Automated Vehicle Electronic Control Unit (ECU) Sensor Location Using Feature-Vector Based Comparisons

Gregory S. Buthker
Wright State University

Follow this and additional works at: https://corescholar.libraries.wright.edu/etd_all

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Repository Citation
https://corescholar.libraries.wright.edu/etd_all/2155

This Thesis is brought to you for free and open access by the Theses and Dissertations at CORE Scholar. It has been accepted for inclusion in Browse all Theses and Dissertations by an authorized administrator of CORE Scholar. For more information, please contact library-corescholar@wright.edu.
AUTOMATED VEHICLE ELECTRONIC CONTROL UNIT (ECU) SENSOR LOCATION USING FEATURE-VECTOR BASED COMPARISONS

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Cyber Security

by

GREG S. BUTHKER
B.S.C.E., Wright State University, 2017

2019
Wright State University
April 29, 2019


__________________________________________
Junjie Zhang, Ph.D.
Thesis Director

__________________________________________
Mateen Rizki, Ph.D.
Chair, Department of Computer Science and Engineering

Committee on Final Examination

__________________________________________
Junjie Zhang, Ph. D

__________________________________________
Jack Jean, Ph. D

__________________________________________
Meilin Liu, Ph.D.

__________________________________________
Barry Milligan, Ph.D.
Interim Dean of the Graduate School
ABSTRACT

Buthker, Greg, S. M.S.C.S., Department of Computer Science and Engineering, Wright State University, 2019. AUTOMATED VEHICLE ELECTRONIC CONTROL UNIT (ECU) SENSOR LOCATION USING FEATURE-VECTOR BASED COMPARISONS.

In the growing world of cybersecurity, being able to map and analyze how software and hardware interact is key to understanding and protecting critical embedded systems like the Engine Control Unit (ECU). The aim of our research is to use our understanding of the ECU’s control flow attained through manual analysis to automatically map and identify sensor functions found within the ECU. We seek to do this by generating unique sets of feature vectors for every function within the binary file of a car ECU, and then using those feature sets to locate functions within each binary similar to their corresponding control function. This feature algorithm is used to locate candidate functions that utilize a sensor, and then examine the structure of each of these candidate functions to approximate the memory-mapped IO address of each sensor. This method was able to successfully locate 95% of all candidate functions and was able to successfully recover 100% of likely sensor addresses within each of those functions.
Contents

1 Introduction .................................................................................. 1

2 Background and Related Work ................................................. 3
  2.1 Introduction into ECUs ............................................................... 3
  2.2 ECU Firmware Design ............................................................. 6
    2.2.1 Sequential Programming ..................................................... 6
    2.2.2 Real-Time Operating System ............................................. 8
    2.2.3 Contemporary Systems ..................................................... 9
  2.3 Additional Work ................................................................. 9
    2.3.1 Alcyone ECU Tuning and Reverse Engineering ...................... 10
    2.3.2 Speed Limiter Removal ................................................... 11

3 Design and Implementation .................................................... 15
  3.1 Initial Reverse Engineering and Manual Analysis ...................... 16
    3.1.1 Architecture and Binary Selection ...................................... 16
    3.1.2 Manual Analysis ............................................................. 17
  3.2 Radare2 Plugin Design ........................................................... 21
  3.3 CFG manual analysis with Radare2 .......................................... 22
    3.3.1 Sensor Function Similarity Across Binaries ......................... 24
    3.3.2 Bottlenecks .................................................................. 26
  3.4 Automated Feature Vector Generation ..................................... 28
    3.4.1 Automatically Parsing Binary Functions .............................. 29
    3.4.2 Data Structure CFG Representation .................................... 29
    3.4.3 Jaccard Index Function Matching ....................................... 31
    3.4.4 Algorithm Theory and Design .......................................... 32
  3.5 Function Parsing and Sensor Location ....................................... 36
    3.5.1 JSON Function Parser .................................................... 36
    3.5.2 Python Sensor Parser .................................................... 38

4 Experimental Results and Evaluation .................................... 41
  4.1 Methodology .................................................................. 41
  4.2 Function Detection Results .................................................. 44
List of Figures

2.1 Subaru SVX Engine Circuit Diagram ....................... 4
2.2 Fuel Map for a JDM Subaru Engine[1] .......................... 5
2.3 AUTOSAR Example FSM[2] .................................. 10
2.4 List of found addresses in the 1993 EG33 ROM ............... 12
2.5 Speed Limiter Representation in IDA .......................... 13

3.1 Example snippet of raw disassembly text from Skuse’s website 18
3.2 Example Black Box Func ................................. 19
3.3 Control function (left) compared to a 1993 EJ22 function (right) 20
3.4 Example Function in R2 ................................. 24
3.5 R2 Example Function Graph ............................... 25
3.6 Control Function with Bottleneck ........................... 27
3.7 Comparison Function with Bottleneck ......................... 28
3.8 UML Diagram for Each Component of the CFG Object ....... 30
3.9 Slightly different functions to illustrate a mismatch ............. 33
3.10 Example Digram Feature Vector ............................ 34
3.11 Example Bottleneck Feature Vector .......................... 35
3.12 Example Func Match Spreadsheet ........................... 37
3.13 Example Sensor Feature Vector Comparison .................... 39

4.1 List of sensors our search was limited to ....................... 42
4.2 List of functions that access some control sensor, along with the sensor type 43
4.3 Basic idea behind sensor finding algorithm .................... 44
4.4 Algorithm Match Comparison, block-level comparisons vs digram comparisons ......................................... 45
4.5 Total distribution of function matches, digram bottleneck feature vectors ........... 46
4.6 Sensor Match Comparison, as percentage of sensors found within each function .................................... 49
Acknowledgment

I would like to take this opportunity to extend my thanks to my parents and close friends for supporting me throughout the writing of and researching for this work.
Abbreviations

BIN — Binary File
CFG — Control Flow Graph
EDM — European Domestic Market
ECU — Engine Control Unit
FSM — Finite State Machine
IDA/IDA Pro — Interactive DisAssembler
IR — Intermediate Representation
JDM — Japanese Domestic Market
JSON — JavaScript Object Notation
OBD — On-Board Display
R2 — Radare2
RegEX — Regular Expression
USDM — United States Domestic Market
Chapter 1

Introduction

In this report we will discuss the design, implementation, and development of our automated ECU sensor function location system utilizing graphing algorithms and Jaccard feature matching. We will also discuss the methodology through which we generated and obtained our manual analysis data, and the process by which we created the tools needed to expand upon our initial set of data. Finally, we will cover the implementation of our algorithm for extracting sensors from the function candidates, and the overall results of our work. Our ultimate goal is to provide the initial groundwork into a method for automatically identifying, emulating, and error checking ECU sensors by comparing them to reused code found in other binaries.

Understanding and identifying code and function segments within binaries is a time-tested method utilized in reverse engineering all throughout the field of cybersecurity. It is used to generate signatures to identify malicious code, to find vulnerabilities in insecure programs, and to understand and emulate how software functions without necessarily running that software in a realistic setting. This method has been used extensively in identifying malware and vulnerabilities in commercial software, but the application of these methods to embedded software is significantly more limited.

Finding vulnerabilities or common segments of potentially vulnerable code utilized in
embedded systems is especially pertinent as embedded software may go for years without being updated to fix a critical flaw or exploit that may be easily patched in a different system. Public and proprietary library software utilized for reading inputs, processing data, or determining control flow of software may also not go going to be completely re-written between differing implementations of software, allowing for an additional common link to persist after a patch may have been created.

Being able to automatically find, match, and partially emulate the functionality of software that is utilized across multiple systems can allow for automated detection of vulnerabilities in both modern and legacy embedded systems. These vulnerabilities can then be used in order to allow for timely recalls, patches, or updates to improve the safety of some of the most commonly-used machinery of the average citizen’s daily life. Understanding the underlying design and discovering potential security flaws in legacy devices can allow us to develop new methods for fixing similar flaws in contemporary binaries.
Chapter 2

Background and Related Work

The contemporary automobile is constructed using a litany of sensors, microcontrollers, and automotive networking connections to manipulate how the vehicle works as you’re driving it. These calculations and adjustments need to be made for even the most minute value, and need to be calculated automatically as you drive for cars to safely and efficiently operate. At the heart of all of these calculations is a microcontroller known as the Engine Control Unit (ECU).

2.1 Introduction into ECUs

ECUs are embedded systems that manipulate and control how automotive engines operate. At their core, they’re responsible for determining the air to fuel injection ratio, the idle speed of the vehicle, and the timings of various valve functions within the engine of a vehicle. They calculate and adjust these parameters of the engine in real-time by reading several sensors located throughout the vehicle that give the ECU a complete picture of how the vehicle is performing at that moment in time. A natural evolution of carburetors in years past, they allow car manufacturers to optimize fuel injection ratio without significantly
sacrificing fuel efficiency or vehicle performance.

Several sensors are used as inputs to fuel the decision making process of the ECU. An example hookup of these sensors is found in 2.1. They include but are not limited to:

- The Airflow Sensor
- The Vehicle Speed Sensor
- The Engine Speed Sensor
- The Coolant/Water Temperature Sensor
- The Ignition Timing Sensor
- The Throttle Position Sensor
- The Knock Sensor
The input sensor values are used to calculate what outputs the ECU needs to write in order to control the operation of fuel injectors, the firing of coils, or the control of the internal throttle. The inputs determine both the amount of fuel and air to be placed into the coil, the timing of the coil, and any adjustments that need to be made to safely fire off the piston at that instant for optimal engine performance.

Older ECUs utilize fuel maps located within a ROM found on-chip to feed dealer-calculated precomputed values based off of sensor readings to the proper ports on the engine. These precomputed table values are calculated during ideal and complicated testing conditions that are emulated by the dealer. Each index in the table corresponds to the amount of fuel that the injector is told to release at those engine conditions. As you can see in the map 2.2, each index holds a corresponding hex value that is indexed based upon the RPM and Load read within the sensors.

Newer ECUs use complex algorithms to generate optimal results based off more intricate sensor readings. Fuel tables located in the older ECUs are generally based off of a best-case testing scenario - tested on new vehicles in ideal conditions. The latter has an
advantage in utilizing results that adapt to the engine in a variety of conditions, including eventual wear and age that will start to adjust the optimal values needed for performance of the engine. However, older ECU fuel tables can and often are reprogrammed by their owners to "upgrade" their performance to match this gradual decline in vehicle functionality.

2.2 ECU Firmware Design

The modern car has up to 40,000,000 lines of code [3] across all of its internal Electronic Control Units. Design of these systems has remained on the cutting-edge of embedded systems programming to guarantee the safety of the drivers and optimal functionality of the engine in spite of the lofty size of its codebase.

2.2.1 Sequential Programming

Within embedded systems, sequential programming is where software events within an embedded system are hard-coded to execute in a specific order, generally with some sort of timing/resource management that impedes execution until some condition has been met[4]. Generally, these systems have some sort of "setup" or initialization function. This is followed by a main "loop" where resources execute in-order after their conditions are met. A very popular example of a basic sequential programming system is a simple program like the LED flash, which flashes an LED at 1hz continuously until the device is disconnected or powered off. An example of this is below:

```c
void main(){
    while (1) { //loop
        digitalWrite(LED, HIGH);
        delay(1000);
    }
```
This type of programming was present in older ECUs, and works for simple systems as they might not need to allocate resources in a real-time manner. Each function is given 100% of the processor to do whatever task it needs, and would not allow the next task to execute until it is done. A rough pseudocode of how an older sequential system would be set up is outlined below:

```c
void main(){
    setup(); // initialize memory, load PC, begin setup
    while (1) {

        health_check(); // check sensor values to make sure the engine is "healthy"
            // for example, check that battery voltage is ... sufficient
        sensor_values = read(clutch, coolant, speed); // get ... values from sensors
        fuel_injection_rate = read_fuel_map(clutch, coolant, ... speed); // use pre-calculated map values to ... predetermined injection
        write(fuel_injection_rate); // tell engine how much ...
    }
}
```
fuel it should inject to which pistons

write_to_OBD(sensor_values); //send sensor values over ...

OBD to display on dash/update if necessary (errors, ...
speed values, etc.)

Older systems would also only support this type of programming, as you would not
have been able to multithread tasks in an older ECU given their single-core and limited
systems. However, these types of programs are insufficient for newer ECUs which often
have much more complex fuel algorithms that might require real-time calculations for each
sensor. The more intensive the individual sensor calculations, sensor software, and func-
tionality gets, the longer the ECU would block. As a result, the more complex the ECUs
become, the more dangerous and difficult a purely sequential system becomes.

### 2.2.2 Real-Time Operating System

The general structure of a more recent (but not fully modern) ECU follows a model
similar to the main loops outlined in 2.2.1, but these loops are often distributed across
multiple cores with a more complex scheduling system. This requires the use of a real-
time-operating-system (RTOS) to schedule, manage, and distribute the cores of the CPU so
that each function can execute on-time. This allows for important sensor reading functions,
health checking functions, and fuel injection functions to be ran concurrently, increasing
their speed and response time and greatly increasing their safety and optimizing their per-
formance. Several contemporary open-source ECUs are implemented on top of an RTOS
to schedule necessary tasks, with a notable one being rusEfi which was built on top of the
open-source platform FreeRTOS[5].
2.2.3 Contemporary Systems

The modern ECU utilizes a mealy finite state machine (FSM) model to simplify the functionality of these engines while optimizing their speed, ensuring that all inputs are accounted for and that errors do not cause undefined behavior within the machine [6]. Paraphrased from FOLDOC [7]: A finite state machine is an abstract structure that consists of a set of "states" which represent the internal variables/parameters of the structure at that point in time. These may be built on top of an RTOS, but they essentially limit the potential functionality of each electronic control unit within the auto system to only fit the parameters of the state they are assigned to. State transitions are explicitly and strictly enforced by the RTOS, further guaranteeing consistency and safety of each embedded component.

AUTOSAR

AUTOSAR is an open agreement between multiple automobile manufacturers to try to get a standard electronic control unit design platform in use for all vehicles [8][2]. These set an additional baseline of security standards that automotive manufacturers must meet to guarantee safe performance and operation of their vehicles. Vehicles using these standards run an AUTOSAR compliant RTOS which manages the runtime environment that handles transitions between the states of the vehicle.

Figure 2.3 outlines the main states specified for use in an AUTOSAR compliant vehicle.

2.3 Additional Work

Throughout my literature review, I was unable to find any similar methodologies to those that we were proposing. I did, however, find some extensive manual analysis work that proved very useful in designing our algorithm and found some other works that at-
tempted to leverage other knowledge of how Engine Control Units worked to try and identify important sub-sections of code or classify them.

### 2.3.1 Alcyone ECU Tuning and Reverse Engineering

Extensive work on the 1980-1990 model year Subaru binaries were done by Phil Skuse and other contributors through his personal website dedicated to the reverse engineering and tuning of Subaru vehicles[1]. The tuning community at large is dedicated to tweaking and editing various parameters of vehicles, including but not limited to the structure of the binary or the values found in fuel maps in order to optimize the performance of the owner’s vehicle to their liking. Numerous online web forums dedicated to reverse engineering, editing, and altering vehicles can be found with extensive documentation on how to extract binaries, what is found within the binaries, and what modifications can be made. Several open-source tools have also been developed to aid in the process, which also allows for the owners of those vehicles to dump and contribute their own binaries for analysis.
The ECUs that they worked with were based on the Mitsubishi M37791 CPU, which is a subset of the M7700 family that would be utilized in several arcade cabinets at the time, and also a few other applications like the Super Nintendo. Their initial binary dumps were created by adopting the disassembler in an open-source emulator at the time, known as MAME[9]. By running two passes to properly process the data length flag of this architecture, they were able to create full binary dumps of the ROM.

Skuse et al were able to acquire the binaries they used for reverse engineering by utilizing a bug in the built-in Subaru select monitor communication protocol. By repeatedly calling the "Read data from ECU address" function over and over, he was able to demonstrate a method through which one could dump the entire binary through those vehicles with a few simple CAN bus instructions. This would dump all 0xFFFF memory locations in roughly 3-4 hours, due to the low baud rate inherent to the vehicles in question.

Skuse himself managed to post a Memory Map of the United States 1993 EG33 ECU that he did the most work on. He also managed to find a list of sensors, how they were used, and the location and structure of the fuel and timing maps (including the one used in 2.2). This list is shown in 2.4. His repository of various ECU binaries, along with his initial analysis done on one, allowed us to start our groundwork without having to reinvent the wheel and pull these binaries from numerous vehicles ourselves.

### 2.3.2 Speed Limiter Removal

Stanislas Lejay, a computer engineer from France who recently moved to Japan, utilized the public information available through the Alcyone website mentioned in 2.3.1. His goal with this was to reverse engineer and understand how the ROM within his Subaru Impreza STI worked, in order to remove the speed limiting function available within the ROM.

Through his research, he was able to locate the speed limiting function and approximate it using a manual analysis method similar to the one that we would base our manual
<table>
<thead>
<tr>
<th>Mode</th>
<th>Address</th>
<th>Parameter</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>F00</td>
<td>8C3D-8C3F</td>
<td>ROM ID NUMBER &quot;YEAR&quot;</td>
<td>See &quot;ROM Images&quot;</td>
</tr>
<tr>
<td>F01</td>
<td>102F</td>
<td>BATTERY VOLTAGE</td>
<td>volts=value*0.08</td>
</tr>
<tr>
<td>F02</td>
<td>1071</td>
<td>VEHICLE SPEED (mph)</td>
<td>mph=value*1.25</td>
</tr>
<tr>
<td>F03</td>
<td>1071</td>
<td>VEHICLE SPEED (km/h)</td>
<td>km/h=value*2</td>
</tr>
<tr>
<td>F04</td>
<td>106F</td>
<td>ENGINE SPEED</td>
<td>rpm=value*25</td>
</tr>
<tr>
<td>F05</td>
<td>1185</td>
<td>WATER TEMP (F)</td>
<td>tempF=(value-82)*9/5</td>
</tr>
<tr>
<td>F06</td>
<td>1185</td>
<td>WATER TEMP (C)</td>
<td>tempC=value-50</td>
</tr>
<tr>
<td>F07</td>
<td>10A2</td>
<td>IGNITION TIMING</td>
<td>advance=value</td>
</tr>
<tr>
<td>F08</td>
<td>1283</td>
<td>AIRFLOW SENSOR</td>
<td>airflow%=(value*100)/255</td>
</tr>
<tr>
<td>F09</td>
<td>1282</td>
<td>LOAD DATA</td>
<td>load=value</td>
</tr>
<tr>
<td>F10</td>
<td>128C</td>
<td>THROTTLE POSITION SENSOR</td>
<td>throttle%=(value*100)/255</td>
</tr>
<tr>
<td>F11</td>
<td>1280</td>
<td>INJECTOR PULSE WIDTH</td>
<td>msec=value*256/1000</td>
</tr>
<tr>
<td>F12</td>
<td>129D</td>
<td>BY-PASS AIR CONTROL</td>
<td>duty%=(value*100)/255</td>
</tr>
<tr>
<td>F13</td>
<td>1291</td>
<td>02 RIGHT HAND</td>
<td>mV=value*5000/512</td>
</tr>
<tr>
<td>F14</td>
<td>1292</td>
<td>02 LEFT HAND</td>
<td>mV=value*5000/512</td>
</tr>
<tr>
<td>F15</td>
<td>12B0</td>
<td>02 R MAX</td>
<td>mV=value*5000/256</td>
</tr>
<tr>
<td>F16</td>
<td>12B1</td>
<td>02 R MIN</td>
<td>mV=value*5000/256</td>
</tr>
<tr>
<td>F17</td>
<td>12B2</td>
<td>02 L MAX</td>
<td>mV=value*5000/256</td>
</tr>
<tr>
<td>F18</td>
<td>12B3</td>
<td>02 L MIN</td>
<td>mV=value*5000/256</td>
</tr>
<tr>
<td>F19</td>
<td>1097</td>
<td>AFR CORRECTION 1</td>
<td>afc=value-128</td>
</tr>
<tr>
<td>F20</td>
<td>109F</td>
<td>AFR CORRECTION 2</td>
<td>afc=value-128</td>
</tr>
<tr>
<td>F21</td>
<td>12A7</td>
<td>KNOCK CORRECTION</td>
<td>correction=value</td>
</tr>
<tr>
<td>F22</td>
<td>12A5</td>
<td>CANISTER PURGE CONTROL</td>
<td>duty%=(value*100)/255</td>
</tr>
<tr>
<td>F23</td>
<td>125F</td>
<td>ATMOSPHERIC PRESSURE</td>
<td></td>
</tr>
<tr>
<td>FB0</td>
<td>1026-1029</td>
<td>DIAG. READ</td>
<td>See &quot;Reading Error Codes&quot;</td>
</tr>
<tr>
<td>FB1</td>
<td>12B9-12BC</td>
<td>DIAG. MEMORY</td>
<td>See &quot;Reading Error Codes&quot;</td>
</tr>
</tbody>
</table>

Figure 2.4: List of found addresses in the 1993 EG33 ROM
Figure 2.5: Speed Limiter Representation in IDA
analysis on. In 2.5, he shows a disassembly within IDA (Interactive DisAssembler) Pro, a popular disassembly tool, of the basic blocks for that function. Through analyzing this, he was able to determine the exact speed at which the ECU limiter kicked in. The speed was measured within the speed sensor by cycling through a high and low edge generated by the sensor - this frequency would be related directly to the speed of the vehicle.

By looking at this, he determined that the best way to bypass the speed limiter within his vehicle was to fool the ECU speed reading by using pulse width modulation (PWM) from an Arduino Teensy 3.2 to bypass that specific speed input on the ECU board. By using this, he was able to bypass the limited speed of about 180km/h and reach speeds of around 220km/h.

```c
while (speed ≥ 180) {
    if (previous_signal_value) {
        signal_value = 0;
    } else {
        signal_value = 1023;
    }
    analogWrite(VSS_TX, signal_value);
    delay(5); // Speed recorded is around 140km/h
    previous_signal_value = signal_value;
}
```
Chapter 3

Design and Implementation

Within the scope of this thesis, we seek to utilize a custom feature-vector generation algorithm to identify useful sensor candidate functions as they appear within both our control binary and any comparison binaries. By manually searching for similar functions that read/wrote to likely and known sensor addresses across each binary, we were able to observe a list of consistent code blocks that showed up in each binary. Our proposed method of automatically identifying sensor functions within ECUs requires generation of a feature vector based on the instructions within that function, the layout of the basic blocks of that function, and the important bottlenecks that influence the full control-flow of that function. This algorithm was designed using a selection of binaries that were chosen to make the observation and generation of these features easier, while still allowing for us to expand to additional architectures in the future. Each feature vector would then be compared using our similarity algorithms, outlined in Subsection 3.4.4, with the highest match candidate for these algorithms containing the likely candidate function for the code to read/write to these sensors. Utilizing our feature vector generation, the function matching algorithm was able to accurately find 95% of our total candidate functions across each test binary, with an over-all reliability index of 90.4%.

Once the function matches were discovered, these candidate functions are then parsed
through our sensor feature generation algorithm to create feature vectors that attempt to represent some sort of basic semantic meaning for sensors within functions. These feature-vectors are generated for each sensor within our function, with the control feature being used as our "sensor match" candidate. From there, the sensor address would be extracted from this block based on how well they matched up with the control sensor features. Using this, our algorithm was able to successfully extract the sensors from 100% of our sample binaries, with an over-all reliability metric of 97.19%.

3.1 Initial Reverse Engineering and Manual Analysis

In order to develop our algorithms, we started with a baseline of an already analyzed ECU binary in order to get an idea of how the functions within it looked and were laid out. The initial set of binary analysis for our research was first accomplished by members of the open-source tuning community. Recall that tuning is the process through which one alters the ECU software or fuel map to optimize the performance of their vehicle[10]. The owner of the ECU source repository, with help from other members of the tuning community, mapped the binaries for the 1993 model of the Subaru EG33 binary and stored their result on the public repository that we gathered our samples from [11]. We utilized these initial results as a baseline going into our later manual analysis, and were able to find similar sensor functions within the other binaries within that repository. This EG33 binary is also our control function and the basis for our comparison data going forward.

3.1.1 Architecture and Binary Selection

Our decision to go with the 1993 EG33 binary as a control for our algorithm was one of convenience. We also needed them to already be extracted from their host vehicle, and available on a public repository, due to our lack of viable research vehicles or tools through the university. Upon learning of the automobile tuning community discussed in Subsection
2.3.1, I was introduced to their public repository of user-submitted binaries that had been run through MAME, disassembled, and uploaded [9]. Due to how completely a few of these binaries were documented by the community through Skuse’s website, we opted to use the M37791 CPU architecture and the ECUs that are designed with this architecture as a baseline for our research. This architecture appears in the EJ18, EJ20, EJ22, EJ20 Turbo, and EG33 model engines that were used in Subaru vehicles from the late 1980s to early 1990s. These engines were used in the Subaru L Series, Legacy, Impreza, and SVX vehicles.

These files were manually extracted from the ECUs of their vehicles, and uploaded to the website to provide a utility for understanding fuel maps for the Tuning community. The choice to use these binaries was due to initial work on reverse engineering the binaries from community members combined with the over-all simplicity of the binaries due to their age. Older ECU binaries, like those from this time period, are designed based on sequential programming to read sensors and process instructions. This is different in numerous ways from contemporary ECUs, however, as today’s vehicles are generally either Real-Time Operating Systems (RTOS) or Finite State Machines (FSM) that handle the sensor reading[12][13][14]. This is talked about at length in Section 2.1.

All of our other binaries came from the same public repository of user-donated M7700 ECU BIN files from 1988 to 1996[1]. These files came from vehicles located in the USDM, the Middle East Domestic Market, the Japansese Domestic Market, and the UK’s domestic market. It is our belief that the difference in year and country of origin causes enough variance in the files to make our theory of searching for functions viable.

3.1.2 Manual Analysis

Skuse’s website has the disassembly dumps of the binaries, but these are missing a lot of critical information we need to understand function control flow and how the code operates. The functions within the code are cut up into very small sub-functions that further
obfuscates where the actual functions begin and end. They included lists of jumps and references to some of the sensors, but only the 1993 binary was labeled to any degree beyond that. An example of the disassembled code from the 1993 binary can be found in figure Figure 3.1 below. Note the conditional branch conditions BEQ and BCC in the first two blocks - understanding where they would go in the disassembly dumps requires manually seeking to that address and is quite cumbersome.

Figure 3.1: Example snippet of raw disassembly text from Skuse’s website

Searching for sensor candidates within our binaries was slow at first, due to an initial lack of reverse-engineering and disassembly tools to work with. We were able to locate
a set of about 26 sample control functions by referring to the list of sensors in Figure 2.4 that wrote to that memory directly. Once we had the control function data written, we then used the code in those functions to try and find similar candidate functions in the other binaries. These were achieved by utilizing regex (REGular EXpression) pattern searches of the provided binaries from our repository, along with some intelligent "best guess" searches to approximate where these functions likely were.

In order to find the sensors and sensor functions in the non-control binaries, we sought to identify functions or code blocks in each vehicle similar to that which utilized the sensors in Figure 2.4. Ultimately, our goal with this search was to just find a series of "black box" functions that were of similar in structure across each binary and seemed to write to similar memory addresses. The idea behind these "black box" functions is that we know the function calling them has some similar usage across each binary, and we know they are writing to sensors in memory, but we do not necessarily know what they are writing or why they are accessing the values in that manner.

Figure 3.2: Example Black Box Func

Figure 3.2 is a rough approximation of how these functions work. Our known functions were usually part of either the reset vector or one of a few other functions that consisted of "mass calls". Each of our M7700 binaries had a reset vector that ran upon initial-
izing the ECU. The program counter loaded the memory address hard-coded at memory address 0xFFFE, starting execution at the byte located there[15]. Mass callers were generally part of the main "function loop" that sequential embedded systems run on repeat during their normal operation. Generally the sensor functions were located within a few of the caller functions, responsible for maintaining the health of the system.

Sensor functions located using this methodology confirmed that the structure of two functions, for example, two functions that read battery voltage across different binaries, initially appeared similar. You can see an example of this from the disassembled code dumps in Figure 3.3. We could also infer that they were the correct functions given that the loads, stores, and writes to memory addresses in those functions were corresponding to those used in our control. Our assumption here was also that the MMIO addresses referred to in these functions point to the same sensors. Going off of these assumptions we set out and defined each of the functions we were able to find here as our manual analysis data.

![Figure 3.3: Control function (left) compared to a 1993 EJ22 function (right)](image)

However, in order for us to properly generate an understanding of the control flow of each of these binaries, and additionally each of the functions within these binaries, we would need a better tool to allow us to map branches, jumps, and other control flow logic automatically within our binaries. The manual analysis method was a tedious and slow process, and it was often very difficult to find matching functions in binaries that had very subtle differences. We also could not find candidate matches manually from some functions due to them being in unexpected locations or having a completely different overall structure. Automating this process, and incorporating more advanced reverse-engineering
techniques into it, was our next step. This would then be used to flush out additional data within our binaries, and would allow us to automate the process of searching for similar functions that we had done manually before.

### 3.2 Radare2 Plugin Design

In order to automate the analysis process from the provided control data shown in Section 3.1, we needed to have a disassembler that could handle the M37791 ECU architecture. IDA Pro has an option that would allow disassembly and analysis for the M37791 architecture [16], but we lacked a license for IDA at the time and also needed to be able to incorporate our disassembler into our future analysis scripts for automated usage. To this end, there also existed a few other disassemblers for the M37791, however, many of these were even more feature-sparse than IDA. One notable open-source disassembler was MAME’s, which was written to assist in the emulation of old arcade cabinets[9]. MAME would only let us dump disassemblies of the code, rather than parse it in real-time and graph functions, however. Scripting would also be extremely difficult, and would require porting MAME to an external library.

Due to the limitations provided by IDA, MAME, and other existing tools, we decided that we needed to go with an open-source reverse-engineering tool that would allow us to:

- parse and disassemble binaries in real time or through scripting
- work with external software or plugins in order to include our new architecture
- any custom plugins for this disassembler would be easy to write and implement

This tool would also have to have a somewhat well-documented API so we could just write our own disassembly plugin on it if the M37791 was not provided to us. This would also allow us to utilize MAME’s open-source analysis framework for this architecture.
Radare2 is an open-source disassembler/static analysis tool that allowed us to easily fit each of our main requirements for the disassembler and also had instructions on how to create our own plugins for their platform. Knowing this, we opted to port the MAME M7700 disassembler over to Radare2 using Radare2’s user-created plugin capabilities[17][18]. Radare2 language plugins could either be written in C or Python, with full instructions for installing and importing the libraries found through Radare2’s own documentation [19]. Radare2 language plugin writing mostly revolves around populating the Radare2 data structures with relevant information on instruction opcode, arguments, and functionality in the assembler plugin and analysis plugin provided.

MAME’s M7700 plugin was already written in C, which made porting it over to Radare simple. The gist of the porting work involved re-writing the hooks to MAME internal libraries into several of Radare2’s built-in libraries. This instructed R2 to write disassembly information to the internal data structure that it uses to interpret instructions in real-time. R2 specifically required our plugin be split into at minimum two parts, one for handling the disassembly of instructions and the other for handling the analysis, semantic meaning, and control flow of instructions.

Radare2 plugins can be scripted to interface with Python through the R2pipe utility, which allowed for us to create Radare2 sessions within our script to analyze binaries as we go.[20] The analysis plugin allowed us to define how the program call and jump functions work, so that R2 can easily map a call to its destination address or follow the multiple branching paths of a conditional jump. Utilizing R2’s built-in control flow graph mapping functions, we can then generate a graphic representation of the control flow using either the .dot format, or through exporting each function node to a JSON representation of that graph.

3.3 CFG manual analysis with Radare2
Within R2, BIN functions are defined when a call instruction is encountered that points to some "valid" executable instructions. The executable address range is defined by the user using the R2 command

\[ e\text{ anal.limits} = true \] (3.1)

to limit the address space and the

\[ e\text{ anal.from} = \text{START_ADDR} \] (3.2)

and

\[ e\text{ anal.to} = \text{END_ADDR} \] (3.3)

commands to define those boundaries. The lower limit of the function boundaries are defined when either another function is found (a call to the middle of a function, for example), or when a valid return instruction, determined by the disassembler, is encountered[21].

Once the functions are defined, they are then broken into basic blocks by analyzing the control flow logic present in each function. These basic blocks are lists of instructions, which are terminated when a branch/jump is encountered. These branch/jump conditions are either an unconditional branch that redirects control flow to a different block or a conditional jump that jumps to one of two addresses based off of a set of prerequisite instructions being met. The control flow logic is also determined by these branch instructions, either unconditional branches (JMP or BRA) or conditional branches (BCC, BEQ, BNE, BBS, etc.)[15]. Unconditional branches are shown within R2’s visual mode/graphic mode as straight blue lines between basic blocks, while conditional branches have the green line for their branch condition and the red line representing their fail condition. When viewing the function as text, all branch destinations are instead represented as ascii blue arrows.
Figure 3.4 contains an example function as it is represented within R2. The blue arrows represent control flow changes encountered by the branch if carry set (BCC) condition on line 0xabd7. Note that this function boundary terminates at the "RTS" condition at the end of the graph.

Function data can also be dumped as a .dot file for viewing, or dumped as raw JSON which preserves the control flow metadata for each basic block within that function. An example function here is shown in Figure 3.5.

The completion of our disassembler plugin allowed us to use the function defining routine within R2’s analysis engine to form more complete understandings of the inter-workings of functions.

### 3.3.1 Sensor Function Similarity Across Binaries

Using the manual analysis data on our Control Binary from Alcyone’s website alongside the other data we had compiled, we were able to manually locate and compile a list of
functions that used each of the sensors provided. Extrapolating upon this initial list of functions from our control binary, we were then able to search for the corresponding functions using two methods, in addition to the method discussed in Subsection 3.1.2:

- Locating large segments of matching opcodes - "Similar functions"

- Searching for a candidate in the same address space as the other function

Candidate function matches were then verified by overlaying the two graphs and making sure that the

- Function control flow was "similar" by analyzing the CFGs
- Instruction OPcodes and their order was the same/similar between the candidate function and control function

The likely sensor address was then extrapolated from the instruction in the control function that accessed that sensor - we found in each of our functions that the sensor addresses corresponded to the same instruction within similar basic blocks of each function.

### 3.3.2 Bottlenecks

One interesting phenomenon we noticed when comparing functions within two binaries was the presence of functions that appear partially or entirely different, but have similar sub-graphs located around a critical node. An example of these critical nodes from our control binary are highlighted in figure Figure 3.6. This function is one of our voltage sensor functions found through our review of the decompiled source code from Alcyone’s website, but represented in Radare2’s graphics engine. As you can see, the basic blocks for this function all "bottleneck" into the highlighted block provided in the figure. This is likely due to test conditions, sensor checks, and other control flow logic present before the bottleneck all "completing" before starting the next stage of execution within that function.

The reason this feature is of interest to our graphing ideals is due to the fact that this behavior seems to remain consistent in corresponding sensor functions, that is, the likely sensor candidate functions in our test binaries. Figure Figure 3.10 below shows the voltage sensor in a middle eastern EG33 binary, the same engine family as our control but a different model year and country of origin. The obvious difference between these graphs is that the routine on the side consists of one large basic block in the Control example, while the instructions in the Comparison example have been split up with conditional jump logic instead.

The extra control flow logic shown in the comparison function results in more incoming edges to the highlighted bottleneck, however, the rest of the incoming edges to
Figure 3.6: Control Function with Bottleneck
the bottleneck remain identical between the Control and Comparison functions. This phenomenon likely has to do with reused code as binaries are updated or slightly changed for different domestic markets. As a result, the over-all structure of the functions appears different but the underlying structure and potential instructions are quite similar - this results in us needing to take special consideration for figuring out how to properly detect these sub-graphs in our automated analysis.

### 3.4 Automated Feature Vector Generation

Using the observations and features noticed in the previous section, our next step was to generate an algorithm to automate the process of finding function candidates, verifying the most likely match, and then parsing those function candidates for their sensors. Our Radare2 plugin brought with it the use of R2pipe to automate and script R2 commands to
automate the CFG generation process in Section 3.3. We would need to use this to create an object model to represent each function within memory, and this model would also require us to access nodes and their parents, and to dump features.

### 3.4.1 Automatically Parsing Binary Functions

R2pipe facilitates our automation of R2 commands to easily, quickly, and consistently load in our binaries, parse them according to our algorithms, and dump them in a manner that is easy to parse. Following our procedure in Subsection 3.1.2, the automated script needs to first seek to a consistent memory location to run R2’s complete recursive analysis on our binary. Our choice for this was the reset vector for each binary, as this function was fairly similar for each binary and would be the first code executed.

The reset vector, and the starting and ending address of the executable code are used to define the boundaries of our code analysis with R2pipe. Then, R2’s automatic analysis engine runs on each binary, generating the function limit parameters mentioned in Section 3.3. Once the function data is compiled, R2 is used to generate a list of all functions that is recursively parsed to generate any other matches. This list of functions is then used to populate information on all function candidates.

### 3.4.2 Data Structure CFG Representation

The analysis plugin of Radare2 outlined in Section 3.2 allows for R2 to construct preliminary control flow information based off of the parameters that are generated from the disassembled opcodes. It also allows for us to dump these graphs as a JavaScript Object Notation (JSON) representation that contains the parameters and opcodes of the disassembled instructions, and also any relevant information for following branches. Using our function data from the previous section, we are able to dump a JSON representation of a function, its boundaries, its location, and the opcodes making it up using R2pipe. Out-
putting our functions as JSON representations allows for us to easily and quickly load a data representation of them into our Python Script.

Using these JSON dumps, we were able to represent complex functions from our M7700 binaries in memory as a linked tree. The tree defined four different object types that allowed for us to precisely pull whatever information we needed from an exact function. Each node in the linked tree represents a basic block within each function, as stated in Figure 3.8.

At the lowest level of these blocks, we have the Instruction class - a representation of an instruction with the opcode and parameter accessible as member variables. Groupings of instructions are loaded into the Block class, which represents a basic block within radare. Basic block node contain pointers to Jump and Fail, their own respective block instructions that specify the next block to visit following the tailing branch condition at the end of a block.

The root node block, along with function address information and mutators that allow for easy recursive traversals of a chain of blocks are stored within the CFG class. The highest-level class is the Function class, which contains the entire CFG data structure and also contains information as to who calls this function and any calls that this function makes to others.
3.4.3 Jaccard Index Function Matching

We sought out to gauge the similarity of two feature vectors by comparing them with the Jaccard Similarity equation. As stated in the book Data Mining: Algorithms, Geometry, and Probability, the Jaccard measurement can be used to measure the distance or similarity between two sets[22]. Our feature vectors end up as sets of unordered instructions, based on their opcodes.

This function is defined as the length of the intersection of two sets divided by the length of the union of those two sets, or

\[ J_{AB} = \frac{|A \cap B|}{|A \cup B|} \]  

(3.4)

where J is the Jaccard index, A and B are the two sets being compared, this can be further simplified as

\[ J_{AB} = \frac{|I_{AB}|}{|U_{AB}|} \]  

(3.5)

with I and U representing the intersection and union, respectively.

This algorithm allows us to limit the impact of the more common but smaller one and two instruction basic blocks that show up in large number in each of our functions, allowing the less common instruction blocks that are more indicative of a unique match to have more weight over-all. It is also an easy to implement function that gives us a reliable metric for the over-all similarity of two sets of features that will likely be different in length and composition.

One problem of note is that a set-wise comparison of two feature vectors does leave out how often a specific feature occurs. For example, the specific number of and order of branches may be a valuable feature for comparing two binaries in the future. Jaccard comparisons merely detect the presence of the grams, but not necessarily the order of the
basic blocks within the entire feature set.

3.4.4 Algorithm Theory and Design

The feature vector generation algorithm was ultimately decided to be split into two parts - a "function detection" algorithm to find the best match function candidates for each of the provided control functions, and a "sensor match" algorithm to locate the likely address of the sensor candidate within each function candidate provided. The design was generated using a combination of features noticed in Section 3.3 and was improved after implementation when we noticed lower than ideal match percentages being generated with our design.

Block-Level Comparison Algorithm

Our initial feature vector generation algorithm was based off of the simplest to design and implement feature we noticed from the comparison data set found above. In Subsection 3.3.1, we noticed that even if the over-all CFG structure of two functions was different, the instructions that made up each basic block were overwhelmingly similar between corresponding basic blocks. Even in the dissimilar bottleneck comparison Figures 3.6 and 3.10, we noticed that matching basic blocks were still identical.

Based on this observation, we created a feature vector where each gram contained the opcodes for all instructions found within a basic block. They would be appended in-order to create a string representation for those blocks. Each index in the feature vector represented one basic block, so the resulting comparison would just compare the over-all similarity of every basic block within two functions.

This design produced very inconsistent results, with a fairly high false-positive rate as smaller basic blocks within functions were given significantly more weight. It also had a high false-negative rate, as if a block differed by a single instruction then it would not match
with our comparison algorithm. This would result in mismatches of similar functions if a basic block differed in only a few instructions, for example the basic blocks outlined in Figure 3.9 show only a few extra instructions differently in the most important basic block - but would result in only a 50% match using our Jaccard comparison formula.

N-gram(digram) Block Decomposition

Learning from the inconsistencies in block-level comparisons, we changed how feature grams are generated and instead decided to decompose each feature vector into N-Gram basic blocks. We decided on N-Gram basic blocks after their use in other classifiers to decompose/recognize strings/words[23, 22]. OPCodes can be similarly represented as words like in the N-Gram decomposition, allowing us to break up instruction listings within basic blocks for better comparisons. As their name suggests, N-Gram basic blocks are split into N-sized smaller pieces[23]. For example, if I wanted to create a 2-gram or digram decomposition of the string "ABCDEFG", it would appear as follows:

\[ABCDEFG_{2-gram} = AB, BC, CD, DE, EF, FG\]  \hspace{1cm} (3.6)

For our algorithm design, each gram represents N adjacent opcodes. In testing, we found that digram feature vectors created the most accurate results across each basic block, as they would allow us to continue to match the smaller blocks while also accurately representing features for the larger blocks in our set. An example digram feature vector appears...
as follows:

```
"BxjhSe": [u"#RS", u"#RR5#RS", u"#RSRS5", u"#RR", u"#RRCCR", u"LOACMP", u"CM#RRCCR", u"#BBCC", u"#BBCC", u"#BBCC", u"#BBCC", u"#BBCC", u"#BBCC", u"#BBCC", u"#BBCC"]
```

Figure 3.10: Example Digram Feature Vector

When comparing features generated using this method to those in Subsection 3.4.4’s Figure 3.9, you would see an increase of the Jaccard match to roughly 70%. This only improves as the number of basic blocks or instructions decrease, and helps to fully minimize the over-all impact of one or two mismatched instructions in each CFG.

**Edge Detection**

Using the digram decomposition produced more positive results, but did not reduce false-positive values enough to accurately match our discovered control functions across each of the binaries that we had manual analysis data for. As a result, we sought to more uniquely identify the structure present in each function through adding in some element of edge detection. Our edge detection algorithm added in several more two-gram features to represent the instructions immediately before and immediately after an edge.

**Control Flow Instruction Filtering**

One problem present in the edge detection grams added into the feature algorithm was that certain instructions like BRA and JMP are semantically identical to two adjacent instructions, but produce different grams. By ignoring instructions like this, we can further generate more generic two-gram feature lists to improve upon this generation. This required us to implement a filter list that would specifically leave out certain instructions. Functionally, our filter list can eliminate any instruction from appearing in features. Ultimately, this method of filtering proved to be inconclusive and was not used in the final dataset generation, however.
Bottleneck Features

Incorporating bottleneck features was caused due to a few sensor functions still not being included in our final set of function candidates after all the other features were incorporated. Each of these sub-functions contained similar bottlenecks but several had very different layouts that resulted in our basic-block and digram comparisons not producing a high enough jaccard index match rating for our liking. As a result, we decided to add in an additional feature to detect the presence of these bottlenecks in order to more accurately match these outliers.

Bottlenecks required a more specialized feature to be created to accurately represent the bottlenecks within our feature vectors. Bottleneck features are instead treated as dictionary sub-entries within the feature vector - essentially, the entire bottleneck is treated as one "feature" within our vector and those are compared independently. This is because the presence of the entire bottleneck is a feature - the entries of the sub-graph are all important in determining if that bottleneck is present. An example of a bottleneck feature within a complete feature vector is shown in Figure 3.11

![Figure 3.11: Example Bottleneck Feature Vector](image)

This new dictionary feature required us to also change our Jaccard comparison algorithm to facilitate this comparison. In order to incorporate the bottleneck feature detection into our feature generation algorithm, we needed to change how functions are compared within our jaccard algorithm and also needed to recursively detect any potential bottleneck in our function control flow graph. When our comparison algorithm encounters a dictio-
nary object, as opposed to a normal string object within our sets, it attempts to compare
to all dictionary objects found within the comparison feature vector (if they exist). If this
bottleneck is found, then the value for the Jaccard index comparison is generated, and then
this jaccard value is averaged with the jaccard value produced for the entire set.

Incorporating this feature allowed us to get the last few missing functions for our
sensor algorithm, but it also skewed the over-all match percentages down a considerable
amount. These changes will be discussed more in the conclusion section of this report.

3.5 Function Parsing and Sensor Location

Once the feature vectors for all functions in each binary were generated, we wrote
two different scripts to properly identify the highest match function candidates within each
binary. The first, our jsonparser script, generates a list of all function jaccard index matches
for each function located within a control binary. The highest index matches are highlighted
and saved to an output file for future use.

3.5.1 JSON Function Parser

The JSON Function Parser script takes the output feature vector JSON file as an input
for processing. As stated in the previous section, each JSON entry corresponds to every
feature vector for each identified function within that binary. Control binaries are defined in
the settings JSON file within the base directory of our Function Parser. Upon loading in the
Feature Vector JSON, each control binary is loaded into memory and the sensor functions
identified within our settings file are pulled from that binary.

For each binary found within the Feature Vector JSON, a spreadsheet is created rep-
resenting how each function within that binary compares to the chosen control functions.
Each control function represents one row in the spreadsheet, with each column representing
the functions from the comparison binary. The Jaccard Index formula is used to compare
the features in each function at that index, with the result being written into the spreadsheet.

An example of one of these spreadsheets is shown in Figure 3.12.

| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |

Figure 3.12: Example Func Match Spreadsheet

Manual analysis for each binary is also loaded in from the code-blocks.xlsx spreadsheet. If manual analysis data exists for the spreadsheet, then the expected function will be labeled in the spreadsheet for that binary. The highest index for each row, that is, the highest percentage match candidate for each row, will be marked with a blue box. If manual analysis data exists, then that function will be marked with a red box. If the manual analysis data matches the function picked by the Jaccard formula, then the box will instead be marked in purple.

Once the spreadsheet is created, a JSON list of the address for each function candidate match for each binary comparison will also be generated. This list is a quick and easy way to instantly parse the match candidates for each function into our sensor parsing algorithm, shown in Subsection 3.5.2.

**Engine-Level Binary Separation**

Our initial draft of the JSON Function Parser attempted to compare each of our engines to the EG33 binary’s completed manual analysis. The match percentages between the functions of the EG33 and the other engine types did not significantly improve as we added more features, so as a result, we changed our comparison algorithm to instead identify a control function for each engine, match each binary to that engine type, and then separate each binary into "categories" based on the most likely match for each engine.
However, we were able to generate additional manual analysis data for several of our binaries and managed to get similarly complete manual analysis data for each of our other main engine types: EJ18, EJ20, EJ20T, and EJ22. This was achieved by comparing all of our binaries to each control within our JSON Function Parser, averaging up the Jaccard match percentages for all function matches in each binary, and then clustering each binary into engine bins based off the control function they had the "highest matches" for.

3.5.2 Python Sensor Parser

Our Python Sensor Parser utilizes the clustered binary lists that were generated in Subsubsection 3.5.1 to determine a "control" function that is used as a "guide" for each sensor location. This control function is the control function defined from the JSON setting found in Subsection 3.5.1 for that engine type. Each function type is loaded into memory using a similar data structure to the one used for the feature vector generation in Subsection 3.4.2, but only the provided functions from the candidate list are loaded into memory.

Sensor Feature Generation

The control functions and the candidate functions go through two similar methods of generating their feature vectors. For the control function, the sensor address is used as an "anchor" for the generation of the feature vector. This anchor automatically parses the function until it finds any instructions that utilize the provided address, and then creates a two-part feature vector centered around that specific address.

Our candidate function matches use a more exhaustive search, creating the same feature vector for any potential operands that fall under the criteria for a "sensor feature". Using our manual analysis data and some information about the binary, we were able to determine that this criteria should be any valid memory address within the RAM/MMIO/IO range of our binary, located in the lower 0x4000 bytes of memory.
Once each "anchor" is defined, the two-part feature vectors are created by

- recursively going back two basic blocks from the block that contains that instruction, creating 2-gram listings of all opcodes, and then creating 2-gram listings of all edges
- iteratively going forward two basic blocks from the block that contains that instruction, creating 2-gram listings of all opcodes, and then creating 2-gram listings of all edges

Any instruction that shows up in more than one basic block within a function is given an additional two feature vectors for each time it shows up. A rough example of how these basic blocks should look is found in Figure 3.13.

![Figure 3.13: Example Sensor Feature Vector Comparison](image)

**Sensor Matching and Extraction**

Using both feature vectors generated in Subsubsection 3.5.2, we compare the first feature vectors and the second feature vectors independently. The jaccard distance generated is then averaged to give an over-all match for two sets of features for each sensor candidate - with the highest match being labeled as the "match" for that candidate functions. Each match represents one (control, non-control) test pair, and works on the basis that the control and non-control functions are corresponding in their respective binaries.
The candidate matches are extracted by looking at the most frequent sensor match candidates for each function. For example, if sensor 0x102f shows up 8 times across 9 potential functions, it would be the "candidate" and would have an 88.88% match. These matches then dumped to a JSON file split off by the binary control engine that they are compared to.
Chapter 4

Experimental Results and Evaluation

Our data was evaluated by generating the complete bottleneck/digram feature vector for each of our provided binaries using our experimental algorithm. These were then evaluated with our Jaccard formula and double-checked with manual analysis to verify that our formula worked. Extracted sensor addresses were also compared to our manual analysis for validation. The full experiment is split into Function and Sensor detection, based upon how we break up the binaries in our feature vector generation algorithms.

4.1 Methodology

We followed the proposed algorithm design outlined in our previous sections and implemented those in three Python scripts that utilize our disassembler to generate our Features. Our experiments and automated scripts require the use of Radare2, R2Pipe, our specially written Disassembler, and the script source code located within my own github repository. Source code for these scripts are provided here:[24][25][26][27]. No additional hardware was required. Each script handles a different stage of our data processing, specifically
• The first script handles generation of all Feature Vectors for all functions within a provided binary

• The second script handles comparison of all Feature Vectors provided with the designated control to find a list of functions

• The third script takes the identified functions, parses sensor candidate addresses, and generates features according to those sensors. It then extracts the sensors that match the designated control

The first and second scripts are specifically used in Function detection, finding of all function match candidates for sensors; the third script is used to parse each corresponding function match to find the potential sensors in the Sensor detection portion of this evaluation.

Our list of sensors to test our searching algorithm was limited to the initial ones found in the pre-existing manual analysis of our designated Control binary, the 1993 USDM EG33. These are specifically outlined in Figure 4.6.

<table>
<thead>
<tr>
<th>Address</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>8C3D-8C3F</td>
<td>ROM ID NUMBER &quot;YEAR&quot;</td>
</tr>
<tr>
<td>102F</td>
<td>BATTERY VOLTAGE</td>
</tr>
<tr>
<td>1071</td>
<td>VEHICLE SPEED (mph)</td>
</tr>
<tr>
<td>1071</td>
<td>VEHICLE SPEED (km/h)</td>
</tr>
<tr>
<td>106F</td>
<td>ENGINE SPEED</td>
</tr>
<tr>
<td>1185</td>
<td>WATER TEMP (F)</td>
</tr>
<tr>
<td>1185</td>
<td>WATER TEMP (C)</td>
</tr>
<tr>
<td>10A2</td>
<td>IGNITION TIMING</td>
</tr>
<tr>
<td>1283</td>
<td>AIRFLOW SENSOR</td>
</tr>
<tr>
<td>1282</td>
<td>LOAD DATA</td>
</tr>
<tr>
<td>128C</td>
<td>THROTTLE POSITION SENSOR</td>
</tr>
</tbody>
</table>

Figure 4.1: List of sensors our search was limited to
Note that the ROM ID NUMBER "YEAR" section above is not a sensor, but is just a label for the ROM. Our manual analysis also revealed that each of the sensor addresses outlined in Figure 4.6 are consistent across all EG33 binaries, which allows for easier manual verification of the algorithm.

Verified through manual analysis, we located a semi-complete list of the functions in which these sensor addresses are utilized. These functions are outlined in Figure 4.5 and are used as the "control functions" for our 1993 EG33 Binary. Note that some sensors are used in the same functions as other sensors, resulting in some redundancy in the matching algorithm for these functions.

```
0x9a56 (Battery Voltage)
0x9f5b (Battery Voltage)
0xa166 (Battery Voltage)
0xa307 (Battery Voltage)
0xae2c (Battery Voltage)
0xd982 (Battery Voltage)
0xe1cd (Battery Voltage)
0xe1cd (Throttle Position Sensor)
0x9be8 (Vehicle Speed)
0x9dce (Vehicle Speed)
0xa59d (Vehicle Speed)
0xa59d (Engine Speed)
0xa9a7 (Vehicle Speed)
0xa9a7 (Engine Speed)
0xafc5 (Vehicle Speed)
0xafc5 (Engine Speed)
0xb5fc (Vehicle Speed)
0xb5fc (Engine Speed)
0xb25a (Knock Correction)
0xa5ec (Engine Speed)
0xb5bf (Engine Speed)
0xc356 (Engine Speed)
0xddcd (Airflow Sensor)
0xda0f (Ignition Timing)
0x9b4e (Water Temp.)
0xab56 (Water Temp.)
```

Figure 4.2: List of functions that access some control sensor, along with the sensor type
Our experimental setup uses our Python Scripts combined with the R2Pipe plugin that was written and developed by the R2 team to automatically parse all binaries in our test directory. The development and layout of these scripts is referenced in Chapter 3. The algorithm to create our feature vectors remains constant for both the control and the comparison binaries, up to including the control binary into our set of comparison binaries to accurately gauge the effectiveness of the comparison algorithm. This ensures consistency of the features and guarantees that our feature vectors are consistent for corresponding binary functions.

Figure 4.3: Basic idea behind sensor finding algorithm

Feature vectors are stored as string representations in JSON files and passed between our scripts to keep track of the function matches, which engine bins the binaries belong to, and the match percentages of any found sensors at the end stage of the program. JSON files were used for ease of reading by humans and our scripts. The final results of our sensor detection algorithm will also be located within the output JSON file of the sensor classification script.

4.2 Function Detection Results
The first stage of our scripts need to use unique feature vectors to accurately and reliably find the corresponding sensor functions in each binary. The over-all structure of the EG33 code is fairly similar; however, there are differences in model year and country of origin that result in differences in function size as extra instructions are added in to compensate for additional features. Verifying the similarity in structure of our EG33 code across-binaries was first done with manual analysis, and then was done with our function matching algorithm to find candidate functions to match with the control function.

![Algorithm Match Comparison](image)

**Figure 4.4: Algorithm Match Comparison, block-level comparisons vs digram comparisons**

Adding in the bottleneck and digram decomposition feature vectors proved to be much more effective for generating features than just utilizing the block-level comparisons. As shown in Figure 4.5, the digram/bottleneck feature generation algorithm produced much more reliable and consistent function match data for the provided EG33 binaries. Digram representations of features allows us to more easily account for one or two instruction differences between basic blocks shared by different binaries, which increases the over all match percentage of functions dramatically.

The bottleneck features are the main source of the lower end of the distribution of our function match data, but also produce higher match candidates in a few functions essential for finding sensors later on. As you can see in Figure 4.5, the bottlenecks can result in
function mismatches as low as 30% - but including them produces a higher over-all match than just using block-level comparisons. Since a few of our sensor functions can only be identified with bottleneck detection, this is a necessary sacrifice for our over-all success rate detailed in Section 4.3.

![Total Distribution of Each Sensor Matches](image)

Figure 4.5: Total distribution of function matches, digram bottleneck feature vectors

Over-all performance of the function detection algorithm with the bottleneck detection algorithm was able to correctly define the EG33 functions with a 90.4% average Jaccard similarity metric. Of the 234 possible functions total within the EG33 binaries, with 26 per binary, 223 were successfully detected with an over 60% match, lining up correctly with the manual analysis data for those binaries. This results in an over-all match of 95%. Figure 4.5 shows the complete distribution of all found functions across each of our EG33 test binaries.
4.3 Sensor Detection Results

Our next experiment used the function match data to parse and identify the sensors within those matches. We achieved this goal by creating feature vectors for each potential sensor within these functions. Potential sensors were defined as memory accesses to the lower-order range of memory. As stated in our coverage of the manual analysis of this binary in the data review, the lower 0x4000 memory locations are where these sensors were located.

By leveraging the features for our sensor detection algorithm, we were able to then use our feature vectors to extract sensor values. Our sensor values were all the same for the EG33 binaries used in this example, but this does not mean that the layout of the sensors were necessarily the same.

These sensor values were found as follows:

- Airflow Sensor: 0x1283
- Battery Voltage Sensor: 0x102f
- Engine Speed Sensor: 0x106f
- Ignition Timing Sensor: 0x10a2
- Knock Correction Sensor: 0x12a7
- Throttle Pos. Sensor: 0x128c
- Vehicle Speed Sensor: 0x1071
- Water Temp. Sensor: 0x1185

Note that these sensors correspond to those shown in Figure 4.6, and match across each of our binaries without fail. These sensors were consistently found in each of our
EG33 test binaries, no matter the model year or country of origin of the binary. This only appears for our EG33 binaries - manual analysis of our EJ18, EJ20, and EJ22 produced different sensor locations even among binaries of the same engine family.

The over-all match percentages are located in Figure 4.6. These matches do not represent the Jaccard index matches of each function, but rather the frequency of the sensors located within the found candidate function matches. Even if the match percentages are not 100%, the correct sensor is still the highest match candidate in 100% of binaries analyzed with this methodology. The over-all match rate for sensors across all binaries is 97.19%, with the United States binaries sharing the highest consistency as they are the most similar binaries.

Likely sources of inaccuracy within this matching algorithm come from potential function mismatches in the function detection algorithm - this sensor extraction algorithm was designed based upon the assumption that each Function algorithm is properly matched to a candidate algorithm. Unless our function feature generation can produce 100% matches, there will always be the potential for a function mismatch in this sensor generation.
Figure 4.6: Sensor Match Comparison, as percentage of sensors found within each function
Chapter 5

Conclusion

In this thesis, a method of automatically identifying sensor locations in an ECU’s ROM was displayed. By automatically loading in a binary in Radare2, we are able to create a data representation of the functions within a binary and their internal control flow graphs. Using these graphs in conjunction with existing manual analysis data from several control binaries, we are able to create feature vectors that can identify the candidate functions that contain and use sensors, and can also locate those sensors in the binary through this method. Our function detection method is able to properly locate 95% of corresponding sensor functions, verified through additional manual analysis. The sensor addresses within those functions were then able to be recovered with a 100% success rate with a 97.19% confidence for our binary sets. These results indicate that this method would be effective for engine binaries of the same family, provided that some manual analysis exists for at least one member of that family. Future work will require generalizing the feature vector generation method to work in a platform-independent manner, and will require additional fine-tuning and testing to ensure that this method produces the acceptable data across all target platforms.
Bibliography


