2021-01-0136 Published 06 Apr 2021

Secure Controller Area Network Logging

Jeremy Daily and Duy Van Colorado State University

Citation: Daily, J. and Van, D., "Secure Controller Area Network Logging," SAE Technical Paper 2021-01-0136, 2021, doi:10.4271/2021-01-0136.

Abstract

Practical encryption is an important tool in improving the cybersecurity posture of vehicle data loggers and engineering tools. However, low-cost embedded systems struggle with reliably capturing and encrypting all frames on the vehicle networks. In this paper, implementations of symmetric and asymmetric algorithms were used to perform envelope encryption of session keys with symmetric encryption algorithms while logging vehicle controller area network (CAN) traffic. Maintaining determinism and minimizing latency are primary considerations when implementing cryptographic solutions in an embedded system. To satisfy the timing requirements for vehicle systems, the memory-mapped Cryptographic Acceleration Unit (mmCAU) on the NXP K66 processor enabled 6.4Mb/sec symmetric encryption rates, which enables logging of multiple channels at 100% bus load. Using AES-128 in Cipher Block Chaining (CBC) mode provides the encryption for data confidentiality. Errors and integrity checks are handled by a Cyclic Redundancy Check (CRC) checksum withing the data and digitally signed SHA256 hash values of the overall encrypted record secured the integrity of the data. A hardware security module (HSM) is utilized to store asymmetric key pairs for key management. The HSM implements Elliptic-Curve Cryptography (ECC) algorithms for key exchanges and digital signatures. Secure collection and secure data uploads to a central server are demonstrated. This work and the source code are open source with the goal of inspiring improved secure communications for vehicle networks.

Introduction

Historically, passenger cars and heavy trucks have been made of various mechanical and thermal systems that convert energy from fuel to kinetic energy. However, modern vehicles incorporate many Electronic Control Units (ECUs) 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. The CAN also is used for 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.

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.

There are a few common attacks that have been done, either by actual hackers or in lab testing, as described in the NMFTA whitepaper. These are considered pain-points for engineering a secure system.

- 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 basic attack that affects data availability.
- Middleperson- 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 impersonate 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 an intermediate representation and determine the proprietary data structures 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 message parameters using fuzzing techniques. Therefore, proprietary information is at risk of being exposed. Fuzzing and 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 access the data. Protecting data availability depends on the system infrastructure and models 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 for cybersecurity.

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, which is a recommended practice for communication and diagnostics among vehicle components. The manufacturers are not legally 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. A 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 paper is to report on a solution to build such a data pool securely and efficiently.

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 in 2015, performed by Charlie Miller and Chris Valasek [3]. The authors 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, et al. described a denial of service attack on embedded networks in commercial vehicles [4]. 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 [5]. 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. This research motivated the inclusion of the error counters in the CAN Logger data record.

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 [6]. 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 that arrived 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 high fidelity CAN monitor. The system cybersecurity was deemed to be out of scope and thus not addressed in the referenced paper. However, if there is any cybersecurity threat, this system will not likely to detect such attack and may 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 [7]. 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.

Current Approach

Due to the complexity and high cost of integrating a new embedded system into the existing vehicle 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 networks. Data volume can grow, and thus, a cloud or server 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 enciphered data is useless and some vehicle owners do not wish to publish their data. As a result, security measures such as cryptographic algorithms are utilized to encrypt, sign, and verify the data.

Organization

The paper provides the following contributions:

1. Detailed documentation regarding the hardware design of the CAN logging device, such that some of the example may provide inspiration with the purpose of increasing cybersecurity posture.
2. Examples of end-to-end security with a hardware security module (HSM) and cloud-based encryption system.
3. Software reference design to implement a cloud-based system with pre-provisioned security devices.

The paper discusses the system requirements, hardware implementation, test and evaluation results, and overall software design to include the middleware and cloud backend. All the source code for the tests and implementations is available on Github [8].

Design Requirements

To carry out the objective, the CAN logger device must securely capture all CAN data under both normal and abnormal (i.e. 100% bus load or error frames) 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. Most requirements were focused on heavy vehicle use cases.

1. The logger must connect to multiple vehicle networks using industry standard connectors. The connector should handle power, ground, CAN1-H, CAN1-L, CAN2-H, CAN2-L, CAN3-H, CAN3-L, J1708-H and J1708L.
2. The logger needs to be inexpensive and easy to manufacture because many devices are essential for efficiently collecting data from various 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 may 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 should 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 not lose data in the event of a power failure. 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 systems up to 24V. However, transients may go up to 30V or more because there are load dumps and reversals associated with inductive loads and starters that create voltage 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 and expandable external storage for storing 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 using open standards.

11. The backend storage system needs to enable secure and scalable access to the data.
12. Users need a local 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.

CAN Logger Hardware

The CAN Logger is designed and built to log CAN messages on operating heavy vehicles. The device is an improved version of the based on two proofs of concept. Both ideas were products from university student projects. More information about these projects can be found on the GitHub repository in [8]. Because there were two earlier versions, the hardware presented herein is the third model, or the CAN Logger 3.

The CAN Logger 3 four-layer printed circuit board (PCB). The board has a dimension of 3.254" by 2.229", which fits in the BUD HP-3651-B enclosure. The board was laid out using Altium Designer, and the design files are publicly available. The main three components that provide the functionality for the CAN Logger 3 are: 1) the NXP K66 ARM Cortex-M4F microprocessor, 2) the ATWINC1500 WIFI module, and 3) the ATECC608A hardware security module.

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.

Microprocessor The circuit to support the NXP K66 processor was inspired by the Teensy 3.6 development board from PJRC.com. The Teensy 3.6 is an open-source hardware design that is compatible with the Arduino Integrated Development Environment (IDE). It is CAN compatible through the FlexCAN library [9]. The device's dual CAN channels, and on-board SD card slot are features that meet the objective of this project. Powered by the K66 ARM Cortex-M4 microprocessor, the Teensy 3.6 has a clock speed up to 180Mhz. Moreover, the K66 also has an embedded mmCAU ColdFire coprocessor that performs cryptographic algorithms such as AES, DES, 3DES, MD5, SHA-1, and SHA-256. The datasheet of the K66 and mmCAU can be found in [10].

The mmCAU can use AES 128 to encrypt data at a rate of 6.4 Mbyte/second in ECB mode. The CAN Logger 3 is able to encrypt and log CAN traffic at 100% bus load. In addition,

FIGURE 1 CAN Logger 3 device showing the buttons and LEDs on the left and the cover removed on the right

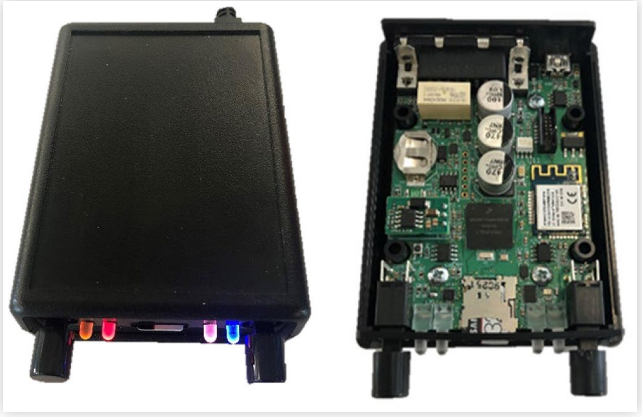


FIGURE 2 CAN logger hardware system block diagram

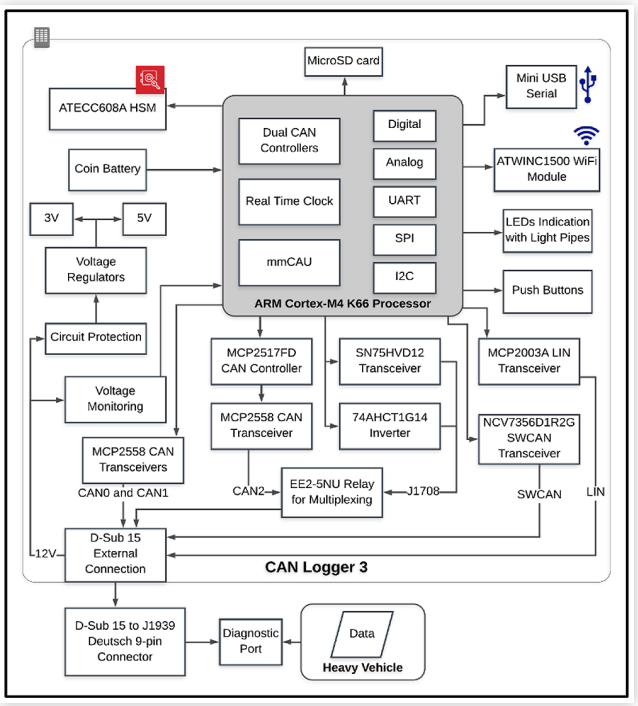


FIGURE 3 CAN Logger 3 showing the ATWINC1500 WiFi module with its hardware switch

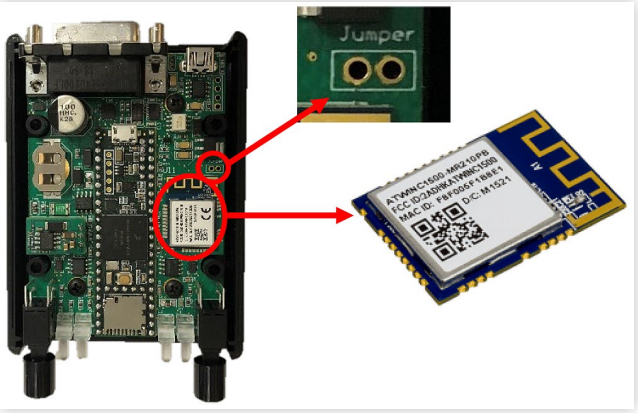


FIGURE 4 CAN Logger 3 and the hardware security module

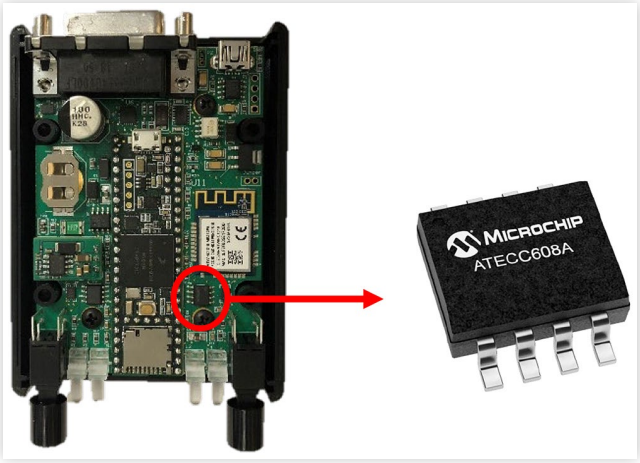
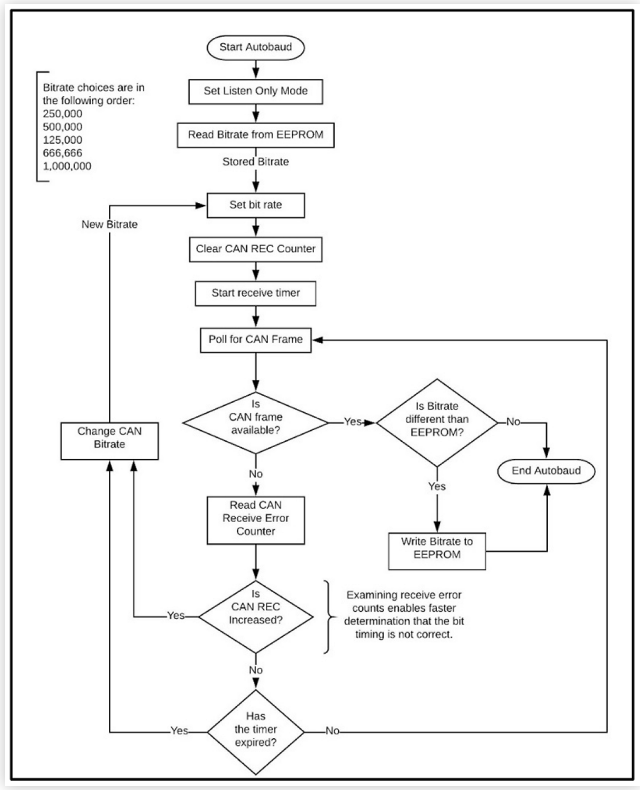


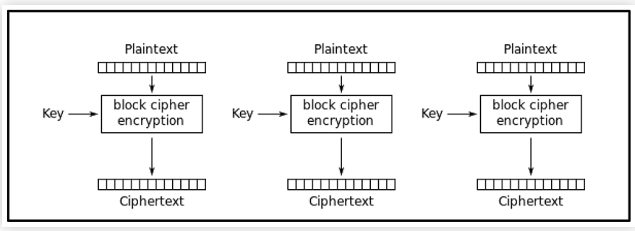
FIGURE 5 Automatic bit rate detection routine



the FlexCAN library has been modified to detect error frames [9]. This microprocessor satisfies the requirements of the project.

ATWINC1500 WIFI Module The CAN Logger 3 is also equipped with the low power consumption ATWINC1500 WIFI module, which features IEEE 802.11 b/g/n and 2.4 GHz ISM band. This allows the CAN Logger to implement WIFI communication to transfer log files to the local computer before uploading or transfer the data straight to the server wirelessly, which is beneficial for the scope of the project.

FIGURE 6 Electronic codebook (ECB) mode encryption



However, this also poses as an attack vector. To mitigate risk in specific applications, a physical switch is made in the design, as seen in Figure 6 above, such that users need to solder and bridge the jumper to enable the WIFI module. Thus, if users do not wish to use the WIFI feature, they can physically disable the ATWINC1500 module. The datasheet of the module can be found in [11].

ATECC608A Hardware Security Module The Microchip ATECC608A hardware security module (HSM) is the key component for the security aspect of the logging process. Information about the module can be found in [12]. The hardware is a cryptographic module 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 read and can only be used internally by the hardware functions. This is a great feature in the cryptographic world where there is always 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 middleperson attacks.

Moreover, the ATECC608 supports cryptographic algorithms including AES-128 encrypt/decrypt, Galois field multiply for generic authenticated encryption block cipher mode, SHA256 & HMAC hash, and especially ECC following P256 NIST, Elliptic-curve Digital Signature Algorithm (ECDSA) following FIPS186-3, and Elliptic-curve Diffie-Hellman (ECDH) following FIPS SP800-56A standards. The reason why ECC is preferably over other algorithms in IoT asymmetric cryptography is that ECC can meet the same security standard with a much smaller key size [13]. A 160-bit ECC key is equivalent to a 1024-bit RSA and Diffie-Hellman, or a 256-bit ECC key is equivalent to a 3072-bit RSA and Diffie-Hellman.

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 [14] shows the superior speed of ECC over RSA. With the cost of approximately $0.75 per piece and the provided features, the ATECC608A HSM was selected as the security module for this project.

Logging Fields

The goal of the logger is to efficiently write CAN messages to an SD card. Writing individual CAN messages (16 bytes) requires overhead when writing to the SD card and 16 byte

blocks are not optimal. Instead, SD card write speeds are much higher when using 512 byte blocks. As such, we designed a 512 byte buffer to write to the SD Card. This buffer contains 19 CAN messages and associated meta data, as shown in Table 2. Each CAN frame has the following pieces of information, as shown in Table 1:

- Channel - representing the physical CAN hardware.
- A real-time clock timestamp (seconds from the epoch)
- A system microsecond counter
- CAN Arbitration Identifier
- Data Length Code
- Eight bytes of CAN Data

The additional data in each frame includes the number of messages received on each channel, the Receive Error Count and the Transmit Error Counts. These additional counters can help determine different events on the network, especially if a so-called bus off attack takes place.

Additional metadata regarding the file name, write time of the last block, and a checksum are included. These often do not change as new data arrives, but filling in the tail of the write buffer with potentially useful information was more desirable than padding with meaningless characters.

If an error condition is detected by the CAN controller, the software will produce an error message by writing a CAN frame to the buffer with the error flag set. The error handling follows the same format and encoding as Linux SocketCAN error handling.

Functional Tests and Results

There are some crucial functions 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].

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] were connected to a terminated CAN bus for testing.

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. The fact that there were messages read 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.

TABLE 1 CAN frame format

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 0xFF if not used.							

TABLE 2 Buffer construction

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			

Bytes	491	492	493	494	495	496	497	498	499	500	501	502	503	504	505	506	507	508	509	510	511
Can0	Can0	Can1	Can2	Can0	Can1	Can 2	T	U	2	-	-	N1	N2	N3	Write Time			CRC32			
uint8_t	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			

Autobaud Function and 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 [9].

The Autobaud process was designed based on the recommendations in J1939-16 Automatic Baud Rate Detection Process [15]. 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 the EEPROM with the correct bitrate setting if the previous one is different and end the Autobaud routine.

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 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.

Symmetric Encryption Using AES-128

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

FIGURE 7 Cipher Block Chaining (CBC) mode encryption

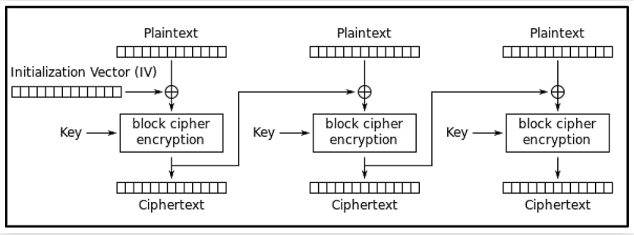


FIGURE 8 Test results for AES-128 CBC mode encryption and decryption

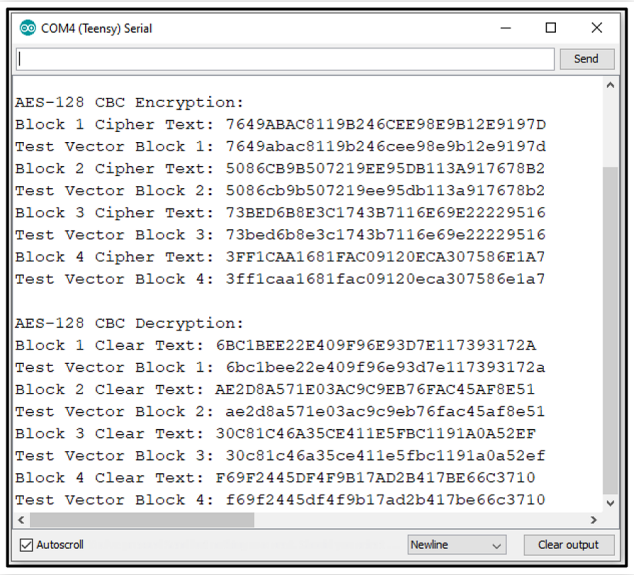


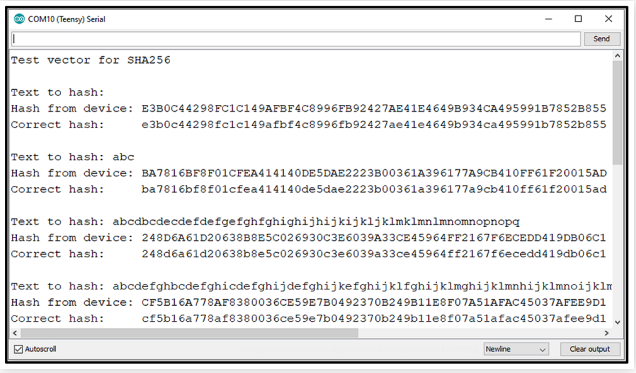
image or text files that have repeated identical data [16]. This is crucial because some CAN frames are periodic, meaning that the same data are sent within the same constant interval. Thus, encrypting CAN data using ECB mode is considered vulnerable. AES in the cipher block chaining (CBC) mode is used 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 6 and Figure 7 illustrate ECB and CBC modes for AES encryption processes, respectively.

The mmCAU uses the cryptolibAESSHA library [17] to implement its AES capability, with an Arduino interface published by Paul Stoffregen [18]. The AES-128 CBC encryption and decryption were tested against NIST test vectors [19]. The main functions from the test code is displayed below:

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

FIGURE 9 Result of the testing for SHA-256



data at full bus load. An Arduino script was written to measure the rate of mmCAU encryption.

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. 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.

Encrypted logging tests were performed at 100% busload for two CAN channels at 1 Mbit/second. The CAN Logger 3 was able to capture and encrypt all messages, which is a rate of 2Mbits/second. To validate this claim, one CAN Logger 3 was programmed to transmit 20,000 messages on each channel on an interval of 130.125 milliseconds. This interval is for an 8-byte message with no stuff bits at 100% load. The CAN Logger 3 that was programmed to encrypt and log the files captured all the messages in the same amount of time. Busload was monitored with the Linux SocketCAN can-utils [20] that showed over 100% busload. While truly exceeding 100% busload is not feasible, the can-utils implementation for SocketCAN is not tuned to make accurate assessments. Therefore, this is an indicator of bus saturation, as opposed to a proof.

Secure Hash Algorithm

SHA-256 hashing is used for one-way mapping data of arbitrary size to a unique fixed-size digest of 32 bytes. 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 function was validated against NIST test vectors [21].

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 9 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.

ECDH Pre-Master Calculation

This test shows the partial concept of ECDH pre-master key exchange by showing the shared secret result from the server (Python) and the client (Teensy). The first step is to generate an ECC key pair for the client. The client public key then will be manually loaded into the server Python script, where the server will generate an ECC key pair for itself and use its private key and the input client public key to calculate a shared secret. The server public key, along with the client keypair generated previously, will be then manually loaded into the Teensy script, where the client will use its private key and the server public key to calculate a shared secret.

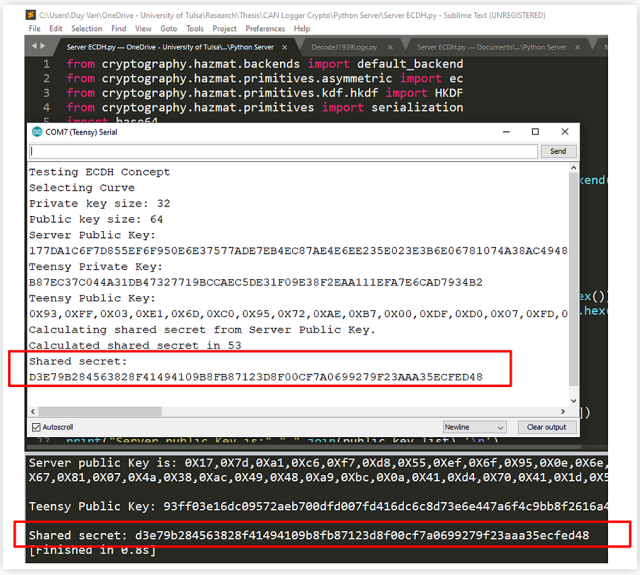
Figure 10 shows that the client and the server calculate the same shared secret. This demonstrates the Diffie-Hellman key exchange concept where public keys are exchanged, and the client and the server can use the other's public key to generate the same shared secret for further use in secure communication.

Connection Interrupt Test

The CAN Logger 3 logs the data by opening, writing, and closing a binary file. If the closing process does not occur, all the data in this current logging session will be lost. The file closing function is only designed to be triggered when network activity has ceased. To verify that the CAN Logger 3 can successfully log a file after being unplugged from the network, the closing process time and the processor running time after disconnection are measured and compared.

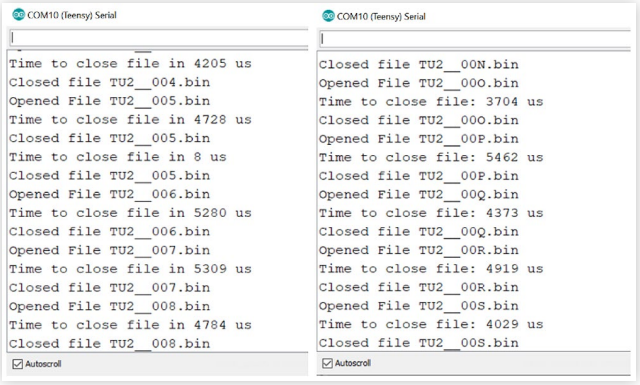
The file closing function in the firmware is modified to print out the time it takes in microseconds. Figure 11 shows the closing time for logging with and without AES encryption. The average time for both is very similar and fluctuates between 4,000-5,000 microseconds.

FIGURE 10 Demonstrating the ECDH concept between a host and client



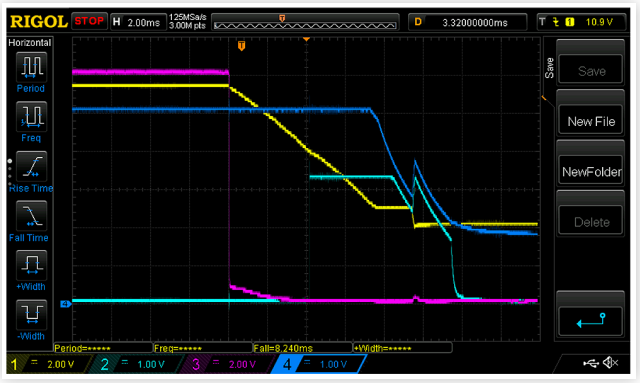
© SAE International.

FIGURE 11 Binary file closing time for non-encrypted version (left) and AES encrypted version (right)



© SAE International.

FIGURE 12 Voltage of safe 12V (yellow), LED indicator (light blue), raw 12V (purple), and processor power (dark blue) when the CAN Logger 3 is unplugged from power



© SAE International.

The next step is to measure the processor running time after the device is disconnected. An oscilloscope is used to monitor the power to the processor along with the raw 12V input from the network and the diode protected 12V from the device at the same time. The firmware is modified to pull an LED high immediately after the file is closed. The voltage of the LED is also monitored by the oscilloscope to determine whether the file closing occurs and how long it takes. Figure 12 shows the voltage traces of the four mentioned signals. The data is then exported and analyzed to measure the desired parameters, as depicted in Figure 13.

The results show that when the device is unplugged, the raw 12V input quickly drops below 1V, which is the first indication that the file closing function should be triggered. The capacitor in the design still supplies power to maintain the voltage regulator output at a normal voltage level of 5V for about 7.5 milliseconds while the device residual power slowly decreases. After that, the Teensy 3.6 input voltage starts to drop and loses power at 8.52 milliseconds when it reaches below its operational voltage of 3.6 V. Moreover, the LED turns on at 4.18 milliseconds after the device is unplugged. The facts that the LED does turn on and the Teensy processor running time outlasts the file closing time indicates the CAN Logger 3 does not lose log data when power connection is interrupted.

FIGURE 13 Rescaling and analyzing the voltage dropped from power interruption

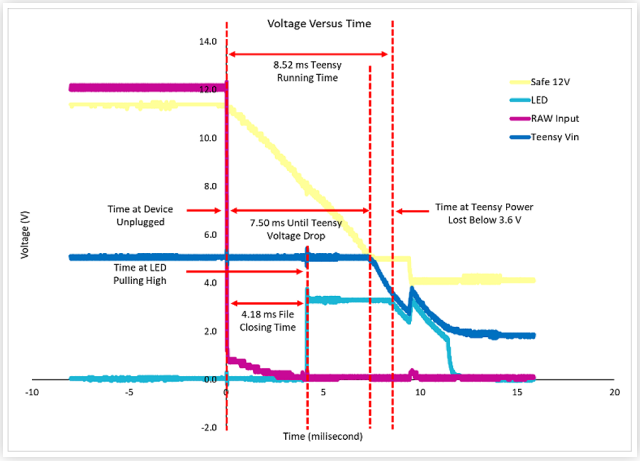
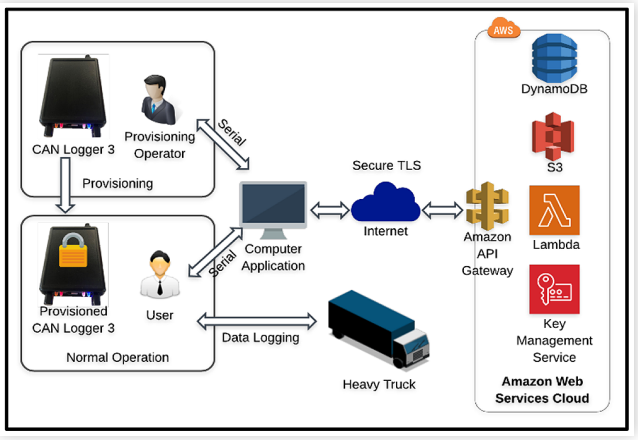


FIGURE 14 CAN Logger 3 system software design overview



In cases where the power loss occurs while there is still residual capacitance, and the device is not isolated from the network, the RAW voltage will not drop suddenly to trigger the file closing function. This can cause the device not to close the file properly. However, the situation is uncommon and not in the scope of the project, and therefore, the problem is not further examined.

System Software Implementation

Overview Process

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

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 14, on the following page.

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

- The local computer with Python application
- The provisioning operator
- The Internet connection with secure TLS
- The AWS third-party
- The owner of the CAN logger

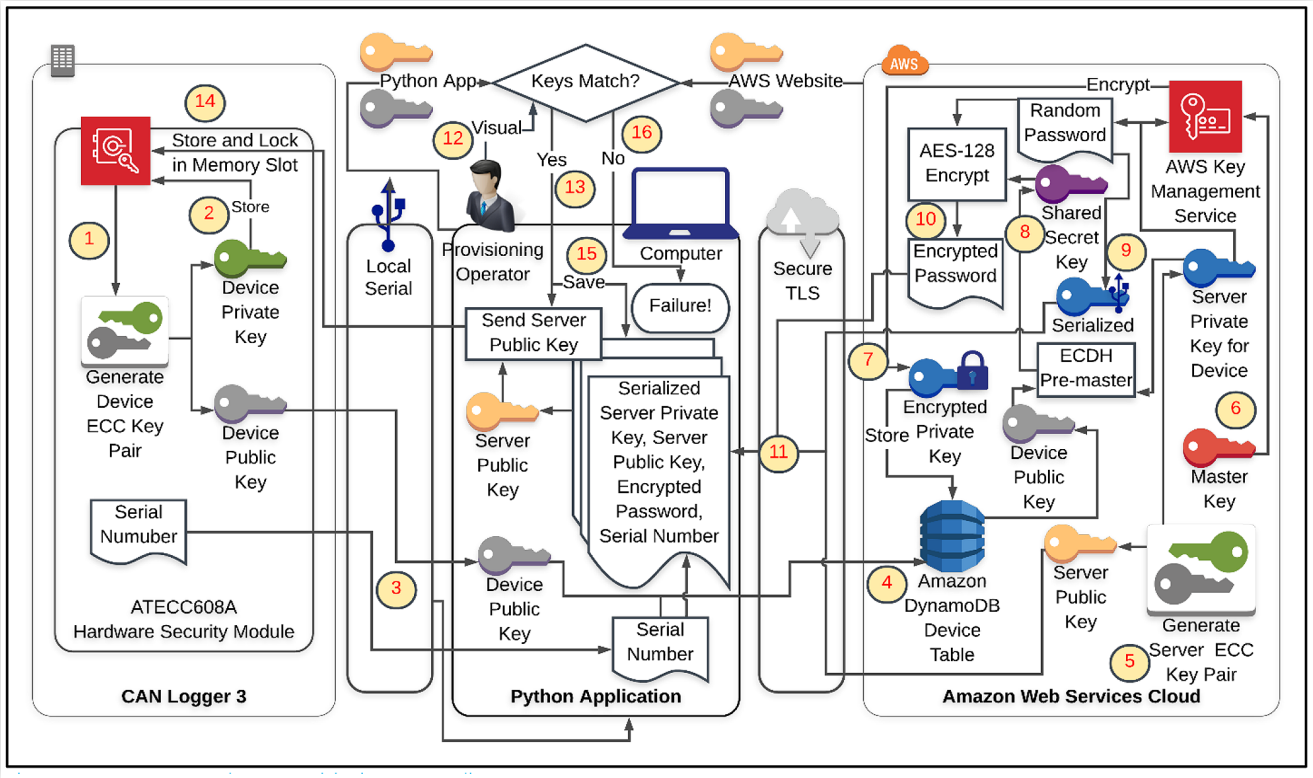
The local computer and the provisioning operator are parts of the device's manufacturing process. Preventing these two factors from being compromised is not in the scope of the CAN logging project but in the security of the local facility itself. As a result, these two factors are assumed to be safe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

Transferring sensitive data via the Internet can be risky. However, by following the industry-standard 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 Web Services (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 the wrong hands 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

FIGURE 15 Secure Key Exchange provisioning process diagram



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.

Provisioning

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 primary purposes of the initial provisioning 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 15 on the following page.

The enumerated descriptions for Figure 15 are described in Table 3.

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 16 and Table 4.

The following table describes Figure 16.

Normal Operation

Secure CAN Logging After the key exchange provision, the device is ready to used for logging sessions. In this process,

FIGURE 16 Serialized server private key password function diagram

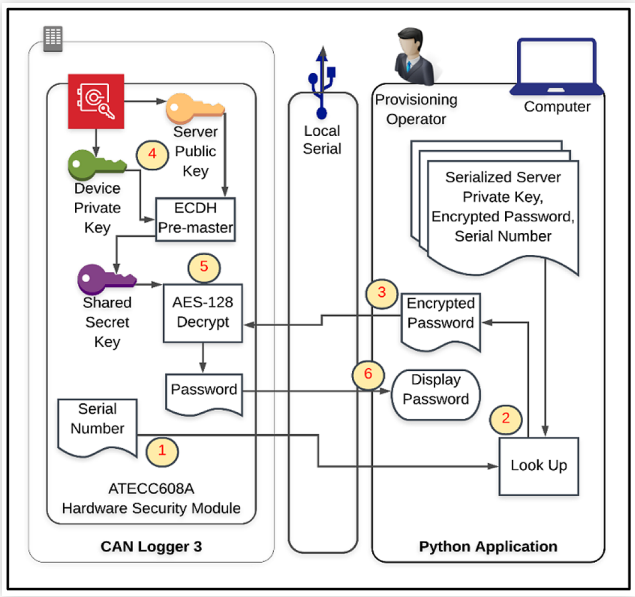


TABLE 3 Secure key exchange provisioning process description

Process	System	Description
1	Embedded Firmware	The ATECC608A hardware security module first generates an ECC key pair, which is 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 show an error message.

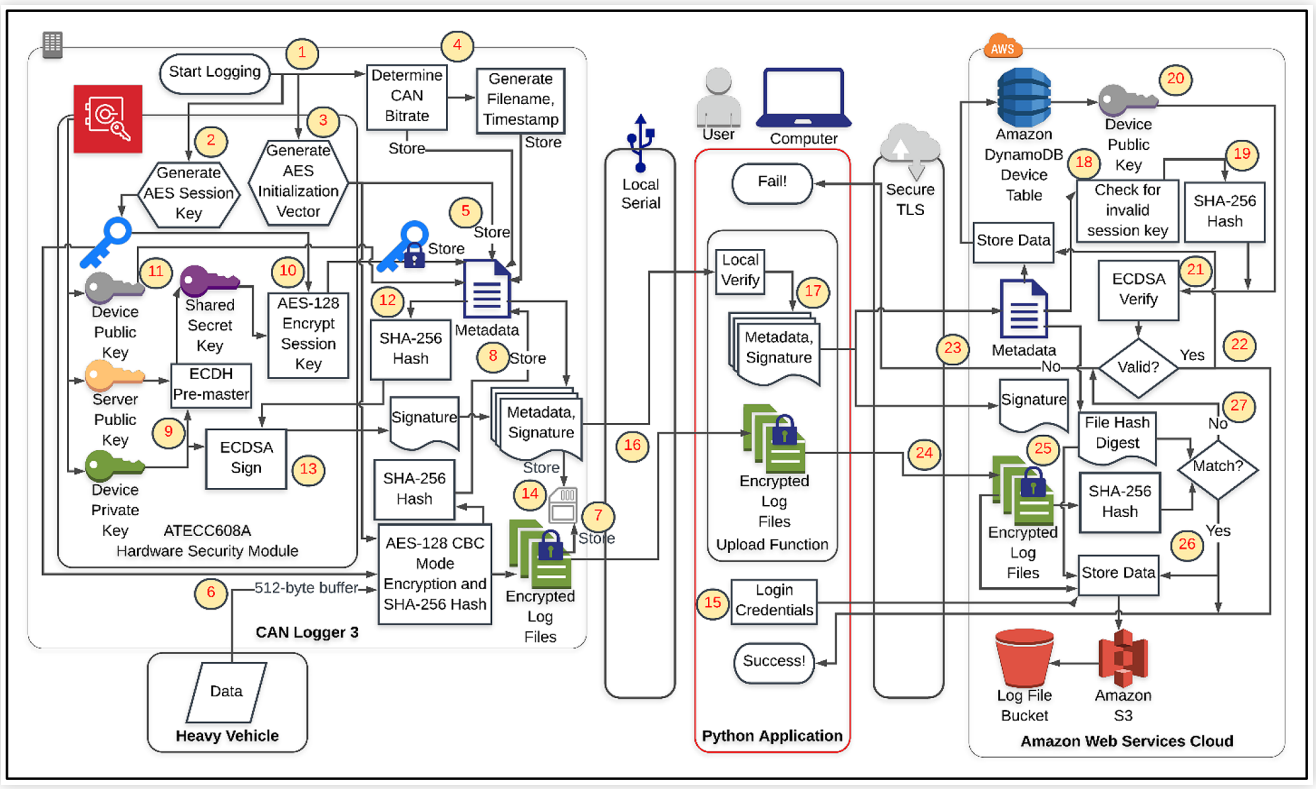
the log data will be encrypted in real time and then signed in the CAN Logger 3 before sending to the server. These steps ensure that the contents of the files are not exposed in storage and while being transmitted to the server through the Internet. Signing the logs verify that the server receives authentic data from the correct sender. The server can then decrypt and analyze the data based on user needs using the calculated shared secret key from the provisioning step. The logging process is depicted in Figure 17.

Every time the device starts a logging session, the ATECC608A generates a 32-byte random number, which is split into a 16-byte AES session key and a 16-byte initialization vector (IV). The CAN data is logged in a way that the logger initially determines the bus bitrate, generates a binary file in the SD card, and starts collecting data to fill up a 512-byte block with CRC-32 checksum included in the last four bytes. When this buffer is full, the logger encrypts the data, writes it to the binary file, resets the buffer, and repeats the process until the logging stops. Because some CAN messages are repeated periodically in a truck network, using AES in ECB mode can pose a potential risk to an AES plain text attack. As a result, the buffer is encrypted with AES-128 CBC mode using the generated AES session key and IV. When logging ceases, the file is closed and written to the SD card. The encrypted log file is then hashed using SHA256 in the program code and signed using the ECDSA ATECC608A function, which produces the file's signature.

Upon generation of the session key and IV, the AES session key is encrypted with AES-128 using the shared secret, which is calculated at the beginning of the logging session with ECDH pre-master function using the device private key and server public key as inputs. The encrypted AES session key and the signature are stored in the metadata text file along with the AES IV, the bitrate, and the filename. The current information in the metadata text file is hashed and signed similarly to the encrypted log file, producing a text file signature to ensure the authenticity of the text file. The text file signature is then appended to metadata text file, which along with the encrypted log file, are then stored in the SD card until being transferred to the server.

At the time of this writing, the clients will need to use the Python user application to upload the data to the server.

FIGURE 17 Logging and uploading process diagram



The clients must log in with their credentials to identify themselves and their device. Through local serial, the device will be connected to the application where the log files information will be displayed on the interface. The clients can then select and upload the desire log files along with their corresponding metadata text files to AWS cloud via secure TLS communication. In theory, the hardware can support automatic uploading through the WiFi subsystem in the CAN Logger 3.

Once the server receives the metadata and the encrypted log files, the server will populate the metadata in the AWS DynamoDB database, and store the encrypted log file in AWS S3 storage service. When the file is needed upon user request, the server will extract the corresponding server encrypted private key and device public key from the database to decrypt the file. The server will first decrypt the server's encrypted private key using AWS key management service. With the server private key and the device public key, the shared secret key is calculated with ECDH pre-master function using the server private key and the device public key. The encrypted AES session key for the file from the database is then decrypted using the shared secret key. The encrypted log file now can be decrypted using AES-128 CBC mode with the AES session key obtained previously and the AES IV from the database. However, before decrypting, the integrity of the file is verified using ECDSA with its signature and device public key to make sure the file has not been tampered with. After the integrity is checked, the file is decrypted before being available to the user.

TABLE 4 Serialized server private key password function description

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.

Middleware (Python User Interface)

The CAN Logger and the server are connected by the Python interface, which is controlled by the user. The GUI is shown in Figure 18 and its functions are described in Figure 19.

FIGURE 18 CAN logger client application interface

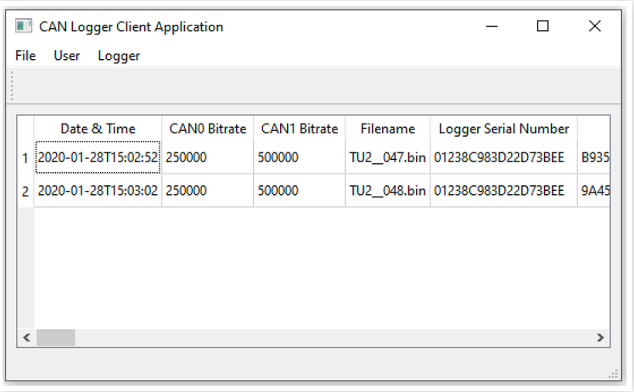
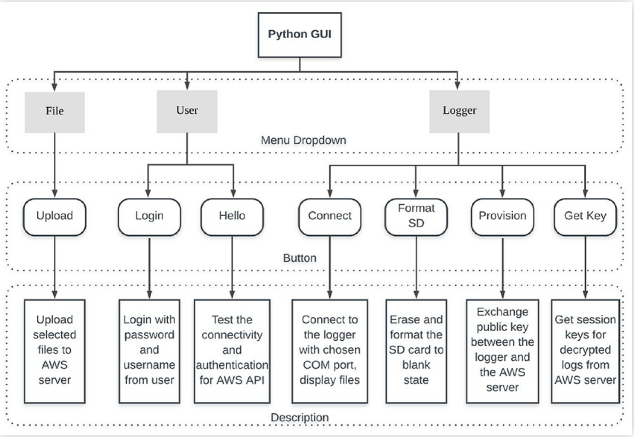


FIGURE 19 Depiction of the button functions in the application



When the clients first run the application, it will automatically ask for username and password with a dialog box to submit to the server and then return a token for further user authentication. This process can also be done by selecting Login button under User menu dropdown. Once the clients have successfully identified themselves, the next step is to connect the logger device by selecting Logger dropdown menu and Connect button to choose the current device serial COM port. This function will parse through all the data on the device SD card and grab all the metadata text files to verify their authenticity with the signature appended at the end of the file using the device public key. When the files are verified, the metadata of all available binary files stored in the SD card is displayed on the Python application, as shown in Figure 18. The clients now can proceed to send the data to the server by clicking on the desire file and selecting Upload button under the File menu dropdown. The application will download the file through local serial and send a POST request to the server with the data.

Beside the mentioned functions, the Python user application also has other buttons with their descriptions shown in Figure 19. The Hello button under the User menu dropdown helps the clients test their connectivity and authentication between their end to the AWS server. A successful dialog box will appear to indicate that the correct API key is used. Under the Logger menu dropdown, the Format SD button will erase and format the SD card on the connected device to blank state. The Provision button should only be used once and at the provisioning process by the manufacture. As described in the provisioning process above, this function exchanges the public keys between the logger and the AWS server securely. Lastly, if the clients want to decrypt a desired file locally, the Get Key function sends the selected file's SHA-256 digest and the device serial number to the server and requests the corresponding plaintext AES session key. The logger must be provisioned before with a securely stored key tied to the serial number for this function to work. The key, once received successfully from the server, can be used to decrypt the encrypted log file.

Summary/Conclusions

The CAN data-gathering project is needed to establish a database of operating heavy vehicle network communication. To accommodate the large scale of CAN logs, the data are uploaded and stored in the cloud. Security mechanism is required to protect data confidentiality and integrity during logging operation that involves the IoT. The approach of secure CAN logger project produces the CAN Logger 3 device, which follows common security standards using symmetric and asymmetric encryption. The functionality of the device has been conducted to fulfill the operational requirements of the project.

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 paper describes 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.
- A public key exchange process between the CAN Logger 3 and the AWS cloud is performed during the

TABLE 5 Logging and uploading process description

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 processor 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.

Process	System	Description
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.

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 must 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 local drive, which is kept by the administrators.

The hardware and software system comprising the CAN Logger 3 outperforms previous systems as it does not drop any frames, even with a saturated CAN bus. Furthermore, the confidentiality of the system is maintained from the moment the CAN traffic is recorded.

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 [8].

Cyber-physical system security, as a field of study, is in its infancy. This paper 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.

References

1. National Motor Freight Traffic Association, Inc., "A Survey of Heavy Vehicle Cyber Security," Updated Jan. 4, 2016.
2. SAE International, "SAE J1939-21 Data Link Layer," Surface Vehicle Recommended Practice, Oct. 2018.
3. Miller, C. and Valasek, C., "Remote Exploitation of an Unaltered Passenger Vehicle," Aug. 10, 2015, http://www.illmatics.com/Remote%20Car%20Hacking.pdf, .
4. Mukherjee, S., Shirazi, H., Ray, I., Daily, J., and Gamble, R., "Practical DoS Attacks on Embedded Networks in Commercial Vehicles," in *ICISS 2016: Information Systems Security* , 2016.
5. Cho, K.-T. and Shin, K.G., "Error Handling of In-vehicle Networks Makes Them Vulnerable," in *CCS '16: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, Vienna Austria, 2016.
6. Shaout, A., Mysuru, D., and Raghupathy, K., "CAN Sniffing for Vehicle Condition, Driver Behavior Analysis and Data Logging," in *2018 International Arab Conference on Information Technology (ACIT)*, Werdanye, Lebanon, 2018.
7. Johanson, M. and Karlsson, L., "Improving Vehicle Diagnostics through Wireless Data Collection and Statistical Analysis," in *2007 IEEE 66th Vehicular Technology Conference*, Baltimore, MD, 2007.
8. Daily, J. and Van, D., "CAN-Logger-3," https://github.com/SystemsCyber/CAN-Logger-3, accessed Dec. 1, 2020.
9. https://github.com/SystemsCyber/FlexCAN_Library, accessed Dec. 1, 2020.
10. NXP, "K66 Sub-Family Reference Manual," Aug. 4, 2018, https://www.nxp.com/docs/en/reference-manual/K66P144M180SF5RMV2.pdf, accessed Dec. 1, 2020.

11. Microchip, "ATWINC15x0-MR210xB IEEE 802.11 b/g/n SmartConnectIoT Module," 2018, https://ww1.microchip.com/downloads/en/DeviceDoc/ATWINC15x0-MR210xB-IEEE-802.11-b-g-n-SmartConnect-IoT-Module-Data-Sheet-DS70005304C.pdf accessed Dec. 1, 2020.
12. Microchip, "Microchip CryptoAuthentication Device," 2017, http://ww1.microchip.com/downloads/en/DeviceDoc/40001977A.pdf. accessed Dec. 1, 2020.
13. Chahar, H., Keshavamurthy, B., and Modi, C., "Privacy-Preserving Distributed Mining of Association Rules Using Elliptic-Curve Cryptosystem and Shamir's Secret Sharing Scheme," *Sādhanā* 42:1997-2007, 2017.
14. Groll, A. and Ruland, C., "Secure and Authentic Communication on Existing In-Vehicle Networks," in *2009 IEEE Intelligent Vehicles Symposium*, pp. 1093-1097, 2009.
15. SAE International, *SAE J1939-16 Automatic Baud Rate Detection Process* (Surface Vehicle Recommended Practice, 2018).
16. Alabaichi, A.M., Mahmood, R., Ahmad, F., and Mechee, M.S., "Randomness Analysis on Blowfish Block Cipher Using ECB and CBC Modes," *Journal of Applied Sciences* 13:768-789, 2013.
17. https://github.com/SystemsCyber/CAN-Logger-3/tree/master/tests/cryptolibAESSHA.
18. Stoffregen, P., "CryptoAccel," https://github.com/PaulStoffregen/CryptoAccel.
19. Dworkin, M., "Computer Security," 2001, https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf.
20. "SocketCAN Userspace Utilities and Tools," https://github.com/linux-can/can-utils.
21. NIST, "SHA256 Test Vectors."

Contact Information

Jeremy Daily,
Jeremy.Daily@colostate.edu

Acknowledgments

We would like to thank and acknowledge Urban Jonson and Ben Gardiner of the National Motor Freight Traffic Association, Inc. (NMFTA). Without their motivation and support, this work would not have been possible.

This work, in part, was sponsored by the National Science Foundation (NSF) under Award number is 1715409 and number 1951224.

The authors would like to thank John Maag and Chris Lute for helping with ideas and providing feedback during the writing process.

Definitions/Abbreviations

AES - Advanced encryption standard cryptography
AWS - Amazon web services
CAN - Controller area network
CBC - Cipher blocker chaining
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
IDE - Integrated development environment
IoT - Internet of Things
IV - Initialization vector
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
RSA - Rivest-Shamir-Adelman asymmetric cryptography

ISSN 0148-7191