Mobile Vehicle Cybersecurity with Onboard Key Management

Iowa State University: ECpE sdmay23-15

Aayush Chanda, Alexander Freiberg, Baganesra Bhaskaran,

Brian Goode, Chau Wei Lim, Michael Roling

Table of Contents

Functional and Non-functional Requirements

The team's project, Mobile Vehicle Cybersecurity with On-Board Key Management, delves into the topic of vehicle communication. Messages sent within a vehicle are transmitted over the controller area network (CAN). The CAN Bus is simply composed of two wires (CAN-High and CAN-Low) which carry each signal from the controllers within the vehicle. These controllers follow a strict series of protocols to ensure each message is not interrupted by another. Communication protocols are set by the Society of Automotive Engineers (SAE) and are defined under J1939. Understanding the key management protocol J1939-91C is a requirement.

A subsequent requirement for the project is the ability to handle the transmitted messages, specifically, within the International Organization of Standardization's format. CAN messages, or frames, are defined by several segments: the controller transmitting, its level of importance, a binary message, and a verification to ensure its integrity. These messages may be extended to include additional information, or more formerly referred to as controller area network flexible data-rate (CAN-FD). Utilizing the benefits of CAN-FD will be essential to the project's success, therefore, it was deemed as a requirement.

The project's final deliverable was to create a key for each message within the vehicle; generating an on-board key uniquely ensures security. It should be noted each vehicle manufacturer could individually define each key for each controller. A notable setback, however, would be the replacement of the controller. The physical device is subject to failure, and ensuring its replacement/key is accepted onto the vehicle is vital; else, other controllers will deem its messages invalid. On-board key generation will allow valid, third-party replacement controllers to be used as well. These attributes forego the otherwise necessary key-communication between vehicle key manufacturers; it is why on-board key generation is the project's final requirement.

Pertinent Standards

- SAE J1939
    - SAE J1939 is a protocol created by the Society of American Engineers designed for the communication between the electronic component units in heavy-duty commercial vehicles, such as trucks and buses.
    - This protocol is the most appropriate and effective for our project because of a few vital advantages that the standards place, including:
        - Standardized method of handling/transmitting more difficult and complicated data across a number of ECUs
        - Developed with the extra “tough” and “harsh” nature/environment that many heavy-duty commercial vehicles operate within in mind
        - Much more functional due to the larger range of data rates (from the previously standardized 250KBps to 1MBps
    - Key Concepts discussed by SAE J1939
        - Uses a 29-bit CAN ID
            - Can switch between 29 bits and 11 bits depending on whether CAN FD is used or not
        - Messages are identified by Protocol Group Numbers (PGNs) that are 18 bits long
        - Signals sent/received to/from ECUs are known as Suspect Parameter Numbers (SPN)

- CAN Flexible Data-Rate (CAN FD)
    - While the original CAN protocol has been used for decades, the increase in innovation and development in the automotive industry has caused the number of messages, along with the size/length of them, to skyrocket, especially in recent years. The maximum message size of 8 bytes was starting to get exceeded, causing manufacturers to figure out overly complicated solutions to work past this.
    - CAN FD fixes this problem in one, common, standardized manner extending off of the original CAN protocol.
        - Maximum message size is increased from 8 bytes to 64
        - Data bitrate is increased from 1MBps to 5 to improve speed
        - Overall increases network bandwidth by 3-8 times
        - Ability to dynamically switch between different message lengths and data rates
            - This also includes being able to switch between an 11-bit identifier and a 29-bit identifier

Engineering Constraints

There are a few main constraints for this project; message capacity, speed, and computational power. The data capacity of CAN/CAN-FD limits the size of messages that can be transmitted to 8 bytes for CAN and 64 bytes for CAN-FD. Additionally, there is a minimum required transmission speed to meet safety requirements. Transmission speed is a fundamental requirement of the project. The ECU's must be able to send these messages so that the systems can react quick enough to protect the user. For example, deceleration must occur the instant the brakes are applied. Signals sent by the brakes, however, must still follow J1939 communication protocols. Ensuring each bit is transmitted in a timely manner is the main engineering constraint. The final constraint applied by the project was the computational power in an ECU. The encryption library chosen to secure these messages being transmitted had to be lightweight enough that the ECU's could run them.

Security Considerations

The primary purpose of the team's project revolves around vehicle security. Security threats are pertinent at both the hardware and software level. Hardware attacks can be made by placing a CAN sniffer device on the CAN bus; the device possesses the ability to read data being sent within the vehicle. A CAN sniffer may then be coupled with another device to manipulate and transmit corrupt data. Signals sent by the brakes to decelerate the vehicle, for example, may be tampered with to be ignored. These manipulations would hinder and cause detrimental functions by the vehicle. Software attacks may be used as well. Pushing illegitimate software through cellular towers, as many vehicles use for communication, would cause security issues at

the fleet level. It should be observed these software manipulations can be controlled from a remote location, therefore, rendering vehicles in real-time control of someone other than its driver. It is of utmost importance the data being sent within a vehicle is only controlled by its driver, and it is legitimate.

These security threats will be managed through an encryption and decryption system among valid ECUs. Messages sent will be encrypted by the **crypto_box** function within TweetNaCl; a lightweight, asymmetric key encryption protocol used to ensure message transmissions are not time consuming, or less than 5mS. Received transmissions will then be decrypted with the respective **crypto_box_open** function. Valid public-private key pair exchanges will allow the transmission of authorized, safe messages across the CAN bus. The overall implementation of TweetNaCl, alongside the other programs used to simulate the system, are described below in great detail.

Implementation and Testing

The implementation of our project involved building a simulated environment in which virtual representations of ECUs could communicate with one another. The communication transmitted over this channel is also designed to be encrypted in order to ensure both ECU authentication and message security. The implementation process can be broken down into four separate segments: initialize and communicate using CAN-FD over the vcan interface, integrate the J1939 standard, implement encryption and decryption into the communications, and implement a key-management protocol.

To enable the simulated communication of ECUs on a CAN bus, the team initialized an interface on a linux machine called vcan0 with a Maximum Transmission Unit (MTU) of 72

bytes. This interface is of type vcan which stands for Virtual CAN. This allows the interface to act in the same manner as an interface connected to a bus of ECUs using either the CAN or CAN-FD protocol. To switch between CAN and CAN-FD or any other CAN protocol, the MTU of the interface must be changed to align to that protocol's standards. The team used an MTU of 72 to ensure compatibility with both CAN and CAN-FD.

Once the interface was initialized, the team then tested ECUs' ability to communicate with one another over this interface by sending raw CAN and CAN-FD frames over the bus. To do so, utilization of a C library called socketcan allows the instantiation of CAN or CAN-FD frames and populates them with information. Communication was tested by sending messages of both valid and invalid lengths to ensure that the interface limited the MTU correctly. Once the interface communication was configured correctly, the socket configuration was altered to comply with the J1939 standard. This was a relatively simple task since socketcan contains flags that allow for J1939 configuration when used in conjunction with CAN or CAN-FD frames. To test the protocol, logic was implemented to determine whether the messages being sent over the socket were of valid length and the frames had valid CAN IDs populated.

The bulk of the time our team spent on this project was in implementing the TweetNaCl encryption protocol which is an extremely lightweight, asymmetric key encryption protocol. TweetNaCl has two main functions: **crypto_box** and **crypto_box_open**. Both functions take in a public key, a private key, a nonce (a number used only once), and an output buffer as parameters. The **crypto_box** function takes plaintext information and encrypts it using the encryption keys and the nonce and places the ciphertext into a buffer. The **crypto_box_open** takes the ciphertext and decrypts it using the same nonce and a valid public-private key pair. While implementing this encryption protocol, it was assumed that keys had already been successfully exchanged prior

to encryption and decryption occurring. This is because the team needed to navigate NaCl's nuances relating to encryption and decryption, but the receiver and transmitter must both have the correct keys to do so.

In addition to the selection of our encryption protocol outlined above, the team was also required to design a key management protocol for this system. The protocol the team designed has two main components: key exchange and key storage. In our proposed protocol, each ECU has its own public-private key pair, and one ECU is designated as the key server. All key pairs for each ECU and the vehicle are considered long-lived meaning they are not reset on startup/shutdown, and must be reset manually. The key server is responsible for keeping track of all ECUs public keys, and it distributes the shared vehicle public-private key pair during initialization. The key server tracks all authenticated ECUs by maintaining a key manifest containing all of the public keys of each ECU as well as the shared vehicle public-private key pair and its own key pair. Upon key exchange, the key server sends a message to each ECU encrypted using its specific public key containing the shared vehicle public-private key pair which is stored for future use by each ECU. Once this initialization is completed, the system is ready to engage in encrypted communication. The team has currently implemented the manifest using indexed text files, one for public keys and one for private keys, stored in long-term memory. Logic to implement the key exchange process is yet to be implemented.

Appendix I

The team is mainly focusing on using a virtual environment, a Linux system running on a virtual machine, to demonstrate the functionalities of the design on CAN FD communication. As mentioned above, the team decided to utilize the CAN and TweetNaCl libraries in C for our implementation. There are a few essential components needed to set up a virtual environment with CAN network communication properties. This can be done by first using some simple commands as listed below (all commands were placed in a bash script to ease the setup):

Note: These commands should be run with a user account with sudo/root privileges.

1. modprobe vcan
    - modprobe is a command to add or remove modules in a Linux system
    - vcan is the virtual CAN interface module in Linux Kernel
    - This command loads the virtual CAN interface module into the system
2. ip link add dev vcan0 type vcan
    - ip command is a tool for system or network administrators to configure network interfaces
    - link is a subcommand to modify and display the states of all network interfaces in a system
    - This command adds a new device named vcan0 with vcan as the device type
3. ip link set vcan0 mtu 72
    - This command set the maximum transmission unit (MTU) of vcan0 to 72 bytes
4. ip link set up vcan0
    - This command brings the device vcan0 up online and running

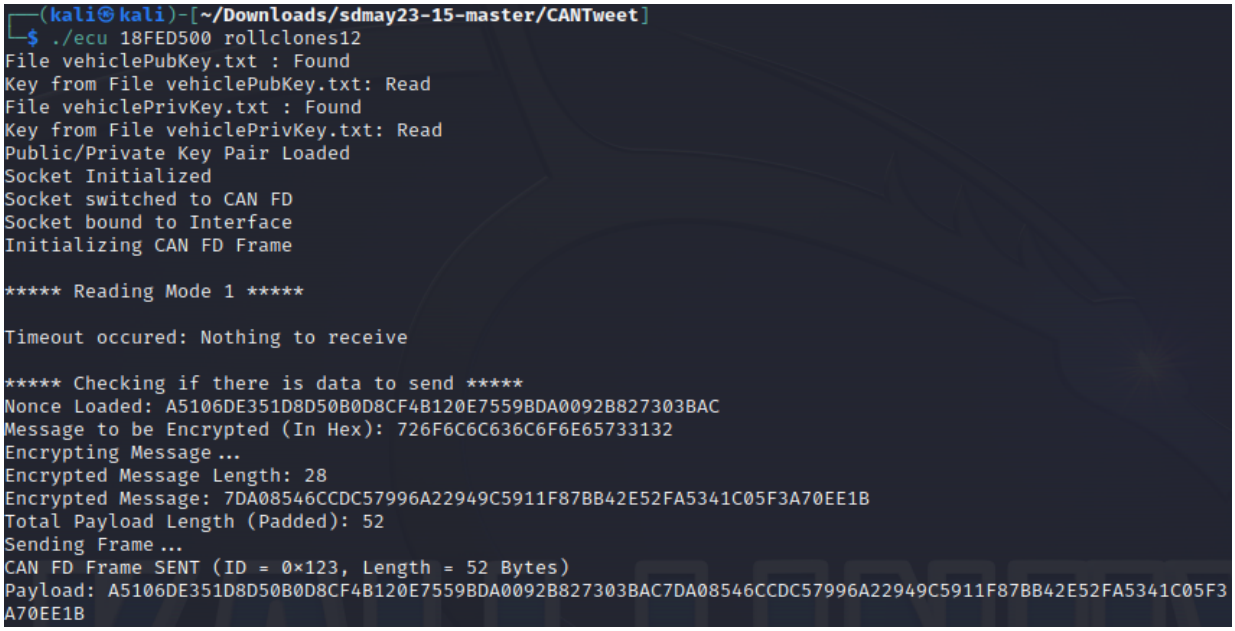
Now, the virtual machine with CAN network communication properties is set up. The following are the steps breakdown of how to demonstrate or test our design implementation in the system:

1. Make sure the bash script (setup_vcan.sh) is ran to set up the virtual CAN device
    - Use command: sh setup_vcan.sh
2. Next, the user could start off with compiling the ecu.c file with message.c file into a single ecu executable file
    - Use command: gcc ecu.c message.c -o ecu
    - ecu.c file
        - It has the properties of an ecu which will receive messages from other ecu, and send messages to other ecu whenever it is needed.
        - Each ecu will always be in receive mode (listening on the socket for a message designated to it). If there is nothing to receive, it will check if there is any message waiting for it to send to another ecu. If there is nothing to be sent, it will go back to receive mode. That is the behavior/process cycle of the ecu.
        - The message will be encrypted before sending to make sure that only the designated ecu could read the message, and only the designated ecu could decrypt the message that is being transmitted.
        - For more detailed information on the encryption and decryption of the ecu, please refer to the “Information and Testing” Section above.
    - message.c file

        - It contains the struct for message to create a message, set value to the message variable, get value of the message variable as well as clearing the value.
        - The functions will be called in the ecu.c file.

3. Create nodes for CAN network communication
    - One terminal window on a Linux system is considered as one node on the CAN network.
    - Each terminal will need to run the executable ecu file created in Step 2.
4. Test the connectivity of nodes by adding valid length of message
    - Example: ./ecu 18FED500 rollclones12
    - Valid message lengths: 8, 12, 16, 20, 24, 32, 48, 64 Bytes

Demonstration of the system:



When there is nothing to receive, it will check for a message to send, if yes, it decrypts and sends the message.

```
***** Reading Mode 2 *****

Data available
Scanning for Frames ...
Frame Received!
Frame Data: A5106DE351D8D50B0D8CF4B120E7559BDA0092B827303BAC7DA08546CCDC57996A22949C5911F87BB42E52FA5341C0
5F3A70EE1B
Nonce Extracted: A5106DE351D8D50B0D8CF4B120E7559BDA0092B827303BAC
Ciphertext Extracted: 7DA08546CCDC57996A22949C5911F87BB42E52FA5341C05F3A70EE1B
Decrypting Message ...
Message: Decrypted!
Plaintext Message: rollclones12
Frame ID              Length of Frame Data          Length of Plaintext Message        Message (Hex)
0×123                 52                            12                                 72 6F 6C 6C 63 6
C 6F 6E 65 73 31 32
```

This screenshot shows that it found data that is available to receive. After receiving the message, it will decrypt it and show the message in plaintext.

```
***** Reading Mode 3 *****

Timeout occured: Nothing to receive

***** Checking if there is data to send *****

Timeout occured: Nothing to receive or to send!
```

After the data is read, the value in the message variable will be cleared, so there will not be any data available to read or send.

Appendix II

The team came up with multiple designs after thorough research of articles regarding security in CAN network communication. The initial proposal design that the team had was to make use of the standard CAN frame. The CAN frame structure allocates portions of bit fields for separate packet information. A 15-bit CRC (Cyclic Redundancy Check) field present within the whole 64 bit CAN frame only provides error detection support by sending checksum code. The team's proposed design was to make use of this CRC field to place the hashed public-private keys (asymmetric key) and also the Message Authentication Code in this 15-bit part of the frame.

The modern security solution of applying cryptography in ensuring the integrity of messages makes it capable for the proposed design to disregard the CRC checksum code and use it for hashed keys and MAC. Since it is placed within the CAN frame, this ensures there is no additional load of traffic which incurs more bandwidth and slower processing time in the CAN network. The embedded CAN frame reaches out to all the ECUs connected in the can network. The receiving ECUs would be able to use the hash to decrypt the key and verify the MAC code to check for the authenticity of the key.

This initial version of the design was a symmetric key exchange as only one key is being passed through the CAN frame. The decision on using a symmetric key exchange was to meet the speed and efficiency of data transmission within the CAN network. As mentioned above, the security solution that would compensate for the absence of the CRC in this design was the abundance of modern cipher algorithms in this case would be lightweight AEAD (Authenticated Encryption with Additional Data) cipher. This is a cipher algorithm that can check the integrity and validate the data communicated within the messages, more specifically CAN payload, to ensure confidentiality and integrity will be maintained throughout the CAN Network. To reduce the delay that this security algorithm might incur, our design also implemented the forward keystream glock generation so that the required key stream would be ready to be encrypted during the transfer of the previous message.

This initial design that met all the project requirements of having a secure key exchange for CAN network communication with faster performance was pitched to the client. The client was impressed with the design but scrapped the proposal due to the constraints in the project's requirements. The CAN network deals with all seven layers of the OSI model and the key exchange protocols will only be able to access or work on the network and application layer (the

upper layer). The exchange protocol would not have exchange to embed or make modification to the physical packet structure of the CAN frame where the proposed initial design would not be able to switch CRC checksum code with hashed key and MAC. In addition, the usage of legacy CAN protocol in the prior design instead of the extended CAN FD protocol limits the packet size that can be sent through the CAN network. This might incur delay which affects the performance of data transmission. These constraints switched our approach on solving the secure key exchange problem with the design that the team has for the project now. The final design that the team was able to implement makes use of the CAN FD protocol which allows 0-64 bytes per of data per frame and integrates asymmetric key exchanges (public-private key usage) for secure CAN communication. The team's research for the project and learnings from the failure of the initial design provided us the space to explore and have deeper understanding on the problem that was trying to be solved.

Appendix III

The design and implementation of Mobile Vehicle Cybersecurity with Onboard Key Management is described above in great detail; any final remarks should be made in regard to the overall vehicle functionality. The team found it quite impressive to learn how a vehicle operates: approximately seventy sensors relaying data to roughly seventy ECUs, each of these ECUs transmitting and receiving information to one another, and all of this being accurately accomplished while traveling seventy miles/hour on uneven terrain. It is simply incredible. Learning to further develop the system, by encrypting/decrypting each message in a timely manner, served as a strong means to completing our undergraduate studies at Iowa State University.

Appendix IV

Gitlab Link: Click Here to See our Source Code!