Automobile Vulnerability Assessment using Ghidra

Robert Hill, Loic Tchuenkou, Vinton Morris
Dr. Kevin Kornegay, Dr. Md Tanvir Arafin
Morgan State University Cybersecurity Assurance and Policy Center, Baltimore, MD

Background

The vehicle subsystems used for this work are from a popular 2020 vehicle. These subsystems involved include the Cockpit Control Unit(CCU), Telematics Unit, Central Gateway, and the Harness Instrument Panel. Ghidra is used throughout this research to analyze firmware received from the vehicle. Ghidra is an open-source reverse engineering software used to disassemble binary machine code back into a more user-friendly language. Qemu is also used to run the firmware in a virtual environment for testing. Qemu is an open -source processor/CPU emulator and virtualizer. This work also involves assessment of vulnerabilities on the CAN bus. CAN is an international Standard Organization (ISO) defined serial communications bus originally developed for the automotive industry to replace the complex wiring harness with a two-wire bus.

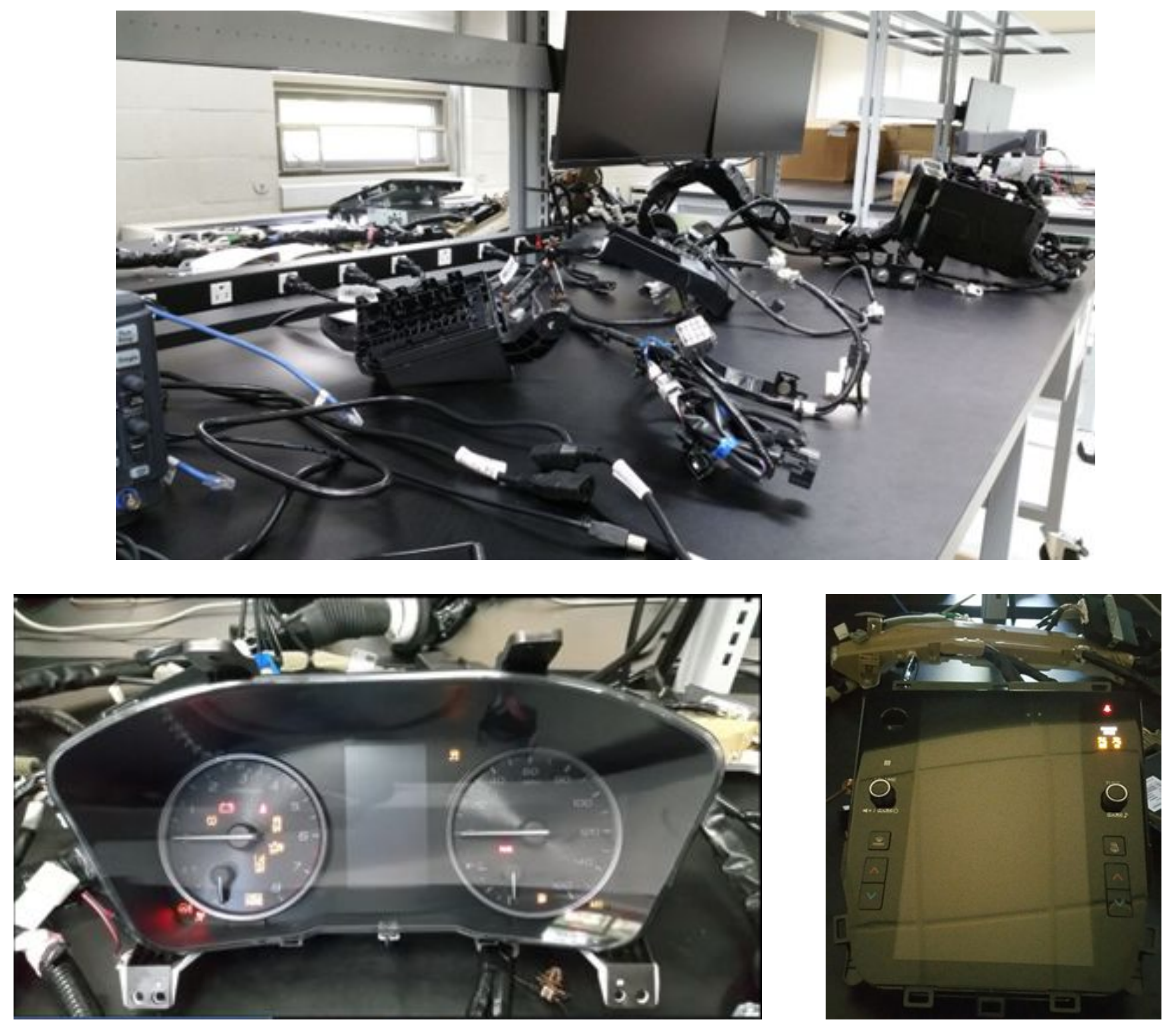


Figure 1: Testbed configuration

Abstract

Today's vehicles have advanced embedded system technology driving the complexity of their navigation, infotainment, and safety systems. The vehicle infotainment system is responsible for the smooth operation of all communication and entertainment functions, including navigation services, music, podcasts, phone calls, and video streaming. This system is also interconnected with many other critical systems to support the vehicle's operation, including external cameras and sensors. This presents the potential for many possible vulnerabilities that can be detrimental to the car's security, leading to many safety concerns in the case of future attacks. This research focuses on the infotainment-related electrical systems of a 2020 vehicle to allow in-situ testing of various cyber techniques and algorithms to mitigate attacks through various control surfaces, i.e., CAN bus, Wi-Fi, and Bluetooth.

Technical Approach

This research was done in stages:

❖ Stage 1: Firmware Exfiltration and Analysis
  ➢ The CCU's NAND flash memory was desoldered from the PCB
  ➢ The firmware was extracted from the flash memory using an eMMC reader
  ➢ The firmware was immediately digested into Ghidra for assessment

❖ Stage 2: Firmware Emulation
  ➢ The Firmware was first emulated in QEMU using the extracted files and the information learned from digesting them in Ghidra.

❖ Stage 3: Vulnerability Testing
  ➢ Messages are sent on the different CAN communication lines, to look for a response or reaction.

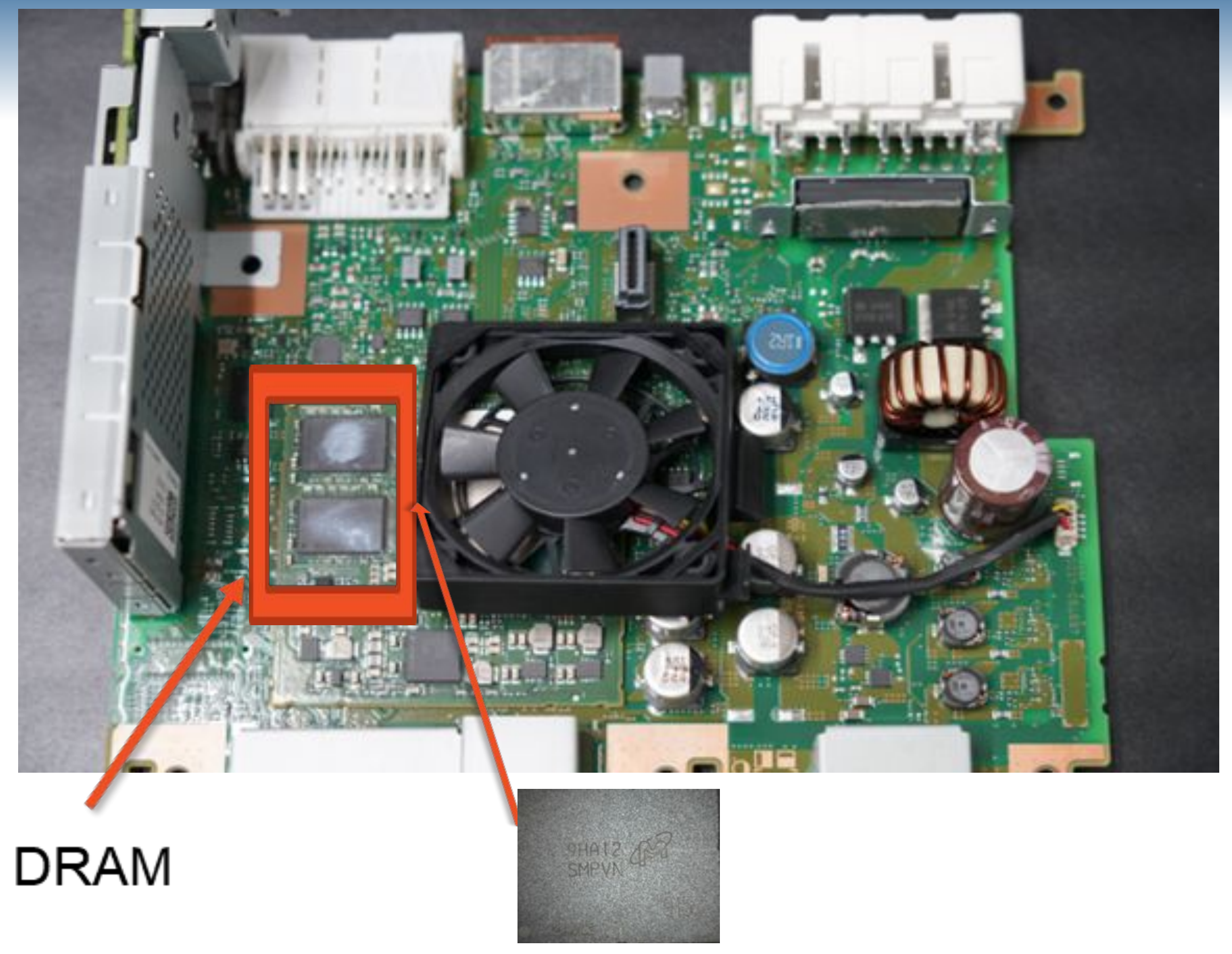
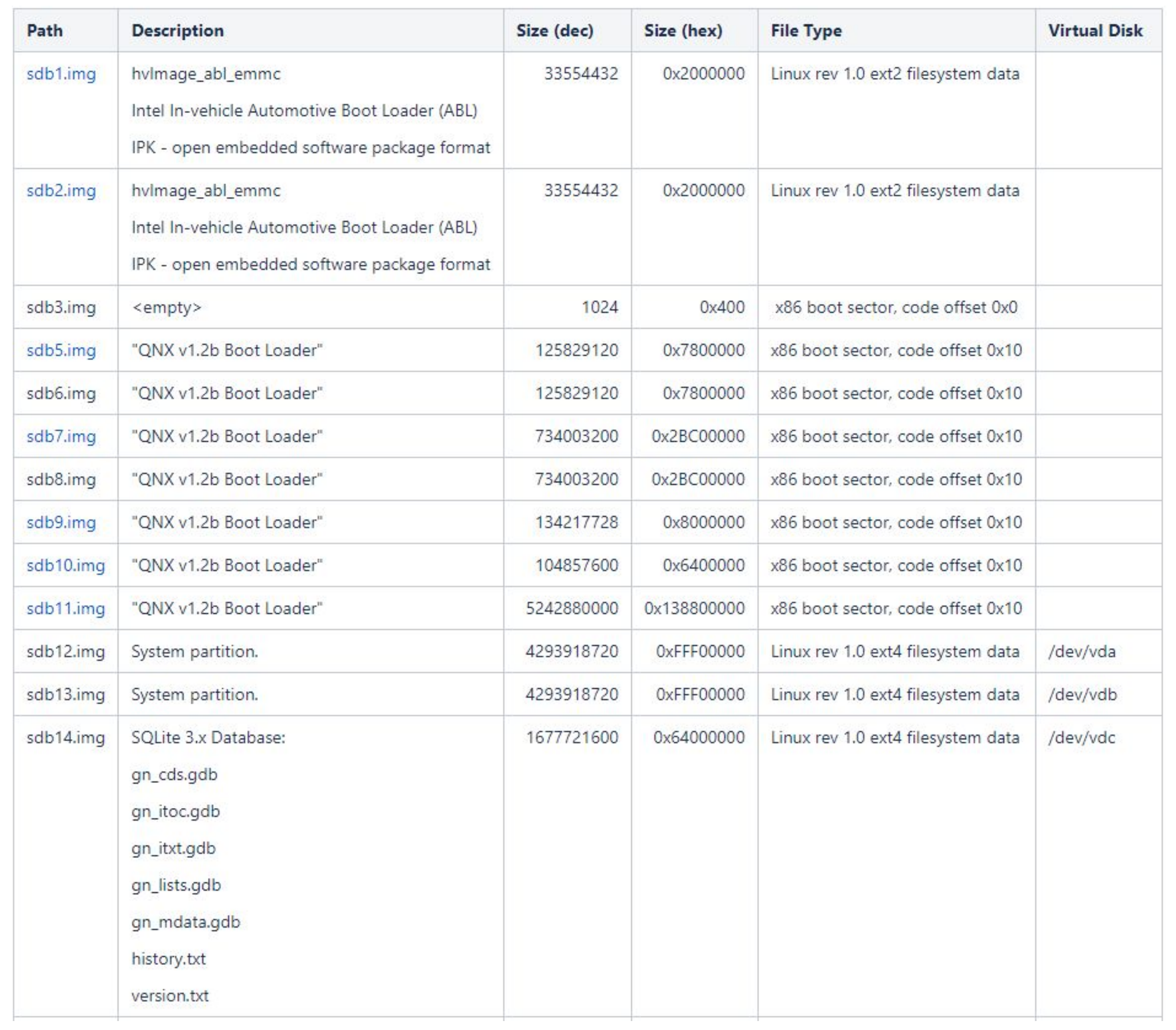


Figure 2: CCU



Path	Description	Size (dec)	Size (hex)	File Type	Virtual Disk
sdb1.img	hvImage_abl_emmc Intel In-vehicle Automotive Boot Loader (ABL) IPK - open embedded software package format	33554432	0x2000000	Linux rev 1.0 ext2 filesystem data	
sdb2.img	hvImage_abl_emmc Intel In-vehicle Automotive Boot Loader (ABL) IPK - open embedded software package format	33554432	0x2000000	Linux rev 1.0 ext2 filesystem data	
sdb3.img	<empty>	1024	0x400	x86 boot sector, code offset 0x0	
sdb5.img	"QNX v1.2b Boot Loader"	125829120	0x7800000	x86 boot sector, code offset 0x10	
sdb6.img	"QNX v1.2b Boot Loader"	125829120	0x7800000	x86 boot sector, code offset 0x10	
sdb7.img	"QNX v1.2b Boot Loader"	734003200	0x2BC00000	x86 boot sector, code offset 0x10	
sdb8.img	"QNX v1.2b Boot Loader"	734003200	0x2BC00000	x86 boot sector, code offset 0x10	
sdb9.img	"QNX v1.2b Boot Loader"	134217728	0x8000000	x86 boot sector, code offset 0x10	
sdb10.img	"QNX v1.2b Boot Loader"	104857600	0x6400000	x86 boot sector, code offset 0x10	
sdb11.img	"QNX v1.2b Boot Loader"	5242880000	0x138800000	x86 boot sector, code offset 0x10	
sdb12.img	System partition.	4293918720	0xFFF00000	Linux rev 1.0 ext4 filesystem data	/dev/vda
sdb13.img	System partition.	4293918720	0xFFF00000	Linux rev 1.0 ext4 filesystem data	/dev/vdb
sdb14.img	SQLite 3.x Database: gn_cds.gdb gn_itoc.gdb gn_itxt.gdb gn_lists.gdb gn_mdata.gdb history.txt version.txt	1677721600	0x64000000	Linux rev 1.0 ext4 filesystem data	/dev/vdc

Figure 3: Portion of the files pulled from the flash memory

Preliminary Results

The firmware was successfully extracted from the flash memory. Some of the files and their descriptions are shown in figure 3. These descriptions were assessed through Ghidra analysis. The firmware was also successfully emulated in virtualized hardware using QEMU. We have not yet been able to get a response from CAN bus messages.

Future Work

Next steps include:

❖ Running the firmware on our own hardware to physically test for potential vulnerabilities.
❖ Further testing of the CAN communication bus and addition of more systems onto the bus.
❖ Extracting firmware from other subsystems in the vehicle testbed.

Acknowledgments

• National Security Agency (NSA)
• Cybersecurity Assurance and Policy Center (CAP)

References

[1] D. Stabili, "READ : Reverse Engineering of," vol. 14, no. 4, pp. 1083–1097, 2019.
[2] T. Huybrechts, Y. Vanommeslaeghe, D. Blontrock, G. Van Barel, and P. Hellinckx, "Automatic reverse engineering of can bus data using machine learning techniques," Lect. Notes Data Eng. Commun. Technol., vol. 13, no. November, pp. 751–761, 2018, doi: 10.1007/978-3-319-69835-9_71.
[3] W. Choi, S. Lee, K. Joo, H. J. Jo, and D. H. Lee, "An Enhanced Method for Reverse Engineering CAN Data Payload," IEEE Trans. Veh. Technol., vol. 70, no. 4, pp. 3371–3381, 2021, doi: 10.1109/TVT.2021.3063261.