On-board Diagnostic(OBD) Reverse Engineering

Murtaza Munaim, Mike Lady, D.J. Waldron, Chris Polis

March 21, 2013

1 Background

OBD, on-board diagnostics, is a manner in which data is retrieved from automobile vehicles. It has been a standard since the 1980's when processing became fast enough to install electronic control units(ECU's) within the car. A Controller Area Network(CAN) is a network type in which there is no central machine handling packets and each device on the network arbitrates communally in order to send messages across the CAN network or bus. The information can be retrieved by reading packet data from the CAN bus which is a bus standard to allow diagnostic tools to interact and query ECU's within the car.

To read data off of the CAN bus you can request that a certain ECU respond with the data corresponding to a parameter identification number. Parameter identification numbers(PIDs) map to a certain function/feature/sensor of the car. This request can be forged into a packet which is sent onto the CAN bus of the car. Data read off of the CAN bus comes as a key, value pair of a parameter identification number and the data representing the PID state that the ECU responded with.

2 Problem Statement

Mentor E-Data is a startup company from San Luis Obispo who is trying to build a product to obtain information regarding automobile motion.Currently they have an application called DrivingBuddy which reads in values from the OBD port in all cars to better gauge how drivers are driving on the road. As well as using OBD data to get driving statistics, DrivingBuddy uses kinematic sensors to better estimate the motion of the car at all times. Currently, they are using predefined PID values that are public knowledge in order to query for information from the car, but this data is not of much use and not available in all car models. The problem Mentor E-Data is facing is the fact that the mapping from PID values of a certain car to the function/feature of the car is not an open standard, and varies greatly between car manufacturers.

The problem needed to be solved for Mentor E-Data was to create a generic way for users of a car to automatically reverse engineer which PID values matched to which features of their own car. Knowledge of which PIDs match to which features is essential as the user can input the results of their aided reverse engineering back into the original DrivingBuddy application, which can then use the reversed PID values to pull more data from the CAN bus. Completely reverse engineering the PIDs of one car is a possible feat. However, the problem of the matter is the shear number of cars you would have to reverse engineer to have enough PID coverage to be useful to customers is very large and not feasible. This project was aimed at a general solution.

3 Analysis and Results

3.1 Interfacing with the Bus

The interface chosen to connect to the OBD port was a bluetooth device which was connected to a CAN bus port underneath the steering wheel of the car. The bluetooth dongle was connected to an ELM327 chip. The

ELM327 chip is a programmed microcontroller which is used to decode messages off of the OBD bus. The ELM327 was used to sample the bits from the physical layer of the CAN bus and decode it based on different specifications. There are multiple protocols at which data from the CAN bus can be read. Each protocol has a different configuration of baud rates, parity and stop bits to initialize the serial communication of receiving and sending packets on the bus. There are about 10 public standards of combinations of baud rate/start bits combinations that the ELM327 chip can support for the CAN bus. The ELM327 chip can decode all 10 of these public standards, and allows a simple serial interface to the decoded data bytes. We did not have to deal with decoding the layer 1 data from the pins into bytes as the chip took care of that for us. The chip also supports configuration commands to manipulate the parameters of the device. This is useful to modify how the device looks like on the network, which also influences which packets the ELM327 will parse.

Messages sent on the bus are like any other routed packet network in that you will not receive packets which are not addressed for your device. This is a sane approach to build a system, but it is not the ideal condition when one is trying to sniff all the traffic on the bus. Also, to make it easier for an end user to use the ELM327 over some serial device, the header bytes, which are usually appended to packets that are sent on the OBD bus, are stripped by the ELM327 on default. Therefore multiple configure parameters had to be set on the device to allow us to read the CAN bus better. One parameter was the keyword-on parameter. Usually the ELM327 would only accept packets addressed to itself or the broadcast address of the CAN bus. The keyword-on parameter enforced this by parsing the "keyword" or address of the packet. Disabling this option allowed alot more data to be read from the CAN Bus to the ELM327 and finally back to the user via serial device. Interfacing with the bluetooth dongle was relatively simple, as the bluetooth device emulated a serial device which enabled you to communicate with the device at a standard baud rate and received data being fed from the ELM327 device. Then standard serial communication could take place, with commands prefixed with the string "AT" being sent to the ELM327 device, and messages without that prefix would be sent directly to the OBD bus.

3.2 Packet decoding and fuzzing

Once communication between basic communication ELM327 and basestation PC was implemented, further investigation into the CAN bus packets could take place. The packet format for a message sent to the CAN bus is as follows:

[Address of ECU][Mode of Operation][Mode Command][Mode Data].

Usually without modifications to the standard ELM327 settings, only the Mode Data is sent back to the user over the terminal interface. To make a generic solution to this problem, we decided to build a generic OBD packet generator, and packet error checker, also known as a protocol fuzzer. The fuzzer would be used to help us better understand the OBD packet structure and the range of outputs we would be able to receive from the OBD bus. We were able to make our packet generation intelligent because we knew which bytes of the packet bytes were describing mode, command, or data.

References found within the ELM327 data sheet indicated that mode 0x1 is the most commonly used data mode and supported by car manufacturers. Thus we concentrated on the data packets coming from this mode. The other modes of the car were also fuzzed, but returned different packet formats(mode and command bytes which should have been 1 or 2 bytes were 3 or 4 bytes). It seemed unfeasible at the time to generate a packet fuzzer for all modes of the OBD packet, and thus mode 0x1 was the target of most of the investigation. It is also the mode where Mentor E-Data was previously extracting data from. A packet fuzzer/decoder for mode 0x1 was written for which the basis of was simple. We would request the ELM327 device to try to read every single command type possible, so from 0-255, the entire range of a byte, which is the byte space given in the packet for commands. This helped us quickly identify the number of PID's on mode 01 that the car supported. This gave us a good start on narrowing down which PIDs to even send, as sending erroring PID's resulted in "NO PACKET" message being errored back by the CAN bus.

3.3 Trying to derive semantic meaning Strategy

Once a master list of valid PID's were gathered for a particular car, more advanced reverse engineering can be done. So, syntactically we know the elements needed to get the car to spit out data. We needed to build a semantic understanding of the data coming back in order to create a map of car features to PID value. The

number of bytes coming back in the Mode Data field of the CAN bus packet was variable, which meant some bytes were single bytes indicating some sort of flag while other fields were full integers describing some analog value from a car sensor. We built a data decoder that would decode bytes from the packet using different possibilities of bytes coming from the bus. Our decode type list was the cartesian product of [[big endian, little endian], [byte, halfword, word], [signed, unsigned]]. This method didn't prove worthwhile, as trying to derive meaning for each PID's data bytes was too cumbersome and didnt provide results in automatically determining relation.

3.4 Car Dump/Difference Strategy

We decided on another strategy. With knowledge of all PID's that the car will respond to, we can create a "car dump" which is the complete capture of all packets from the output of the car's CAN bus when it responds to our CAN bus fuzzer. We decided on a common output format of JSON so that our data collection could be easily used for further data analysis.

We know that there is a PID in the car that will change when a certain action in the car is taken. An action can be turning on the headlights, putting on the seatbelts, hitting the brake or accelerator pedals. By doing a single action in the car, creating a car dump and then repeating this test multiple times, we can compare data of previous runs and try to correlate the state of PID data with the action that took place when the car dump was taken. These car dumps were very useful for trying to reverse which PID's match to which action.

We developed some python code to do a difference algorithm on the car dumps of various runs in order to determine which PID's changed when a car action/feature was activated. We noticed that sometimes the data from subsequent car dumps would diff incorrectly, even when no car action was taken. We realized that there were multiple PID values that would change regardless of if an action was taken. This posed a problem for the simple idea of diffing the data from car dumps. To overcome this, we would collect data in this pattern: 1) No action taken, 2) Dump car stats, 3) Action is taken, 4) Dump car stats, 5) Action is untaken, 6) Dump car stats. This 6-step pattern would allow us to remove the PID values that changed constantly regardless of the state of the car being unchanged in our testing. This requires the user to take multiple car dumps per feature the user is trying to reverse engineer.

Again, even after removing the PID values that changed during a halted car, the number of PID's that changed during the selected car action was too large and not enough to narrow down to a single PID. The next step we took was to take multiple runs of the 6 step process. Then, once we know the set of PID's that changed during the same action, we can find the PID that changed the most out of the multiple runs, and then make a guess at the PID that matches to the action we are trying to investigate. This multiple step, filtered process did yield some results, yielding a final list between 2-3 PIDs that would be the "best guess" of which PID matched to which feature. However, more testing is required in order to see how well this process works on a wide selection of cars.

4 Conclusions

The work done with Mentor E-Data was an interesting and challenging problem to pick at proprietary protocols in hopes of trying to extract meaningful information out of it. Much more work can be done in terms of packet decoding, not just on other modes, but mode 0x1 itself. Near the end of the research with the tool, more device options which could have enabled to give more options were discovered. One such option allows us to modify the address of the client device. Further work would allow us to try to spoof the existence of many devices in order to see what sort of other packets can be read from the bus when using different addressing modes.

5 Work Done

Implementation of code and reference sheets: http://bit.ly/WWvkkU