Non-Intrusive system behavior tracing using power consumption

by

Adan Flores

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2020

Author's Declaration

Abstract

This thesis presents a novel non-intrusive method of creating a map of black-box systems behavior to commands by observing the power consumption of the system. We expand our method in the form of a flexible framework which supports communication abstraction layer to send commands on different channels. The framework also integrates data processing and the opportunity of adding a validation module that framework users can utilize to detect anomalies in the system. The framework furthermore builds on the idea of anomaly detection by providing two operating modes: one for generating training data for machine learning models, and one for run-time validation of the systems behavior. From the experiments, we confirm the effectiveness of the framework for creating a map of the systems power consumption against a series of queries. Our experiments also show some of the challenges presented with this method when different commands induce similar behaviour in the system.

Acknowledgements

I would like to begin by thanking my academic advisor Dr. Sebastian Fischmeister for his guidance and for giving me the opportunity of being part of his research team.

Thanks to the ECE department and faculty, for providing me with the knowledge and assistance I required during my studies.

Finally I would like to thank "Consejo Nacional de Ciencia y Tecnología" for funding my studies and making all of this possible.

Dedication

This thesis is dedicated to my parents, Elva Ramirez and Alberto Flores, who have always supported me and never stopped believing in me; and to my family and friends, whom I left behind to pursue this opportunity.

Table of Contents

List of Tables **ix**

List of Figures **x**

1 Introduction **1**

1.1 Related Work 2

1.2 Problem Statement and Assumptions 3

1.3 Contributions 4

1.4 Organization of the Thesis 4

2 Background **5**

2.1 Power Tracing 5

2.2 Time series 6

2.3 Statistical classification 10

3 Framework **12**

3.1 Software architecture 14

3.2 Data acquisition 15

3.3 Framework running modes 17

3.3.1 Training mode 18

3.3.2 Normal mode 20

3.4 Shared memory 21
3.5 Communication Abstraction Layer 23
3.6 Data processing 25

4 Experiments **30**
4.1 I2C devices 30
4.1.1 Results 33
4.2 Custom system 34
4.2.1 Command space 36
4.2.2 Results 37
4.2.3 New command space 41
4.2.4 Classification of traces 43
4.3 EMP water pump 43
4.3.1 Command space 45
4.3.2 Results 46
4.3.3 Classification of traces 49
4.4 Sterling truck 50
4.4.1 Command space 51
4.4.2 Results 52
4.4.3 Classification of traces 55

5 Future Work and Conclusion **56**
5.1 Lessons learned 56
5.2 Future Work 57
5.3 Conclusion 58

References **60**

APPENDICES . . . **63**

A Additional plots for Section 4.2 . . . 63

A.1 Read accelerometer value . . . 64

A.2 Read temperature value from sensor 1 . . . 65

A.3 Read ambient temperature from infrared sensor . . . 66

A.4 Read ambient light value . . . 67

A.5 Read temperature value from sensor 2 . . . 68

A.6 Read temperature from MCU sensor 1 . . . 69

List of Tables

2.1 Distance comparison between Euclidean and DTW computations for time series. . . . 8

3.1 Default values for digitizer arguments. . . . 16

4.1 Command space for custom system under test. . . . 36

4.2 Map of traces to command space. . . . 40

4.3 New command space after removing commands that generate repeated traces. 41

4.4 Map of traces to new command space. . . . 43

4.5 Metrics for support vector machine classification of custom system's data. . . 43

4.6 Command space for pump ECU. . . . 45

4.7 Map of traces to new command space. . . . 49

4.8 Metrics for support vector machine classification for pump's data. . . . 50

4.9 Command space for truck. . . . 51

4.10 Metrics for support vector machine classification using truck's data. . . . 55

4.11 Average DTW distances between command *"Request engine temperature 1"* and the rest of the command space. . . . 55

List of Figures

2.1 Diagram showing where the resistor is added to the system. 5

2.2 Example of a trace from Section 4.2. 6

2.3 Euclidean distance vs DTW distance [22]. 7

2.4 Plots of time series to compare 10

2.5 Example of a hyperplane dividing a set into classes. 11

3.1 System architecture. 12

3.2 Software architecture. 14

3.3 Digitizer card ATS9462 by AlazarTech. 15

3.4 Data acquisition setup. 17

3.5 Training mode sequence diagram. 19

3.6 Normal mode sequence diagram. 20

3.7 Interaction between processes using shared memory. 21

3.8 Communication Abstraction layer. 23

3.9 Screenshot of power trace from I2C device. 25

3.10 Plot of power trace using raw data. 26

3.11 Low pass filter based on fast Fourier transform [21]. 27

3.12 Plots of power traces after applying a rolling mean. 28

3.13 Plots of power traces after applying an iterative rolling mean with a window size of 10. 29

4.1 Photo of I2C system for experiment. 31

4.2 System diagram for I2C experiment. 32
4.3 Screen captures of traces from each I2C device. 33
4.4 Photo of custom system for experiment. 34
4.5 System diagram for custom system experiment. 35
4.6 Plots of selected traces for each command sent to the system under test. Only one iteration of the rolling mean with a window 10 was computed for this step. 39
4.7 Plots of selected traces for new command space in Table 4.3. 1000 iterations of the rolling mean with a window 10 was computed for this step. 42
4.8 Photo of EMP pump system for experiment. 44
4.9 Plots of traces for each command sent to the pump ECU. 1000 iterations of the rolling mean with a window of 10 were computed for each trace 48
4.10 Photo of Sterling truck used in experiment. 50
4.11 Plots of traces for each command sent to the Sterling truck. 1000 iterations of the rolling mean with a window of 10 were computed for each trace. . . 53
4.12 J1939 log showing request of PGN FEEE. 54

1 Plots from random samples of read accelerometer value command 64
2 Plots from random samples of read temperature value from sensor 1 command 65
3 Plots from random samples of read ambient temperature from infrared sensor command 66
4 Plots from random samples of read ambient light value command 67
5 Plots from random samples of read temperature value from sensor 2 command 68
6 Plots from random samples of Read temperature from MCU sensor 1 command 69

Chapter 1

Introduction

With an estimated 93% market share for embedded hardware in 2015 [17] and with the promise of a connected world through the Internet of Things (IoT) looming in the horizon, it is not surprising that more and more attackers target embedded platforms in search of potential vulnerabilities they can exploit for their advantage. In a paper published by Vijeet M., and Ashish S. [11], the authors list some examples of frequent attacks used on modern embedded systems that aim to steal information, damage the system, or take control of the system. Those examples include:

- Tampering with the physical system, whereby the attacker aims to obtain data from the system by tapping into a communication bus.
- Spoofing of data, whereby the attacker introduces tampered data packages into the bus with the end goal of persuading the system into thinking that the source of the package is valid.
- Malware attacks, in which the attacker introduces malicious pieces of code into the system that run unnoticed by the user and the system itself, to steal data or manipulate the behaviour of the system.

Attacks like the ones previously mentioned have serious economic repercussions for individuals, companies, and governments. A report by the government of the United States of America [14] estimates that cyber attacks took a toll on the U.S. economy of between \$57 and \$109 billion dollars in 2016 alone. In 2019, businesses in the United Kingdom lost around \$243,000 per incident [10]. From the point of view of the average computer user, victims lost, on average, $142 to hackers in 2017. In the USA alone, 143 million Americans were affected by cybercrime in the same year [4].

Companies do not only see a loss of capital whenever they find themselves victims of cybercriminals. they have to invest capital to minimize the exposure to the attacks. In May 2019, Deloitte published the results of a survey [7] in which financial institutes report their spending range towards cybersecurity. On average, financial institutions spend 10% of their IT budget on cybersecurity. From the point of view of company revenue, financial institutions sometimes spend as much as 0.9% of their revenue in an aim to protect themselves from attacks.

In 2015, security experts Charlie Miller and Chris Valasek hacked a Jeep vehicle by gaining access through an infotainment ECU [6]. The attackers then updated the firmware of a microcontroller (Renesas V850) with malware that allowed them access to the Controller Area Network (CAN) bus, which they then used to gain remote control of the vehicle. In 2016, Researchers from Keen Security Lab were able to gain access to the CAN bus of a Tesla Model S vehicle from a distance of about 19 km [19]. This attack was possible by making the car connect to a malicious hotspot set up by the attackers. Once the vehicle connects to the attackers' hotspot, the attackers then exploit a vulnerability in the system to get access to a shell and subsequently send commands through the CAN bus [18].

But even attacks like the ones previously mentioned can, in theory, be detected without the need to install any additional software in the system. In a paper published by Bridges et al. [2], the authors create power profiles of a CPU for fixed tasks. The authors then use the profiles to detect any variations when malware is present in the system. We can use a similar approach to detect anomalies in systems by creating power profiles of a system when it responds to external requests.

In this thesis, we propose a non-intrusive technique to create a map of system behaviour without the need to modify the firmware of the system under test. We will implement the technique in the form of a framework that deploys on multiple systems. The framework can also obtain training data or, with more development, detect anomalies in run-time.

1.1 Related Work

Power tracing, although novel, is already proving to be a good solution for creating profiles of systems and aid in the detection of anomalies. Clark et al. researched run-time malware detection by creating profiles of the power consumption of medical devices [3]. The authors would gather traces of the system's power consumption in an idle state and then infect the system with malware. With an infected system, the authors can then create specific profiles on how the system's power consumption behaves when running with malware. Later these profiles can aid in the detection of malware and identify known malware with good accuracy. In [23], Wei et al., the authors' research micro-architectural attacks and their impact in power traces. In their work, the authors confirm that the attacks introduce noise in the system, which proves useful to detect the anomalies with the power traces. In [2], Bridges et al. detect anomalies in tasks by creating power profiles of several tasks before and after infection. One important thing to notice is that for the approaches mentioned in [2] and [3] depend on the systems running fixed processes. If the processes or systems were to change not because of malware, but because of an update to the system, the models would have to be retrained. In [13], Moreno et al. propose a framework for monitoring the power consumption of a CPU and determine which block of source code is running. This method, however, requires to reorder the instructions of the code to maximize distinguishability. The framework adds a stage in the compilation process responsible for reordering the instructions. Lamichhane et al. later expanded on the previous technique by adding support for multitasking systems by using a control flow graph [9].

1.2 Problem Statement and Assumptions

As we saw on the Section 1.1, the different works focused on creating profiles that can be used in anomaly detection. All of this shows that machine learning is being applied for side-channel analysis. With our research, we aim to add a new methodology to gather data with our framework to contribute in to machine learning side-channel analysis.

In this thesis, we want to address the following problem statement: Given a black-box system with observable inputs, perform system identification to map system inputs to side channel behaviour.

We established the following assumptions for this research:

- The command space is known beforehand: The purpose of this research is to show that it is possible to create a map of the system's inputs to side-channel behaviour. Prior knowledge of the system's messages has to be known. Finding or guessing the command space is outside of the scope of this research.
- Systems under test are embedded systems: This assumption limits the scope of the research to embedded systems. Although we can deploy the framework on general-purpose computational systems, the complexity of said systems might introduce a lot

of noise in the traces. Thus, embedded systems are more desirable for our research because of the recurring behaviour these systems have with a limited command space.

- System input channels use standard communication protocols: We assume that the input channels of the systems under test use standard protocols such as I2C, Serial, UDP, SPI, USB, CAN, J1939, etc. We do not consider cases where the system has proprietary non-disclosed protocols used in the input channels. We only consider standard protocols so we can take advantage of the already available drivers in Linux and the information available online.

1.3 Contributions

The main contributions of this research are:

- A framework that can obtain power traces of a system under test in an autonomously way by sending a series of commands to the system's input.
- We provide a modular architecture to facilitate the integration of a validator module to execute online anomaly detection.
- We give the framework the flexibility of being deployed on multiple systems by using an abstraction layer to reconfigure the framework easily.
- We designed the framework to run in two modes: The first mode will collect data that can train a model. The second mode could detect anomalies in run-time mode with the addition of the validator module.

1.4 Organization of the Thesis

The rest of the thesis is organized as follows. Chapter 2 provides the reader with some background of power tracing, time series, and machine learning. Chapter 3 presents the proposed framework and its components. Chapter 4 contains the systems on which we ran our experiments and the results we obtained for each system. Finally in Chapter 5 we discuss the lessons learned from our research, the future work, and share our conclusions.

Chapter 2

Background

2.1 Power Tracing

A power trace is a collection of data samples showing the power consumption of a system over time. We can use the samples to create a map of a system, as shown in the works presented in Section 1.1. One approach to obtain the power traces is by placing a resistor in series at the entrance of voltage to the system as shown in Figure 2.1. We can then capture the voltage drop of the resistor over time to generate the trace. Said method was used in [13]

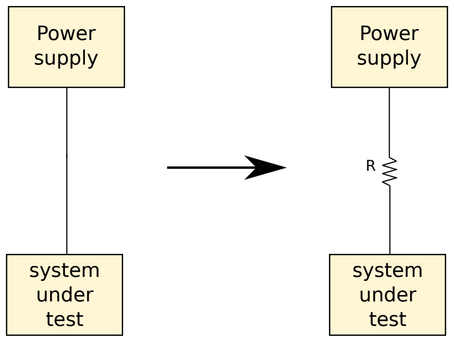


Figure 2.1: Diagram showing where the resistor is added to the system.

In [8], the author describes how to calculate the average power consumption of a processor. The author also explains how power consumption is a function of the instructions the processor is executing. When we send a request to a system via a command through an input channel, the processor will execute a set of instructions associated with that request. To the power consumed by the processor, we can add the power consumed by other active modules as part of the request sent to the system.

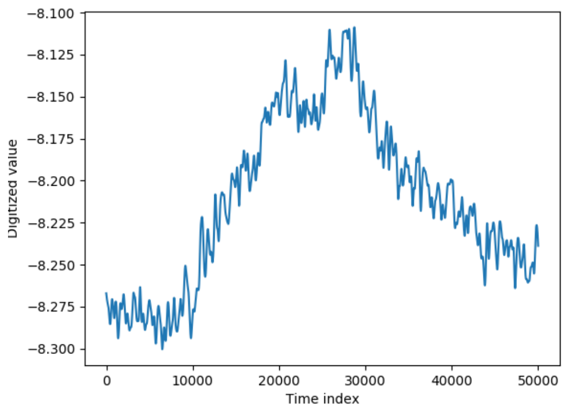


Figure 2.2: Example of a trace from Section 4.2.

2.2 Time series

Time series is a series of data points ordered in time [15]. Time series models are often used to understand the causes that produce the data or to fit a model to create a forecast. By this definition, we can see that our power traces described in Section 2.1 are time series. This allows us to use standard time series analysis methods on the power traces.

One useful advantage of time series analysis is the measurement of similarity between two time series. One method to calculate the similarity between two time series is to calculate the Euclidean distance. The Euclidean distance between the two series is given by the following equation described in [1]:

$$E_d\left(s_1, s_2\right) = \sum_{t=0}^{T} \left(s_1(t) - s_2(t)\right)^2$$

We can say that s_1 is equal to s_2 if the Euclidean distance is zero. We can quickly see that using the Euclidean distance would be a problem. In an ideal world, when capturing the time series under the same scenarios, we will always get the same data points every time. But in practice, that is not always the case. For instance, power traces contain power consumption of the system in general. Although sending a command several times might yield similar power traces, the power consumption of the CPU will vary depending on what tasks are being executed by the operating system. Noise is also another factor to consider. For example, noise can be caused by a probe or by the power supply of the system. If one of the two time series is shifted or stretched, the Euclidean distance will be different than what we expect every time. Dynamic time warping (DTW) helps in the situation previously described. DTW computes multiple paths for a data point from one time series to another and keeps the path that is the most similar to the first data point [22]. Using DTW, we can calculate the similarity of different time series and create clusters using the measurement. DTW gives us great flexibility but with the downside of over-fitting our models.

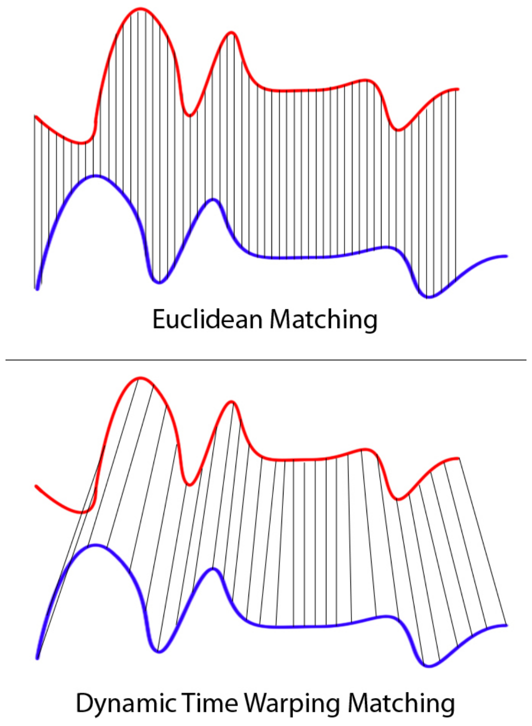


Figure 2.3: Euclidean distance vs DTW distance [22].

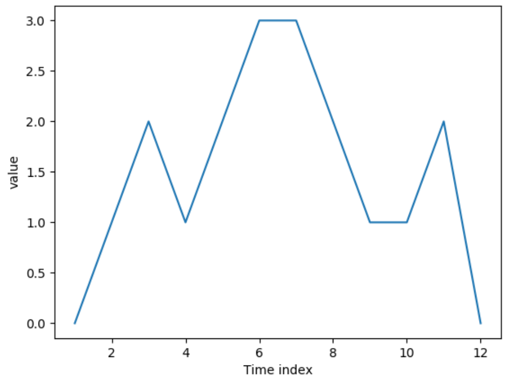
To demonstrate the differences between the Euclidean and DTW distances, we created some time series and a simple python script to compute the distances from one time series to other eight time series. Time series 2 to 5 are similar to our time series used as a reference. Time series 6 to 9 are different than our reference. Plots at the end of this chapter show a visual representation of the data for each time series.

For our python script, we loaded the data into individual NumPy arrays. Then we calculated the Euclidean distance from time series 1 to the rest of the time series by using the equation previously mentioned in this section. To calculate the DTW distance, we used the FastDTW library from author Kazuaki Tanida [20]. Table 2.1 shows the results of the operations.

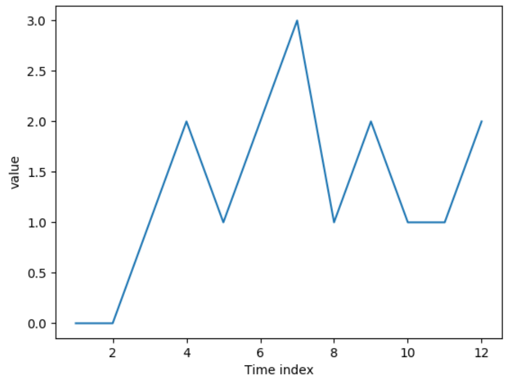
Time series	**Euclidean distance**	**DTW distance**
Series 1 to series 2	3.4641	3.0
Series 1 to series 3	4.1231	3.0
Series 1 to series 4	3.3166	4.0
Series 1 to series 5	4.5823	2.0
Series 1 to series 6	5.0990	10.0
Series 1 to series 7	6.4031	11.0
Series 1 to series 8	6.7823	14.0
Series 1 to series 9	8.0	18.0

Table 2.1: Distance comparison between Euclidean and DTW computations for time series.

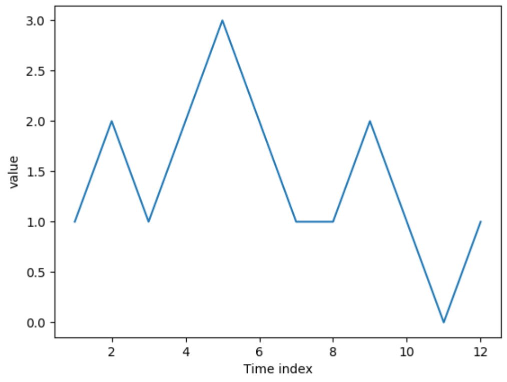
As we can see in Table 2.1, the Euclidean distance calculation yields similar results across all of the time series, while the DTW distance gives us a greater distance for those time series that are different than the reference. This might seem like a contradiction to our previous statement, where we say that Euclidean distance could not accurately compute a good distance for time series with data shifting. The reason why we get similar values throughout all Euclidean computations is because of the low data sample for our time series. In contrast, DTW can calculate greater distances between series that are not similar to our reference than those that are similar but contain data shifting and data variations. This is because DTW will try to look for the closest path for the current point from our first time series. When the time series is different, the path will increase, adding to the total distance between the two time series.



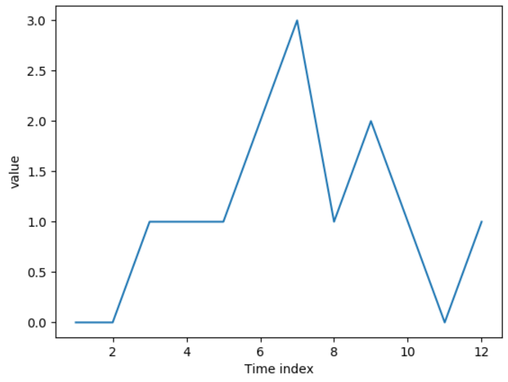
(a) “Series 1”. Data to compare.



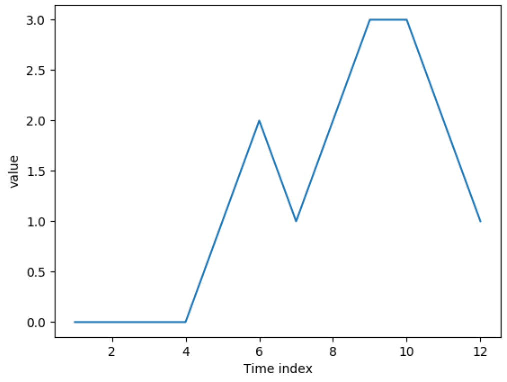
(b) “Series 2”.



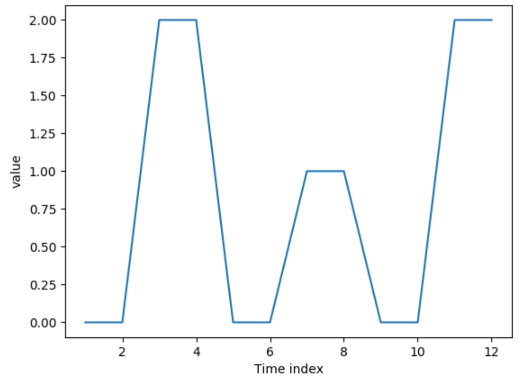
(c) “Series 3”.



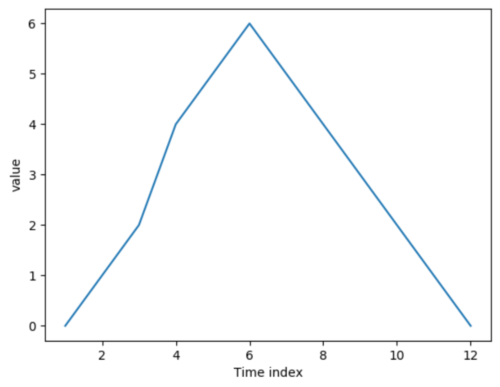
(d) “Series 4”.



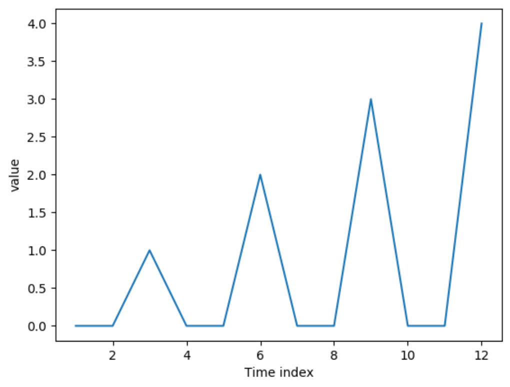
(e) “Series 5”.



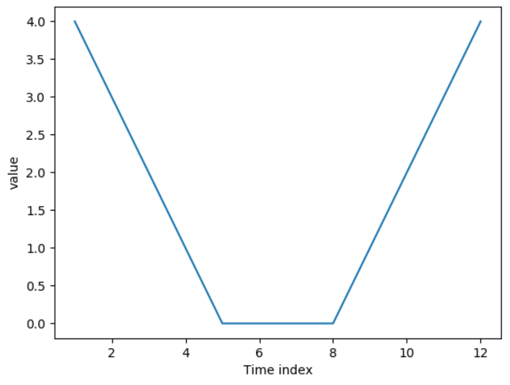
(f) “Series 6”.



(g) “Series 7”.



(h) “Series 8”.



(i) “Series 9”.

Figure 2.4: Plots of time series to compare

2.3 Statistical classification

Statistical classification is a supervised learning approach that aims to categorize new, unlabeled information based upon its relevance to known, labeled data. The algorithms that sort the data are called classifiers [5]. Statistical classification follows a set of rules established by the user ahead of time. It is essential to evaluate if the classification is yielding successful results. For this we can check for different evaluation metrics. These are some of the metrics we will be taking a look at in this research:

- **Accuracy**: Number of correct predictions divided by the total number of predictions [12].
- **Precision**: Number of correct predictions divided by the sum of correct predictions and false positives [12].
- **Recall**: Number of correct predictions divided by the sum of correct predictions and false negatives [12].

For this research we will use support vector machines for obtaining classification metrics from our data. Support vector machine is a supervised learning algorithm that is used for classification, regression, and outliers detection [16]. We decided to use this approach because of the support for high dimensional spaces and it is a supervised learning algorithm which we can take advantage of since we will know the labels beforehand. Support vector machines establish hyperplanes to separate two or more sets [24].

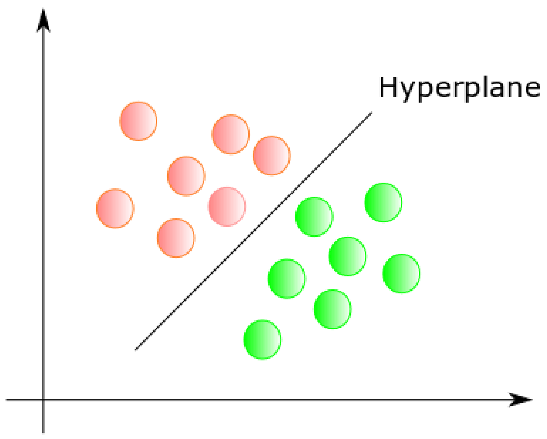


Figure 2.5: Example of a hyperplane dividing a set into classes.

Chapter 3

Framework

In this chapter, we will explain how the framework is structured and how the framework's main components work and interact with each other at a high level. We will begin by introducing the system architecture shown in Figure 3.1.

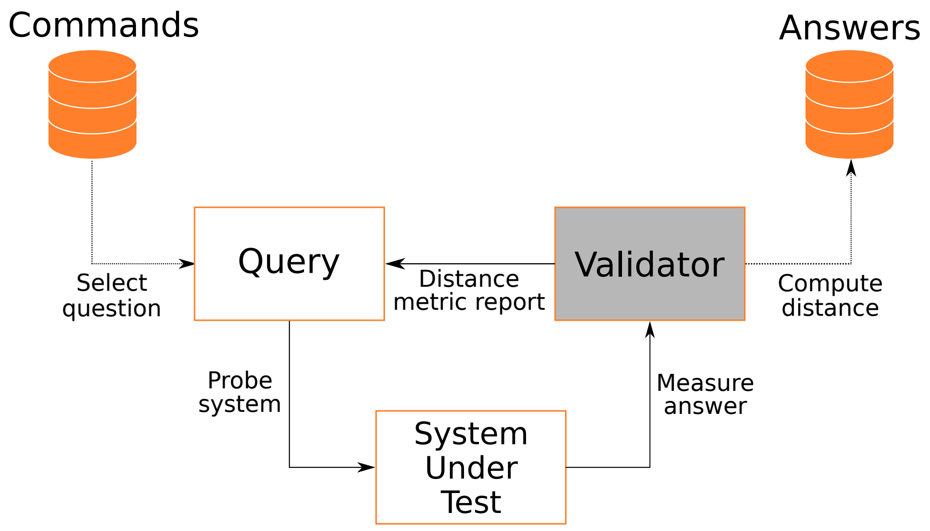


Figure 3.1: System architecture.

The system architecture gives us a general idea of how the main components interact with each other. The framework has access to a list of messages that are available to be sent to the system under test. The framework probes the system by sending a message selected from the list. The validator module then checks the behaviour of the system. The validator module has a model of the system which we generate with data obtained from previous controlled experiments. The validator can then compute a distance metric report using data from previous experiments. The framework can utilize the metric report to decide which message it should send next.

The scope of this thesis does not include the validator module, and it is mainly focused on the development of the framework. The following sections of this chapter will explain in more detail how the framework is structured and how the modules and components of the framework work.

3.1 Software architecture

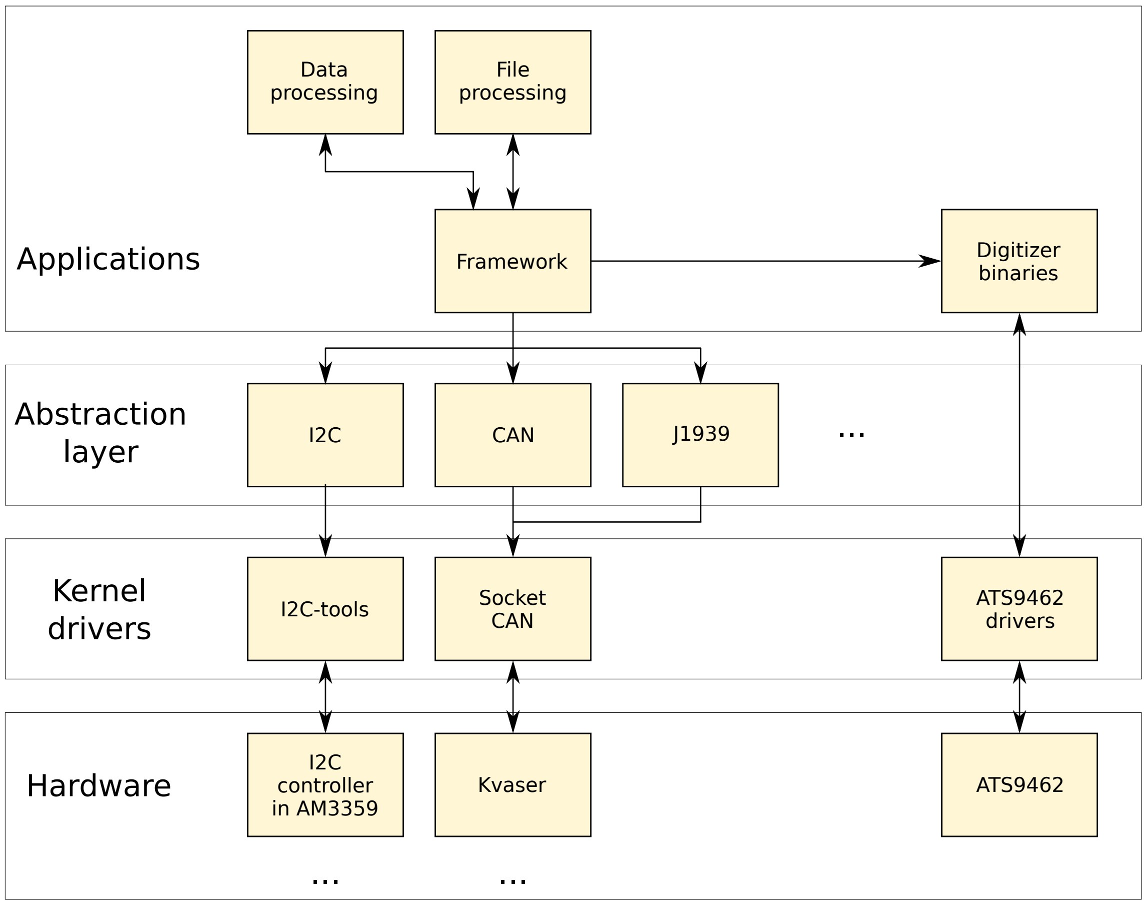


Figure 3.2: Software architecture.

The design of the framework's architecture has the objective of being able to use the framework on more than one system under test. Target systems that we want to create power profiles might have different input bus protocols; therefore, the design included an abstraction layer between the framework application and the kernel drivers. Using the abstraction layer, we can swap communication protocols while setting up the framework for an experiment without the need for significant changes in the framework's code. Thus, avoiding any unnecessary interaction by the user and minimizing the room for error while preparing for an experiment. Another advantage of using an abstraction layer approach is the flexibility that the framework possesses to integrate additional communication protocols. When a system under test requires the use of a communication protocol that is not present in the framework, the framework maintainer can add a new communication module to the abstraction layer. The new module will have to provide the required interfaces to the framework application and manually added to the framework. The user of the framework can then change the protocol to the new one without knowing the internals of the new module and without changing the application to call any new interface. Section 3.5 offers a more detailed look at the abstraction layer.

Another major focus of the design of the framework is modularity. By identifying the main features of the framework, it is possible to break the features down into modules. Such is the case of the data processing and file processing modules. The idea of creating a module for each process is similar to that of the abstraction layer. The modules keep the implementation of the feature hidden from the user and allows the maintainer of the framework to add, change, or remove code from the module without having a big impact on the application.

3.2 Data acquisition

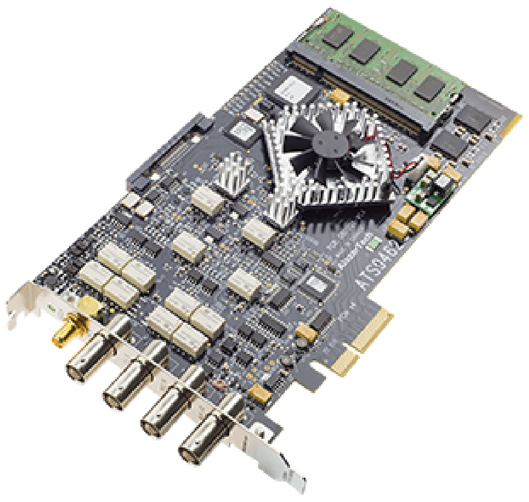


Figure 3.3: Digitizer card ATS9462 by AlazarTech.

The framework captures the data from the system under test by utilizing a digitizer PCI Express interface card. The card is a model ATS9462 manufactured by AlazarTech, which allows the capturing of data from two simultaneous channels.

For the framework to be able to control the digitizer, the framework uses a binary file that has the necessary code to configure the digitizer and begin the capturing of the data. The binary takes as input the following arguments:

- Number of samples
- Range of sample
- Time of capture
- Number of channels
- Name of output file

The arguments previously mentioned are coded in the framework as part of the user pre-configured settings. The values of the settings are set to default values, as shown in the next table. These values will be part of the experiments throughout Chapter 4 except for the number of channels that will vary.

Parameter	**Default value**
Number of samples	10MS/s
Range of sample	400mV
Time of capture	1000s
Number of channels	1
Name of output file	experiment

Table 3.1: Default values for digitizer arguments.

The time of capture is kept at a high value because the framework automatically stops the capture once it finishes sending commands to the system. Ideally, we want to keep the digitizer capturing data and have the framework stop the capture whenever we consider that enough data was collected. Thus, we use a value such that the data capture will not be interrupted when the framework is still issuing commands to the system under test.

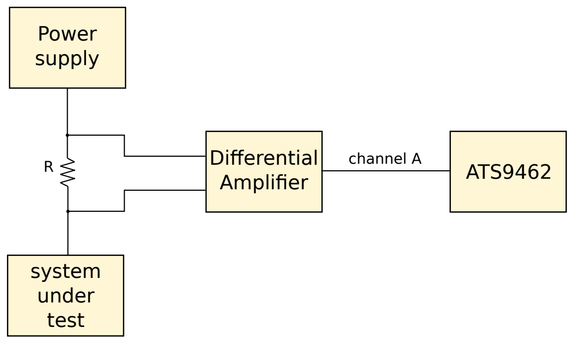


Figure 3.4: Data acquisition setup.

Figure 3.4 shows the connection between the digitizer card to the system under test. It would be easy to assume that the card is connected directly to the system; however, this presents a problem. By attaching a probe directly to the power line of the system, we are effectively adding the probe to the overall circuitry. This additional circuitry can lead to changes in the behaviour of the power consumption and thus introducing unwanted noise to the traces[**Reference**]. To work our way around this issue, we introduce a resistor (represented by R) between the power supply and the system under test. However, the value of the resistor must be small so that the voltage drop caused by the resistor does not affect the functionality of the system. Nevertheless, adding a small resistor leaves us with a new issue. Because the value of the resistor is small, the voltage drop will be in the order of millivolts. Since the digitizer card capture range starts at 200mV, we have to add a circuit that outputs the voltage drop across the resistor and amplifies the signal so that our digitizer card can capture the data. The solution is to add differential amplifier stage at the input of the digitizer card.

3.3 Framework running modes

One of our objectives is that our framework can provide an easy way to gather data for machine learning side-channel analysis. We also want the framework to allow researchers to expand upon the current framework and use it as a platform to perform online side-channel analysis. Thus, we designed two running modes for our framework: training mode and normal mode. In training mode, the framework loads a list of commands to send to the system under test. The framework then sends each command and captures the respective trace while creating processes to convert the files and transform the data in parallel. The framework will do this continuously without any input. In contrast, in normal mode, the framework sends a command, waits for the file and data to be processed, and then it proceeds to send the next command and repeat the process. The following subsections offer a more detailed explanation for each mode.

3.3.1 Training mode

When running the framework in training mode, it is important to note that the main objective is to capture data that will be useful to train a model. Thus, the framework does not have to wait for any additional input or feedback before sending the next message.

Another difference between the training mode and the normal mode is the creation of the monitor thread. The monitor thread's objective is to monitor the availability of files or data that need processing by checking two queues and the number of framework's host PC cores available for the framework to use. One queue contains files that need to be processed, and the other queue contains data that needs to be processed. If any of the queues are not empty, the monitor thread will check if there are PC cores not being used by the framework. If there is a core available, the monitor thread will create the corresponding process based on the queue's contents.

The file processing process is in charge of converting the raw data to a text file. The process then parses the file and splits the file into individual traces, or trims down the data using user pre-configured parameters in the framework. Finally, the process saves the resulting traces in the form of a CSV file. We decided to use the CSV format because its human readable, it can be processed by almost all applications, and it is simple to parse. Although CSV comes with some disadvantages such as: no distinction between text and numeric values, no way to represent binary data, and poor support of special characters, the data in the CSV files are only positive or negative decimal values that require no special formatting.

The data processing process takes the CSV file generated by the file processing process so it can apply a moving average to the data using user pre-configured parameters to clean the noise from the data. Section 3.6 contains more information regarding this process.

The image below shows the sequence that the framework follows in this mode:

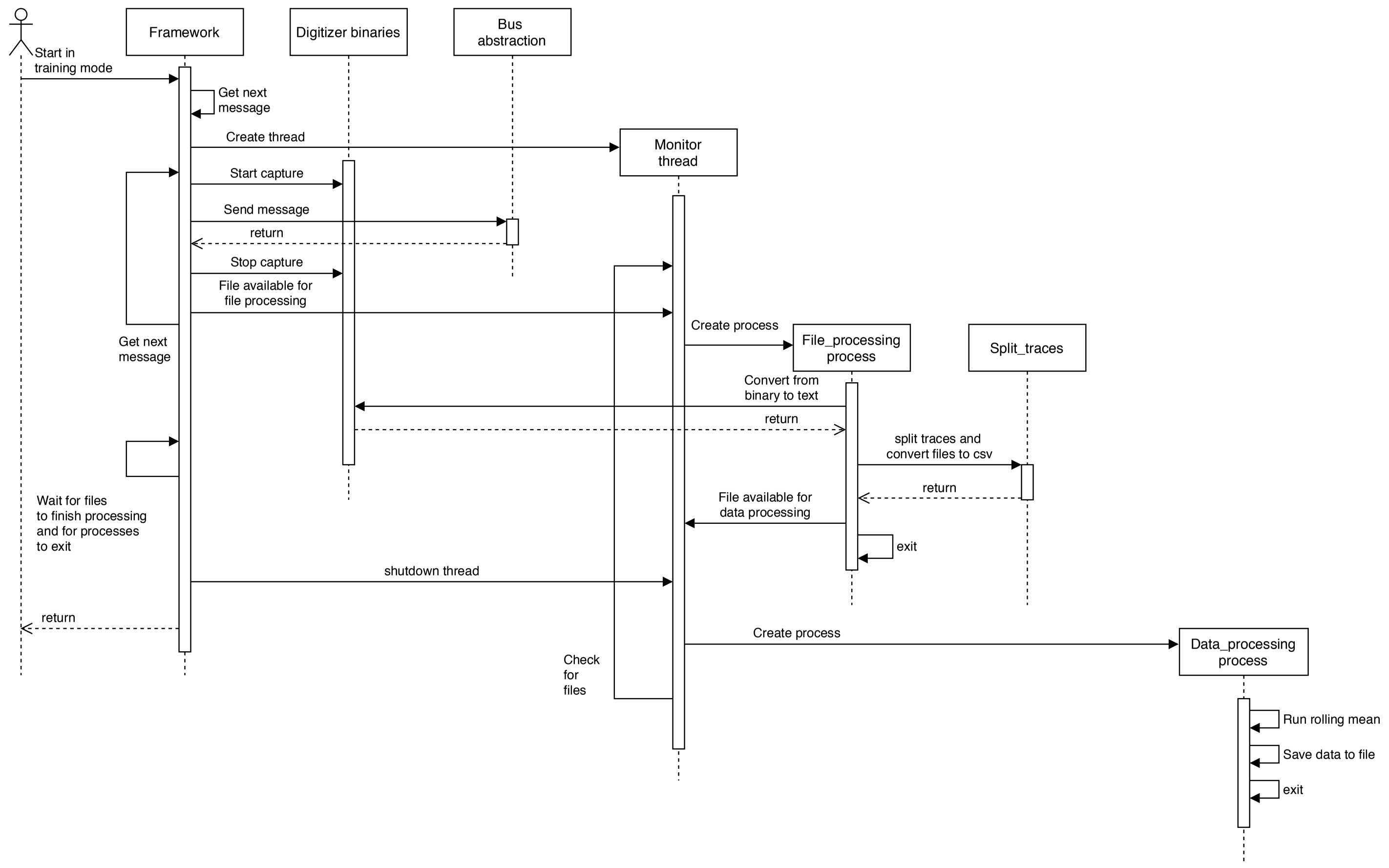


Figure 3.5: Training mode sequence diagram.

3.3.2 Normal mode

Running the framework in normal mode follows a similar sequence to that of the training mode. The main difference of the running mode is that the application no longer creates the monitor thread. The absence of the thread is because the data obtained from the system under test must be fed to a distance calculator and compare against the database to decide which message follows. In contrast, in training mode, we have a list of all the messages to send to the system.

The absence of the monitor thread and the processes created by the thread does not mean that the overall functionality and data obtained are any different. In fact, the same code that is used by the processes is used by the framework in running mode. This is the result of the modularity approach mentioned in Section 3.1. This approach allows us to change how each module behaves with both training and normal modes being able to benefit from the changes without the need for additional coding effort.

Figure 3.6 shows the interaction between the framework, the digitizer binaries, the communication abstraction layer, and the data processing module.

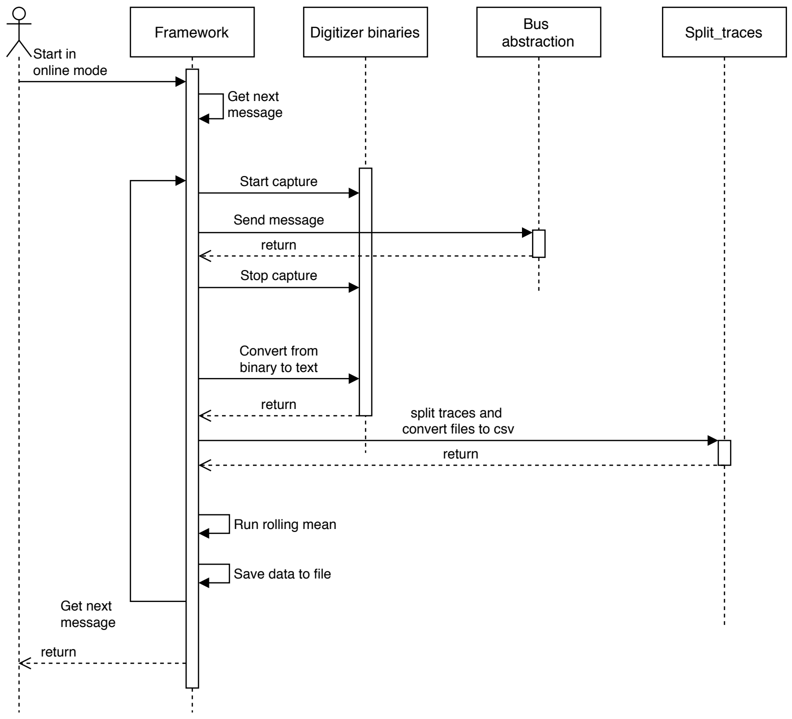


Figure 3.6: Normal mode sequence diagram.

3.4 Shared memory

We designed our framework to use multi-threading and multiprocessing to take advantage of the framework's host PC multi-core CPU to reduce bottle-necking. Since some of the operations are independent of each other, we split them into different from each other. We used three processes: one for the framework, one for processing the files, and one for processing the data. Furthermore, we added an additional thread to the framework's main process that is in charge of creating the necessary processes when necessary.

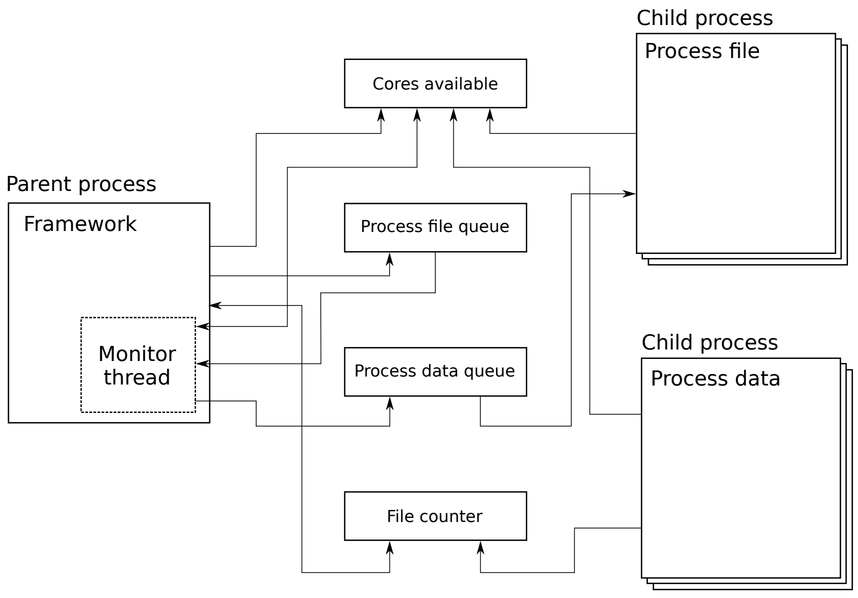


Figure 3.7: Interaction between processes using shared memory.

Processes need to transfer data between each other in the framework. The interaction between processes in the framework uses blocks of shared memory. Locks control access to shared resources. The framework uses one lock per memory block. When a process needs to access the shared resource, its main thread must first acquire the lock. If some other thread is already using the lock, the requester must wait for the lock to be released. The process for acquiring and releasing the locks uses the synchronization APIs from Python's multiprocessing library, which are blocking function calls. In Figure 3.7, we can see the blocks of shared memory. Next, we will talk about each memory block and its purpose.

- **Cores available**: This is a variable that contains the number of cores available to create processes. When the monitor thread sees that some files or data needs to be processed, the thread will read the value of the variable and see if the value is greater than zero. If the value is different than zero, the monitor thread creates the corresponding process and decreases the value of the counter by one. When a process finishes (including the framework application), it releases the core it was using and adds the free core to the counter by increasing the counter by one.

- **Process file queue**: This is a list that contains all the files that require to be processed and trimmed down to a single power trace. This list is accessed by the framework application and the file processing process. Looking at this from the point of view of a producer-consumer problem, the application is in charge of producing a raw file, and the monitor thread to create a process that will consume the file. When no file is available, the monitor thread will skip the process creation.

- **Process data queue**: Much like the queue previously mentioned, this is a list that contains all the files in which data needs to be processed. This list is accessed by the file processing process and the data processing process. Following the example from the previous queue, the file processing process creates the necessary data, and the data processing process consumes the resource.

- **File counter**: This is a variable that contains the number of files that are yet to be processed. This variable takes into account the moment the file containing the raw data is generated by the application, till the moment the data processing processed finishes and generates the processed data file. This variable main usage is to know when there are no more files to process and safely exit the application.

3.5 Communication Abstraction Layer

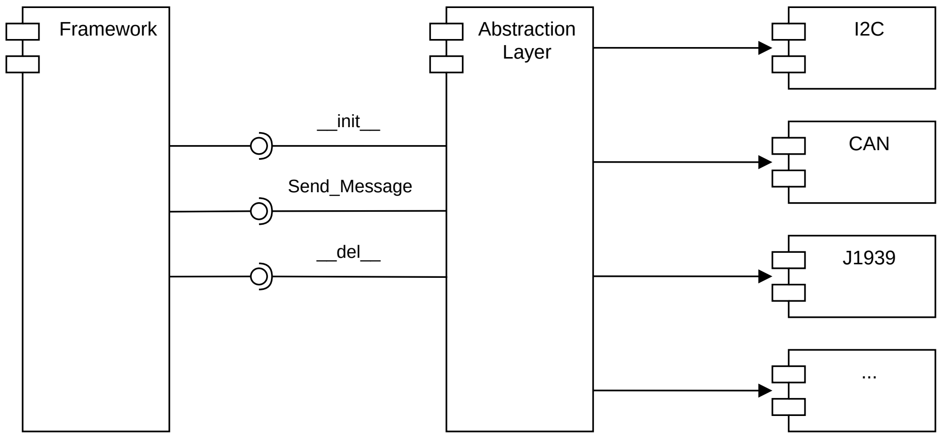


Figure 3.8: Communication Abstraction layer.

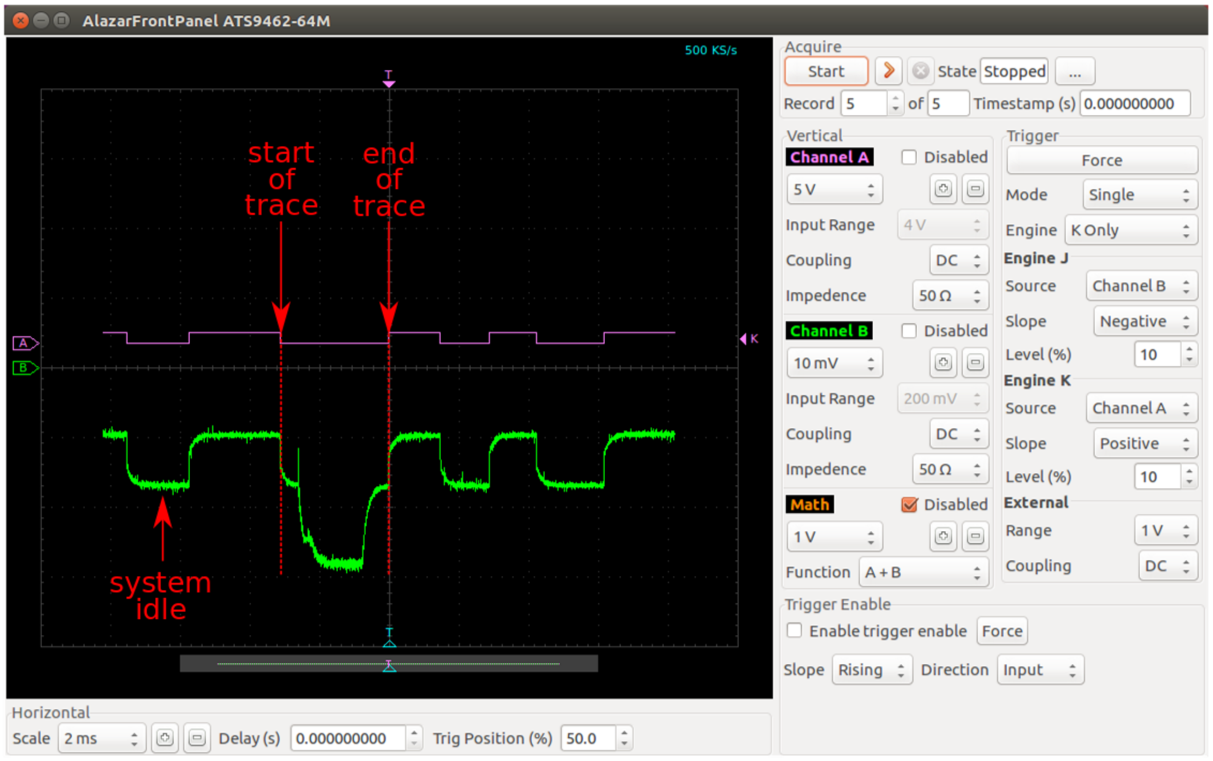
One of the framework's objectives is to be able to be used to map inputs to side-channel behaviour on any blackbox system with observable inputs and outputs. Since every system under test might have different communication protocols for its inputs, the framework must change between communication protocols without the need for major rework to the framework. The solution to this problem is to create an abstraction layer. This layer will provide the necessary interfaces to set up the communication bus, send a message, and close the bus. These interfaces allow the user to choose the desired communication protocol in the pre-configured parameter of the framework without changing any code or functionality in the framework. Following this approach allows the user to focus only on the design of the experiment rather than having to spend extra effort in modifying the application itself.

Another advantage of this strategy is the fast integration of new communication protocols into the framework without needing to modify the application code. A framework maintainer can add a new communication protocol by creating a new module that provides the required interfaces. The maintainer is then free to add any processing that the module might require before sending the message to the system under test.

The interfaces supported by the abstraction layer are the following:

- **__init__**: This interface initiates the bus. It contains the necessary code to set the right interface, speed, etc. Note that this constructor interface is not accessed directly by the user. When the framework creates the object, the constructor executes the code in this method.
- **Send_Message**: This interface sends a message out to the system under test. It contains the code necessary to adjust the format of the arguments. It takes as arguments the target device address, data to send, and register offset if supported.
- **__del__**: This interface closes the bus. It contains the code needed to clean up the module and close down the communication with the bus. Note that like the constructor interface, this destructor interface is not accessed directly by the user. When the framework deletes the object, the destructor is invoked and runs the code in this method.

3.6 Data processing



AlazarFrontPanel ATS9462-64M

500 KS/s

Acquire
Start
State Stopped
Record 5 of 5 Timestamp (s) 0.000000000

Vertical
Channel A Disabled
5 V
Input Range 4 V
Coupling DC
Impedence 50 Ω

Channel B Disabled
10 mV
Input Range 200 mV
Coupling DC
Impedence 50 Ω

Math [✓ Disabled]
1 V
Function A + B

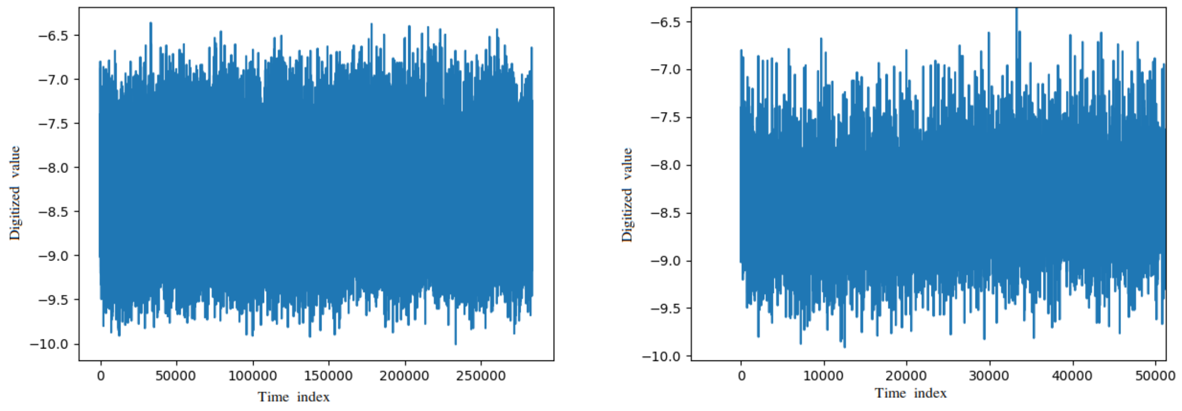
Trigger
Force
Mode Single
Engine K Only
Engine J
Source Channel B
Slope Negative
Level (%) 10
Engine K
Source Channel A
Slope Positive
Level (%) 10
External
Range 1 V
Coupling DC

Trigger Enable
Enable trigger enable Force
Slope Rising Direction Input

Horizontal
Scale 2 ms Delay (s) 0.000000000 Trig Position (%) 50.0

Figure 3.9: Screenshot of power trace from I2C device.

Our first results obtained during the first experiment in Section 4.1 were showing unique power traces when sending commands to the system under test. The power traces obtained were distinguishable from the power consumption of the system under test idle state. Figure 3.9 shows a screen capture of one of the early tests from Section 4.1. Channel B shows a clear difference between the system in idle mode and when the system receives a command. However, we stumbled across a new problem when performing the experiments in Section 4.2.



(a) Full plot. (b) Trimmed plot.

Figure 3.10: Plot of power trace using raw data.

Figure 3.10 shows the plot of the raw data obtained by the framework from a system under test in an experiment from Section 4.2. On the left side, we can see the plot of the entire data we collected, and on the right side, we can see a trimmed version of the data showing only the first 50000 data samples. The trace shows no obvious discernable features in the traces compared to the one in Figure 3.9. This is due to the fact that some components of the system are producing high-frequency signals. Removing the high-frequency signals results in a more defined signal of interest which in our case is the overall power consumption of the system over longer periods of time than those of the high-frequency signals. These longer periods of time can be interpreted as lower frequency signals than the high-frequency dominating the time series. Figure 3.11 shows an example of a low pass filter being applied to a signal.

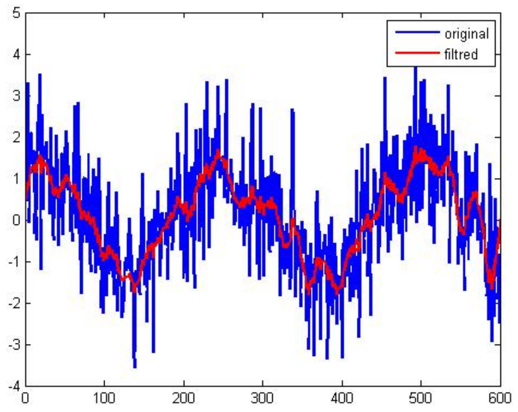


Figure 3.11: Low pass filter based on fast Fourier transform [21].

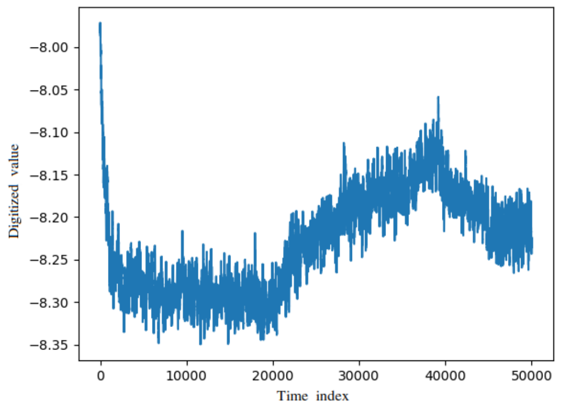
There are several ways to filter out high-frequencies of a signal. The method we decided to use in our research is a rolling mean calculation. The rolling mean is the result of computing the mean of a subset of orderly sequential values where the subset size is defined by a positive integer value called "window size". The set of values is then shifted to the right, adding a new value to the set and dropping the last value from the set. The process repeats itself until all the values in the main set are covered.

The rolling mean can be described using the following formula:

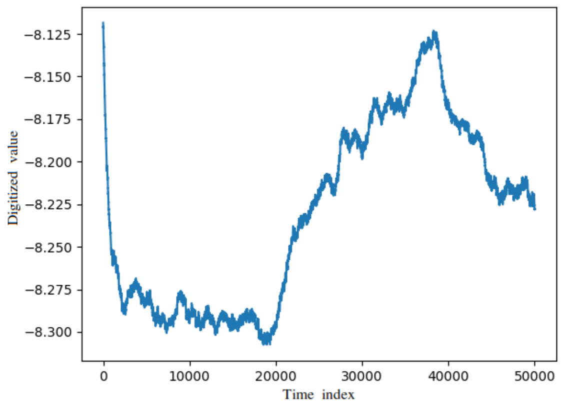
$$RM = \frac{x_1 + ... + x_n}{n} \tag{3.1}$$

Where RM is the rolling mean value or moving average value, $x_1 + ... + x_n$ is the set of values in the window, and n is window size value.

When we compute the rolling mean, we can see that the high-frequency elements starts to minimize and details from the trace start to appear. Figure 3.12a shows the result of a rolling mean with a window size of 100 points, and Figure 3.12b shows the results of computing a rolling mean of 1000 points.



(a) Window size of 100.

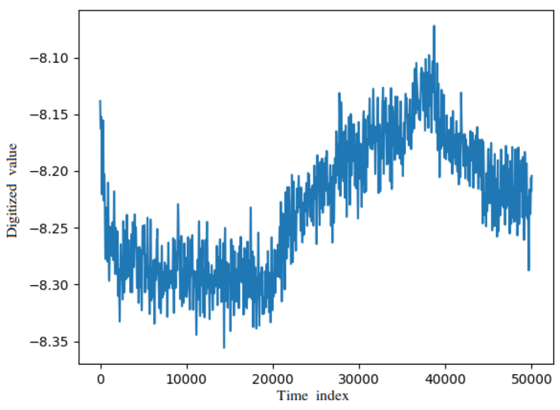


(b) Window size of 1000.

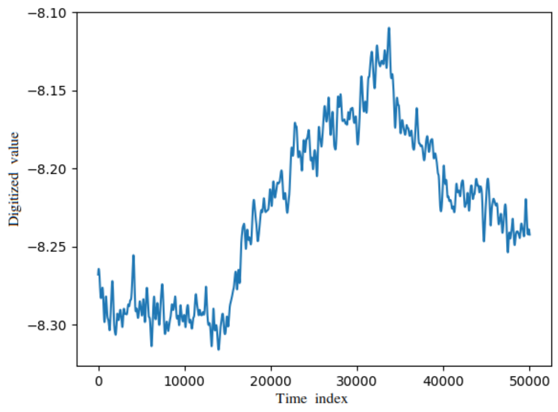
Figure 3.12: Plots of power traces after applying a rolling mean.

As we can appreciate in the Figure 3.12, the rolling mean can significantly reduce the high-frequency components from the original data shown in Figure 3.10b compared to that from Figure 3.12b.

Another approach that expanded on the use of the rolling mean was to repeat the operation several times using the same window. This approach allows us to remove the high-frequency elements while having smaller window sizes. As a result, we minimize the loss of data and features present in the trace. The following figures show the results of computing a rolling mean with a window size of 10 points. The figure on the left iterated the computation 100 times, and the one on the right iterated the computation 1000 times.



(a) 100 iterations.



(b) 1000 iterations.

Figure 3.13: Plots of power traces after applying an iterative rolling mean with a window size of 10.

After experimenting with different window sized and iterations, we designed the rolling mean method with flexibility in mind. The user can configure the number of iterations to run the rolling mean on the data set, the window size of the rolling mean, and the number of data points to keep.

Chapter 4

Experiments

In this chapter, we describe the experiments we conducted using the framework described in Chapter 3 on various systems. We begin by gathering power traces from individual I2C devices to confirm that different systems can generate distinct power traces. Next, we collect power traces of a custom-built embedded system to verify that one system can generate multiple distinct power traces when the framework sends different commands to the system. Afterward, we gather power traces from a water pump system to confirm that systems using ECUs and J1939 can generate distinct power traces. Finally, we gather traces from a cargo truck to confirm that a system with multiple ECUs can generate distinct power traces.

4.1 I2C devices

The purpose of the experiment in this section was to observe changes in the power consumption of the system while sending commands to the system. We wanted to confirm whether or not systems have distinct power traces that we can map to system inputs. To do this, we gathered four different I2C sensors and sent I2C commands to each sensor and captured the power consumption of the sensors individually.

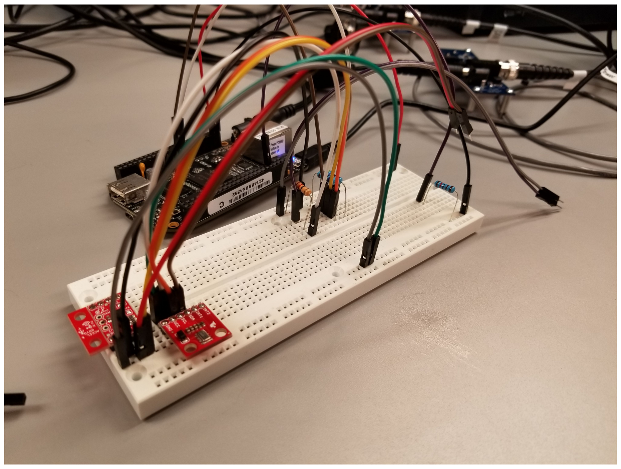


Figure 4.1: Photo of I2C system for experiment.

The setup for this experiment consisted of a BeagleBone Black card, the digitizer card (described in 3.3) with two differential amplifiers for two channels, a synchronization module for the digitizer, and four I2C devices. The diagram in Figure 4.2 shows the connection of all devices. One important thing to note in this experiment is that the framework resides in the BeagleBone Black for this experiment. The framework is inside the board to have direct access to the I2C bus. Because the framework is not running on the computer where the digitizer card is, we created a synchronization module for this experiment. The framework uses the synchronization module to start and stop the acquisition of data the same way as if the framework is running in the digitizer workstation.

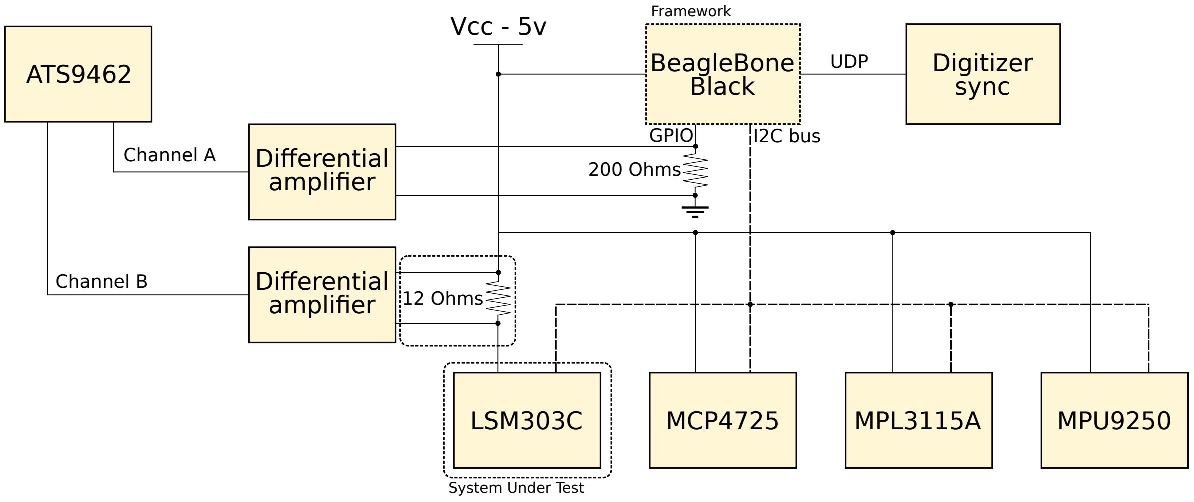


Figure 4.2: System diagram for I2C experiment.

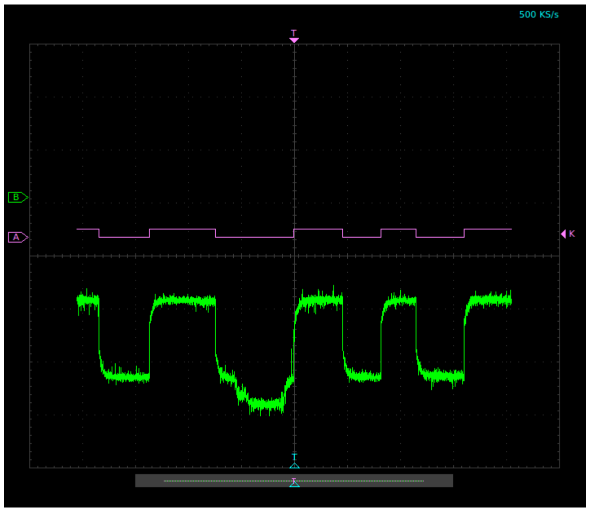
Our system under test in this experiment consists of four I2C devices:

- **LSM303C**: Six degree of freedom inertial measurement unit
- **MCP4725**: 12 bit digital to analog interface
- **MPL3115A**: Precision pressure sensor with altimetry
- **MPU9250**: 3-Axis gyroscope, accelerometer, and magnetometer

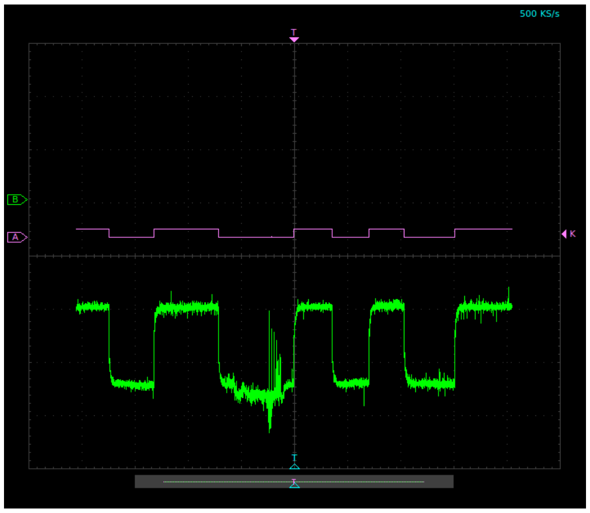
This experiment was repeated for each of the I2C devices to confirm that each I2C device generated distinct power traces. For each measurement, we moved the 12 Ohms resistor to the power line of the I2C device that we were testing. Figure 4.2 only shows channel B measuring the power consumption of device LSM303C.

This experiment also used the second channel of the digitizer card. The purpose of this channel is to allow the framework to know when a trace begins and when it ends more precisely. The second channel is measuring the output of a GPIO pin from the BeagleBone Black. The GPIO toggles once to indicate that the trace has begun and twice to indicate the end. Figure 3.9 has labels showing where the start and end of the trace are. This functionality is exclusive to the experiments using the BeagleBone Black card, and we plan to port it to the framework as part of the future work.

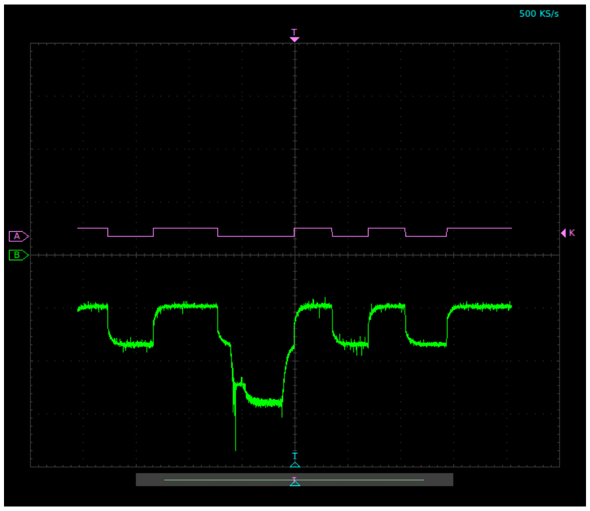
4.1.1 Results



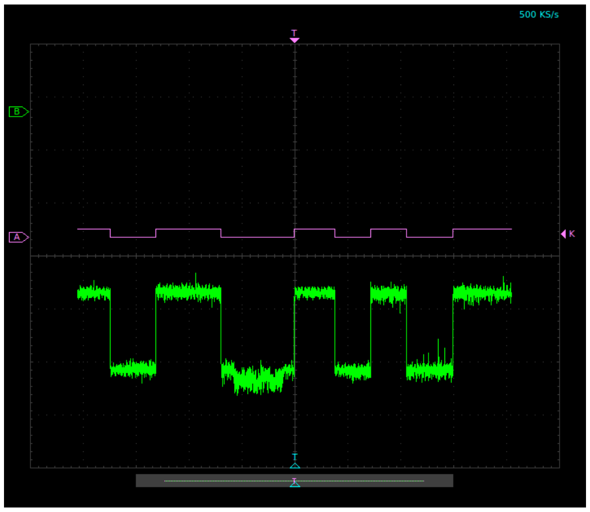
(a) Trace of LSM303C.



(b) Trace of MCP4725.



(c) Trace of MPL3115A2.



(d) Trace of MPU9250.

Figure 4.3: Screen captures of traces from each I2C device.

We selected one command per each I2C device by reading the datasheets of the devices. These commands would prompt the I2C device to do a reading of its sensor and return the computed value. Figure 4.3 contains a capture of the digitizer tool to see a live visual representation of the power consumption of the system under test when we send the selected command to the I2C device. As we can see, each of the traces generated is different from each other. This confirms that we can generate distinct traces for different devices when we send a command to the system.

4.2 Custom system

The purpose of this experiment is to determine if distinct power traces can be generated for input commands when we take into account the power consumption of the system as a whole. For this experiment, we have built a custom embedded system. This setup aims to mimic a small and simple embedded system with a microprocessor and a microcontroller, both accessing slave devices. Much like the experiment in Section 4.1, this experiment also uses a second channel to generate reference points to mark the beginning and end of the traces.

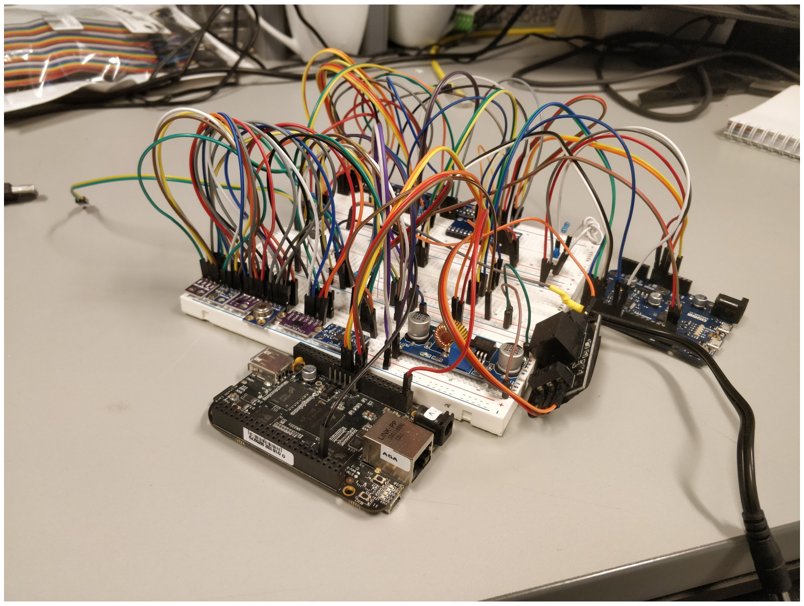


Figure 4.4: Photo of custom system for experiment.

The setup for the second experiment consisted of a custom-built system. Similar to the setup in Section 4.1, there is a BeagleBone Black and I2C sensor. In addition to the components previously mentioned, there is an Arduino Zero, a digital multiplexer, and a voltage regulator. The biggest difference between this experiment and the last one is the measuring point. For this experiment, the power consumption is measured at the entrance point of the voltage for the system, as shown in Figure 4.5.

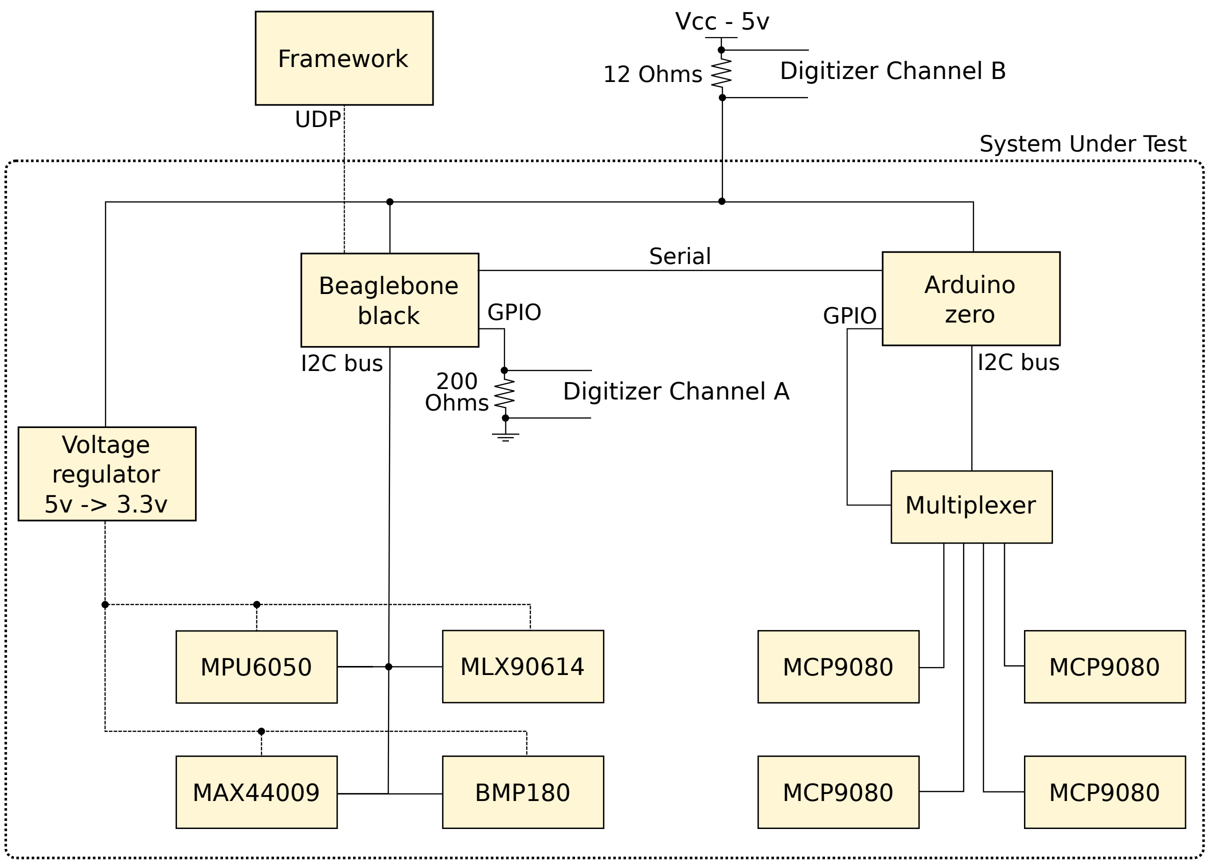


Figure 4.5: System diagram for custom system experiment.

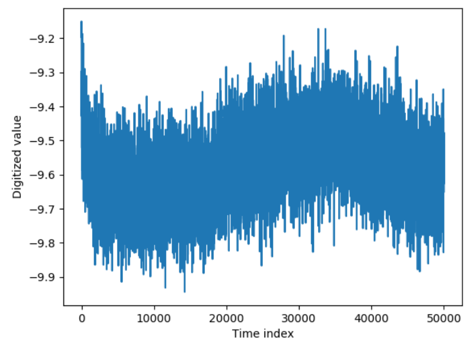
4.2.1 Command space

Command to send
Read accelerometer value from sensor 1
Read temperature value
Read gyroscope value
Read ambient temperature from infrared sensor
Read object temperature from infrared sensor
Read ambient light value
Read temperature value from sensor 2
Read pressure value 1
Read pressure value 2
Read pressure value 3
Read pressure value 4
Calculate mean temperature value from MCU sensors
Read temperature from MCU sensor 1
Read temperature from MCU sensor 2
Read temperature from MCU sensor 3
Read temperature from MCU sensor 4

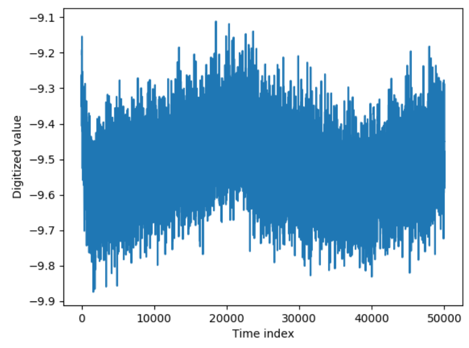
Table 4.1: Command space for custom system under test.

The command space for this system consists of reading values from multiple sensors. Some of the values come from the same sensor. The framework will send the commands to the system via UDP. The microprocessor will get the request and distribute it to the right module. For this experiment, we set the framework for training mode, and we sent each command 100 times to the system.

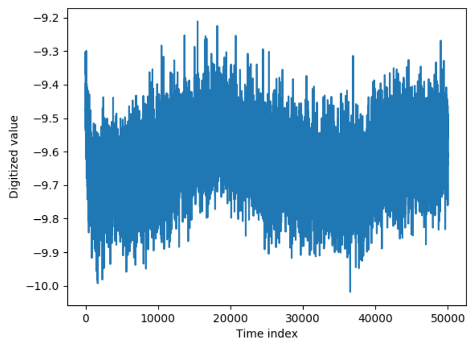
4.2.2 Results



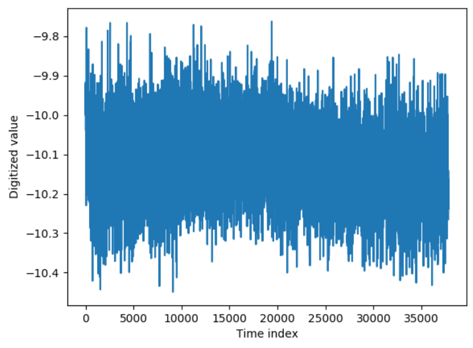
(a)



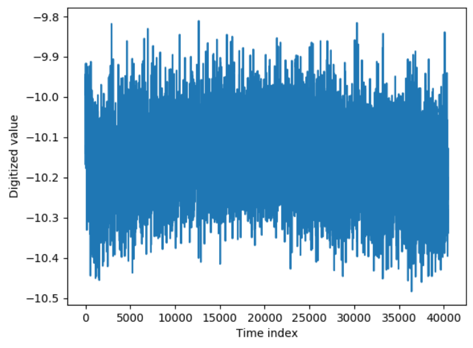
(b)



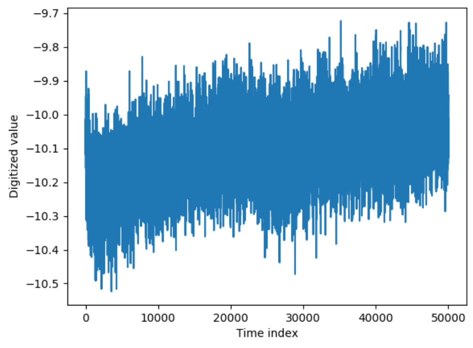
(c)



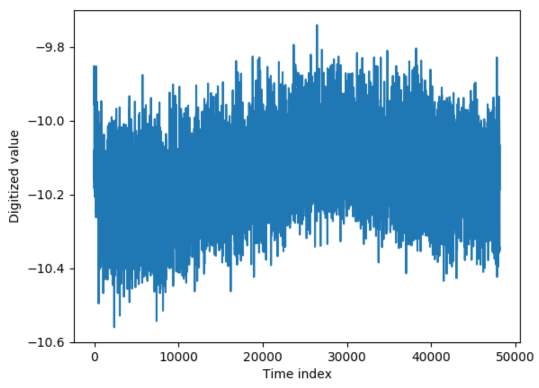
(d)



(e)

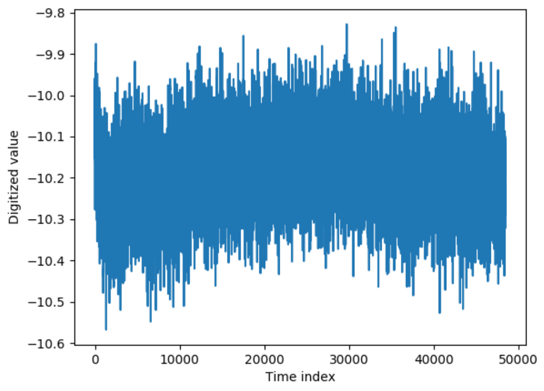


(f)



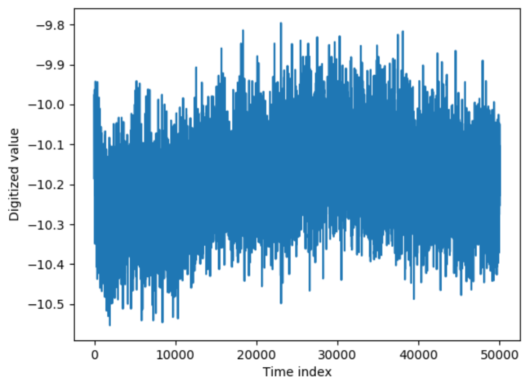
−9.8
−10.0
−10.2
−10.4
−10.6
Digitized value
0
10000
20000
30000
40000
50000
Time index

(g)



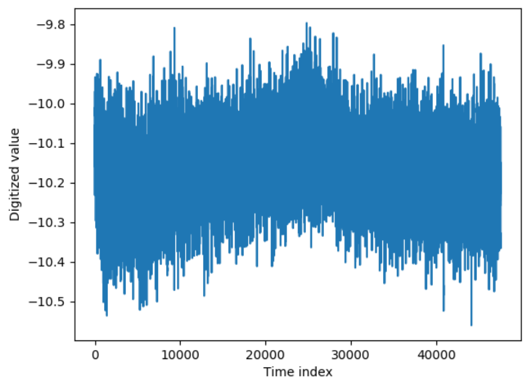
−9.8
−9.9
−10.0
−10.1
−10.2
−10.3
−10.4
−10.5
−10.6
Digitized value
0
10000
20000
30000
40000
50000
Time index

(h)



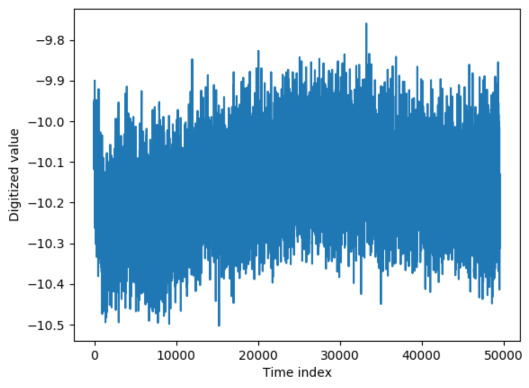
−9.8
−9.9
−10.0
−10.1
−10.2
−10.3
−10.4
−10.5
Digitized value
0
10000
20000
30000
40000
50000
Time index

(i)



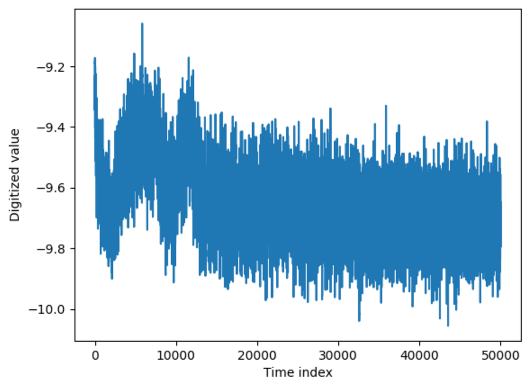
−9.8
−9.9
−10.0
−10.1
−10.2
−10.3
−10.4
−10.5
Digitized value
0
10000
20000
30000
40000
Time index

(j)



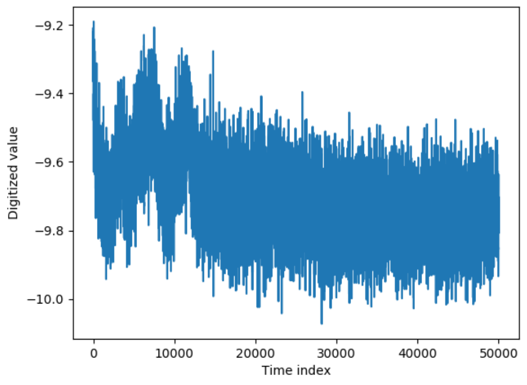
−9.8
−9.9
−10.0
−10.1
−10.2
−10.3
−10.4
−10.5
Digitized value
0
10000
20000
30000
40000
50000
Time index

(k)

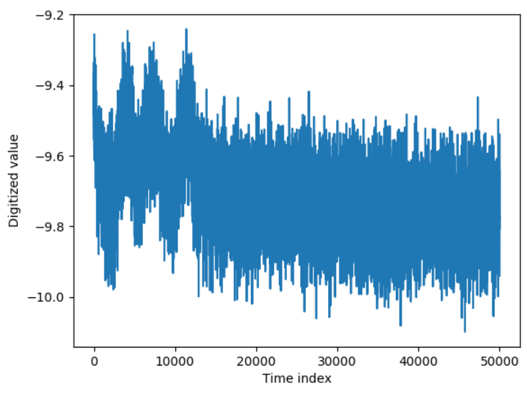


−9.2
−9.4
−9.6
−9.8
−10.0
Digitized value
0
10000
20000
30000
40000
50000
Time index

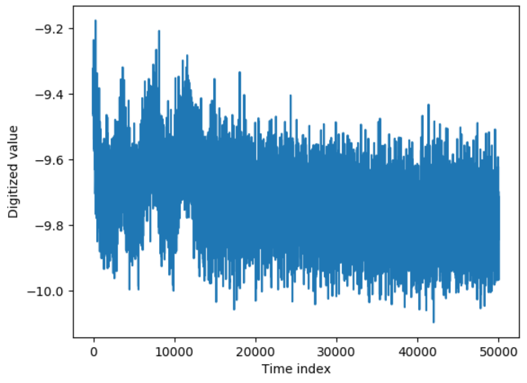
(l)



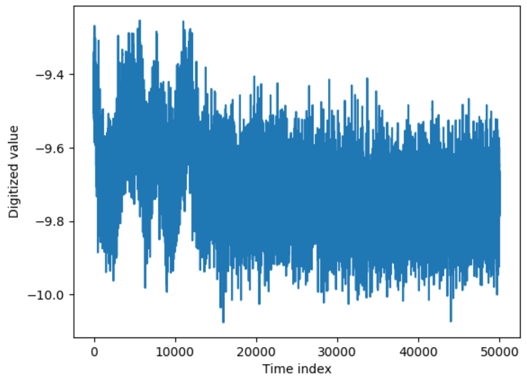
(m)



(n)



(o)



(p)

Figure 4.6: Plots of selected traces for each command sent to the system under test. Only one iteration of the rolling mean with a window 10 was computed for this step.

Figure 4.6 displays plots selected from the collection of data for each of the commands sent to the system processed using a single iteration of the rolling mean with a window of 10. Visual inspection of the plots indicates similar results to the ones obtained in the previous experiment where the traces are mostly distinguishable from each other. Table 4.2 maps the plots to their corresponding commands from the command space.

Command to send	**Trace**
Read accelerometer value	4.6a
Read temperature value from sensor 1	4.6b
Read gyroscope value	4.6c
Read ambient temperature from infrared sensor	4.6d
Read object temperature from infrared sensor	4.6e
Read ambient light value	4.6f
Read temperature value from sensor 2	4.6g
Read pressure value 1	4.6h
Read pressure value 2	4.6i
Read pressure value 3	4.6j
Read pressure value 4	4.6k
Calculate mean temperature value from MCU sensors	4.6l
Read temperature from MCU sensor 1	4.6m
Read temperature from MCU sensor 2	4.6n
Read temperature from MCU sensor 3	4.6o
Read temperature from MCU sensor 4	4.6p

Table 4.2: Map of traces to command space.

One thing to note is that some of the traces are similar to each other, as it is the case with Figures 4.6. The similarity between traces is due to the fact that similar traces are generated by commands triggering a reading from the same module in the system. Although the commands access different registers from the module and the type of data sent back by the module is different, it is possible that the module treats all requests in the same way but only sends back the data corresponding to the request.

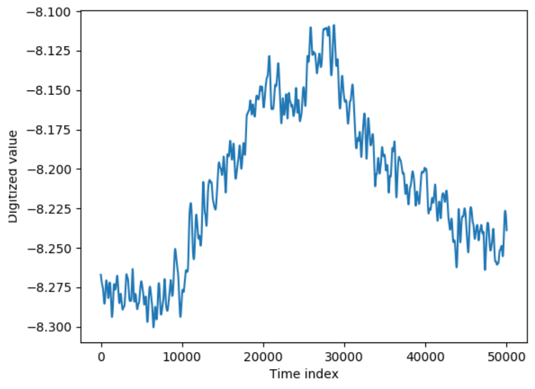
If we filter out the repeated traces, we are left with the following command space that will guarantee distinct traces for each command:

4.2.3 New command space

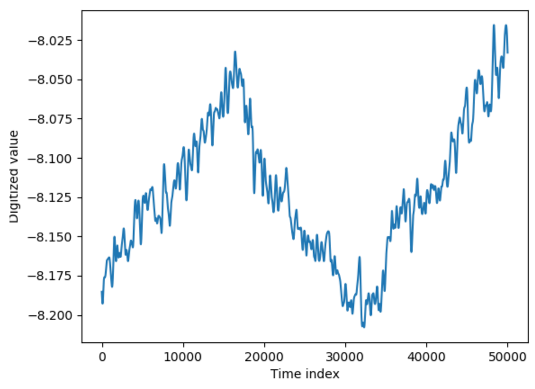
Command to send
Read accelerometer value
Read temperature value from sensor 1
Read ambient temperature from infrared sensor
Read ambient light value
Read temperature value from sensor 2
Read pressure value 1
Read temperature from MCU sensor 1

Table 4.3: New command space after removing commands that generate repeated traces.

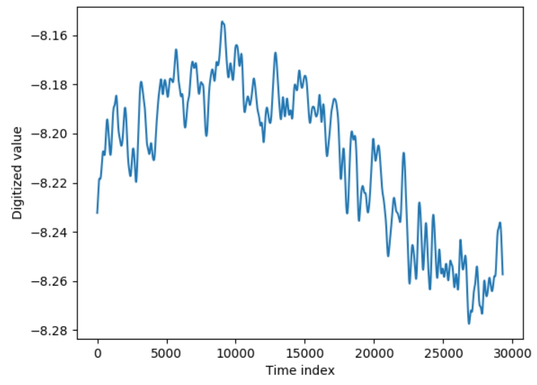
The resulting data shown in Figure 4.6 still contain a lot of noise and has to be removed. To remove the noise, the data processing module described in Section 3.6 runs 1000 iterations of the rolling mean computation with a window of 10 and generates new data files. Figure 4.7 shows the new plots for the new data.



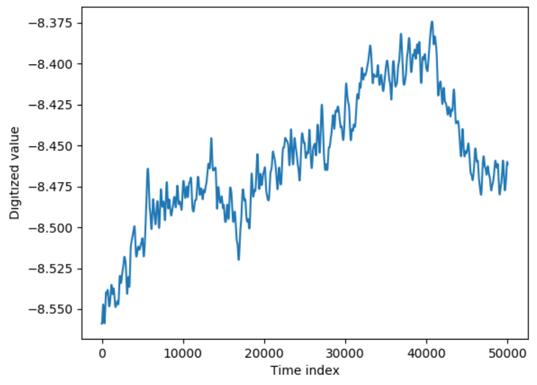
(a)



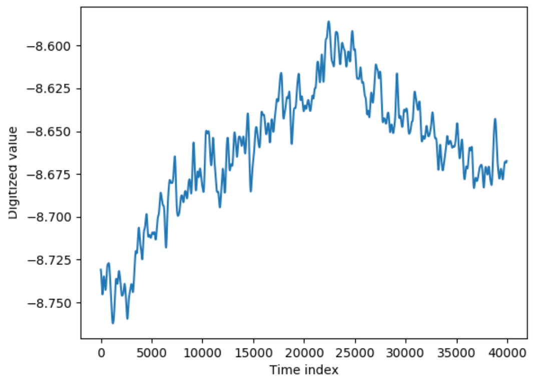
(b)



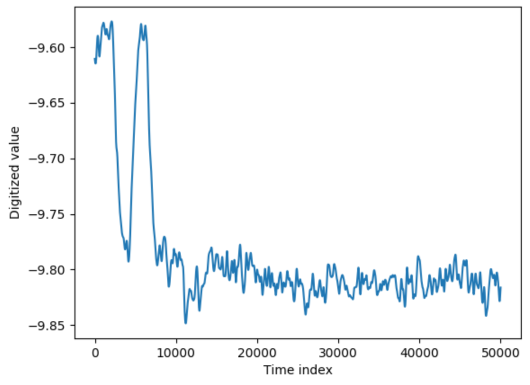
(c)



(d)



(e)



(f)

Figure 4.7: Plots of selected traces for new command space in Table 4.3. 1000 iterations of the rolling mean with a window 10 was computed for this step.

The plots in the Figure 4.7 show a more clean and defined plot. We have included more samples of plots for each command in the appendix for a visual confirmation that each command generates reproducible traces.

Command to send	**Trace**
Read accelerometer value	4.7a
Read temperature value from sensor 1	4.7b
Read ambient temperature from infrared sensor	4.7c
Read ambient light value	4.7d
Read temperature value from sensor 2	4.7e
Read temperature from MCU sensor 1	4.7f

Table 4.4: Map of traces to new command space.

4.2.4 Classification of traces

Using the data collected in Subsection 4.2.2, we can proceed to classify the traces. We chose to use support vector machines for our classification method. We decided to use support vector machines because the method is effective in high dimensional spaces. The method also uses supervised learning, which we can use to our advantage since we know the labels of each data sample.

We used 100 data samples for each of our commands from the command space in table 4.4. Out of our pool of samples, we randomly selected 70% of samples per command for training our model and 30% of samples per command for testing our model.

Using the support vector machines algorithm from the scikit-learn library for python, we obtain the metrics in Table 4.5.

Metric	**Percentage**
Accuracy	97.78%
Precision	98.00%
Recall	97.76%

Table 4.5: Metrics for support vector machine classification of custom system's data.

4.3 EMP water pump

The purpose of this experiment is to confirm if by using a command space of J1939 messages, the system can generate distinct power traces. For this experiment we are using a system that has an ECU with a J1939 bus. Since the ECU is used to control an actuator, we focused our command space on messages that change the behaviour of the actuator which we then mapped to the power traces we obtained using the framework.

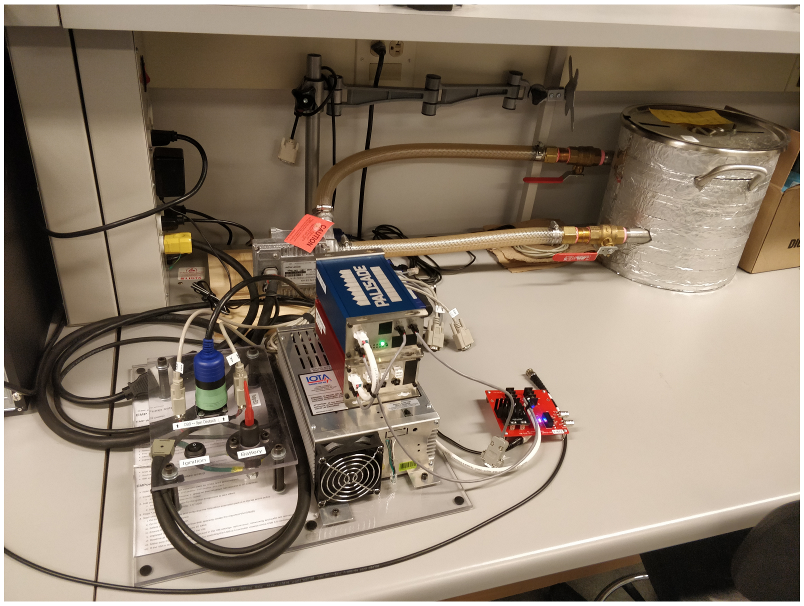


Figure 4.8: Photo of EMP pump system for experiment.

The setup for the third experiment consisted of an ECU-controlled fluid pump system which is used to train students in vehicle penetration testing and reverse engineering of automotive ECUs. For our experiment, we only consider the ECU and disregard any other components of the system.

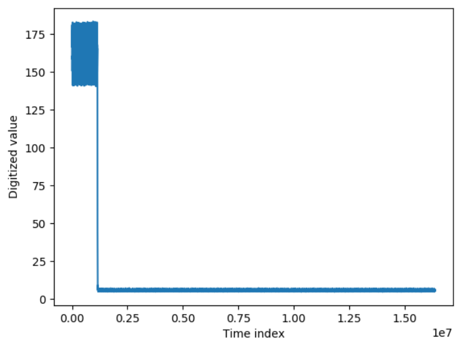
4.3.1 Command space

Command to send
Motor: off — speed: 1000 RPM
Motor: off — speed: 500 RPM
Motor: off — speed: 100 RPM
Motor: off — speed: 0 RPM
Motor: off — speed: 1250 RPM
Motor: off — speed: 1375 RPM
Motor: off — speed: 1626 RPM
Motor: off — speed: 3000 RPM
Motor: on — speed: 1000 RPM
Motor: on — speed: 500 RPM
Motor: on — speed: 100 RPM
Motor: on — speed: 0 RPM
Motor: on — speed: 1250 RPM
Motor: on — speed: 1375 RPM
Motor: on — speed: 1626 RPM
Motor: on — speed: 3000 RPM
Motor: reverse — speed: 1000 RPM
Motor: reverse — speed: 500 RPM
Motor: reverse — speed: 100 RPM
Motor: reverse — speed: 0 RPM
Motor: reverse — speed: 1250 RPM
Motor: reverse — speed: 1375 RPM
Motor: reverse — speed: 1626 RPM
Motor: reverse — speed: 3000 RPM

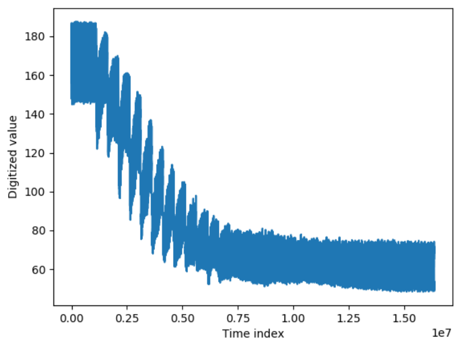
Table 4.6: Command space for pump ECU.

The command space for this system consists of changing the system's actuator speed. The actuator of the system has three modes: off, on, reverse. We have defined a set of eight commands to change the actuator's speed to different values. This set of eight speeds has been repeated for each one of the actuator modes for a total of 24 commands to send.

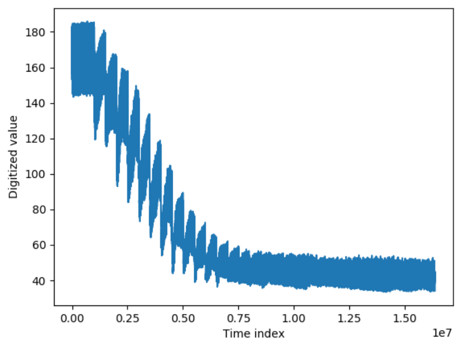
4.3.2 Results



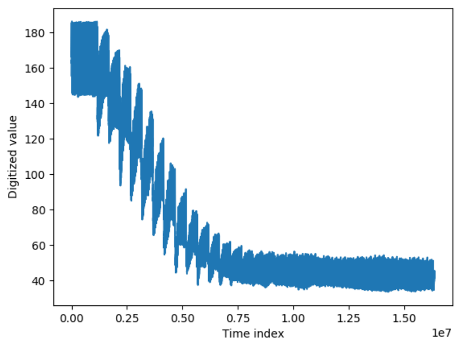
(a)



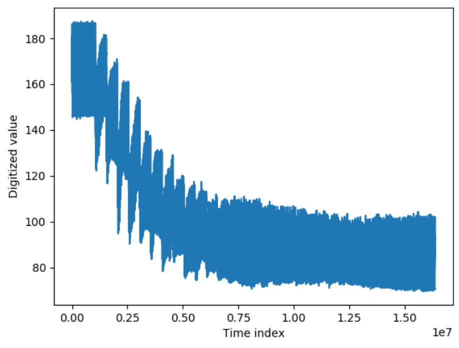
(b)



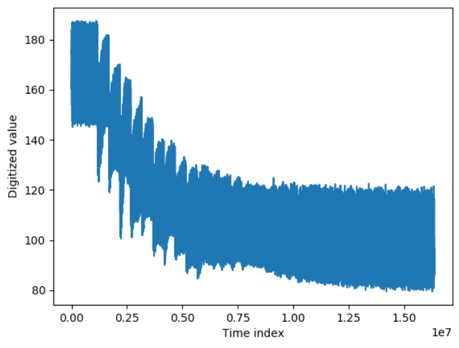
(c)



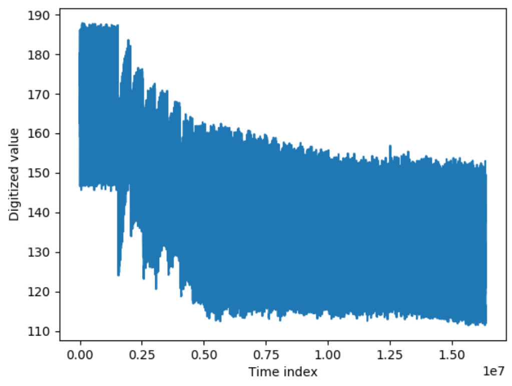
(d)



(e)

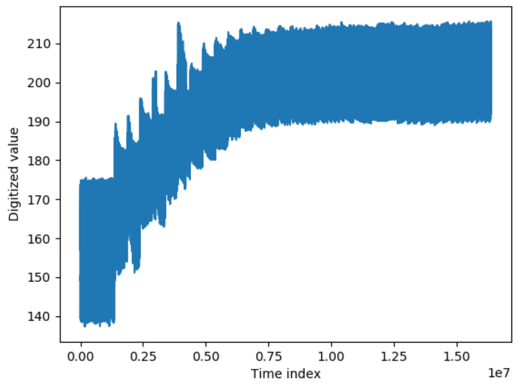


(f)



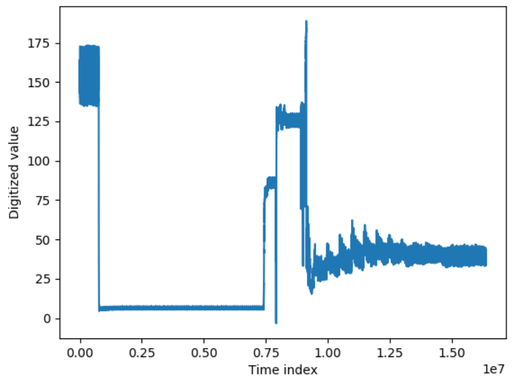
Digitized value
Time index
1e7

(g)



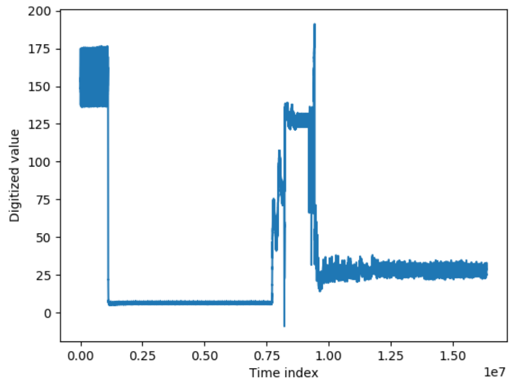
Digitized value
Time index
1e7

(h)



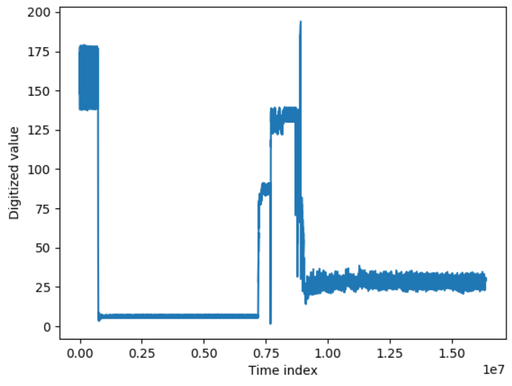
Digitized value
Time index
1e7

(i)



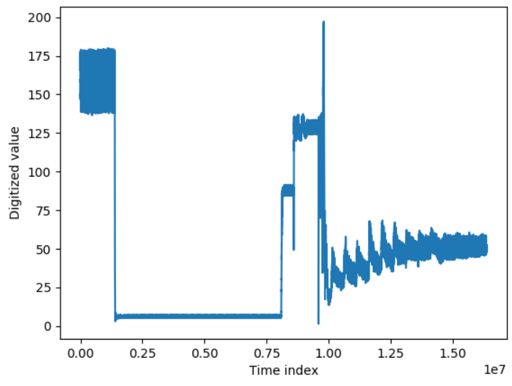
Digitized value
Time index
1e7

(j)



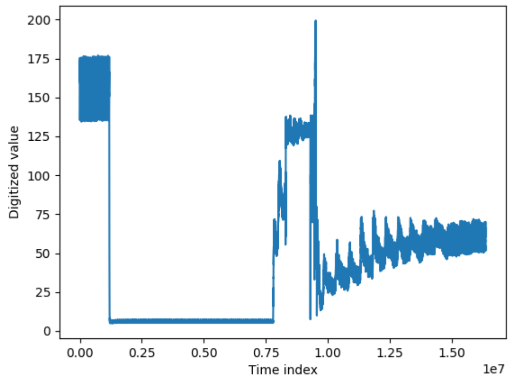
Digitized value
Time index
1e7

(k)

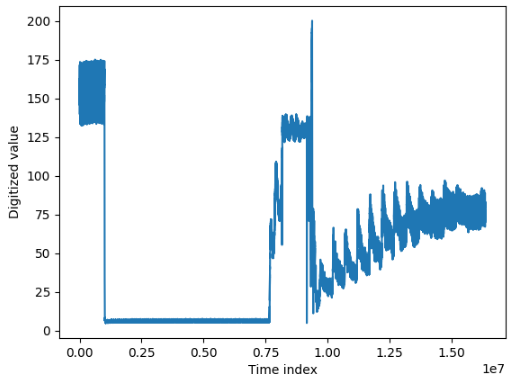


Digitized value
Time index
1e7

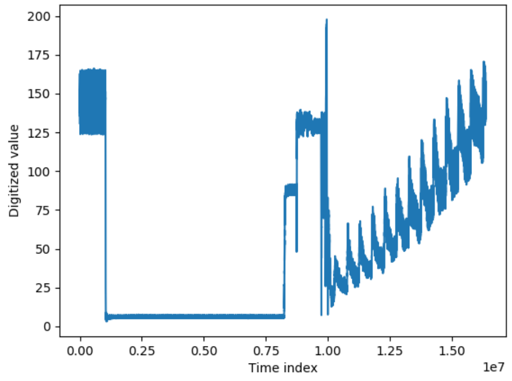
(l)



(m)



(n)



(o)

Figure 4.9: Plots of traces for each command sent to the pump ECU. 1000 iterations of the rolling mean with a window of 10 were computed for each trace

The plots in Figure 4.9 show clear plots for each command. One thing to notice is that only one plot for when the motor is off is shown in Figure 4.9a since all of the traces for when the motor is off are the same.

When selecting the values for the actuator's speed, the values will have to vary by a minimum factor to produce traces that are significantly different, otherwise when attempting to classify the traces, the classifier will not be able to calculate a good distance between traces and will end up grouping similar traces. The following pair of plots show the previous issue: 4.9c resembles 4.9d, 4.9e resembles 4.9f, 4.9j resembles 4.9k, and 4.9l resembles 4.9m.

Command to send	**Trace**
Motor: off — speed: 1000 RPM	4.9a
Motor: off — speed: 500 RPM	4.9a
Motor: off — speed: 100 RPM	4.9a
Motor: off — speed: 0 RPM	4.9a
Motor: off — speed: 1250 RPM	4.9a
Motor: off — speed: 1375 RPM	4.9a
Motor: off — speed: 1626 RPM	4.9a
Motor: off — speed: 3000 RPM	4.9a
Motor: on — speed: 1000 RPM	4.9b
Motor: on — speed: 500 RPM	4.9c
Motor: on — speed: 100 RPM	4.9d
Motor: on — speed: 0 RPM	4.9a
Motor: on — speed: 1250 RPM	4.9e
Motor: on — speed: 1375 RPM	4.9f
Motor: on — speed: 1626 RPM	4.9g
Motor: on — speed: 3000 RPM	4.9h
Motor: reverse — speed: 1000 RPM	4.9i
Motor: reverse — speed: 500 RPM	4.9j
Motor: reverse — speed: 100 RPM	4.9k
Motor: reverse — speed: 0 RPM	4.9a
Motor: reverse — speed: 1250 RPM	4.9l
Motor: reverse — speed: 1375 RPM	4.9m
Motor: reverse — speed: 1626 RPM	4.9n
Motor: reverse — speed: 3000 RPM	4.9o

Table 4.7: Map of traces to new command space.

4.3.3 Classification of traces

We used 5 data samples for each of our commands from the command space in table 4.4. We randomly selected 70% of the samples for training our model and 30% of the samples for testing our model.

Metric	**Percentage**
Accuracy	92.10%
Precision	89.89%
Recall	93.33%

Table 4.8: Metrics for support vector machine classification for pump's data.

4.4 Sterling truck

The purpose of this experiment is to gather power traces of a true black-box system. We decided to use a cargo truck to expand on the ECU and J1939 results from Section 4.3. Another factor for deciding to use the truck is how attackers are targeting automotive vehicles in recent years as discussed in [6] and [19].

Figure 4.10: Photo of Sterling truck used in experiment.

The setup for the final experiment consisted of a Sterling truck. We connected a resistor in the fuse box of the vehicle to measure the power consumption, and we connected the framework to the J1939 bus of the vehicle.

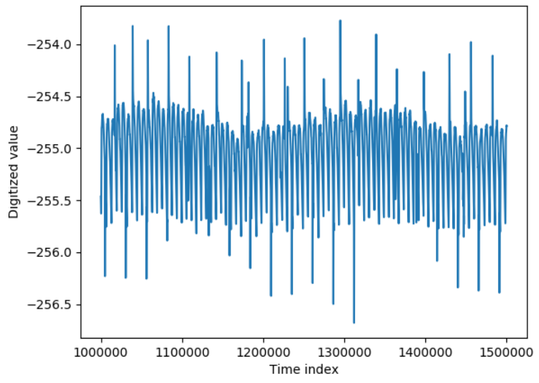
4.4.1 Command space

Command to send
Request engine temperature 1
Request engine fluid level
Request dash display
Request vehicle electrical power 1
Request air supply pressure

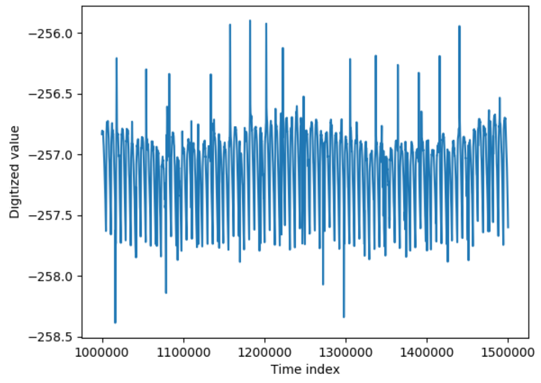
Table 4.9: Command space for truck.

The command space selected for this system consists of general requests using PGN 59904. By using PGN 59904, we can request the ECUs to send specific PGNs to the bus even if the message is not scheduled or requested by the system itself. For this experiment we chose five PGNs with various information of the vehicle.

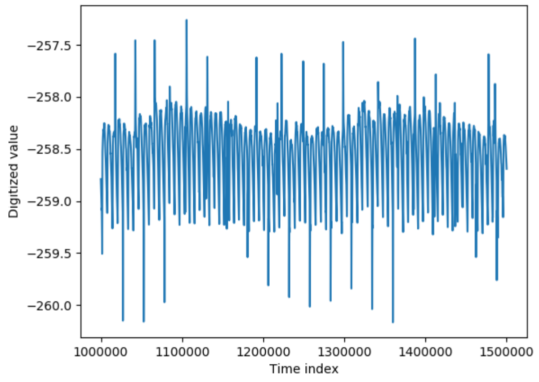
4.4.2 Results



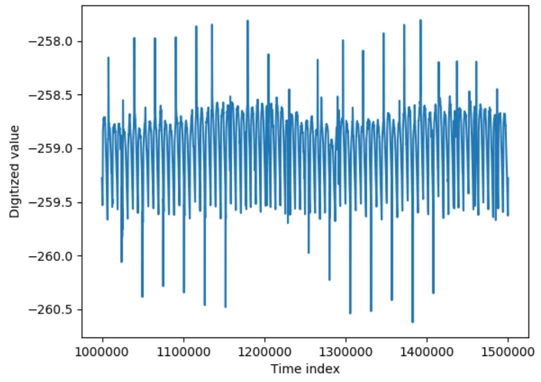
(a)



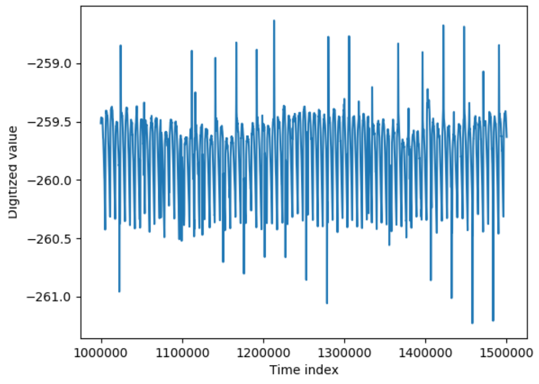
(b)



(c)



(d)

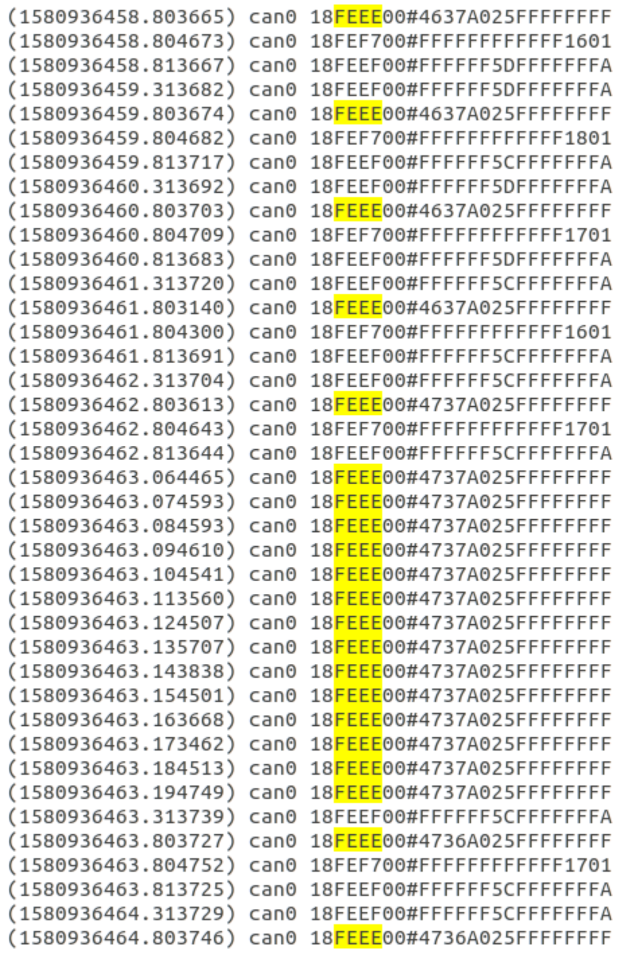


(e)

Figure 4.11: Plots of traces for each command sent to the Sterling truck. 1000 iterations of the rolling mean with a window of 10 were computed for each trace.

Figure 4.11 shows a sample of a trace for each of the commands in the command space. The traces are zoomed to a section where the message should have been received by the system. Although the framework was able to capture traces, the results for this experiment are inconclusive as there is no visual indication that the traces obtained are distinguishable from each other. An explanation as to why we do not get the desired outcome in the traces could be given by the fact that we have a large time window from the moment the framework starts capturing the data, to the moment when we send the message to the system. Another factor to consider is that the framework itself is running in a Linux machine and the context switching of the system must be taken into account, making it more difficult to calculate the start of the trace.

In addition to the power traces we obtained from this experiment, a capture of the J1939 log shows the system answering to the requests.



```
(1580936458.803665) can0 18FEEE00#4637A025FFFFFFFF
(1580936458.804673) can0 18FEF700#FFFFFFFFFFFF1601
(1580936458.813667) can0 18FEEF00#FFFFFF5DFFFFFFFA
(1580936459.313682) can0 18FEEF00#FFFFFF5DFFFFFFFA
(1580936459.803674) can0 18FEEE00#4637A025FFFFFFFF
(1580936459.804682) can0 18FEF700#FFFFFFFFFFFF1801
(1580936459.813717) can0 18FEEF00#FFFFFF5CFFFFFFFA
(1580936460.313692) can0 18FEEF00#FFFFFF5DFFFFFFFA
(1580936460.803703) can0 18FEEE00#4637A025FFFFFFFF
(1580936460.804709) can0 18FEF700#FFFFFFFFFFFF1701
(1580936460.813683) can0 18FEEF00#FFFFFF5DFFFFFFFA
(1580936461.313720) can0 18FEEF00#FFFFFF5CFFFFFFFA
(1580936461.803140) can0 18FEEE00#4637A025FFFFFFFF
(1580936461.804300) can0 18FEF700#FFFFFFFFFFFF1601
(1580936461.813691) can0 18FEEF00#FFFFFF5CFFFFFFFA
(1580936462.313704) can0 18FEEF00#FFFFFF5CFFFFFFFA
(1580936462.803613) can0 18FEEE00#4737A025FFFFFFFF
(1580936462.804643) can0 18FEF700#FFFFFFFFFFFF1701
(1580936462.813644) can0 18FEEF00#FFFFFF5CFFFFFFFA
(1580936463.064465) can0 18FEEE00#4737A025FFFFFFFF
(1580936463.074593) can0 18FEEE00#4737A025FFFFFFFF
(1580936463.084593) can0 18FEEE00#4737A025FFFFFFFF
(1580936463.094610) can0 18FEEE00#4737A025FFFFFFFF
(1580936463.104541) can0 18FEEE00#4737A025FFFFFFFF
(1580936463.113560) can0 18FEEE00#4737A025FFFFFFFF
(1580936463.124507) can0 18FEEE00#4737A025FFFFFFFF
(1580936463.135707) can0 18FEEE00#4737A025FFFFFFFF
(1580936463.143838) can0 18FEEE00#4737A025FFFFFFFF
(1580936463.154501) can0 18FEEE00#4737A025FFFFFFFF
(1580936463.163668) can0 18FEEE00#4737A025FFFFFFFF
(1580936463.173462) can0 18FEEE00#4737A025FFFFFFFF
(1580936463.184513) can0 18FEEE00#4737A025FFFFFFFF
(1580936463.194749) can0 18FEEE00#4737A025FFFFFFFF
(1580936463.313739) can0 18FEEF00#FFFFFF5CFFFFFFFA
(1580936463.803727) can0 18FEEE00#4736A025FFFFFFFF
(1580936463.804752) can0 18FEF700#FFFFFFFFFFFF1701
(1580936463.813725) can0 18FEEF00#FFFFFF5CFFFFFFFA
(1580936464.313729) can0 18FEEF00#FFFFFF5CFFFFFFFA
(1580936464.803746) can0 18FEEE00#4736A025FFFFFFFF
```

Figure 4.12: J1939 log showing request of PGN FEEE.

In Figure 4.12 we can see that PGN FEEE is broadcasted on the bus periodically before being sent multiple times in a row as a result of the requests by the framework. This confirms that the system is responding to our requests. We can assume that the trace might contain the behaviour we expect, but we are unable to pinpoint the exact start of the trace. In Section 5.2 we propose a solution to have a better method of delimiting the power traces more accurately.

4.4.3 Classification of traces

For the classification of power traces that we obtained in this experiment, we used 25 data samples for each of our commands from the command space in table 4.9. As in Subsections 4.2.4 and 4.3.3, we randomly selected 70% of samples per command for training our model and 30% of samples per command for testing our model. We obtained the following metrics in Table 4.10.

Metric	**Percentage**
Accuracy	38.22%
Precision	36.80%
Recall	40.74%

Table 4.10: Metrics for support vector machine classification using truck's data.

The low metrics that we obtained are expected since the power traces are really similar to each other. This can be calculated by computing the DTW distance of the traces. Table 4.11 shows the DTW distances of the power trace of the first command from Table 4.9 to all of the commands that we sent to the system. We used five different samples for each command and calculated the average.

Command to compare	**Average DTW Distance**
Request engine temperature 1	1161217.5988
Request engine fluid level	1045310.2158
Request dash display	1076973.2522
Request vehicle electrical power 1	1177055.2404
Request air supply pressure	1272766.7989

Table 4.11: Average DTW distances between command *"Request engine temperature 1"* and the rest of the command space.

As we can see, the DTW distances in Table 4.11 are all similar which can explain why the classifier metrics are so low. One way we can aid the classifier is by making the data capture more accurate with the addition of a second channel to mark the start and end points of the power trace to avoid having power traces with noise generated by other components of the system.

Chapter 5

Future Work and Conclusion

5.1 Lessons learned

In this thesis, we presented four different experiments in Chapter 4. We designed a fifth experiment that did not provide the expected results, but it is important to highlight the results of this experiment.

The setup for the fifth experiment consisted of a DSLR CANON T3i camera with a modified USB power adapter with a resistor that allowed us to monitor the power consumption of the camera. The purpose of this experiment was to see if, by using different firmware for the camera, we could obtain different power traces for the same commands in the command space.

One of the main challenges of the experiment using the DSLR camera was to find the right resistor to capture the power traces. At first, we tried using the same resistor values as those used in experiments in Sections 4.1 and 4.2, but the camera would have trouble taking pictures as the shutter would not close. As a result, the camera would lockout, forcing us to remove the power source from the camera and reapply power to get it working again. Later we found that the camera was sensitive to voltage drops caused by the resistor. Therefore, we used a resistor of 0.1 Ohms for this experiment to avoid a significant voltage drop that could affect the camera's operations.

The traces obtained from this system were not helpful and seem only to contain background noise. The only visible change in the traces is when the camera opens and closes the shutter, but other than that, the rest of the commands from the command space were not generating distinct power traces. Our explanation for this behaviour is low computational complexity of the system's logic to address command requests. When we send a command to the camera to change the settings of the camera, the resources needed to change the settings are not complex enough to be distinguished by the operations being carried by the OS in the background, which are more complex than saving variables for later use. Once the actuator of the camera is triggered, the camera draws more power, and the variables that we previously set come into play. This leaves us with only one command that generates a somewhat defined trace, and the only other variable we can control is how long the shutter remains open. We could have obtained better results if we could isolate the power consumption of only the processor without the peripherals of the device.

One important detail to point out is that although the power traces did not generate the expected results, the data captured by the framework did vary from firmware to firmware. By using a custom firmware found online, the trace for the "take photo" command was slightly bigger in size than when using official firmware from the manufacturer. This by itself is not enough proof that our experiment was successful, but it could be an indicator that traces might change from firmware to firmware when issuing the same commands.

5.2 Future Work

Some of the next steps to extend on the work presented in this thesis are:

- *Increase the accuracy of the power traces with the help of a second channel.* Much like the experiments in Sections 4.1 and 4.2 where a GPIO pulse delimits the traces, the framework can take advantage of this approach and integrate a module that can add references to the starting and ending of a trace. An approach like this might yield better results for experiments like the one in 4.4.

- *Add an abstraction layer for the digitizer.* The intention of moving the digitizer APIs behind an abstraction layer is to provide the framework with the flexibility to switch between digitizer tools or effectively move the framework to another system and having a remote connection to the computer where the digitizer resides.

- *Implement and integrate anomaly detection module.* With the addition of a validation module, the framework can become a powerful tool to aid in the detection of anomalies by comparing a power trace gathered from a system to those from a database

5.3 Conclusion

In this thesis, we presented a framework which is used to obtain power profiles to aid in mapping system inputs to side-channel behaviour. We reviewed the architecture of the framework and discussed the main features of the framework. We also demonstrated the effectiveness of the framework through a series of experiments from which we learned the following:

- Systems can produce distinct power traces when we send different commands to the system.
- Classification of the obtained traces is possible as shown in the classification metrics with accuracy, precision, and recall values of 90% or more for experiments in sections 4.2 and 4.3
- Different firmware might produce different results.

Some considerations to keep in mind when designing experiments using the framework are:

- Some commands will trigger functionality that shares power consumption with other commands. Therefore it is important to limit the command space to commands that will only generate distinct traces.
- When using commands to set an actuator value, define the test in such a way that the values will produce traces with sufficient distance between each other to avoid classification issues.
- The commands sent to the unit must trigger some complex behaviour in the system to induce a higher power consumption that can produce distinct traces.

Finally, some of the properties of the framework and the traces obtain are discussed next:

- The framework has the flexibility of being deployed in multiple systems thanks to the abstraction layer that allows for new communication modules to be developed and integrated with ease.

- With the map of the power profiles to the command space, it is possible to detect anomalies by capturing a power trace and comparing it to the database of known profiles.

- Given a power trace obtained while the system received a request, it is possible to know which command was requested by comparing the power trace to the profile map.

References

[1] Michael R. Berthold and Frank Höppner. On clustering time series using euclidean distance and pearson correlation. 2016.

[2] Robert A. Bridges, Jarilyn M. Hernández Jiménez, Jeffrey Nichols, Katerina Goseva-Popstojanova, and Stacy J. Prowell. Towards malware detection via CPU power consumption: Data collection design and analytics (extended version). 2018.

[3] Shane S. Clark, Benjamin Ransford, Amir Rahmati, Shane Guineau, Jacob Sorber, Wenyuan Xu, and Kevin Fu. Wattsupdoc: Power side channels to nonintrusively discover untargeted malware on embedded medical devices. 2013.

[4] Jamie Condliffe. Hackers stole $172 billion from people in 2017, 2018 (accessed March 12, 2020). https://www.technologyreview.com/f/610043/hackers-stole-172-billion-from-people-in-2017/.

[5] DeepAI. Statistical classification, 2019 (accessed April 21, 2020). https://deepai.org/machine-learning-glossary-and-terms/statistical-classification.

[6] Alex Drozhzhin. Black hat usa 2015: The full story of how that jeep was hacked, 2015 (accessed March 12, 2020). https://www.kaspersky.com/blog/blackhat-jeep-cherokee-hack-explained/9493/.

[7] Sam Friedman and Nikhil Gokhale. Pursuing cybersecurity maturity at financial institutions, 2019 (accessed March 12, 2020). https://www2.deloitte.com/us/en/insights/industry/financial-services/cybersecurity-maturity-financial-institutions-cyber-risk.html.

[8] Kauffman, Sean. Static transformation of power consumption for program tracing and software attestation, 2017 (accessed March 12, 2020).

[9] Kamal Lamichhane, Carlos Moreno, and Sebastian Fischmeister. Non-intrusive program tracing of non-preemptive multitasking systems using power consumption. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2018.

[10] Hiscox Ltd. Hiscox cyber readiness report 2019, 2019 (accessed March 12, 2020). https://www.hiscox.com/documents/2019-Hiscox-Cyber-Readiness-Report.pdf.

[11] Vijeet H. Meshram and Ashish B. Sasankar. Security in embedded systems : Vulnerabilities , pigeonholing of attacks and countermeasures. 2016.

[12] Shervin Minaee. 20 popular machine learning metrics. part 1: Classification & regression evaluation metrics, 2019 (accessed April 21, 2020). https://towardsdatascience.com/20-popular-machine-learning-metrics-part-1-classification-regression-evaluation-me

[13] Carlos Moreno, Sean Kauffman, and Sebastian Fischmeister. Efficient program tracing and monitoring through power consumption – with a little help from the compiler. 2016.

[14] The United State's Council of Economic Advisers. The cost of malicious cyber activity to the u.s. economy, 2018 (accessed March 11, 2020). https://www.whitehouse.gov/wp-content/uploads/2018/03/The-Cost-of-Malicious-Cyber-Activity-to-the-U.S.-Economy.pdf.

[15] National Institute of Standards and Technology. Nist/sematech e-handbook of statistical methods, 2012 (accessed March 12, 2020). https://www.itl.nist.gov/div898/handbook/pmc/section4/pmc41.htm.

[16] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[17] Inc. QY Research. Global embedded systems market trends, survey, growth and analysis by 2018-2025!!!, 2018 (accessed March 11, 2020). http://www.digitaljournal.com/pr/4006746.

[18] Ling Liu Sen Nie and Yuefeng Du. Free-fall: Hacking tesla from wireless to can bus. 2017.

[19] Olivia Solon. Team of hackers take remote control of tesla model s from 12 miles away, 2016 (accessed March 12, 2020). https://www.theguardian.com/technology/2016/sep/20/tesla-model-s-chinese-hack-remote-control-brakes.

[20] Kazuaki Tanida. Dynamic time warping (dtw) algorithm with an o(n) time and memory complexity, 2019 (accessed March 12, 2020). https://pypi.org/project/fastdtw/.

[21] Berghout Tarek. Low pass filter based fft, 2019 (accessed April 22, 2020). https://www.mathworks.com/matlabcentral/fileexchange/71002-low-pass-filter-based-fft.

[22] Alexander Tolpygo. Dynamic time warping: Time series analysis ii, 2016 (accessed March 12, 2020). https://sflscientific.com/data-science-blog/2016/6/3/dynamic-time-warping-time-series-analysis-ii.

[23] Shijia Wei, Aydin Aysu, Michael Orshansky, Andreas Gerstlauer, and Mohit Tiwari. Using power-anomalies to counter evasive micro-architectural attacks in embedded systems. *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 111–120, 2019.

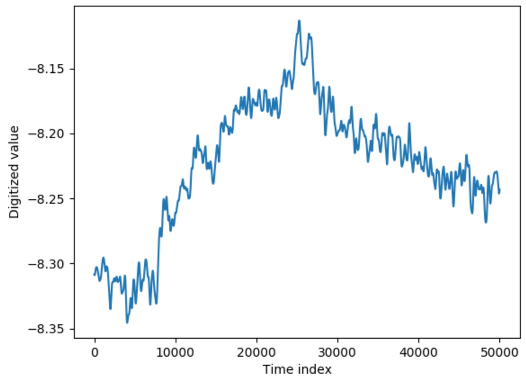
[24] Peter Wittek. Support vector machine, 2014 (accessed April 21, 2020). https://www.sciencedirect.com/topics/computer-science/support-vector-machine.

APPENDICES

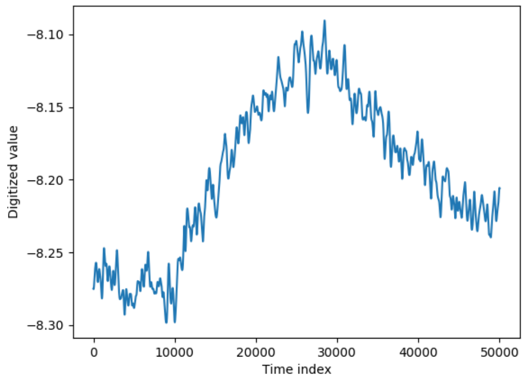
A Additional plots for Section 4.2

In this section, we have included additional plots for the experiments executed in Section 4.2. We have randomly chosen four samples of power traces for each command from Table 4.4 to show the reader the similarities between power traces for each command.

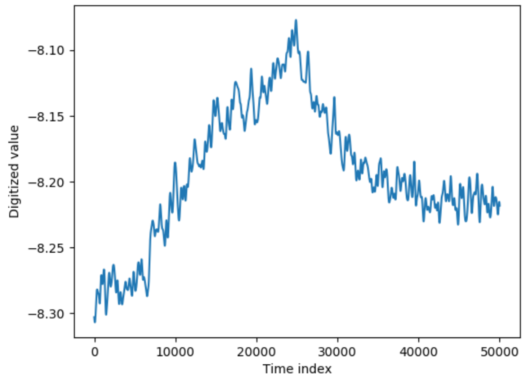
A.1 Read accelerometer value



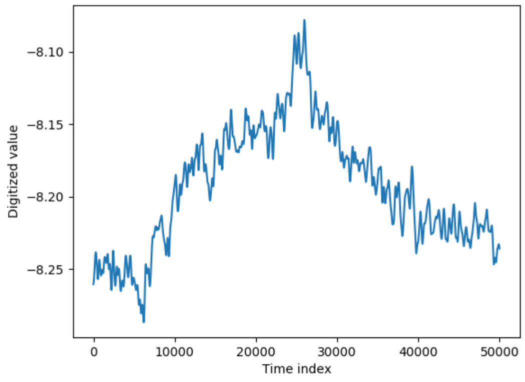
(a)



(b)



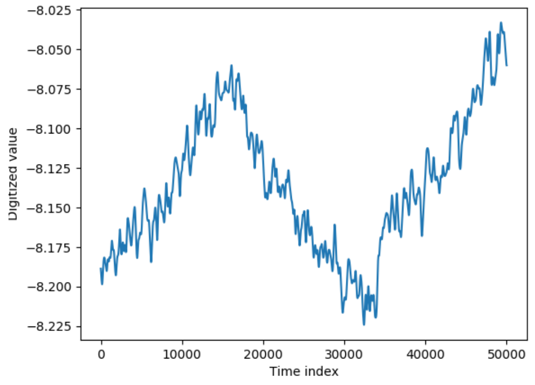
(c)



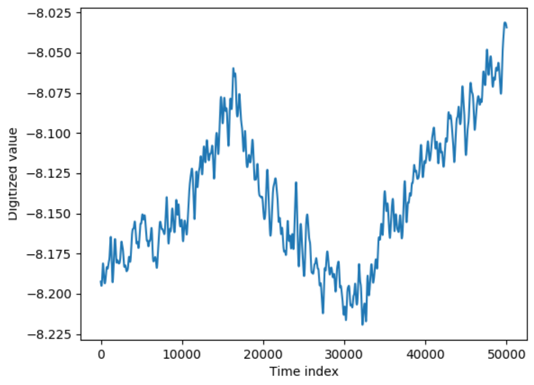
(d)

Figure 1: Plots from random samples of read accelerometer value command

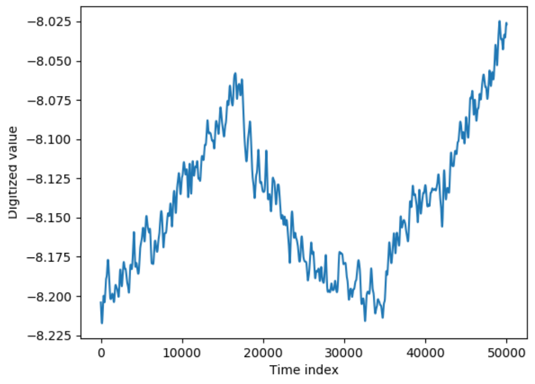
A.2 Read temperature value from sensor 1



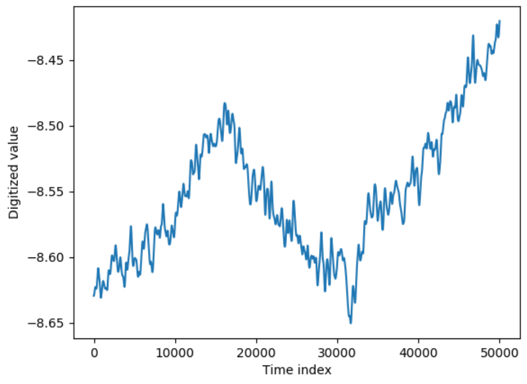
(a)



(b)



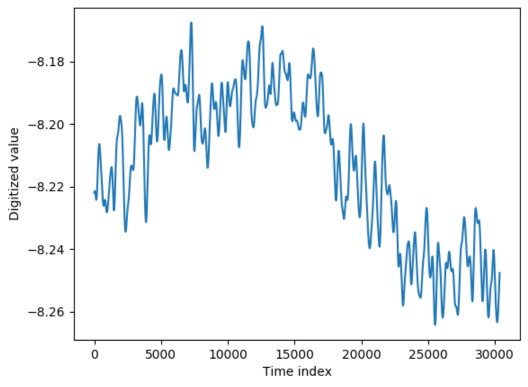
(c)



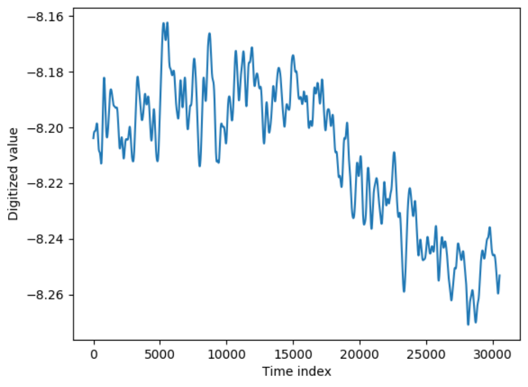
(d)

Figure 2: Plots from random samples of read temperature value from sensor 1 command

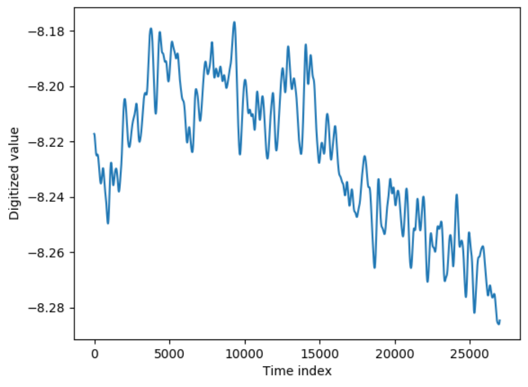
A.3 Read ambient temperature from infrared sensor



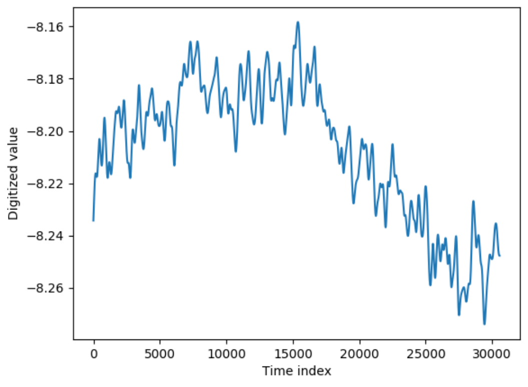
(a)



(b)



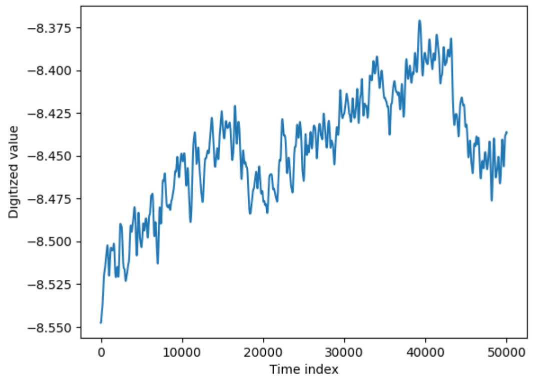
(c)



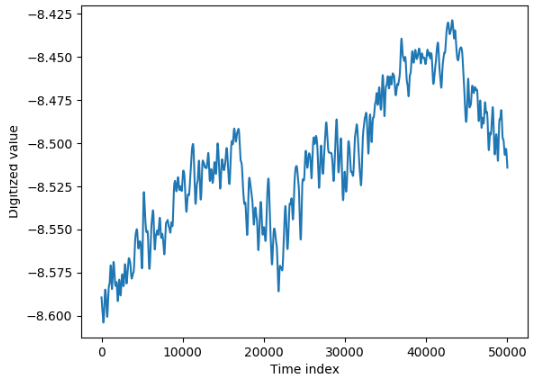
(d)

Figure 3: Plots from random samples of read ambient temperature from infrared sensor command

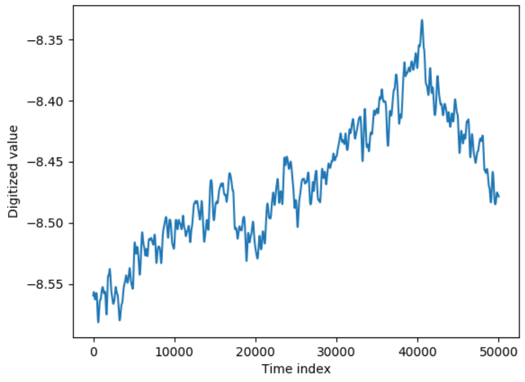
A.4 Read ambient light value



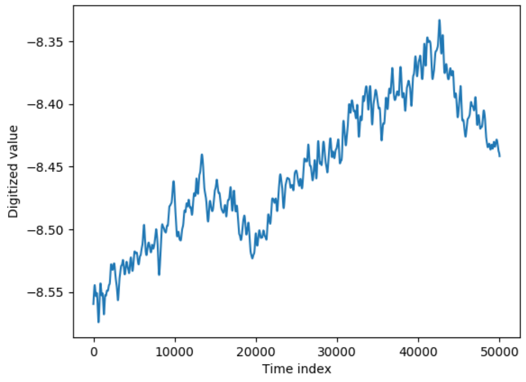
(a)



(b)



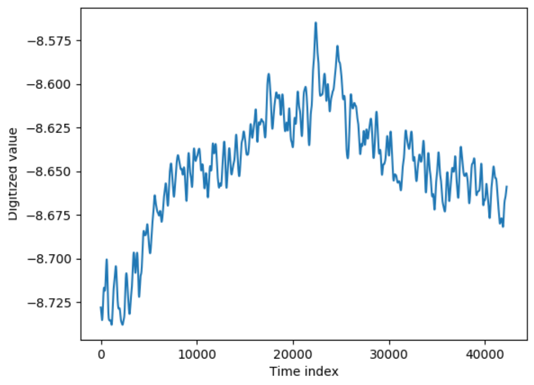
(c)



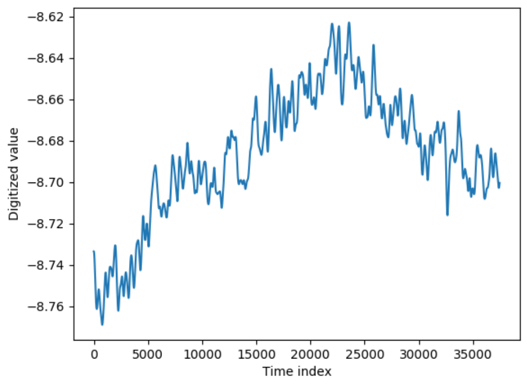
(d)

Figure 4: Plots from random samples of read ambient light value command

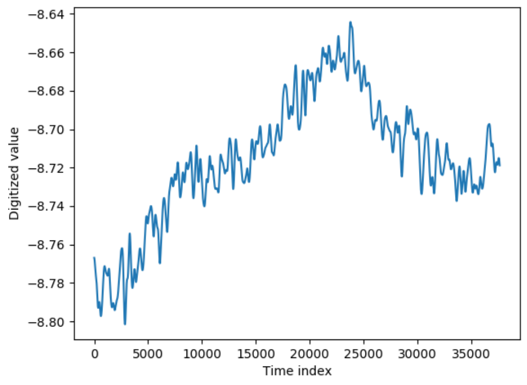
A.5 Read temperature value from sensor 2



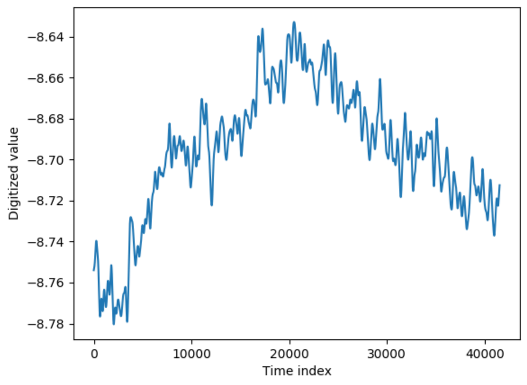
(a)



(b)



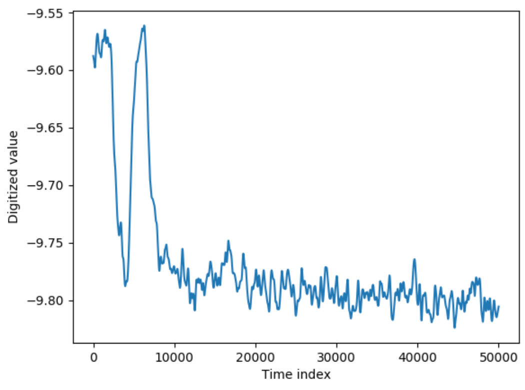
(c)



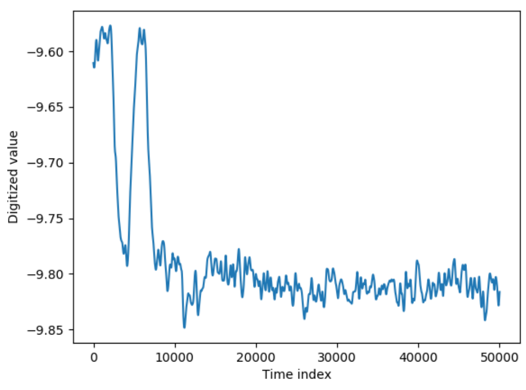
(d)

Figure 5: Plots from random samples of read temperature value from sensor 2 command

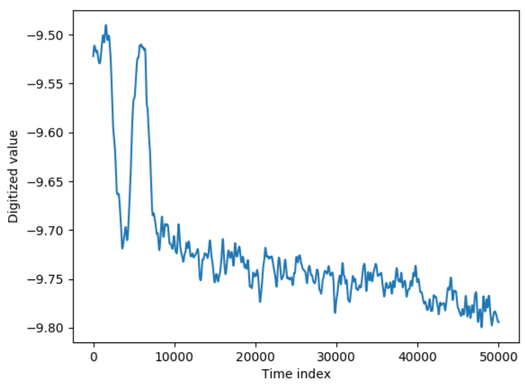
A.6 Read temperature from MCU sensor 1



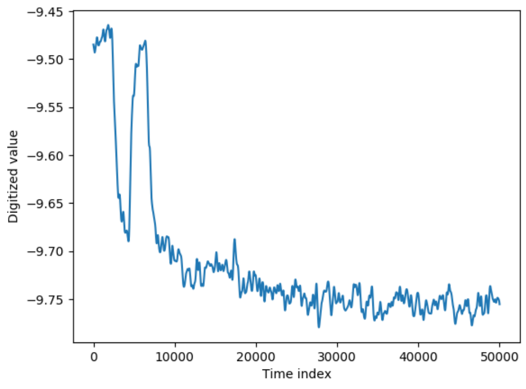
(a)



(b)



(c)



(d)

Figure 6: Plots from random samples of Read temperature from MCU sensor 1 command