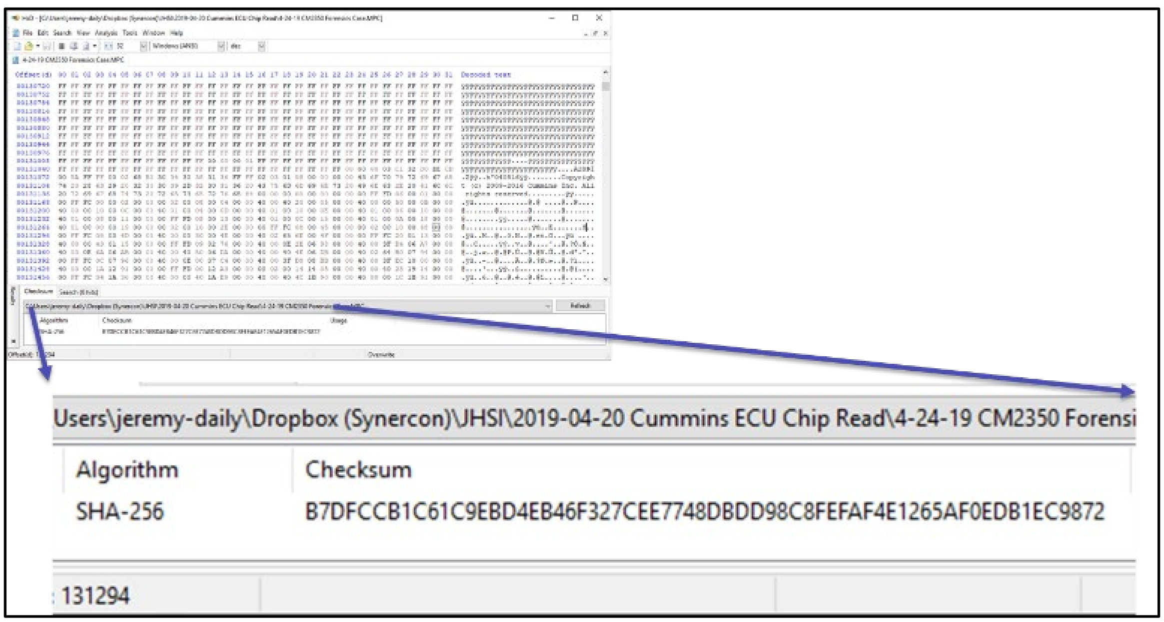


Figure 5-26. Example calculation of the SHA-256 cryptographic hash digest in the HxD hex editor

The process of computing and preserving a hash for the forensic image is the same regardless of the tool used to extract the data.

E. Data Decoding and Analysis

With the forensic image extracted and preserved, an analysis can be done to determine interesting data within the binary image. In a vehicle crash reconstruction, determining the data structures and values for the sudden deceleration record is of interest. To this end, we present the techniques and results of decoding the SDDR and the Data Plate. These pieces of data can provide some indication of an attributable event.

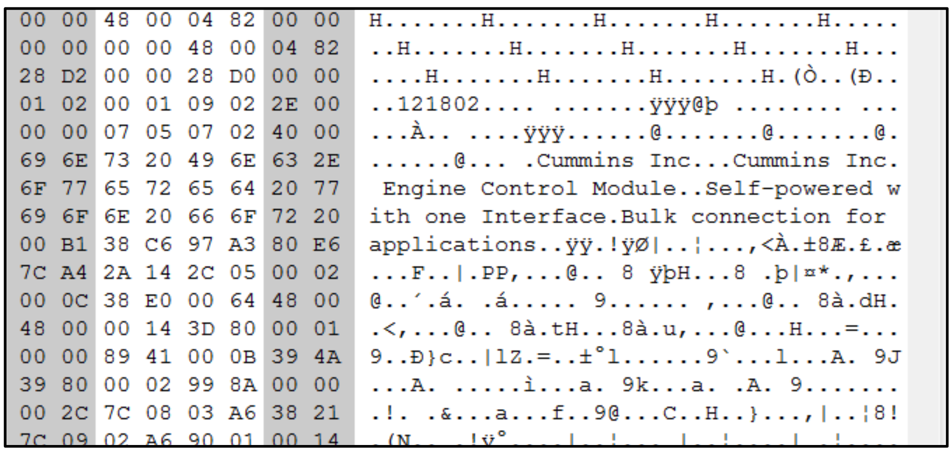


Figure 5-27. A binary image from the ECM showing some human readable snippets

The screenshot of the forensic image shown in Figure 5-27 reveals human readable data. This suggests the contents of the memory are not encrypted while stored in non-volatile, which is a prerequisite for performing a binary analysis.

i. Data Plate

The data plate contains descriptors of the engine and ECM. An example for the CM870 in this study is shown in Figure 5-28 with its corresponding data stored in the EEPROM shown in Figure 5-29.

Engine Dataplate Report

Engine Type	ISX 02	Ecm Code	AB10402.22
Engine Serial Number	79076145	Software Phase	6.5.4.2
Unit Number	25175	Extraction Date	04-02-2018 05:48:00

ECM Information

Module Name	CM870
Ecm Code	AB10402.22
Software Phase	6.5.4.2
ECM Serial Number	23052876
ECM Part Number	3683289

Engine Information

Engine Model	ISX 02
Engine Build Date	N/A
Engine Serial Number	79076145
Do Option	1325
SC Option	11145

Vehicle Information

Vehicle Identification Number (VIN)	4V4NC9TG25N391063
Vehicle or Equipment Year	
OEM Vehicle Equipment Model	STA15
Customer Name	central
Customer Location	utah
Vehicle Unit Number	25175

Figure 5-28. Data plate from the Cummins PowerSpec report for a CM870

```
........................u5..p...)..N¸.....
..............................^CMMNS.......
...........¶.1.¯¨!H...................FC0
P388..r.°......TS.83Ù._ÂL..AB10402.22
       ..+....-
        ..                      STA15          4
V4NC9TG25N391063                   central
  utah               25175      497294010049729
452003409167300497294340049728955004972882
60049729267004972951800
 .XÃnjdb@@00000000000000000000000000000..
... . .....d.!..... .........ì...........
```

Figure 5-29. ASCII decode contents from the EEPROM record found in the CM870

F. ECM Cloning: The Virtual Chip Swap

Physically moving a memory bearing chip from one ECM to another is often a challenging prospect. The process requires a steady hand or specialty tools. The 8-pin EEPROMs found in passenger vehicle airbag modules are easy examples of this technique. However, the high pin count and/or BGA chips make physically swapping the chips almost impossible.

Since the in-circuit programming techniques have both read and write capabilities, the data from the module under investigation can be read and written to a surrogate ECM. Since the binary image from the initial read is forensically preserved, with a SHA-256, it provides a canonical copy of the original ECM. This forensic image can be written to other ECMs many times over without compromising the integrity of the image. This can be useful when investigating fault codes and their origination.

The process of reading from one ECM and writing to another is called cloning. It is dubbed a virtual chip swap because only the data is moving, not the physical chip. The process makes a bit-for-bit copy of the flash memory from one chip to another.

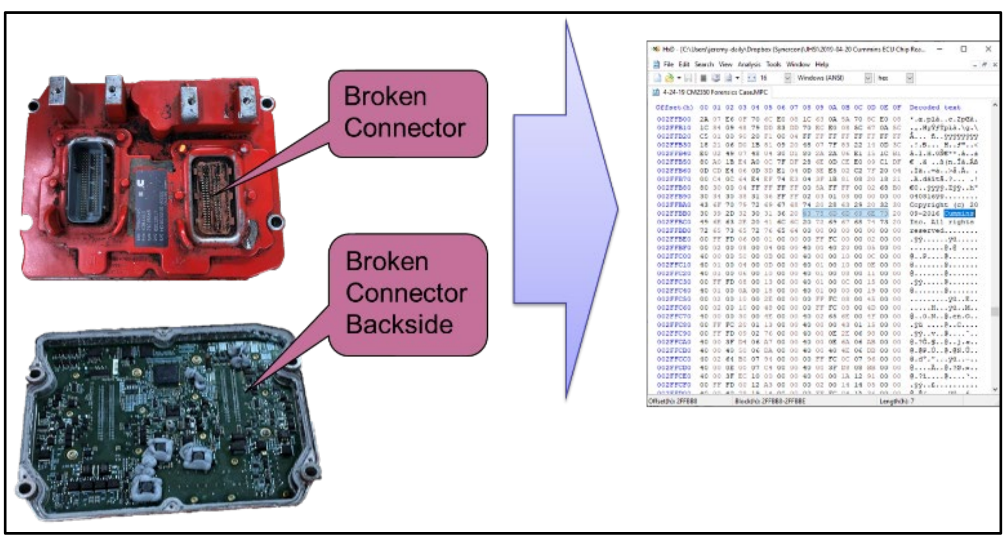


Figure 5-30. Extracting the binary record from the broken ECM

A visual representation of the chip cloning process is shown in Figure 5-30. The ECM images on the left of the figure show the front and back of a broken control module. The data is extracted and held as file contents. A screenshot of the data in a hex editor is representative of the data. Immediately after the binary image data is saved to a computer, the file contents need to be cryptographically hashed to demonstrate the contents were not altered.

The reverse process of taking a binary image and writing it to an ECU is shown in Figure 5-31. This figure shows the same binary that was extracted and hashed being imaged onto an exemplar ECM. The ECM depicted in the figure is connected to a vehicle emulation device capable of providing power, ignition, and CAN communications. From this device, a service tool can be used to download the data from the new ECM, but the data would be from the original ECM.

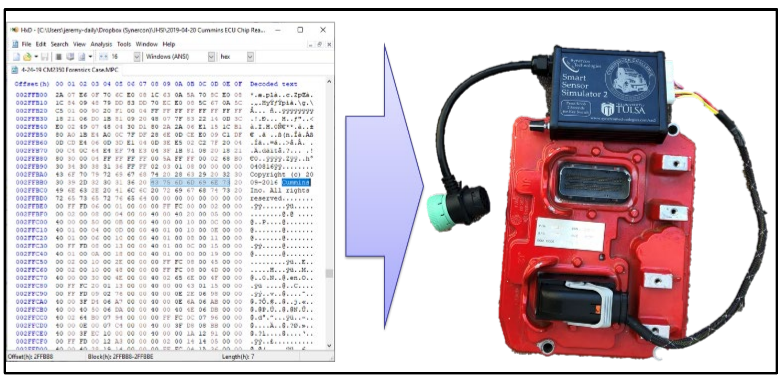


Figure 5-31. Uploading or 'flashing" an exemplar ECM with the forensic binary image of the broken ECM

The process to perform the chip cloning using the KTAG system is built into the programming software. As shown in Figure 5-32, the cloning function is an automated process activated by pressing the Clone ECU button.

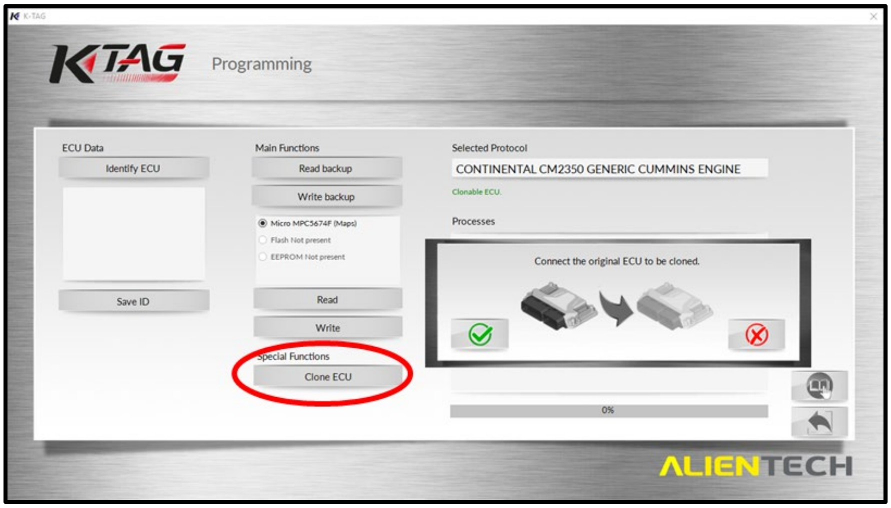


Figure 5-32. The *Clone ECM* function within the KTag software provides cloning capabilities

After following the on-screen instructions from the KTAG software, the data from the broken ECM will be on the new ECM. However, there are a few data elements that do not get transferred. A comparison of the Cummins PowerSpec Data Plate from two different ECMs is shown in Figure 5-33. The ECM data plate labeled as A is the surrogate ECM that will have its contents replaced. The VINs are called out to show the changes after the cloning process. ECM B is the ECM that has the data that needs to be transferred to the donor or surrogate ECM.

After the cloning process, the Data Plate information from the two ECMs were compared again, as shown in Figure 5-34. All the data from the surrogate ECM was replaced except the part number and serial number. This infers these data elements are stored in different memory bearing devices than the flash memory. Often processors have unique IDs that can be used for part tracking.

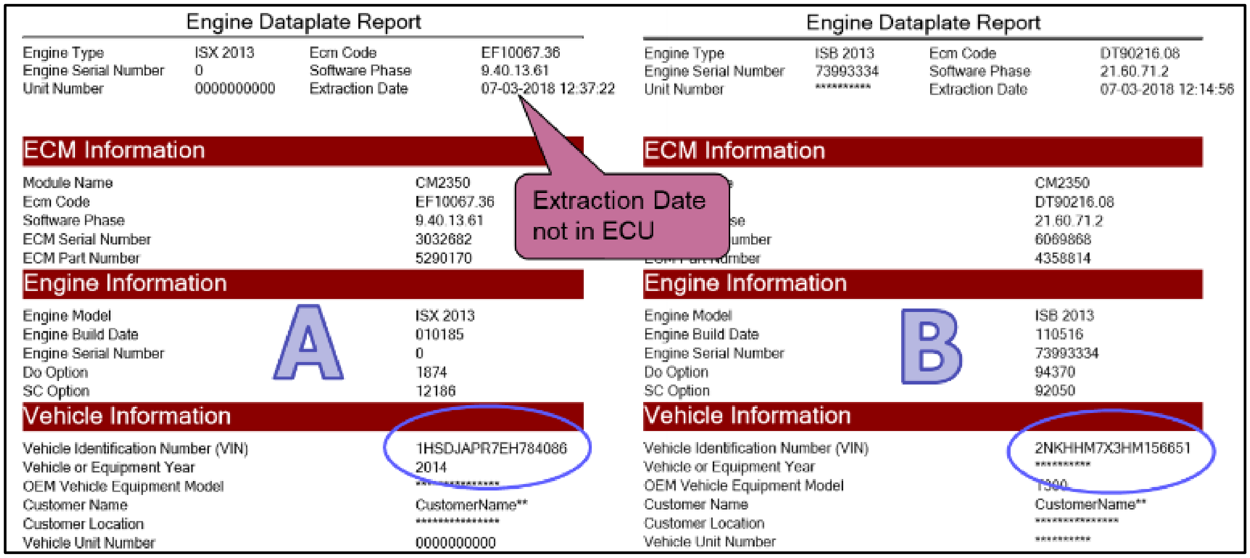


Figure 5-33. Data Plate from 2 different ECMs (A and B) before cloning. ECM B will be copied over to ECM A

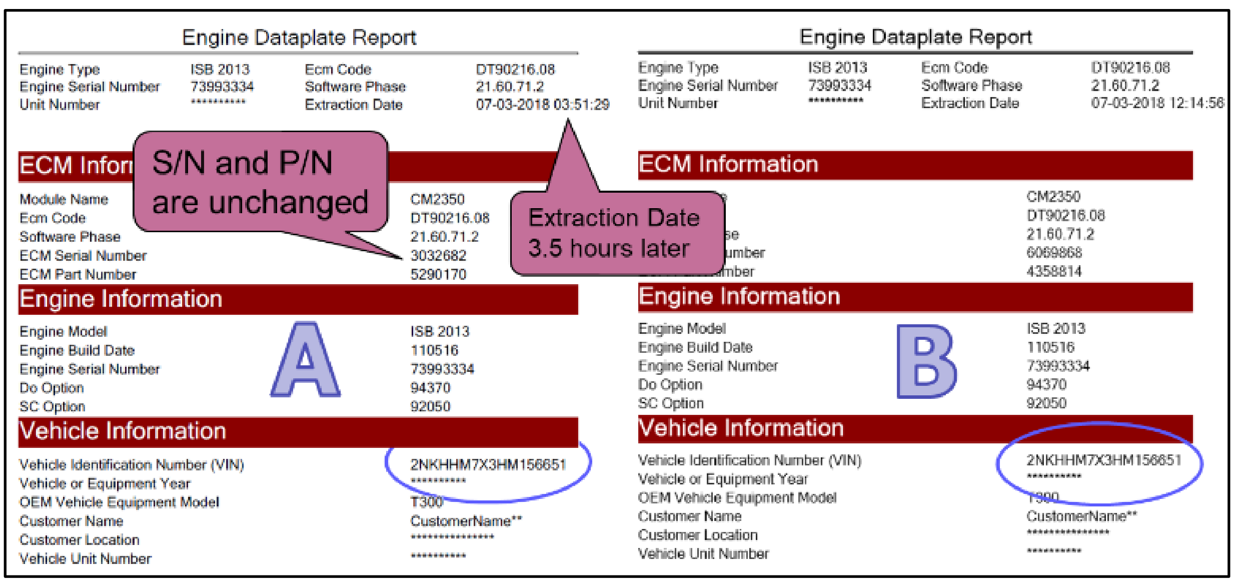


Figure 5-34. Data Plate information after the image from B is flashed onto A

Once the binary image has been transferred to the surrogate ECM, a traditional download procedure is enabled (assuming the surrogate ECU is in proper working order). During the cloning process, it is important to save the binary file and calculate the hash function of the device. This digital image can be used repeatedly to write to ECUs. Since the SHA hash digest is known, the image can also be verified before each use. One of the uses of the binary image is manual decoding where an investigator determines the meaning of the data in the flash memory as it relates to sudden deceleration events.

G. Decoding Sudden Deceleration Events

The data of interest in crash reconstructions are often associated with the sudden deceleration event where vehicle speed, engine speed, engine load, throttle position, brake, clutch, cruise and lamp status are tabulated for 75 seconds (60 seconds before a trigger and 15 seconds afterwards). An example table of partial data from the Cummins PowerSpec report is shown in Figure 5-35. The data shown in this figure is contrived because it was generated in a laboratory to have a speed of 150 mph, which is unrealistic. The data in the ECU was generated using a vehicle speed signal generator. The generator was a small microprocessor development board called the Teensy 3.2. It was programmed using the Arduino language to change frequency of a pulse width modulated signal set to a 50% duty cycle.

The initial challenge is to identify the hundreds bytes associated with the sudden deceleration record from the 3 million bytes in the binary record. This process requires some pattern matching within the records. Since we have a known repeated record of 150mph, we needed to determine how 150mph is encoded. Our first attempt was to follow the SAE J1939-71 recommendations where speed is encoded with 2 bytes and the least significant bit value is 1/256 km/h. However, this conversion did not work as there were no repeated patterns with the value. Instead, we tried to encode speed directly as 1/256 miles per hour per bit. This encoding scheme would suggest 150 mph would be encoded as 150*256 = 38,400. This decimal value represented as a little endian (Motorola format), 16-bit hexadecimal number is 0x9600. Searching for 0x9600 in the binary file resulted in the repeated pattern shown in Figure 5-36.

Record 1

Time (Seconds)	Vehicle Speed (mph)	Engine Speed (rpm)	Engine Load (%)	Throttle (%)	Brake Status	Clutch Status	Cruise Status	Lamp Status
-16	150	0	0.0	0.0	-	-	-	On
-15	150	0	0.0	0.0	-	-	-	On
-14	150	0	0.0	0.0	-	-	-	On
-13	150	0	0.0	0.0	-	-	-	On
-12	150	0	0.0	0.0	-	-	-	On
-11	150	0	0.0	0.0	-	-	-	On
-10	150	0	0.0	0.0	-	-	-	On
-9	150	0	0.0	0.0	-	-	-	On
-8	150	0	0.0	0.0	-	-	-	On
-7	150	0	0.0	0.0	-	-	-	On
-6	150	0	0.0	0.0	-	-	-	On
-5	150	0	0.0	0.0	-	-	-	On
-4	150	0	0.0	0.0	-	-	-	On
-3	150	0	0.0	0.0	-	-	-	On
-2	150	0	0.0	0.0	-	-	-	On
-1	150	0	0.0	0.0	-	-	-	On
0	150	0	0.0	0.0	-	-	-	On
1	89	0	0.0	0.0	-	-	-	On
2	6	0	0.0	0.0	-	-	-	On

Figure 5-35. Table produced by Cummins PowerSpec from a laboratory generated speed record

```
          00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0x0FB1D0  00 00 00 00 00 00 00 00 0C 31 00 00 00 00 00 00   ........ 1......
0x0FB1E0  00 00 00 73 00 02 1E FA 15 9B 85 8A 12 F6 01 D2   ...s.. ú.....ö.Ò
0x0FB1F0  79 DC 00 00 96 00 00 00 00 00 00 00 00 00 00 01   yÜ..............
0x0FB200  00 00 96 00 00 00 00 00 00 00 00 00 00 01 00 00   ................
0x0FB210  96 00 00 00 00 00 00 00 00 00 00 01 00 00 96 00   ................
0x0FB220  00 00 00 00 00 00 00 00 00 01 00 00 96 00 00 00   ................
0x0FB230  00 00 00 00 00 00 00 01 00 00 96 00 00 00 00 00   ................
0x0FB240  00 00 00 00 00 01 00 00 96 00 00 00 00 00 00 00   ................
0x0FB250  00 00 00 01 00 00 96 00 00 00 00 00 00 00 00 00   ................
0x0FB260  00 01 00 00 96 00 00 00 00 00 00 00 00 00 00 01   ................
0x0FB270  00 00 96 00 00 00 00 00 00 00 00 00 00 01 00 00   ................
0x0FB280  96 00 00 00 00 00 00 00 00 00 00 01 00 00 96 00   ................
0x0FB290  00 00 00 00 00 00 00 00 00 01 00 00 96 00 00 00   ................
0x0FB2A0  00 00 00 00 00 00 00 01 00 00 96 00 00 00 00 00   ................
0x0FB2B0  00 00 00 00 00 01 00 00 96 00 00 00 00 00 00 00   ................
0x0FB2C0  00 00 00 01 00 00 96 00 00 00 00 00 00 00 00 00   ................
0x0FB2D0  00 01 00 00 96 00 00 00 00 00 00 00 00 00 00 01   ................
0x0FB2E0  00 00 96 00 00 00 00 00 00 00 00 00 00 01 00 00   ................
0x0FB2F0  96 00 00 00 00 00 00 00 00 00 00 01 00 00 96 00   ................
0x0FB300  00 00 00 00 00 00 00 00 00 01 00 00 96 00 00 00   ................
0x0FB310  00 00 00 00 00 00 00 01 00 00 96 00 00 00 00 00   ................
```

Figure 5-36. A pattern showing the decoded 150mph record

To further test the theory of an encoding scheme where the LSB is 1/256 mph, the speed value coming off the 150 mph constant, which is 89 mph, as shown in Figure 5-35, is determined. Since we have a theory of the position for the speed record being every 14 bytes, we identified the likely position corresponding the 89 mph entry in the PowerSpec SDDR record. The value of 0x588A is highlighted in blue in Figure 5-37. The value of 0x588A is 22,666 in decimal. Using the conversion rate of 1/256 per mph, the resulting operation is 22,666/256 = 88.54 mph, which rounds to 89 mph, thus confirming the location and structure of the SDDR in memory.

```
0x0FB4C0 00 00 00 00 00 00 00 00 00 01 00 00 96 00 00 00  ................
0x0FB4D0 00 00 00 00 00 00 00 01 00 00 96 00 00 00 00 00  ................
0x0FB4E0 00 00 00 00 00 01 00 00 96 00 00 00 00 00 00 00  ................
0x0FB4F0 00 00 00 01 00 00 96 00 00 00 00 00 00 00 00 00  ................
0x0FB500 00 01 00 00 96 00 00 00 00 00 00 00 00 00 00 01  ................
0x0FB510 00 00 96 00 00 00 00 00 00 00 00 00 00 01 00 00  ................
0x0FB520 96 00 00 00 00 00 00 00 00 00 00 01 00 00 96 00  ................
0x0FB530 00 00 00 00 00 00 00 00 00 01 00 00 58 8A 00 00  ............X...
0x0FB540 00 00 00 00 00 00 00 01 00 00 06 58 00 00 00 00  ...........X....
0x0FB550 00 00 00 00 00 01 00 00 00 74 00 00 00 00 00 00  .........t......
0x0FB560 00 00 00 01 00 00 00 08 00 00 00 00 00 00 00 00  ................
0x0FB570 00 01 00 00 00 01 00 00 00 00 00 00 00 00 00 01  ................
```

Figure 5-37. Confirming a speed record of 89mph

Using a similar strategy for the changing parameters within the SDDR records, a decoding scheme was determined. The results of the binary positions and subsequent conversions are shown in Figure 5-38 with a confirming example from the SDDR record in PowerSpec. However, there are still switch parameters represented by single bits that are constant through the PowerSpec records. Since there are no unique changes, we need to find or generate a new record with changing switch values. To do this, a method called middle-person attach was used to create arbitrary PowerSpec records. But first, we need to understand the J1939 network traffic for the Sudden Deceleration Records.

Byte	3 & 4	5 & 6	7 & 8	9 & 10
Hex	07 BE	26 98	00 99	17 84
Convert to Decimal	1,982	9,880	153	6020
Resolution	1/256 mph/bit	1/8 RPM/bit	1/4 %/bit	1/256 %/bit
Actual Number	7.74 mph	1,235 RPM	38.25%	23.52%
	Vehicle Speed	Engine Speed	Throttle	Engine Load

Record 2

Time (Seconds)	Vehicle Speed (mph)	Engine Speed (rpm)	Engine Load (%)	Throttle (%)	Brake Status	Clutch Status	Cruise Status	Lamp Status
11	6	840	14.6	32.3	-	On	-	-
12	6	1071	38.6	45.5	-	-	-	-
13	8	1235	23.5	38.3	-	-	-	-
14	8	1287	0.0	8.0	-	-	-	-
15	8	952	0.0	22.8	-	On	-	-

Figure 5-38. Decoding results for variable parameters in the PowerSpec records

H. CAN Data Analysis

Understanding the vehicle network and diagnostic communications is necessary for being able to solve the mystery of the switch bits. The first step is to record the data using the CAN Logger. A photograph of the inside of the CAN Logger 2 used for this research is shown in the upper left of Figure 5-39 and the assembly of the CAN Logger 2 and its corresponding cable is shown in the remainder of the figure.

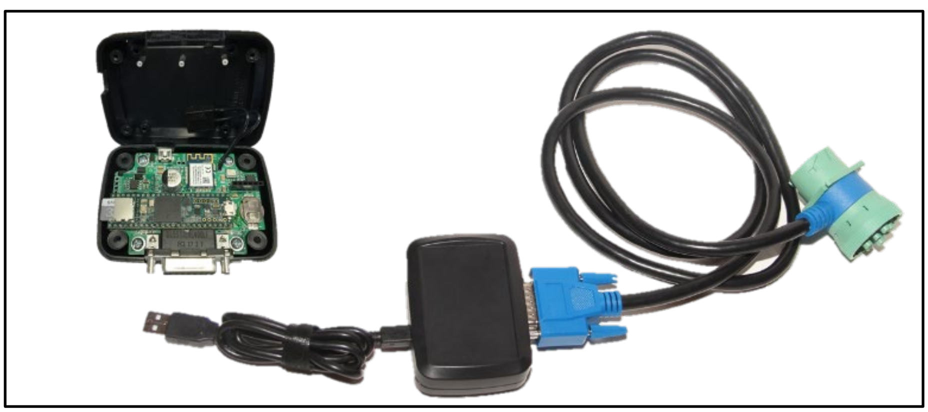


Figure 5-39. CAN Logger 2 to compare binary image with CAN network traffic

The CAN Logger 2 is capable of logging network traffic to an SD card without missing frames. Based on the work in [111], it was hypothesized that the memory contents will be reflected in the CAN traffic. Therefore, a section of memory corresponding to a SDDR record was identified as shown in Figure 5-40. The goal was to find this memory record in the CAN traffic. Since CAN messages are limited to 8-byte frames, a transport protocol as defined in SAE J1939-21 Data Link Layer [2] is used to transfer memory contents over CAN. This means the messages for data transport will carry 7 bytes per frame of memory contents with the first byte used as an indexing counter to ensure ordered delivery. The data identified in Figure 5-41 show the messages with the same contents as the ones in memory shown in Figure 5-40.

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B
000F31BC	00	00	00	00	00	00	00	00	00	00	00	00
000F31C8	00	00	00	00	00	00	00	00	00	00	00	00
000F31D4	00	00	00	00	0C	31	00	00	00	00	00	00
000F31E0	00	00	00	73	00	02	1E	FA	15	9B	85	8A
000F31EC	12	F6	01	D2	79	DC	00	00	96	00	00	00
000F31F8	00	00	00	00	00	00	00	01	00	00	96	00
000F3204	00	00	00	00	00	00	00	00	00	01	00	00
000F3210	96	00	00	00	00	00	00	00	00	00	00	01
000F321C	00	00	96	00	00	00	00	00	00	00	00	00
000F3228	00	01	00	00	96	00	00	00	00	00	00	00
000F3234	00	00	00	01	00	00	96	00	00	00	00	00
000F3240	00	00	00	00	00	01	00	00	96	00	00	00
000F324C	00	00	00	00	00	00	00	01	00	00	96	00

Figure 5-40. Raw data from the binary image

1479051111.967139	can0	18EBFF00	08	04	AF	00	03	03	6C	00
1479051111.976171	can0	18EBFA00	01	45	1A	A0	00	00	04	26
1479051111.986092	can0	18EBFA00	02	1E	FA	15	9B	85	8A	12
1479051111.996095	can0	18EBFA00	03	F6	01	D2	79	DC	00	00
1479051112.006143	can0	18EBFA00	04	96	00	00	00	00	00	00
1479051112.016150	can0	18EBFA00	05	00	00	00	00	01	00	00
1479051112.026145	can0	18EBFA00	06	96	00	00	00	00	00	00
1479051112.026713	can0	18EBFF00	09	04	01	66	00	04	01	94
1479051112.036134	can0	18EBFA00	07	00	00	00	00	01	00	00
1479051112.046148	can0	18EBFA00	08	96	00	00	00	00	00	00
1479051112.056151	can0	18EBFA00	09	00	00	00	00	01	00	00
1479051112.066164	can0	18EBFA00	0A	96	00	00	00	00	00	00
1479051112.076132	can0	18EBFA00	0B	00	00	00	00	01	00	00
1479051112.086140	can0	18EBFA00	0C	96	00	00	00	00	00	00
1479051112.086713	can0	18EBFF00	0A	04	03	01	B9	04	04	03
1479051112.096128	can0	18EBFA00	0D	00	00	00	00	01	00	00
1479051112.106137	can0	18EBFA00	0E	96	00	00	00	00	00	00
1479051112.116171	can0	18EBFA00	0F	00	00	00	00	01	00	00

Figure 5-41. CAN Traffic from the log file matching the data in the binary image from a direct chip read

With the understanding of how the network traffic carries the SDDR records, we can insert a device to manipulate the data in transit to set switch events in the records. From these manipulated records, the encoding for the switches can be determined.

i. Middle person Manipulation

A middle person is a hardware device that breaks the direct link from one node to another. In this case the direct link is the CAN bus from the ECM to the vehicle diagnostics adapter. If this link is broken with a man-in-the-middle, then the data on the ECM side of the link can be changed as it is transferred over to the VDA side of the link. This means the middle person hardware needs to have 2 CAN channels, one for each link. Since the CAN Logger 2 has these channels, it was reprogrammed to be a middle person.

The setup for the middle person is shown in Figure 5-42. The device in the upper left of Figure 5-42 is the Smart Sensor Simulator 2, which is connected to the ECM (not shown). The SSS2 creates a truck-like network and a 9-pin diagnostic port. The other link shown in Figure 5-

42 is the Cummins Inline 7 vehicle diagnostics adapter. The middle person is the CAN Logger 2 with an adapter cable that breaks the direct link. This cable has the J1939 traffic from the ECM connected with channel 1 on the CAN Logger 2 and Channel 2 is connected to the VDA. Thus, the processor in the CAN Logger 2 is the man in the middle.

The logic programmed into the middle person looks for the pattern associated with the SDDR events. Once it detects a CAN frame containing the switch data is to be forwarded to the VDA, it manipulates the data frame to set a switch bit. The corresponding PowerSpec report is acquired and the changed switch status is located. From this change, the encoding scheme can be determined.

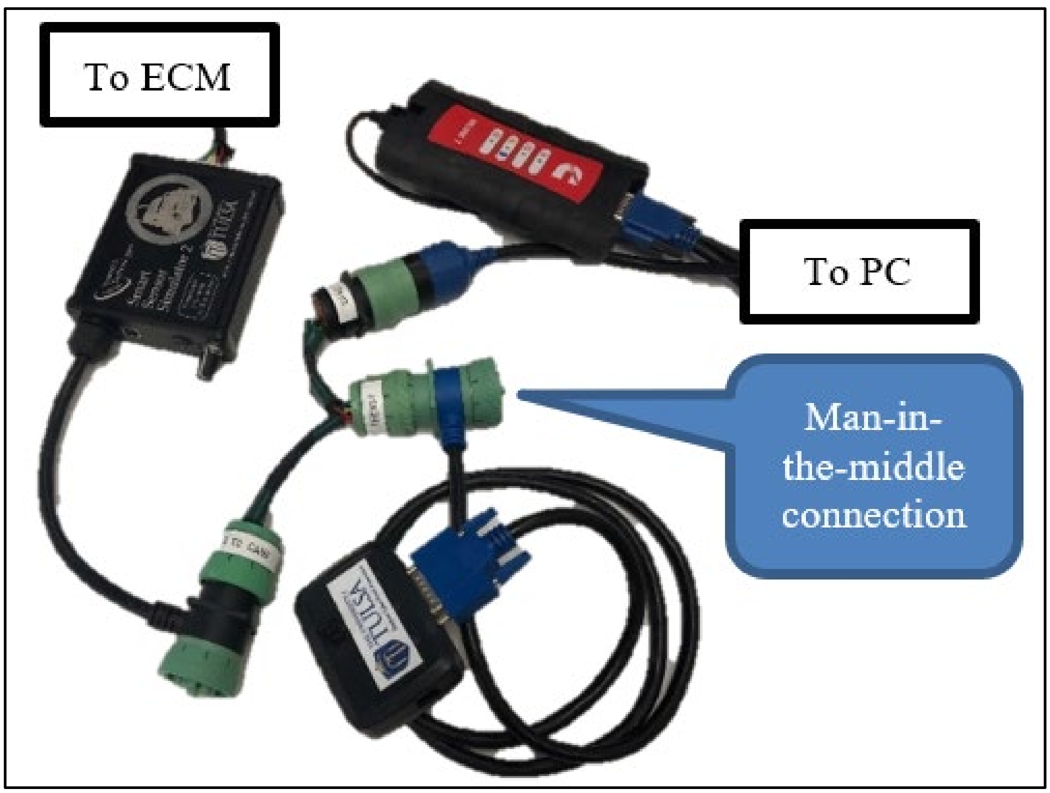


Figure 5-42. Using the CAN Logger 2 as a Man-in-the-Middle

ii. Decoded Results

After mapping the binary switch data to engineering results (i.e. translating the data), a routine was built to decode all the switch data in the SDDR. The results of the decoding from the example is shown in Figure 5-43.

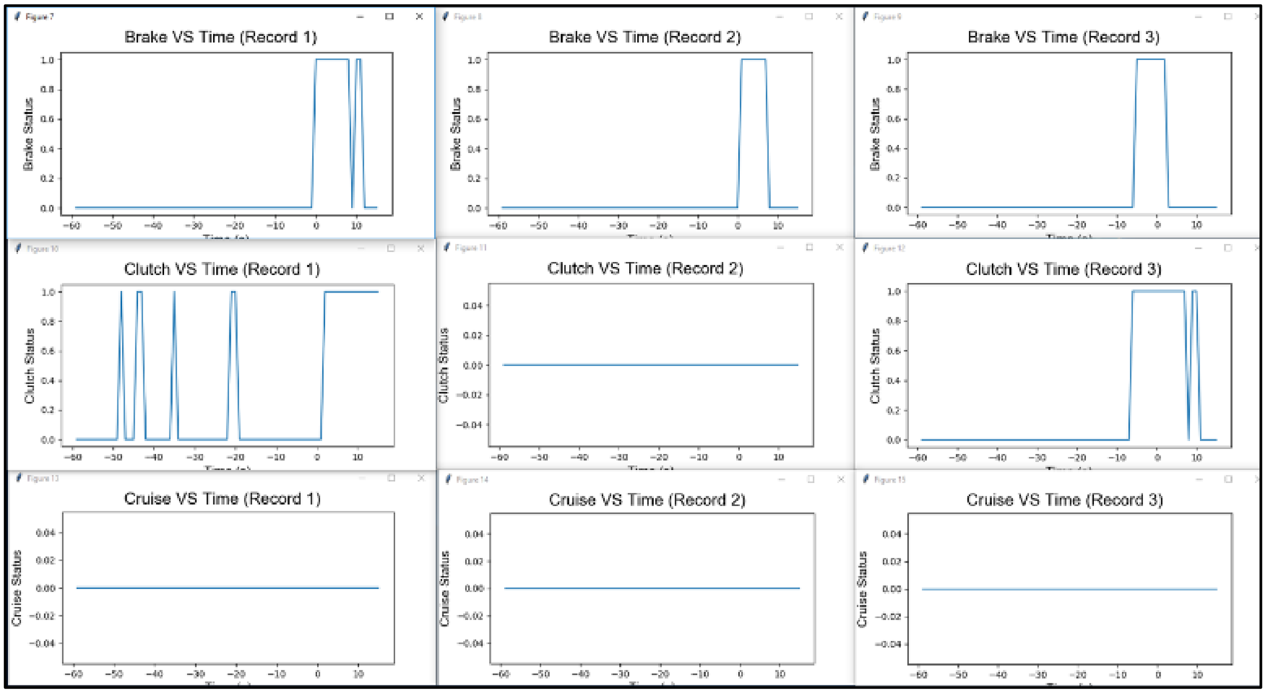


Figure 5-43. Results of decoding the switch states for all three available sudden deceleration records

With the complete key to each line of the table in the SDDR records, we can build a tool to create a custom version of the Cummins PowerSpec report generator. The results of the Python-based implementation is shown in Figure 5-44.

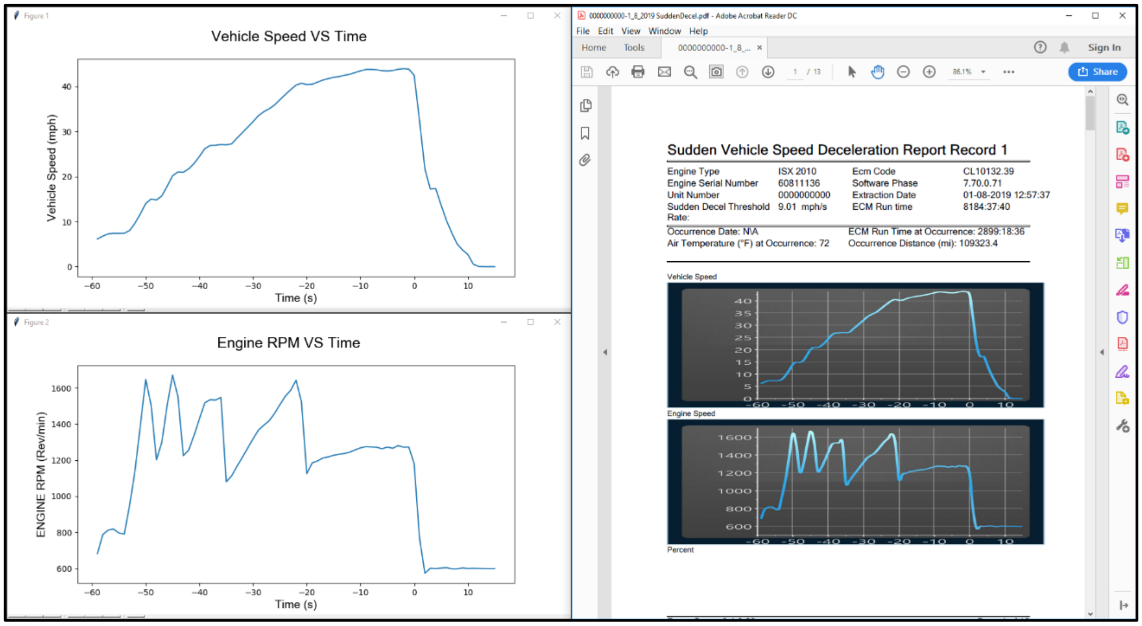


Figure 5-44. Comparison of the custom decoding of the vehicle speed and engine speed records compared to the Cummins PowerSpec report

I. Discussion

The methods presented in this paper for the CM870, CM871, and CM2350 may only apply specifically to the ECMs tested. It is not uncommon for manufacturers to introduce revisions to circuit boards during production. Chip part numbers or configurations may be different in other CM870/871/2350 ECMs with different circuit board revisions.

The virtual chip swap methodology may not be a replacement for conventional chip swaps in all cases. Damage to the ECM that interrupts the connections between the JTAG header and the processor may prevent successful communication. This could include localized impact damage in the processor region or severe burning that consumes the circuit board.

The method of binary chip imaging may be the best way to preserve the fault code data in an ECU. Since the startup routines do not take place, the ECU is not initialized and does not perform its diagnostic routines. This keeps the fault codes preserved, with an ability to reset an ECU with a forensically sound digital image.

If the ECU is cloned, the new ECU with the data of interest can be powered back on to do a download. Many times, the investigator may not want to set new fault codes. When performing a download on the cloned ECU, there are 2 methods to try to achieve a fault free environment: 1) connect the ECU into a surrogate vehicle in good operating order or 2) connect the ECU to a bench harness with a complete simulation of the fault free environment. Each of these strategies is challenging because fault codes can arise from configuration differences between the surrogate vehicle/bench harness and the ECU with the new data. For example, a bench harness would need to have a CAN based Variable Geometry Turbo (VGT) that matched the ECU in question. If the programming on the VGT is different, a new fault code may be introduced.

With the technique explained in this research, the vehicle/bench harness can be updated to address new fault codes. Once the fault code is mitigated, then the ECU can be flashed back to its original state using the forensically sound ECU binary image. The process can be repeated until there are no new fault codes present. Furthermore, forensic investigators can examine the differences in the binary images to see presence and preservation of the fault codes. The direct interpretation of the fault codes from the binary image is out of scope for this research. However, preliminary analysis suggests a more complicated linked data structure with contributions to fault code data scattered throughout the binary records. Nevertheless, the work presented herein provides a fundamentally sound method to drastically reduce the risk of data spoliation through the creation of additional fault codes.

J. Chapter Conclusions

A forensically sound method for imaging, preserving, and analyzing data from a Cummins CM870/871/2350 electronic control module was explained in detail. When a Cummins-powered heavy vehicle has damage to the electronic control module containing the heavy vehicle event data recorder, data may not be able to be extracted using vehicle diagnostics tools. Invasive and low-level techniques for extracting non-volatile memory contents were described that use board level techniques with the available JTAG port. Additional chip level data extraction techniques can also provide access to the data through a chip reader.

Once these data are obtained and preserved in a forensically sound manner, the binary record was decoded and presented to show typical HVDER data, like engine speed, vehicle speed, accelerator pedal position, and other status flag data.

The memory contents from the ECM can be written to a surrogate module. The data from this surrogate module can be downloaded and decoded with traditional maintenance and diagnostic software. This was described as a virtual chip swap.

The research also shows the ECM is not turned on during the binary imaging. Therefore, diagnostic trouble codes from the ECM are preserved in its as-found state. In other words, the digital forensic technique of extracting memory contents through the JTAG port does not introduce any new fault codes.

Cryptographic hashing of the forensic data provides a mechanism to verify the original digital forensic record, which makes the technique presented forensically sound. Finally, the decoding for the HVEDR binary record was presented so investigators can decode the forensic record without the need to a surrogate ECM. The techniques in this chapter provide a new method for extracting data from heavy vehicle ECMs.

Chapter 6. Thesis Conclusions

A. Abstract Restatement

A large database of CAN network traffic from operational heavy trucks is a beneficial resource that gives the trucking industry a better understanding of those vehicles' cybersecurity aspects. An intrusion detection algorithm can be developed based on the database to protect heavy trucks from potential cybersecurity threats. Therefore, an affordable CAN logger device was designed to gather CAN data. Moreover, the device must also be secure for the CAN logging process to prevent the log data information and integrity from being compromised. Thus, encryption and digital signature were introduced and implemented in the CAN logger design.

Practical encryption is an important tool in improving the cybersecurity posture of data loggers and engineering tools by providing confidentiality. Implementations of symmetric and asymmetric algorithms were used to perform envelope encryption of session keys with symmetric encryption algorithms. Maintaining determinism and minimizing latency are primary considerations when implementing a cryptographic solution in an embedded system. To satisfy the stringent requirements for truck systems, the mmCAU on the NXP K66 processor found on the Teensy 3.6 development board was evaluated for potential use in heavy vehicles. Results show the K66 mmCAU can encrypt heavy vehicle CAN traffic at a rate over 6 Mbps per second and, along with the CAN Logger 3, successfully log all the data at 100% bus load. Using AES-128 in CBC mode, the log data is encrypted in a manner such that the overall data is encrypted instead of each code block being individually scrambled as in the ECB mode. AES CBC mode provides high entropy encryption for data confidentiality; however, it does not include error and integrity checks. The error verification of the encrypted buffer is handled by a CRC checksum and the integrity check uses an Elliptic Curve Digital Signing Algorithm (ECDSA). An ATECC608A HSM is explored and utilized to securely store key pairs for key management and implement an ECC algorithm to sign the data for integrity verification. The ATECC608A is also used to generate shared secret keys using ECDH for secure data storage and management. Secure collection and secure data transportation to a central server are crucial areas of focus for a practical cybersecurity implementation.

B. Contribution Restatement

The CAN logging project has gathered a significant amount of heavy truck CAN traffic with more than 11 billion messages for the database, and more data is still being collected. Moreover, a CAN logger device with an AWS cloud system has been designed for the project to provide secure data collection and storage by implementing cybersecurity measures following the industry standards. There is also a user-friendly client application GUI for users to manage their data between the device and the AWS server. The log data from the project can only be accessed by its owner and the project administrators; however, the CAN logging project hardware and source codes are made available to the trucking industry as well as the public with the hope that it can be applied to increase cybersecurity posture in heavy vehicles, and its documentation can be found on the GitHub repository [23].

Cyber-physical system security, as a field of study, is in its infancy. This thesis represents a concrete example of designing an entire data logging system (i.e. device, front-end and back-end) with cybersecurity as a primary objective. The CAN Logger 3 project demonstrates the economics and feasibility of incorporating cybersecurity as a design requirement. This body of work should be useful for inspiring future designs that incorporate CAN bus, hardware security modules, and system level communications.

C. Future Work

Even though some features are not within the current project scope, they are still needed to be tested to completely validate the CAN logger design. On the other hand, there are ideas that can be implemented to improve the CAN logging project design. The following list describes some possibilities for future work:

- The Single-Wire CAN feature needs to be validated.
- The LIN feature needs to be validated.
- The CAN2 feature needs to be fully implemented in the firmware by utilizing the ACAN2517 library [85].
- The J1708 feature needs to be fully implemented in the firmware.
- J1708 and CAN2 needs to be detected automatically for multiplexing by using the REC from CAN2, which is similar to the autobaud feature.
- The WiFi feature can be implemented where data stored on the CAN logger can be transferred to the local computer client application wirelessly for convenience. WPA2 should be used in this application for security.
- With the current configuration, the server public key stored in the ATECC608A HSM is validated through the provisioning process and cannot be changed after that. Therefore, this could be costly to revoke that public key because it requires the HSM to be replaced. Public key certificates can be added to the design using the ATECC608A HSM memory slot to efficiently validate the server public key.

- An RSA asymmetric encryption scheme can be implemented in such a way that the CAN logger uses a server's RSA 2048-bit public key stored on the ATECC608A HSM to encrypt AES session keys, which will be decrypted using the corresponding RSA private key residing on the server. By replacing the ECDH shared secret with this method, the design can be improved by reducing the number of keys that the system has to manage because the ECDH shared secret key is eliminated. Moreover, it is also more difficult for attackers to hack the CAN logger if they have the physical device because it will require more resources to crack the RSA encryption than reverse-engineering the device and exploiting the ECDH shared secret function on the device. For the current configuration, there is a low risk that the device is physically compromised because the users are trusted to keep their devices secure. However, if a device is stolen and hacked, critical information from only that device is exposed, and the administrators can revoke it from the server without negatively affecting the entire system.

REFERENCES

[1] NMFTA, "A Survey of Heavy Vehicle Cyber Security", September 21st , 2015, updated Jan. 4, 2016, http://www.nmfta.org/pages/HVCS

[2] SAE, J1939-21 Data Link Layer Standard, October 3rd, 2018, https://www.sae.org/standards/content/j1939/21_201810/

[3] SAE, J1939 Standards Collection on the Web, https://www.sae.org/standardsdev/groundvehicle/j1939a.html

[4] FMCSA, "2019 Pocket Guide to Large Truck and Bus Statistics", January 9th, 2020, https://www.fmcsa.dot.gov/safety/data-and-statistics/2019-pocket-guide-large-truck-and-bus-statistics

[5] Dr. Charlie Miller and Chris Valasek, "Remote Exploitation of an Unaltered Passenger Vehicle", Aug. 10th, 2015, "http://illmatics.com/Remote%20Car%20Hacking.pdf"

[6] Subhojeet Mukherjee, Hossein Shirazi, Indrakshi Ray, Jeremy Daily and Rose Gamble, "Practical Denial-of-service Attacks in Embedded Networks of Commercial Vehicles," Proceedings of 12th International Conference on Information Systems Security (ICISS), 2016

[7] Kyong-Tak Cho and Kang G. Shin, "Error Handling of In-vehicle Networks Makes Them Vulnerable", Proceedings of the 2016 ACM SIGSAC Conference on Compute and Communications Security:1044-1055, Vienna, Austria-Octoboer 24-28,2016, doi:10.1145/2976749.2978302

[8] A. Shaout, D. Mysuru and K. Raghupathy, "CAN Sniffing for Vehicle Condition, Driver Behavior Analysis and Data Logging," 2018 International Arab Conference on Information Technology (ACIT), Werdanye, Lebanon, 2018, pp. 1-6

[9] Mathias Johanson and Lennart Karlsson, "Improving Vehicle Diagnostics through Wireless Data Collection and Statistical Analysis", Vehicular Technology Conference, 2007. VTC-2007 fall. 2007 IEEE 66th Vehicular Technology Conference

[10] Daily, J., DiSogra, M., and Van, D., "Chip and Board Level Digital Forensics of Cummins Heavy Vehicle Event Data Recorders," SAE Technical Paper 2020-01-1326, 2020

[11] DG Technologies, "DG Technologies Product Pinouts and Industry Connectors Reference Guide", April 23rd, 2014, https://www.dgtech.com/wp-content/uploads/2016/04/Pinouts_ICR.pdf

[12] Amazon, SAE J1939 Type-1 to Type-2 Male to Female 9 Pin Adapter, https://www.amazon.com/Dalagoo-Adapter-Connector-Diagnostic-Trackers/dp/B07QWD2CLN

[13] PEAK System, PCAN-USB, https://www.peak-system.com/PCAN-USB.199.0.html?&L=1

[14] Intrepid Control Systems, Value CAN 4, https://www.intrepidcs.com/products/vehicle-network-adapters/valuecan-4/

[15] Vector, CANlog, https://www.vector.com/int/en/products/products-a-z/hardware/canlog/

[16] DG Technologies, DPA-5 Kit, https://www.dgtech.com/store/dpa-5-kit.html

[17] Nexiq, USB-Link 2, https://www.nexiq.com/Product/Detail/124032

[18] Cummins, Cummins INLINE 7, https://shopcummins.ca/products/cummins-inline-7-datalink-adapter?variant=5655481483295

[19] CSS Electronics, CANedge2, https://www.csselectronics.com/screen/product/can-lin-logger-wifi-canedge2

[20] GitHub, NMFTA CAN Logger Repository, https://github.com/Heavy-Vehicle-Networking-At-U-Tulsa/NMFTA-CAN-Logger

[21] GitHub, NMFTA CAN Logger photo, https://github.com/Heavy-Vehicle-Networking-At-U-Tulsa/NMFTA-CAN-Logger/blob/master/docs/AssembledPrototype.jpg

[22] GitHub, CAN Logger 2 Repository, https://github.com/Heavy-Vehicle-Networking-At-U-Tulsa/CAN-Logger-2

[23] GitHub, CAN Logger 3 Repository, https://github.com/SystemsCyber/CAN-Logger-3

[24] NXP, K66 Sub-Family Reference Manual, K66P144M180SF5RMV2, Rev. 4, August 2018, https://www.nxp.com/docs/en/reference-manual/K66P144M180SF5RMV2.pdf

[25] GitHub, FlexCAN Library Repository, https://github.com/Heavy-Vehicle-Networking-At-U-Tulsa/FlexCAN_Library

[26] PJRC, Teensy Schematic, https://www.pjrc.com/teensy/schematic.html

[27] Microchip, MCP2517FD CAN Controller Datasheet, http://ww1.microchip.com/downloads/en/DeviceDoc/MCP2517FD-External-CAN-FD-Controller-with-SPI-Interface-20005688B.pdf

[28] Electronics Tutorials, Oscillator, https://www.electronics-tutorials.ws/oscillator/crystal.html

[29] Learning about Electronics, Bypass Capacitor, http://www.learningaboutelectronics.com/Articles/What-is-a-bypass-capacitor.html

[30] ISO, ISO 11898-1:2015 Road vehicles — Controller area network (CAN) — Part 1: Data link layer and physical signaling, December 2015, https://www.iso.org/standard/63648.html

[31] Microchip, MCP2557FD/8FD CAN FD Transceiver Datasheet, https://www.mouser.com/datasheet/2/268/20005533A-1067161.pdf

[32] ISO, ISO 11898-2:2016 Road vehicles — Controller area network (CAN) — Part 2: High-speed medium access unit, December 2016, https://www.iso.org/standard/67244.html

[33] Diodes Incorporated, 74AHCT1G14 Inverter Datasheet, https://www.diodes.com/assets/Datasheets/74AHCT1G14.pdf

[34] Texas Instruments, SNx5HVD1x 3.3V Transceiver Datasheet, http://www.ti.com/lit/ds/symlink/sn75hvd12.pdf

[35] GitHub, Smart Sensor Simulator 2 Repository, https://github.com/SystemsCyber/SSS2

[36] Texas Instruments, AN-915 Automotive Physical Layer SAE J1708, http://www.ti.com/lit/an/snla038b/snla038b.pdf

[37] KEMET, EC2/EE2 Miniature Signal Relays Datasheet, https://content.kemet.com/datasheets/KEM_R7002_EC2_EE2.pdf

[38] Analog Devices, ADG1633/ADG1634 Switches Datasheet, https://www.analog.com/media/en/technical-documentation/data-sheets/ADG1633_1634.pdf

[39] ON Semiconductor, NUD3124, SZNUD3124 Automotive Inductive Load Drive Datasheet, https://www.onsemi.com/pub/Collateral/NUD3124-D.PDF

[40] Electronics Hub, Flyback Diode, April 17, 2019, https://www.electronicshub.org/flyback-diode-or-freewheeling-diode/

[41] Microchip, MCP2003/4/3A/4A LIN J2602 Transceiver Datasheet, http://ww1.microchip.com/downloads/en/devicedoc/20002230g.pdf

[42] ON Semiconductor, NCV7356 Single Wire CAN Transceiver Datasheet, https://www.onsemi.com/pub/Collateral/NCV7356-D.PDF

[43] Microchip, ATECC608A Hardware Security Module Datasheet, http://ww1.microchip.com/downloads/en/DeviceDoc/40001977A.pdf

[44] Sparkfun, I2C at the Hardware Level, https://learn.sparkfun.com/tutorials/i2c/i2c-at-the-hardware-level

[45] Amiruddin Amir and Rifi Sari, "Selecting key generating elliptic curves for Privacy Preserving Association Rule Mining (PPARM)", 2015, 10.1109/APWiMob.2015.7374930

[46] Andre Groll and Christoph Ruland, "Secure and Authentic Communication on Existing In-Vehicle Networks", 2009, 1093 - 1097. 10.1109/IVS.2009.5164434

[47] Atmel, WINC1500 WiFi Module Datasheet, http://ww1.microchip.com/downloads/en/devicedoc/atmel-42376-smartconnect-winc1500-mr210pa_datasheet.pdf

[48] AVX, AV Multilayer Ceramic Transient Voltage Suppressors Datasheet, http://datasheets.avx.com/TransFeedGeneral.pdf

[49] Bel, Surface Mount PTC 0ZCG Series Datasheet, https://www.belfuse.com/resources/datasheets/circuitprotection/ds-cp-0zcg-series.pdf

[50] Comchip, ACURA107-HF Schottky Diode Datasheet, https://www.mouser.com/datasheet/2/80/ACURA107-HF_RevA970010-1480717.pdf

[51] Littlelfuse, SMA6J TVS Diodes Series Datasheet, https://www.littelfuse.com/~/media/electronics/datasheets/tvs_diodes/littelfuse_tvs_diode_sma6j_datasheet.pdf.pdf

[52] muRata, OKI-78SR Voltage Regulator Datasheet, https://www.murata.com/products/productdata/8807037992990/oki-78sr.pdf?1583754815000

[53] ON Semiconductor, NCP1117LP Voltage Regulator Datasheet, https://www.onsemi.com/pub/Collateral/NCP1117LP-D.PDF

[54] Ohms Law Calculator, Voltage Divider Online Calculator, http://www.ohmslawcalculator.com/voltage-divider-calculator

[55] E Switch, PB400 Series Pushbutton Switch Datasheet, https://sten-eswitch-13110800-production.s3.amazonaws.com/system/asset/product_line/data_sheet/234/PB400.pdf

[56] ON Semiconductor, 8-pin SOIC Dual-Channel Phototransistor Output Optocoupler Datasheet, https://www.onsemi.cn/PowerSolutions/document/MOCD217M-D.PDF

[57] Hirose, UX60S Mini-B Connector Datasheet, https://media.digikey.com/pdf/Data%20Sheets/Hirose%20PDFs/UX60S.pdf

[58] OSRAM, KG R971 ChipLED Datasheet, https://dammedia.osram.info/media/resource/hires/osram-dam-2493936/LG%20R971.pdf

[59] Bivar, SLP-150 Lightpipe Datasheet, https://www.bivar.com/parts_content/Datasheets/SLP3-150-XXX-X.pdf

[60] Amphenol, D-Sub 15 Datasheet, https://www.amphenol-icc.com/media/wysiwyg/files/documentation/datasheet/inputoutput/io_dsub_d8656.pdf

[61] GitHub, CAN Logger 3 CAN0 and CAN1 Test, https://github.com/SystemsCyber/CAN-Logger-3/blob/master/tests/Test_CAN0_and_CAN1_no_CAN2/Test_CAN0_and_CAN1_no_CAN2.ino

[62] SAE, J1939-16 Automatic Baud Rate Detection Process, November 5th, 2018, https://www.sae.org/standards/content/j1939/16_201811/

[63] Ashwaq Alabaichi, Ralan Mahmood, Faudziah Ahamad, and Mohammed Mechee, "Randomness Analysis on Blowfish Block Cipher Using ECB and CBC Modes", Journal of Applied Sciences, 2013, doi:10.3923/jas.2013.768.789

[64] Hong, S. & Im, J. & Islam, Sm Mazharul & You, Jaehee & Park, Y.. (2017), "Enabling Energy Efficient Image Encryption using Approximate Memorization", Journal of Semiconductor Technology and Science. 17. 465-472. 10.5573/JSTS.2017.17.3.465

[65] Alhassane, Diallo, (2015), "Enhancement Of Bluetooth Security Authentication Using Hash-Based Message Authentication Code (HMAC) Algorithm", 10.13140/RG.2.1.1196.0082

[66] GitHub, Do-It-Yourself ECB Penguin Repository, Pakesson, https://github.com/pakesson/diy-ecb-penguin

[67] GitHub, cryptolibAESSHA Library Repository, https://github.com/SystemsCyber/CAN-Logger-3/tree/master/tests/cryptolibAESSHA

[68] GitHub, CryptoAccel Repository, Paul Stoffregen, https://github.com/PaulStoffregen/CryptoAccel

[69] Dworkin, Morris, "Computer Security" , NIST Special Publication 800-38A, 2001 Edition, https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf

[70] GitHub, AES 128 CBC Test, https://github.com/SystemsCyber/CAN-Logger-3/blob/serverless/tests/AES-128_CBC_Test/AES-128_CBC_Test.ino

[71] GitHub, mmCAU AES-128 Test Script, https://github.com/SystemsCyber/CAN-Logger-3/blob/master/tests/cryptolibAESSHA/cryptolibAESSHA.ino

[72] GitHub, SHA256 Library Repository, https://github.com/SystemsCyber/CAN-Logger-3/blob/master/libraries/sha256.zip

[73] NIST, SHA256 Test Vectors, https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Standards-and-Guidelines/documents/examples/SHA256.pdf

[74] GitHub, Rweather Repository, SHA256 Test Vectors, https://github.com/rweather/arduinolibs/blob/master/libraries/Crypto/examples/TestSHA256/TestSHA256.ino

[75] GitHub, SHA256 Test Script, https://github.com/SystemsCyber/CAN-Logger-3/blob/master/tests/sha256-test/sha256-test.ino

[76] GitHub, SparkFun ATECCX08A Library Repository, https://github.com/SystemsCyber/SparkFun_ATECCX08a_Arduino_Library

[77] GitHub, ATECC Key Configuration Script, https://github.com/SystemsCyber/CAN-Logger-3/blob/master/tests/ATECC_Key_Configuration_Test/ATECC_Key_Configuration_Test.ino

[78] GitHub, SparkFun ATECCX08A Source Code, https://github.com/SystemsCyber/SparkFun_ATECCX08a_Arduino_Library/blob/master/src/SparkFun_ATECCX08a_Arduino_Library.cpp

[79] GitHub, ECDH and ECDSA Python Test Script, https://github.com/SystemsCyber/CAN-Logger-3/blob/master/tests/ECDH%20and%20ECDSA%20Sign.py

[80] GitHub, ECDH and AES Test Script, https://github.com/SystemsCyber/CAN-Logger-3/blob/master/tests/ECDH_and_AES/ECDH_and_AES.ino

[81] AES Online Calculator Tool, http://testprotect.com/appendix/AEScalc

[82] GitHub, ECDSA and Verify Python Test Script, https://github.com/SystemsCyber/CAN-Logger-3/blob/master/tests/ECDSA%20Verify.py

[83] GitHub, ECDSA Sign Test Script, https://github.com/SystemsCyber/CAN-Logger-3/blob/master/tests/ECDSA_Sign/ECDSA_Sign.ino

[84] GitHub, ECDSA Verify Test Script, https://github.com/SystemsCyber/CAN-Logger-3/blob/master/tests/ECDSA_Verify/ECDSA_Verify.ino

[85] GitHub, ACAN2517 Library Repository, Pierre Molinaro, https://github.com/pierremolinaro/acan2517

[86] GitHub, MCP2517 Loopback Test Script, https://github.com/SystemsCyber/CAN-Logger-3/blob/master/tests/MCP2517_LoopBack_Test/MCP2517_LoopBack_Test.ino

[87] GitHub, J1708 Test Script, https://github.com/SystemsCyber/CAN-Logger-3/blob/master/tests/J1708_Test/J1708_Test.ino

[88] GitHub, Voltage Monitoring Test Script, https://github.com/SystemsCyber/CAN-Logger-3/blob/master/tests/Voltage_Monitoring_Test/Voltage_Monitoring_Test.ino

[89] GitHub, CAN Logger 3 Source Code, https://github.com/SystemsCyber/CAN-Logger-3/blob/master/CAN_Logger_with_Autobaud_and_Requests_no_CAN2_AES_CBC_SHA256/CAN_Logger_with_Autobaud_and_Requests_no_CAN2_AES_CBC_SHA256.ino

[90] GitHub, LED with Button Test Script, https://github.com/SystemsCyber/CAN-Logger-3/blob/master/tests/Test_LED_with_Button/Test_LED_with_Button.ino

[91] Github, WiFi101 Arduino Library Repository, https://github.com/arduino-libraries/WiFi101

[92] GitHub, WiFi Firmware Check Test Script, https://github.com/SystemsCyber/CAN-Logger-3/blob/serverless/tests/WiFi_Firmware_Test/WiFi_Firmware_Check.ino

[93] GitHub, CAN Logger 3 WiFi Test Script, https://github.com/SystemsCyber/CAN-Logger-3/blob/CAN-Logger-WiFi/tests/WiFi/CAN_Logger_3_WiFi_no_SD_logging/CAN_Logger_3_WiFi_no_SD_logging.ino

[94] GitHub, CAN Logging through WiFi Python Script, https://github.com/SystemsCyber/CAN-Logger-3/blob/CAN-Logger-WiFi/tests/WiFi/TCPclient%20for%20CAN%20Logging.py

[95] PJRC, Time Library, https://www.pjrc.com/teensy/td_libs_Time.html

[96] GitHub, FlexCAN.cpp Library File Source Code, https://github.com/SystemsCyber/FlexCAN_Library/blob/arduinoCompatible/src/FlexCAN.cpp

[97] GitHub, Error Definition, https://github.com/SystemsCyber/FlexCAN_Library/blob/arduinoCompatible/src/error.h

[98] GitHub, Bit Stuffing Error Generator Test Script, https://github.com/SystemsCyber/CAN-Logger-3/blob/master/tests/Bit_Stuffing_Error_Generator/Bit_Stuffing_Error_Generator.ino

[99] SAE, J1939-71 Vehicle Application Layer Standard, February 11th, 2020, https://www.sae.org/standards/content/j1939/71_202002/

[100] GitHub, Local Computer Python ClientApp, https://github.com/SystemsCyber/CAN-Logger-3/blob/master/clientApp/CANLoggerClient.py

[101] Colorado State University, CAN Logger Project Website, https://canlogger.auth.us-east-2.amazoncognito.com/login?client_id=58tl1drhvqtjkmhs69inh7l1t3&response_type=token&scope=email+openid&redirect_uri=https://systemscyber.github.io/CAN-Logger-3/loggers.html

[102] Amazon Web Services, Access Resources with API Gateway and Lambda with a User Pool, https://docs.aws.amazon.com/cognito/latest/developerguide/cognito-scenarios.html

[103] GitHub, CAN Logger 3 Provisioning Firmware, https://github.com/SystemsCyber/CAN-Logger-3/blob/master/CAN_Logger_3_Teensy_Provisioning/CAN_Logger_3_Teensy_Provisioning.ino

[104] GitHub, CAN Logger Format Convert GUI, https://github.com/Heavy-Vehicle-Networking-At-U-Tulsa/Log-File-Format-Converter/blob/Log-File-Converter-with-PYQT5/Log%20File%20Format%20Converter%20GUI.py

[105] GitHub, Pretty J1939 Repository, NMFTA, https://github.com/nmfta-repo/pretty_j1939

[106] GitHub, CAN Data Analyzer, https://github.com/Heavy-Vehicle-Networking-At-U-Tulsa/NMFTA-CAN-Logger/blob/DecodingGUI/Utilities/Decoding_GUI/DecodeJ1939Logs.py

[107] GitHub, Vehicle Speed Plotting Script, https://github.com/Heavy-Vehicle-Networking-At-U-Tulsa/Log-File-Format-Converter/blob/Log-File-Converter-with-PYQT5/plot_vehicle_speed.py

[108] GitHub, Engine Speed Plotting Script, https://github.com/Heavy-Vehicle-Networking-At-U-Tulsa/Log-File-Format-Converter/blob/Log-File-Converter-with-PYQT5/plot_RPM.py

[109] Bortolin, R., van Nooten, S., Scodeller, M., Alvar, D. et al., "Validating Speed Data from Cummins Engine Sudden Deceleration Data Reports," SAE Int. J. Passeng. Cars – Mech. Syst. 2(1):970-982, 2009

[110] Messerschmidt, W., Austin, T., Smith, B., Cheek, T. et al., "Simulating the Effect of Collision-Related Power Loss on the Event Data Recorders of Heavy Trucks," SAE Technical Paper 2010-01-1004, 2010

[111] Daily, J., Kongs, A., Johnson, J., and Corcega, J., "Extracting Event Data from Memory Chips within a Detroit Diesel DDEC V," SAE Technical Paper 2015-01-1450, 2015

[112] McKemmish, R., 2008, in IFIP International Federation for Information Processing, Volume 285; Advances in Digital Forensics IV; Indrajit Ray, Sujeet Shenoi; (Boston: Springer), pp. 3–15.

[113] Johnson, J., Daily, J., and Kongs, A., "On the Digital Forensics of Heavy Truck Electronic Control Modules," SAE Int. J. Commer. Veh. 7(1):72-88, 2014, https://doi.org/10.4271/2014-01-0495.

[114] MPC5674F Microcontroller Reference Manual, last accessed on Nov 16, 2019 from https://www.nxp.com/docs/en/reference-manual/MPC5674FRM.pdf

[115] Ng, J., PEmicro, personal communication, July. 2018.

LIST OF ABBREVIATION

AES	Advanced encryption standard
AWS	Amazon web services
BGM	Ball grid array
BOM	Bill of materials
CAN	Controller area network
CBC	Cipher blocker chaining
DA	Destination address
DRO	Diagnostic read-out
ECB	Electronic codebook
ECC	Elliptic-curve cryptography
ECDH	Elliptic-curve Diffie-Hellman
ECDSA	Elliptic-curve digital signature algorithm
ECU	Electronic control unit
GUI	Graphical user interface
HSM	Hardware security module
HVEDR	Heavy vehicle event data recorder
JTAG	Joint test action group
KMS	Key management service
LIN	Local interconnect network
IAM	Identity and access management
IC	Integrated circuit

IDE	Integrated development environment
IoT	Internet of Things
IV	Initialization vector
MITM	Man-in-the-middle
mmCAU	Memory-mapped crypto acceleration unit
NIST	National Institute of Standards and Technology
NMFTA	National Motor Freight Traffic Association
NSF	National Science Foundation
PCB	Printed circuit board
PGN	Parameter group number
PTC	Resettable fuse
RSA	Rivest-Shamir-Adelman asymmetric cryptography
SA	Source address
SDDR	Sudden deceleration data report
SMT	Surface-mount technology
SOF	Start of frame
SPN	Suspect parameter number
SSS2	Smart sensor simulator 2
SWCAN	Single-wire CAN
S3	Simple storage service
TCP	Transmission Control Protocol
TLS	Transport layer security
TVS	Transient voltage suppressor
VDA	Vehicle diagnostic adapter

VIDA	Vehicle information and diagnostics for aftersales