Final Report

Made by: SDMay23-14
Name(s): Ryan Scehovic, Ryan Campbell, Cody Stricker, Levi Jansen, Drake Ridgeway, Riley Lawson, Josue Torres

Appendix I:

Setup / Operation Manual

1. Virtual Machine - We use virtual box (https://www.virtualbox.org/) to set up and run a low power VM running Raspbian OS.
2. In the vm use the terminal to install python 3, using: sudo apt install python3
3. Then download CAN with: sudo apt install python3-can
4. After that download SocketCAN with “pip install socketcan”
5. Clone the repository from here into your preferred directory
6. Make sure you “git checkout routing”
7. Navigate to sdmay23-14/Socket_CAN_Tests using “cd sdmay23-14/Socket_CAN_Tests”
8. Type “./startup.sh” and click enter in your terminal. What this does is it initializes the virtual CAN busses, which can be seen if you type “ifconfig” in your terminal and hit enter. The virtual CAN busses are simulated as network interfaces, but can only be accessed via programs utilizing SocketCAN.
9. Now that we have our network set up with our virtualized CAN busses, open 4 tabs. In one of them, you will want to run “python Bidirectional.py”. In another one run “python Bidirectional2.py”. In the third one run “getFrames.py”. In the fourth one run “getFrames2.py”.
    a. **What these do:** Bidirectional are your bridges that are sending and receiving frames, whether it be standard CAN 2.0 frames to the ends vcan0 or vcan2, or whether it be FD for vcan1. The getFrames populate the vcan0 and vcan2, which those messages will be picked up by the Bidirectional bridges.
10. In the getFrames terminals, you will see standard messages being sent of just 8 bytes of data. In the Bidirectional terminals, you will see CAN messages being picked up and packed into FD frames and sent. Additionally, you will see FD frames received by the opposing Bidirectional file as it’s sent. We created a randomizer to ruin the hash + monotonic counter so we could demonstrate that our verification functions do work.
11. To see an even further implementation, you can run “./runMaster_2.sh” which will run all of the previous functions, but in new windows unfortunately rather than tabs. Additionally, it will pull up 2 ECUs from each vcan network so you can see and confirm that messages are properly routed to sent onto each network when needed.

Project Structure and Individual Demos Explanation

When navigating to the Socket_CAN_Tests directory you will see several files. The three main functional components of our program are sendFrames.py, packFrames.py, and receiveFrames.py. Each of these components has a fairly straightforward function: sendFrames.py reads CAN frames from a file, determines which ones need to be routed across the bus, and sends them onto the vcan0 virtual network. Next, packFrames.py reads each individual CAN message from the vcan0 network and packs them into a CAN FD frame while also appending our two validity checkers: a CMAC tag and a monotonic counter value and sends those frames onto the vcan1 network. Lastly, receiveFrames.py reads packed CAN FD frames from the vcan1 network and unpacks them while checking their integrity and validity on the system. The unpacked and validated frames are sent onto the vcan2 network and are read by the devices (ECUs) on the other side of the network.

With this structure there are also several files for testing the functionality of our program. We built an ECU representation which looks for messages that would be intended for a specific device and outputs the values it finds. These mock-ECUs are found within the ECUdemo folder and can be run individually or all at once using the provided shell file.

Another set of files used for testing are the example man in the middle attacks that are located within the ValidationDemo folder. This folder has mock attacks that can be fed into the program to prove functionality with our CMAC validation and monotonic counter catching.

Evolution of Design since 491:

Going from 491 into 492, we had a few design changes, such as not using any hardware and just focusing on software. Once we shifted our focus, we went into the implementation phase, where we took our ideas, and turned them into functional code. We started developing prototypes using C, utilizing stdin and stdout in order to pass data between programs. Our first phase included CMAC functions, and reading messages from a file. From there, we brainstormed ideas for packing frames, and made critical design decisions for how the container for our messages operate. Many of the decisions made came down to how many bytes would be allocated for things such as a freshness value, CMAC tag, etc. After this, came the development of a piece of software that processes messages and packages them to be sent out onto the network.

Once many of the design decisions were made, we switched gears and started using python to develop the CAN security mechanisms. Most of what was done in C was taken and transformed into a python variant. This is where things started to gain traction, and actual results were seen. With the help of our client and advisor, a program was made that packed five messages into a CAN FD frame, added the necessary CMAC tag and freshness value, and then sent out onto a virtual CAN network. Once we were able to get the packed frames sent out onto the network, we started developing a way to route the messages to various nodes, or ECUs. This is where a lot of the complexity came from, as we were running into issues with implementing full-duplex communication over the network bus, and thus not able to route the messages to the desired nodes. This was resolved by refactoring the code, and turning it into more digestible pieces. From this point is where we started wrapping things up and preparing a demonstration for the work we accomplished.

Functional and Non-functional Requirements:

The Mobile Vehicle Security Bridge software will be able to detect instructions sent by a malicious user, whether they be replayed instructions that a hacker reads while sniffing on the local CAN network, or artificial. These fraudulent messages will be ignored which will allow the vehicle to keep functioning normally.

The final version of the software is intended to be loaded into a device which will act as a component of a vehicle. The component is a box that needs to be durable, able to withstand any situation, holding the technology inside. The user shouldn't have to worry about their car being tampered with due to this device. The user should never need to activate and run the code or worry about its inclusion in the vehicle. The product passively sits, defending any cyber attack against the user's vehicle that may come its way. If the user attempts to hack their own car, this device will prevent that.

Relevant Standards:

We followed our engineering constraints and standards from 491 and that did not change. The engineering standards we followed were J1939 and AES-128. J1939 allows us to use CAN to communicate packets on the CAN bus. J1939 is a higher layer that allows more complex communication. AES-128 is an encryption protocol that uses a 128 bit key to encrypt and decrypt a block of messages. We used these in our second semester of this project.

Engineering Constraints:

The engineering constraints are mainly in the form of software performance and compatibility. Our software is required to be capable of reading J1939 vehicle CAN frames and keeping them in-tact through their entire journey through our code. The performance of a vehicle running our software should not be affected and function as it did before. We are required to keep our software lightweight to allow it to be supported by a vehicle CAN network. The software's final intention is to be loaded into a small device attached to a physical CAN network, so it must run with the idea of a single input and output. Message rejection is required to be handled automatically without any user interaction or GUI display. The software is required to act on its own without management or initiation by an outside source. The software must be able to run for the duration that a vehicle's CAN network is in use, this is to say that the software is not allowed to pause or reboot while running on the system.

Security Concerns + Countermeasures:

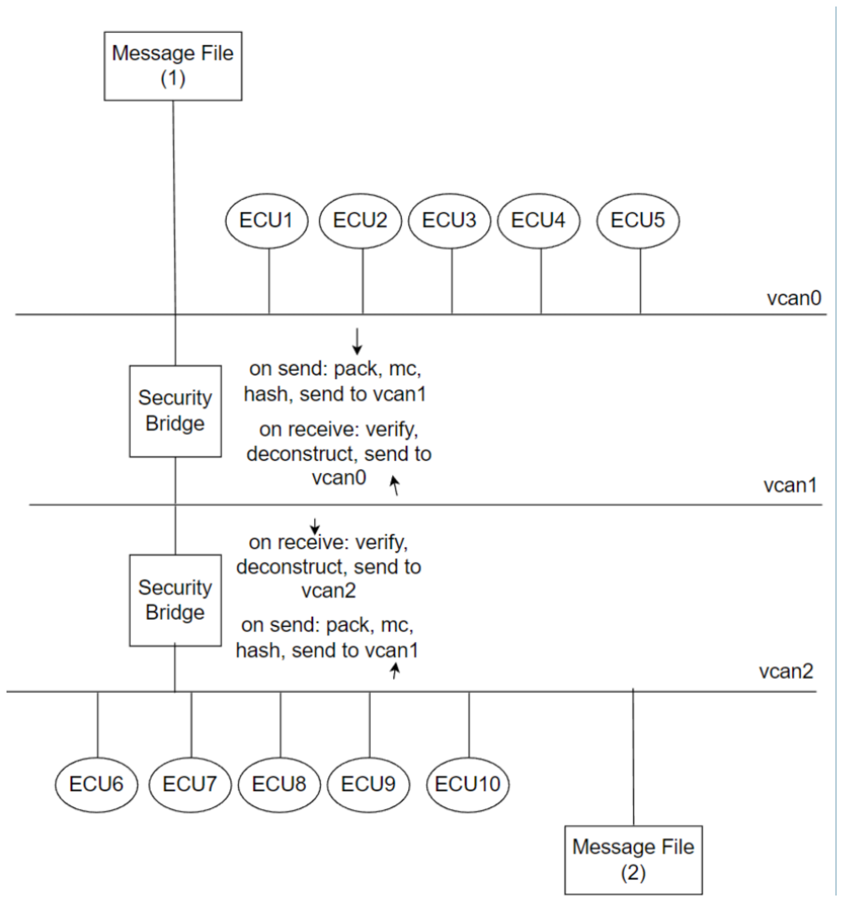
- **Physical Security**
    - The buses are stored deep within the car, so the adversary wouldn't be able to tamper with the buses without physically destroying the car. More than likely they would destroy parts of the car and the ECUs before they'd physically be able to tamper with the bridges that handle messages.

- **Digital Security**
    - The only concern for this would be confidentiality. We did include encryption and decryption as a future work project, though we didn't get it implemented. It's not too much of a concern as the hacker doesn't know the secret key used for the AES-CMAC function. It's not given to us for this project, so we can assume that the hacker won't be able to get it either. The only thing they would be able to interpret about our messages are the counter at the very end as that is the only thing very apparent about the messages being sent on the FD bus, which would be vcan1. Like it's stated, encryption would help obfuscate this in the future, but as of now, it doesn't pose too much of a threat.

Implementation Details:

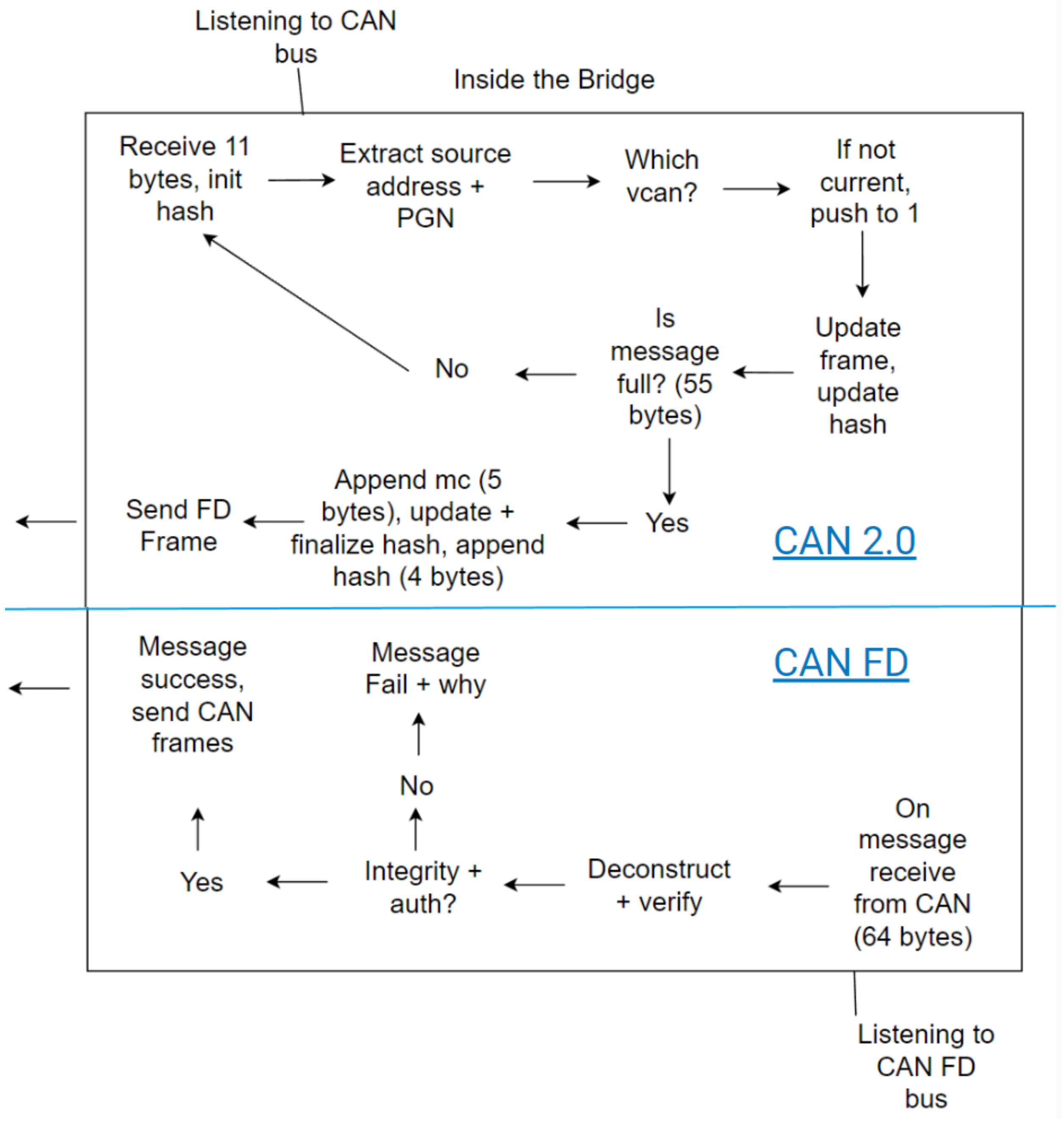
How we went about this was upgrading basic CAN 2.0 Frames which have a data payload length of 8 bytes and extending it out to 64 bytes. The frame architecture that allowed us to do this was CAN FD, which extends the data payload length and allows for multiple CAN 2.0 messages to fit within. How we did this was by taking in both the arbitration ID value and the data payload of the old CAN 2.0 frames, and packing a total of 11 bytes per message into CAN FD. How we utilized CAN FD was by packing 5 CAN 2.0 messages, which would provide us a use for 55 of the 64 bytes. The remaining bytes would be filled in by (5 bytes) the monotonic counter, which would provide a freshness value, and (4 bytes) the AES-CMAC hash function, which would provide integrity in the forms of a digital signature.

How the monotonic counter works, is per FD frame sent, the monotonic counter value is increased by one, hence the term "monotonic". The CMAC hash function was used to loop over each message in the frame, take note of it, update the hash value for each message, and finally the counter. Then it would be finalized and appended as the last 4 bytes in the 64 byte FD frame before being sent.


Message File (1)
ECU1
ECU2
ECU3
ECU4
ECU5
vcan0
Security Bridge
on send: pack, mc, hash, send to vcan1
on receive: verify, deconstruct, send to vcan0
vcan1
Security Bridge
on receive: verify, deconstruct, send to vcan2
on send: pack, mc, hash, send to vcan1
vcan2
ECU6
ECU7
ECU8
ECU9
ECU10
Message File (2)

With the picture above, this is our general virtual structure for simulating a CAN bus in a vehicle network. We made two Message Files, denoted Message File (1), and Message File (2) to have CAN 2.0 frame messages that have a data payload length of 8 bytes. What these messages are destined for are the opposite vcan's that they get distributed on. For example, if a message file has messages being deployed onto vcan0, then those would be messages set for the ECUs as a destination on vcan2. The purpose for this is so we can demonstrate the use of our Security Bridge, which has the packing, **AES CMAC**, and **Monotonic Counter** functions so we can provide two pillars of security to Vehicular Networks: **Authenticity and Integrity**.

Upon receiving a frame from either vcan0 or vcan2, we will pack it into our FD frame, and if it's full (5 CAN messages received), we will add the counter, add the hash, finalize the message and send it to vcan1, which is the FD CAN bus that can handle the 64 bytes for data payloads.

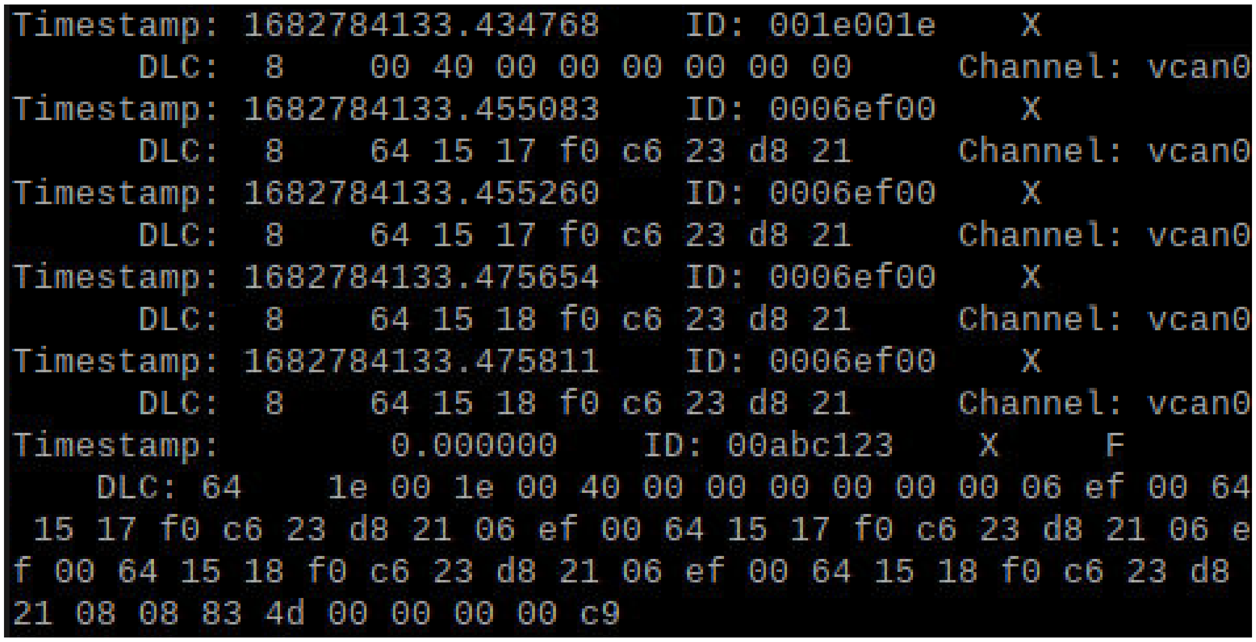

Listening to CAN bus
Inside the Bridge
Receive 11 bytes, init hash
Extract source address + PGN
Which vcan?
If not current, push to 1
Update frame, update hash
Is message full? (55 bytes)
No
Yes
Append mc (5 bytes), update + finalize hash, append hash (4 bytes)
Send FD Frame
CAN 2.0
CAN FD
Message success, send CAN frames
Message Fail + why
No
Yes
Integrity + auth?
Deconstruct + verify
On message receive from CAN (64 bytes)
Listening to CAN FD bus

With this picture, it is demonstrated how our bridge actually interprets messages whether they are CAN 2.0 messages or CAN FD messages. For CAN 2.0 messages, it receives the 11 bytes, packs it into an FD frame by extracting the source address, PGN, and data, verifying that it is destined for the opposite-end vcan by doing a source address lookup. If the frame is full, it will append the counter and the hash for all of the data excluding the hash itself for the data payload section of the FD frame. Then, it gets sent to vcan1. Upon receiving FD frames, the bridge breaks down all the different individual messages and verifies the CMAC tag with the known secret-key to ensure the integrity of the received message and ensure that it was sent from a validated sender on the network who also has the secret-key. Additionally, with the

Monotonic Counter value, it checks it against the last known 64 values to ensure that it is not stale, meaning less than that of the smallest Monotonic counter value, and ensuring that it has not been seen before. This helps against replay attacks. If the verification test passes, the message is broken down to 5 individual CAN 2.0 messages and sent to whatever CAN 2.0 bus the receiving bridge is currently connected to, either vcan0 or vcan2.

Testing Processes & Results:

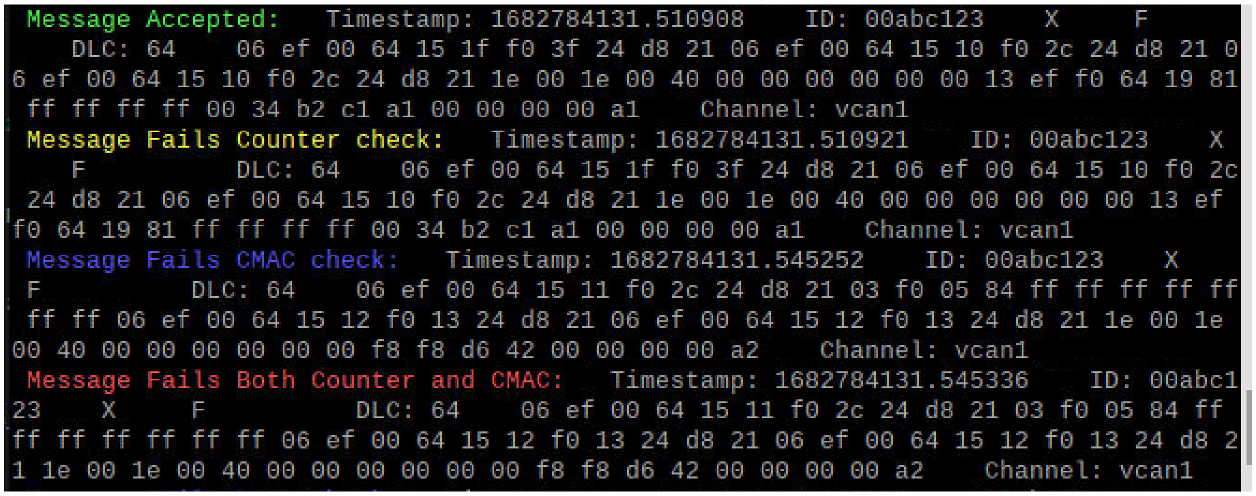
To test, we ran different shell scripts that would send different sets of CAN frames through our system to see if the CAN FD frames were built correctly. After checking if they were properly built, we tested if they could be sent/received from the bridges and then properly distributed to ECUs based off of the PGN and Source Address of the CAN frames packed inside the CAN FD frame. Below here, you can see an example of 5 CAN frames being read by a Bridge and then the FD frame being sent from the Bridge:



```
Timestamp: 1682784133.434768    ID: 001e001e    X
      DLC:  8    00 40 00 00 00 00 00 00     Channel: vcan0
Timestamp: 1682784133.455083    ID: 0006ef00    X
      DLC:  8    64 15 17 f0 c6 23 d8 21     Channel: vcan0
Timestamp: 1682784133.455260    ID: 0006ef00    X
      DLC:  8    64 15 17 f0 c6 23 d8 21     Channel: vcan0
Timestamp: 1682784133.475654    ID: 0006ef00    X
      DLC:  8    64 15 18 f0 c6 23 d8 21     Channel: vcan0
Timestamp: 1682784133.475811    ID: 0006ef00    X
      DLC:  8    64 15 18 f0 c6 23 d8 21     Channel: vcan0
Timestamp:        0.000000    ID: 00abc123    X     F
    DLC: 64    1e 00 1e 00 40 00 00 00 00 00 00 06 ef 00 64
 15 17 f0 c6 23 d8 21 06 ef 00 64 15 17 f0 c6 23 d8 21 06 e
f 00 64 15 18 f0 c6 23 d8 21 06 ef 00 64 15 18 f0 c6 23 d8
21 08 08 83 4d 00 00 00 00 c9
```

Each CAN frame takes up 11 bytes in the CAN FD frame because we are taking its PGN (2 bytes), Source Address (1 byte), and Data Payload (8 bytes). As the CAN frames are added a CMAC tag is being updated and then after the final CAN frame is added, the CMAC tag is finalized and the first four bytes of it are added to the CAN FD frame. We couldn't take all 16 bytes from the CMAC tag because of our limited space, so we had to pick a realistic number which we considered in the 3-6 byte range. We ultimately chose 4, because we also needed to fit a freshness value (also known as monotonic counter) and the odds are very low that the first 4 bytes of a CMAC tag of a tampered message would line up with our valid CMAC tag. For our freshness value we allocated 5 bytes because that allows us to send 2^40 messages (5 bytes = 40 bits) before having to reset our key. That gives us approximately 1 trillion CAN FD frames that we can send, and of course we never hit that number, but in a real life situation with how many computers and user inputs a vehicle has, there are thousands of messages being sent during operation so we wanted to allocate plenty of space so that our system wouldn't have to go through a key reset all the time.

Below we have a screenshot showing a message being accepted and the 3 ways a message failures can occur:



```
 Message Accepted:   Timestamp: 1682784131.510908    ID: 00abc123    X     F
    DLC: 64    06 ef 00 64 15 1f f0 3f 24 d8 21 06 ef 00 64 15 10 f0 2c 24 d8 21 0
6 ef 00 64 15 10 f0 2c 24 d8 21 1e 00 1e 00 40 00 00 00 00 00 00 13 ef f0 64 19 81
 ff ff ff ff 00 34 b2 c1 a1 00 00 00 00 a1     Channel: vcan1
 Message Fails Counter check:   Timestamp: 1682784131.510921    ID: 00abc123    X
    F          DLC: 64    06 ef 00 64 15 1f f0 3f 24 d8 21 06 ef 00 64 15 10 f0 2c
 24 d8 21 06 ef 00 64 15 10 f0 2c 24 d8 21 1e 00 1e 00 40 00 00 00 00 00 00 13 ef
f0 64 19 81 ff ff ff ff 00 34 b2 c1 a1 00 00 00 00 a1     Channel: vcan1
 Message Fails CMAC check:   Timestamp: 1682784131.545252    ID: 00abc123    X
 F          DLC: 64    06 ef 00 64 15 11 f0 2c 24 d8 21 03 f0 05 84 ff ff ff ff ff
 ff ff 06 ef 00 64 15 12 f0 13 24 d8 21 06 ef 00 64 15 12 f0 13 24 d8 21 1e 00 1e
00 40 00 00 00 00 00 00 f8 f8 d6 42 00 00 00 00 a2     Channel: vcan1
 Message Fails Both Counter and CMAC:   Timestamp: 1682784131.545336    ID: 00abc1
23    X     F          DLC: 64    06 ef 00 64 15 11 f0 2c 24 d8 21 03 f0 05 84 ff
ff ff ff ff ff ff 06 ef 00 64 15 12 f0 13 24 d8 21 06 ef 00 64 15 12 f0 13 24 d8 2
1 1e 00 1e 00 40 00 00 00 00 00 00 f8 f8 d6 42 00 00 00 00 a2     Channel: vcan1
```

The message is accepted if none of our validation checks fail. If any single validation check fails then we don't want to accept the message because we know the contents of it got tampered with or it was a replayed message. This helps prevent two very common attacks on a CAN system because if a hacker is able to continuously replay a message, for example like the message for accelerating, clearly, that would be a problem and could cause a serious accident. There are occurrences where a message received could fail both checks, for example, if they have Message 1, which is telling the engine to accelerate and start replaying that on the system, but then they also want to add in telling it to steer left, they might take that message and try to modify the contents of it and then send it. This would cause the CMAC check to fail, because we recalculate our CMAC tag with the received frame and if it doesn't match up, we know something was tampered with. The freshness value check would also get triggered because a message with that value would have already been seen.

Work Context:

- **Related Products:**

Other related products that are relevant to our design are software and features that are built for cars, such as using an app to control the temperature of a car/RV or being able to recognize the speed or temperature of your engine. There are plenty of utilities out there that utilize and recognize what your car is doing using the CAN network to determine gas mileage and collect data tailored to that particular automobile. This applies to all vehicles and since CAN is in almost every automobile the assurance that our cars, buses, and RVs (for instance) won't be taken control of without our knowing. We used tools that aided our progression such as Open Garages, utilized AES encryption, and there are many tools that allow you to read, write, and analyze a can network that can be located at can-awesome (GitHub).

- **Related Literature:**

Jeep Cherokee Hack

In 2015, hackers remotely hacked into a Jeep Cherokee with someone inside it, driving at 70 MPH down the interstate. The driver explains that his car, on its own, started blasting the AC at its coldest temperature, the radio started to switch channels on its own, to which the driver tried to stop it by turning the dial and hitting the power button, which did not work. The volume of the radio was being tampered with as well, putting the radio up to full volume. Then, the windshield wipers got turned on at full speed and sprayed the wiper fluid on the windshield. Lastly, the screen inside the car displayed a picture of the two hackers who took over his Jeep Cherokee. Very shortly after the picture was displayed, the attack immediately stopped because the hackers lost connection.

The hackers used a Zero-Day exploit that gave the hackers control of the vehicle over the Internet. The exploit allowed them to send commands through the entertainment system into the dashboard, gas, brakes, steering, etc. This was very relevant to our project because we were tasked with designing code to defend an attack such as this.

Progressive Insurance Dongle Hack

In 2015, Corey Thuen warned of security flaws in Progressive's "Snapshot ODC-II" Dongle. This dongle is used to track a customer's driving habits. Thuen tested this device on a 2013 Toyota Tundra, this vehicle was found to operate with no security at all. No encryption and the firmware is not signed nor validated in any way, and no secure boot function. The CANbus in the vehicle is in the same network as the basic vehicle functions. Such as braking, accelerating, steering, etc. This is a prime situation for a Man-In-The-Middle attack since the bus has no security, anything that can get into the bus can communicate with the vehicle. So anyone who can control the dongle has full control of the vehicle. This is relevant to our project because our code can defend against a Man-In-The-Middle attack such as this; any message that is not validated by the CMAC is completely ignored in our program.

Appendix II:

- Version 1:

  Initially we intended to write our project using the C programming language. This was planned because we had the most experience with this language. However, we ran into issues with compatibility such as C not having many resources and libraries related to SocketCAN, a tool we had planned to use as a foundation for our project. After a few weeks of work and meetings with our client it was clear that the program was going to need an overhaul with a language change, and we turned to python. This original version had a few basic features such as message sending and receiving and a template for CMAC tag, but due to the complications with the tools we were using, this was as far as we had gotten.

- Version 2:

  Using the foundation we had built using the C programming language we overhauled our program converting the code into python. With python, integration with the SocketCAN library was much cleaner with far fewer lines of code for the same tasks done in C. The basic functions of the program were able to be rebuilt in just about three days of effort for the team. In this version of the project, we had several basic features implemented such as message reading, messages sent over frames as well as a start to more complex additions like message packing, counter implementation, and a proper CMAC validation system.

- Version 3:
  The final version of our program consisted of code evolving from version 2 to now incorporate a better structure of sending CAN frames, receiving CAN frames & building CAN FD frames, and then sending those built FD frames. For this we had a three file structure with files being named sendFrames.py, packFrames.py and receiveFrames.py. This structure is what eventually evolved into our current bidirectional bridge files. Once we were able to send messages and receive them going one way, we were able to use a lot of the same logic to reverse it and send/receive frames the other way as well, except now we had to account for rules around when our bridge should be reading vs sending messages. Luckily message processing happens so fast and there are so many messages being sent on the system, it'll never just be stuck for seconds at a time, many messages are coming in fractions of a second apart, so this made it easy for us to allow it to receive a CAN frame and add it to the CAN FD frame and then afterwards checking if there was any CAN FD frame that it needs to unpack coming from another bidirectional bridge.

Appendix III:

We learned many things throughout this year and the project as a whole. An example of something we learned as a team was that a solution is definitely possible for cybersecurity in the most simple vehicles, vehicles you would not expect to have/need cybersecurity. One other thing that we learned is you hear about many stories of people scrapping their entire project and starting from scratch. As a programmer, you would never want to experience that, nor do you think it'll ever happen to you. But, it happened to us because of our switch from C to Python. We had some code to reference from C to Python, but changing did end up being for the better.

We learned how important communication is in a group, especially one as large as ours. When seven people are in a group working together towards a common goal, information can often be lost between people. We started working in smaller subgroups as we wanted to avoid losing information between people and, in theory, work faster and come together later to combine work. However, this did not go as planned as some subgroups did not fully understand what each group was working on, other than their own. We decided to scrap the groups and work together as one focusing on one big goal. This resulted in everyone knowing what they were working on and encouraged others to work together. Being in a group of seven was difficult, however, we knew communication was very important going into this. Even so, we all still learned more about communication and the benefits it brings forward.