Decoding the GMLAN Network

Reverse engineering Chevrolet Volt CAN messages using General Motors GDS2 software, Arduino-based CAN logging hardware, and artificial-intelligence.

Overview

The objective of reverse engineering controller area network or “CAN” messages from a vehicle communication network is universal to car hackers wishing to speak the language of their car. Automakers guard the proprietary decoder ring or can database file, called a DBC, to obfuscate the messages on the vehicle network, making cars more secure and harder to modify. For someone trying to understand and adapt the use of the components in their vehicle understanding CAN messages is critical.

This document explores the resources a highly motivated CAN hacker might have available, and methodology to use these resources to decode vast portions of the vehicle CAN network without manually annotating data, a prohibitively laborious process. The GMLAN network is used here as it is Maxwell Vehicle’s primary objective, however the strategies employed can be generalized to almost any vehicle.

Vehicle CAN Busses

The internet of things model that has been widely adopted by many automakers utilizes serial communication busses to aggregate logic and control of the intelligent systems in the vehicle, allowing individual nodes to communicate information relevant to their specific function to the rest of the systems in the vehicle. This network consists of messages that have an identifier, called a CAN ID, and a data payload of up to 8 bytes. This information is encoded into a CAN frame. CAN protocols, hardware layers and transceivers are not described in this exercise past assuming an engineer has successfully configured a device with a CAN transceiver to sniff data from the vehicle bus.

CAN networks are designed with the database file exposed, and the encoding of messages is known. Without the DBC file, the encoding and resulting dependencies are difficult to ascertain. The payload of certain data packets can be interpreted by using CAN reverse engineering tools to visually monitor and annotate the data in real time for certain functions, like vehicle or engine RPM, where this information is readily presented to the driver. Past this relatively narrow set of functions, decoding the bus becomes more complicated.

CAN Tools

In attempting to decode the CAN frames, the fundamental problem is annotation. Unannotated, raw CAN frames are easily sniffed from an an active bus, but lack context without an understanding of the function associated with a given ID and data. With good CAN logging hardware and software, available for a little as $85 from Comma Ai with their Panda OBDII kit, one can begin to manually sniff and annotate frames to produce a reverse-engineered DBC file. Unfortunately, it is often the case that most messages cannot be decoded without more insight into the vehicle behavior.

Traditionally, technicians have used diagnostic tools conforming to the OBDII protocol to diagnose cars. These devices follow a unique subset of the CAN protocol defined for UDS or the universal diagnostic system, that generic OBDII readers use to gather diagnostic information about the vehicle. These devices request information from devices on the CAN bus using the CAN network, but the request and response messages and independent of the network's standard operation and are generally absent when the OBDII tool is removed.

Some automakers have developed proprietary tools that use the vehicle CAN bus for more in-depth diagnosis. These tools often perform additional functions like programming and module testing, which are outside the scope of OBDII. General Motors developed a software tool called GDS2, or Global Diagnostic System 2, which utilizes a Bosch VCI (Vehicle Communication Interface) to display information from the vehicle controllers on a technician's computer. This display includes annotations for vehicle data labeled with units in real time.

Some of these messages act as requests for data, while others are interpreted from the bus and displayed, decoded in the GDS2 software.

Reverse Engineering Methodology

Pairing the annotated data, as viewed using GDS2, with raw CAN frames by synchronizing and correlating the data can provide direct insight into the vehicle behavior, without the need for manual CAN data annotation. The method described here uses a combination of CAN logging, computer vision, and machine learning to correlate CAN frames to annotated data from GDS2. The reverse engineering process flowchart breaks the methodology into a series of simple steps.

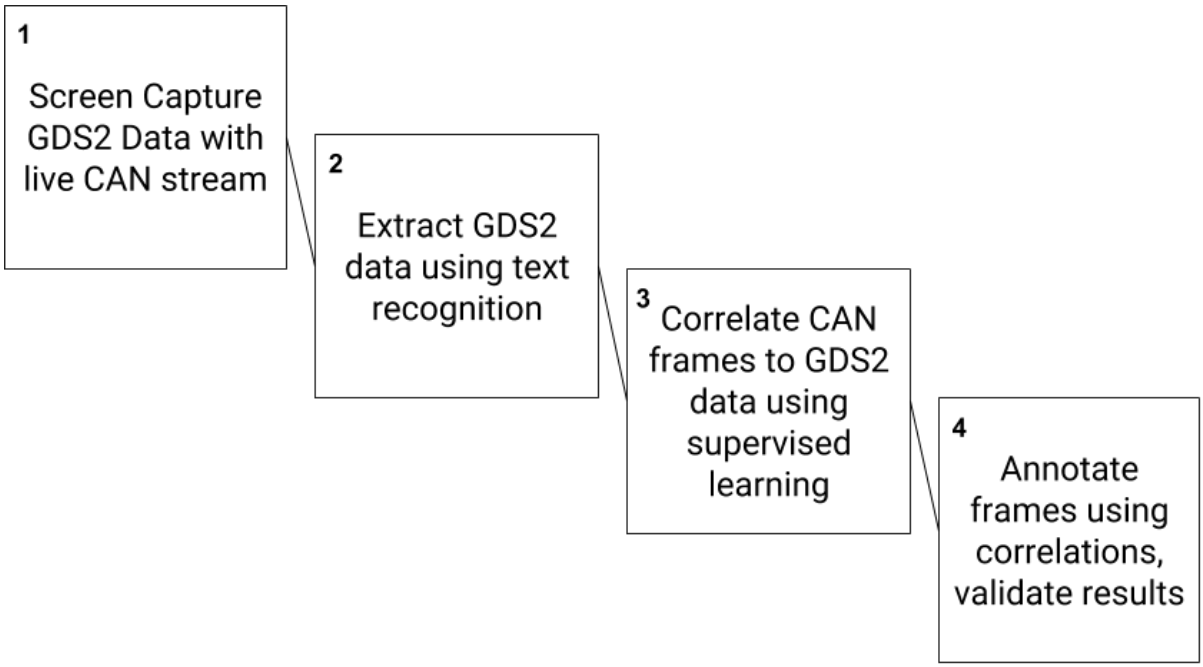

Figure 1 - Automated Reverse Engineering Process Flowchart

Step 1: Simultaneously Capturing GDS2 and CAN data

The first step in the process simultaneously harvests the raw and annotated data from the network. The CAN data stream is saved to a .csv file which includes a relative timestamp, ID, and data. This information is also streamed on screen alongside the GDS2 window, allowing for relatively loose correlation between the CAN frame recording time and the GDS2 recording time, assuming the latency of both systems from bus to display is approximately the equal. We will discuss later why this assumption is safe, and how it is accommodated. The GDS2 display is recorded using screen recording software, producing a video file that contains the recorded CAN stream and annotated display of specific values. A sample image from one of these recordings is included below for reference. The CAN stream is included on the left, while the GDS2 software display is on the right, with the Value column updating in real time.

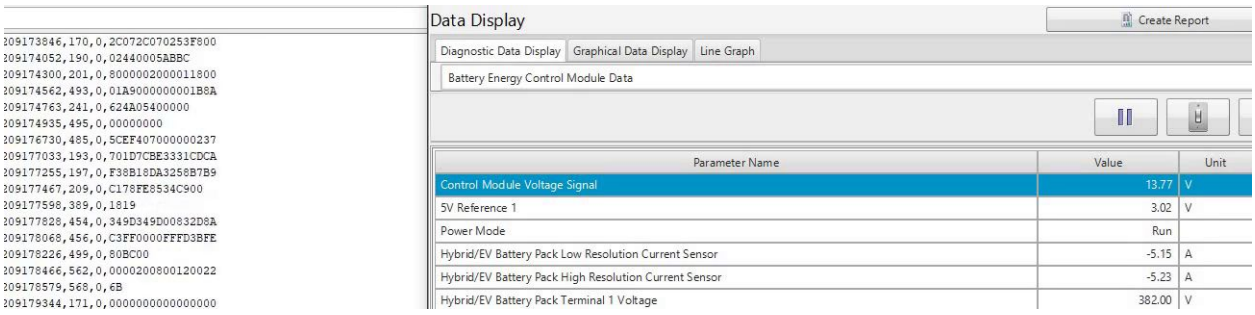

Figure 2 - Screen Capture of raw CAN and GDS2 data.

Step 2: Extracting the GDS2 Data

The screen capture must now be turned into annotated data in a .csv or .json format for use in step 3. In order to take the screen capture and convert it to .csv data, we employ text recognition using pytesseract.

At this point in the process, the data migrates from the logging computer, a Windows environment running GDS2 and the Arduino Serial Monitor, to Unix where OpenCV and tesseract process video frames individually. A bash script using ffmpeg tool splits the video into individual frames, saving them to a new directory called “output_images” where they are read during text recognition.

It is important to note that the screen recorder we are using captures video at 10FPS, so a 115 second video yields 1150 frames. The 10FPS sample rate is included in the conversion process to timestamp the outputs. With the screen capture split up into individual frames, tesseract text recognition processes the images and decodes the text in the frame. Tesseract splits up the text into individual lines, and our filtration method separates the lines based on labels. These labels are user-specified at the beginning of the process, in order to ensure that only the data of interest from the recording is being captured. This text parsing process is not generalizable and relies on knowledge of the GDS2 format to omit the need for text detection (i.e. finding where the text is) prior to recognition (translating to text).

Extracted data is filtered based on knowledge that the data is either signed or unsigned, will always only have a decimal between numerals, and may only prepend numerals with a “-” sign. This formatting knowledge makes cleaning the detected data simple.

The result of this process is a file “parameter_name.csv” which is “Hybrid EV Battery Pack Terminal 1 Voltage.csv” in the training example. This file is that parameter’s data, encoded by “time, value”

Step 3: Correlating CAN Frames to GDS2 Data

Now that we have two concurrently recorded datasets in .csv formats. Of note, the raw CAN log should always be started before the screen capture, and stopped after, so that the raw data is not the limiting factor. Because of unknown latencies in the CAN logger and GDS2 stack, it is nearly impossible to exactly correlate the GDS2 display at start of the screen capture to a raw frame recording time. By recording a short interval before and after, we are able to

to a specific CAN frame being this information is fed to a machine learning model to generate correlation between raw-data and and the annotated information. The result is a list of annotations for the raw data, pairing a byte or set of bytes transmitted by a specific CAN ID to a parameter name as extracted from GDS2. This annotation also includes scaling and offset

information for the raw binary data to translate between the value displayed in GDS2, and the binary decoded from the CAN bus.

Some critical knowledge of the bus is essential when setting up and feeding the model. Stated here explicitly, we know that message data for numerical values is either unsigned or 2s complement. Data may be broken up into multiple bytes into big-endian groupings. We also know that data may not be broken up into even byte chunks, with information encoded in 1,2,4,8,12,16 or more bits, sometimes starting mid-byte. Finally, we know that the sample rate of the GDS2 data and the raw bus data is not necessarily the same. The 10FPS sample from the video log, coupled with the GDS2 sample rate, may yield an update interval that is not necessarily consistent with the interval of message transmission on the bus. The key is that the general behavior of the data is the same over time, and the larger the sample the pool, the more information the model has to work with when correlating the two datasets.

Step 4: Annotating and Validating Frames

Because the model will produce annotations based on a confidence threshold, with some degree of uncertainty or error, these correlations will need to be validated by the reverse engineer.

Training and testing the model using known data offers a straightforward way to validate the system's accuracy with little headache. In this system example, we developed and tested using "Hybrid EV Battery Pack Terminal 1 Voltage" which produces unsigned-values between 280.00 and 400.00 and has known correlation to CAN ID 711, starting at bit position 31 with 12 bits of length, big-endian. This value has no offset, but is scaled by 0.125 from the binary representation. The C manipulation of this raw data from an 8-byte buffer is;

```
 hv_battery_voltage = 711Msg.buf[4] << 4;
hv_battery_voltage += 711Msg.buf[5] >> 4;
hv_battery_voltage *= 0.125;
```

The learning model applied to detecting the data need not include the bit-shifts required to make the data useful, but it should provide a location, bit-range and size of the decoded data.

Once the development moves past known-data to un-annotated areas, a CAN data viewer, like Comma Ai's Cabana software, coupled with the Panda CAN logger, should check the model against the GDS2 data by plotting the value side-by side. This process must be exercised until the model has proved to be sufficiently accurate.

The final annotation may be encoded in a .dbc file, so that they may be used by traditional CAN engineering tools. This step can be done with a DBC generator, or manually using Comma's Cabana during the verification process.

References

PyTesseract

- https://pypi.org/project/pytesseract/
- https://github.com/tesseract-ocr/tesseract/wiki/Command-Line-Usage#tsv-output-currently-available-in-305-dev-in-master-branch-on-github
- https://www.pyimagesearch.com/2017/07/10/using-tesseract-ocr-python/

OpenCV

- https://opencv-python-tutroals.readthedocs.io/en/latest/