IEC-61850 Protocol Analysis and Online Intrusion Detection System for SCADA Networks using Machine Learning

by

Shivam Patel
Bachelor of Engineering, Gujarat Technological University, 2014

A Report Submitted in Partial Fulfillment
of the Requirements for the Degree of

MASTERS OF ENGINEERING

in the Department of Electrical and Computer Engineering

Supervisory Committee

IEC-61850 Protocol Analysis and Online Intrusion Detection System for SCADA Networks using Machine Learning

by

Shivam Patel
Bachelor of Engineering, Gujarat Technological University, 2014

Supervisory Committee

Dr. Xiaodai Dong, Department of Electrical and Computer Engineering
Supervisor

Dr. Issa Traore, Department of Electrical and Computer Engineering
Departmental Member

Abstract

Supervisory Committee

Dr. Xiaodai Dong, Department of Electrical and Computer Engineering
Supervisor
Dr. Issa Traore, Department of Electrical and Computer Engineering
Departmental Member

Nowadays, industrial network security has become a major threat. In order to detect and prevent any type of attack on the industrial networks it is necessary to understand the communication protocols used by them. Hence, the first part of the report would review research done on IEC (International Electro Technical Commission) -61850 protocol employed in electric substation environment. In the second part of the project, an online intrusion detection system (OIDS) for SCADA networks which uses machine learning for detection is implemented. OIDS is a testbed which emulates a typical SCADA network and it consists of both attack and defense toolkits. SNORT is used for detecting the attack traffic based on the machine learning weights. The machine learning weights are obtained by training the collected traffic using the logistic regression algorithm.

Table of Contents

Supervisory Committee ii
Abstract iii
Table of Contents iv
List of Tables v
List of Figures vi
Acknowledgments viii
Chapter 1 Introduction 1
Chapter 2 IEC-61850 Protocol 7
Chapter 3 IEC-61850 Network Traffic Analysis 15
Chapter 4 SCADA Network and MODBUS protocol 30
Chapter 5 Online Intrusion Detection System 38
Chapter 6 Traffic Generation and Dataset processing 48
Chapter 7 Intrusion Detection using SNORT 62
Chapter 8 OIDS Configurations 70
Chapter 9 Future Work and Conclusion 73
Bibliography 75
Appendix A Basic Encoding Rules (BER) 77
Appendix B Modbus Function Codes 79
Appendix C Intrusion Detection Algorithm Implementation 80

List of Tables

Table 1: IEC-61850 standard document parts [5] 7
Table 2: Cases where Alarm is set to ‘1’ by Modbus reader 62
Table 3 : Defense Wall Configurations 75
Table 4: Description of the BER Identifier 81

List of Figures

Figure 1: IEC-61850 Substation Architecture [6] 9
Figure 2: Data Modelling approach [7] 10
Figure 3: Object name structure IEC-61850 [7] 11
Figure 4: IEC-61850 Communication Profiles [6] 12
Figure 5: Goose Frame Structure [8] 16
Figure 6: GOOSE Message frame captured in Wireshark [8] 17
Figure 7: GOOSE PDU fields [8] 18
Figure 8: MMS Protocol Stack [7] 20
Figure 9: MMS PDUs [10] 21
Figure 10: Initiate request PDU [10] 23
Figure 11: Confirmed Request PDU [10] 27
Figure 12: IEC-61850 pcap reader sample CSV output 29
Figure 13: SCADA Zones of Power Plant [11] 31
Figure 14: MODBUS Communication Stack [12] 34
Figure 15: Example MODBUS Architecture [12] 35
Figure 16: MODBUS Frame [12] 35
Figure 17: MODBUS Transaction (Error Free) [12] 37
Figure 18: MODBUS Transaction (Error) [12] 37
Figure 19: Online Intrusion Detection System Architecture (After [2]) 40
Figure 20: Tank System HMI [2] 41
Figure 21: Nova web console [2] 43
Figure 22: Nexpose Web Console [2] 45
Figure 23: Samurai Tool menu [2] 46
Figure 24: Wireshark 48
Figure 25: Starting Mod Slave server 50
Figure 26: Starting the HMI and honeyd 50
Figure 27: Wireshark capturing packets on Defense wall 51
Figure 28: pump_speed_reg.sh 52
Figure 29: tank_level_normal.sh 52
Figure 30: modify_threshold_normal.sh 53
Figure 31: pump_speed_conti.sh 53
Figure 32: pump_speed_attack.sh 54
Figure 33: tank_level_attack.sh 54
Figure 34: modify_threshold_attack.sh 54
Figure 35: dos.sh 55
Figure 36: Dataset processing in offline module 55
Figure 37: Function code 16 request packet 56
Figure 38: Address of the tank system [2] 57
Figure 39: Function code 16 response packet 57
Figure 40: Function code 3 request packet 58
Figure 41: Function code 3 response packet 58
Figure 42: Modbus reader output CSV file 62
Figure 43: Snort Workflow [14] 65

Figure 44: Snort Decoder configurations 66
Figure 45: Intrusion Detection Algorithm 67
Figure 46: Modbus preprocessor 70
Figure 47: Alert Generation 71
Figure 48: Enabling rule in preprocessor.rules 71
Figure 49: Generated Alerts 72
Figure 50: Virtual Network Settings [2] 74
Figure 51: MODBUS Function Codes [12] 83

Acknowledgments

I would like to thank:

Dr. Xiaodai Dong for trusting me and giving me this opportunity.

Our collaborators, Dr. Tao Lu and Yizhou Zhu (MASc student, University of Victoria) for helping me with this project.

Shivam Patel

Chapter 1 Introduction

1.1 Main Purpose

Today, mostly all the electric substations are managed with the help of substation automation system. IEC (International Electro Technical Commission) -61850 is a standard which uses comprehensive object oriented data model and Ethernet technology to allow various Intelligent Electronic Devices (IED) communicate with each other in an electric substation environment. Despite of many obvious advantages provided by IEC-61850 substations over the traditional substations, the power supplying companies are still being very careful about its implementation due to its security vulnerabilities. In fact, researchers in [1] have discovered vulnerabilities and weaknesses in IEC-61850 standard such as lack of encryption of GOOSE messages, no implementation of firewalls inside the IEC-61850 network and most importantly lack of implementation of an intrusion detection system in IEC-61850 network. Motivated by on-going collaborations with Fortinet Corp., a security company, in order to implement an intrusion detection system inside the IEC-61850 network it is necessary to conduct the following steps:

1. Research IEC-61850 standard.
2. Analyze and understand IEC-61850 traffic data.
3. Develop a Java code which reads IEC-61850 data [Pcap file] and extract the list of desired features in a CSV which can be used for machine learning.

Thus, the main purpose of the first part of this project is to form the basis for creating an online intrusion detection system inside IEC-61850 network.

Supervisory control and data acquisition (SCADA) is a software system used to automate and/or monitor industrial processes in various vertical markets: manufacturing, transportation, energy management, building automation, and any other field where real time operational data is used to make decisions [2]. Until early 2000's it was believed that SCADA networks were electronically isolated from rest of the networks and hence industries were stressing more on physical security of the network. In 2010 Stuxnet, a malicious computer worm attacked Iran's nuclear program. Stuxnet specifically targeted Iranian programmable logic controllers (PLC) and caused the fast spinning centrifuges to tear themselves apart. This was a major security incident which made people realize the urgent need to provide SCADA network security. In order to accomplish this a proper study of industrial network security has to be accomplished under real corporate environments. This requires logging all communication packets and this slows down the message transmission rate which is impractical for the corporate networks. To overcome this problem, Liao Zhang designed a software based testbed [2] which could simulate large scale SCADA networks. Now, this testbed simulates a network and can provide some basic intrusion detection using SNORT rules. There has been some research done on providing intrusion detection for SCADA networks via machine learning as in [3]. Also, Hongrui Wang in [4] provided intrusion detection for SCADA networks using machine learning algorithm in Bro (an open source intrusion detection tool). But, it does not involve actual implementation and combining the machine learning results with rule/signature based

detection to enhance the accuracy of the detection. Therefore, it was necessary to design an online intrusion detection system with the following features:

1. It can simulate a large scale SCADA network whose topology can be modified easily. This has been accomplished by Liao Zhang in [2].
2. Generate real time attack and normal SCADA network traffic.
3. Offline module: This module includes a Java code that can process Modbus traffic generated in the testbed and generate a dataset with desired features and label for machine learning. Finally, a code that uses a machine learning algorithm (logistic regression in our case) to train the dataset and generate weights for each feature [5].
4. A defense system that uses the machine learning weights obtained by offline training and provides online (i.e., real time) detection using SNORT.

1.2 IEC-61850 Protocol

Earlier substation automation systems were simple and they used straight forward, system specific communication protocols. Currently, computers and domain specific applications are used to enhance and optimize the management of more intelligent substation equipment. These equipments gave the advantage of multi-tasking operating systems, relational database systems and state-of-the-art graphical display technology. These devices used in the substation were manufactured by different companies and each of them would use their own substation automation protocol. Due to this, the equipments in the substation were unable to communicate with each other. This caused the need to

develop a unified international standard to support seamless co-operation between products from different vendors. The main objective set for the IEC standard were as follows:

1. Develop a single protocol for a complete substation considering the modelling of different types of data required by the substation.
2. Define basic services required to transfer data.
3. High inter-operability between systems from different vendors.
4. A standard format for storing the data.
5. Define a test procedure for the equipment which conforms to the standard.

Domain experts from 22 countries worked in three different IEC groups from 1995 to form IEC-61850 standard which tackles and responds to the above mentioned objectives. This standard reduces the configuration and maintenance cost to great extent by taking advantage of comprehensive object oriented data model and Ethernet technology. In order to make the standard less domain dependent, the committee stressed on data semantics. This resulted into carving out most of the communication details and made the standard quite difficult to understand.

1.3 Online Intrusion Detection System (OIDS) Features

As this system uses the testbed designed by Liao Zhang in [2], the below list of features is a combination of the testbed features and the additional features added as a part of this project:

1. Industrial devices, e.g. programmable logic controller (PLC) are simulated and they can use industrial protocols to communicate with each other [2].
2. OIDS can emulate a large scale SCADA network with the help of PLC.

3. Actual industrial process is demonstrated by the OIDS.
4. OIDS includes protocol oriented attack tool kits which can be used by researchers to carry out the attacks easily.
5. OIDS is easy to deploy and operate. Also, it is easy to install the OIDS on a personal computer with low cost.
6. The new OIDS detection system can gather traffic data which can be given to the offline training module of the OIDS.
7. Offline training module is capable of generating dataset with desired features and labels. It also trains the dataset with logistic regression algorithm to generate weights for each feature in the dataset.
8. OIDS detection system has the ability to detect network intrusion based on the machine learning weights of different traffic features.

1.4 Outline of Report

The outline of this report is organized as below:

Chapter 2 describes various details of IEC-61850 protocol like Data modelling, substation architecture, communication models used by the protocol. These details are necessary to understand how the communication takes place in a substation environment using IEC-61850 protocol.

Chapter 3 describes the packet structure for different types of IEC-61850 communication models. It also includes details on the IEC-61850 reader (Java code developed as a part of this project that parses pcap file containing Manufacturing message

specification (MMS) traffic. This code uses the parsing functions developed by Yizhou Zhu) that can be used to generate a CSV file with desired features from IEC-61850 pcap file.

Chapter 4 gives details on SCADA networks and MODBUS protocol.

Chapter 5 describes all the components of Online intrusion detection system.

Chapter 6 shows how to generate both attack and normal traffic using the OIDS. It also describes how to use MODBUS reader (Java code developed that parses pcap file containing MODBUS traffic. This code is an extension of Yizhou Zhu's base code written for parsing pcap file) for generating the data set i.e., CSV file that will include all the features and labels required for machine learning training.

Chapter 7 introduces SNORT and describes how SNORT is configured to provide detection based on machine learning weights.

Chapter 8 describes how to configure the Online Intrusion detection system, so that it can be used for detection.

Chapter 9 concludes the report and suggests future work.

Chapter 2 IEC-61850 Protocol

2.1 Overview

IEC (International Electro Technical Commission) defines the way of communication and information exchange between the devices in the electric substation environment. Substation Automation Systems generally manage the electric substation by using computers and domain specific applications. The IEC-61850 documentation is divided into 10 parts as shown in Table 1. The document starts by giving some basic idea about the standard from part 1 to part 3. Then part 4 gives the description of the project management requirements in the IEC-61850 enabled electric substation. The parameters required for physical implementation are specified in part 5. Part 6 gives the description of the XML based language for IED (Intelligent Electronic device) configuration. Part 7 gives idea of core concepts of the standard. Part 8 shows how to map the data objects to presentation and Ethernet link layer. Mapping Sampled Measured Values (SMV) to point-to-point Ethernet is shown in part 9.

Part	Title
1	Introduction and Overview
2	Glossary
3	General Requirements
4	System and Project Management
5	Communication Requirements for Functions and Device Models
6	Configuration Description Language for Communication in Electronic Substation related to IEDs
7	Basic communication structure for Substation and Feeder Equipment
8	Specific Communication service mapping (to MMS and to Ethernet)
9	Specific Communication service mapping (from Sampled Values)
10	Conformance Testing

Table 1: IEC-61850 standard document parts [6]

2.2 Intelligent Electronic Device (IED)

The utility communication standard in the past have assumed that the readers would have some domain specific knowledge. Consequently, the standards contained a lot of implicit domain information which is difficult to interpret for the outsiders for example software engineers. IEC-61850 standard falls in this same category, in order to understand the logical concepts of the standard it is necessary to have a basic idea of intelligent electronic devices (IED). IED is a microprocessor based controller of power system equipment, which is capable to send or receive data/control to and from an external source [6]. IED is essentially a computer which contains one or more microprocessor, memory and a collection of communication interfaces like Ethernet interfaces, serial ports and USB ports. However, in order to facilitate the domain specific processing, it may also contain some digital logics.

IEDs are classified based on their function. Some common types of IEDs are relay devices, voltage regulators, circuit breakers and so on. IED can also perform multiple functions with the help of its general purpose microprocessor. It is also possible to run some kind of operating system like Linux on IED.

2.3 Substation Architecture

A typical IEC-61850 substation architecture is shown in Figure 1. As shown in figure the substation network is connected to the outside network through a secure gateway. The remote operators and control centers outside uses Abstract Communication Service

Interface (ACSI) defined in part 7 of the standard documentation in order to query and control the devices inside the substation.

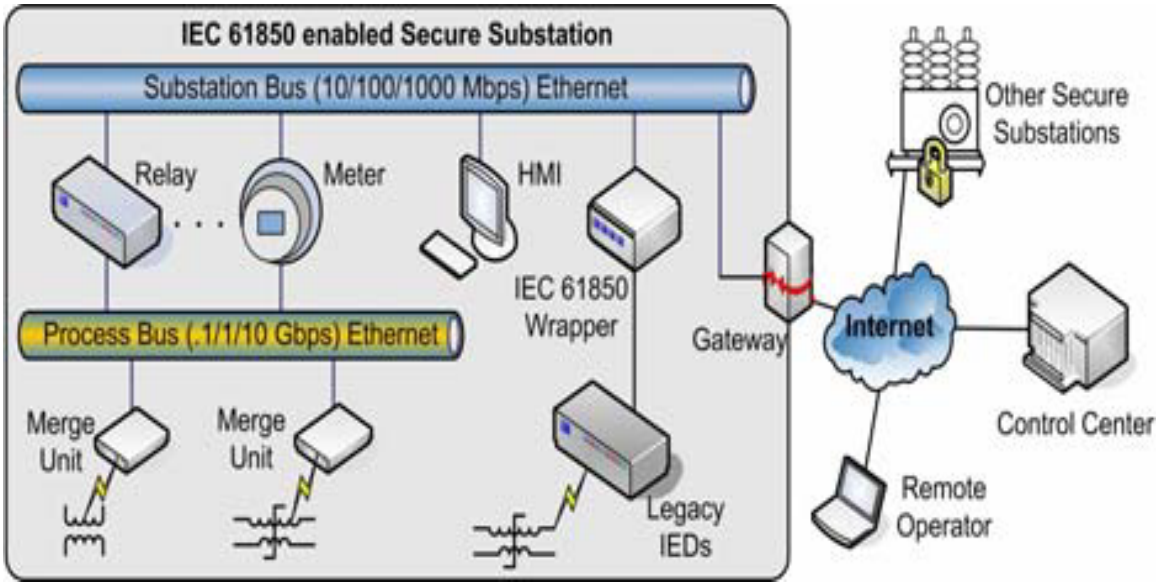


Figure 1: IEC-61850 Substation Architecture [6]

There are two kinds of communication bus in the substation which connects all the IEDs:

Substation Bus: It carries all the requests/responses and generic event substation messages (refer to 2.5 Communication Profiles for details). There is generally only one global substation bus. It is realized by a medium bandwidth Ethernet network.

Process Bus: It connects the IEDs to traditional devices like merge units as shown in Figure 1. There can be more than one process bus inside the substation and it is realized by high bandwidth Ethernet networks.

The three main kind of data active in IEC-61850 substation network are: ACSI requests/responses, Generic Substation Event (GSE) messages and sampled analog values. As our main focus is providing intrusion detection and as the process bus is not directly

linked with the gateway, it is not possible to attack the data carried on the process bus. Hence, the communication on the process bus i.e., sampled analog values are not considered in this research.

2.4 Data Modelling

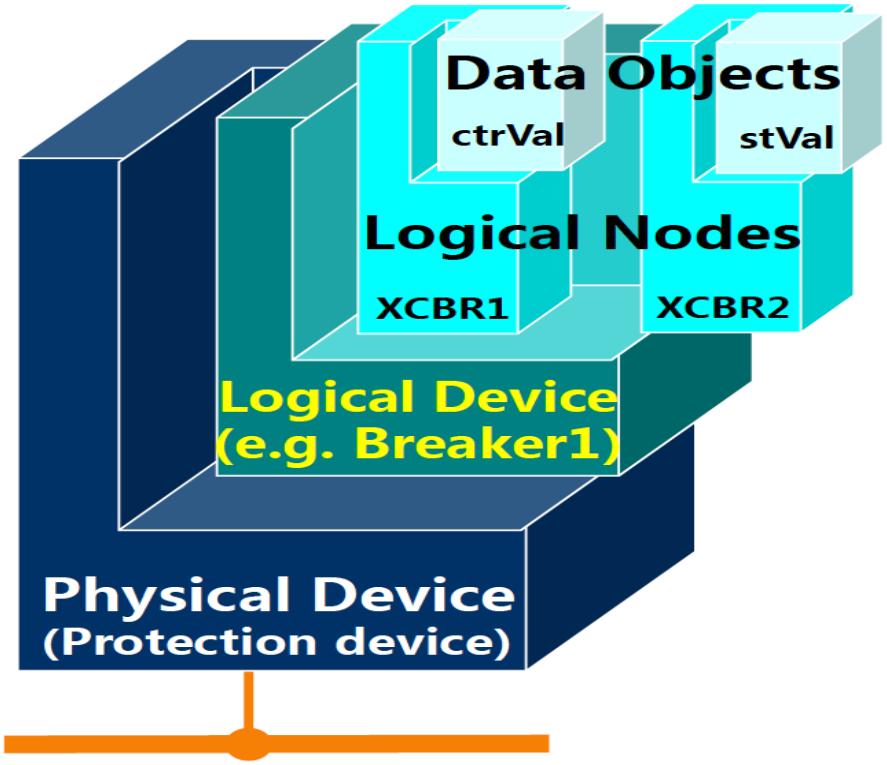


Figure 2: Data Modelling approach [7]

Physical device: IEC device modelling begins with a physical device. A physical device is the device that can connect to the network and can be accessed through the network address. Each physical device can contain one or more logical device. Hence, IEC allows the physical device to act as gateway for multiple devices.

Logical device: Logical device is a collection of Logical nodes. Example of a Logical device is a Breaker.

Logical node: Logical node is the named grouping of data and associated function or services. Example of Logical node is **XCBR: Circuit Breaker**. Each Logical node contains one or more elements of data and each element has a unique name.

Example Data Model:

- **Physical Device:** IED (Intelligent electronic device which is a microprocessor based controllers of the power system equipment which is able to send or receive data/ control to and from the external source)
- **Logical device:** Breaker / Relay
- **Logical Node:** XCBR (Circuit breaker)
- **Data:**

  Loc: determines if the operation is local or remote.

  OpCnt: Operation count

  Pos: Position

OBJECT NAME STRUCTURE:

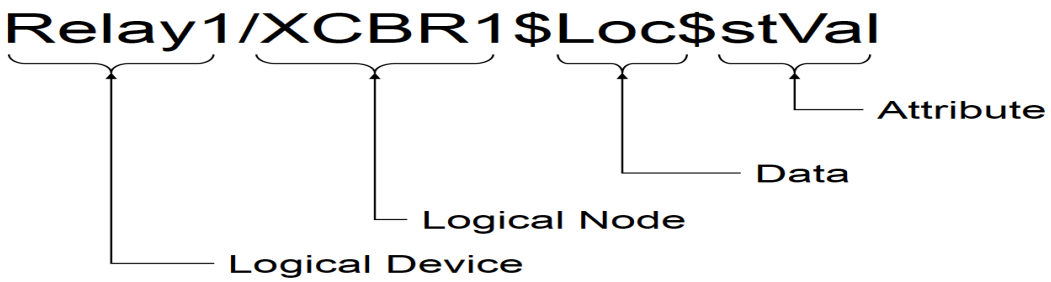


Figure 3: Object name structure IEC-61850 [7]

2.5 Communication Profiles

The IEC-61850 substation interactions can be grouped into following three categories: data monitoring/reporting, data gathering/setting, and event logging. In the IEC-61850 standard all the inquiries and control activities are modelled as getting or setting the values of corresponding data attributes. On the other hand, data monitoring/reporting interactions provides an efficient way to track the system status. Hence the first two types of interactions are very important in the IEC-61850 standard.

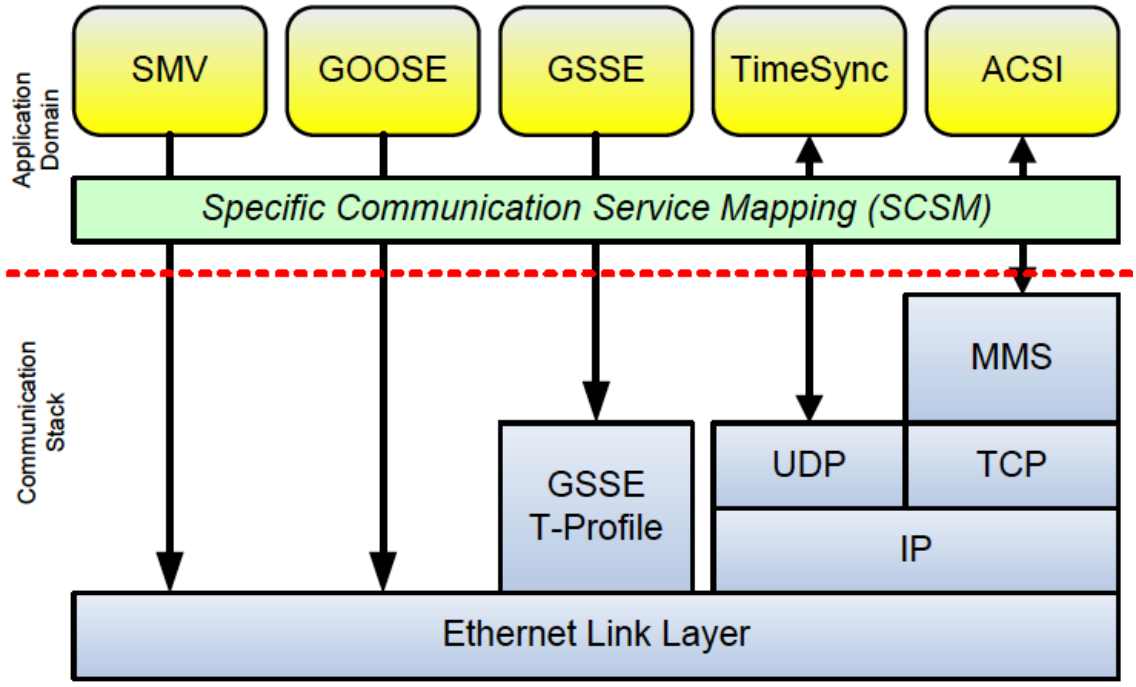


Figure 4: IEC-61850 Communication Profiles [6]

In order to realize the above mentioned interactions IEC-61850 standard has defined a fairly complicated communication structure as shown in Figure4. The standard has defined five types of communication profiles as shown in Figure 4 out of which this report is only focussing on ACSI and GOOSE (Generic Object Oriented Substation Event). As

mentioned earlier, the main reason for choosing these two profiles is that the messages for both the profiles are carried out on the global substation bus which is connected to the internet via gateway. Hence, ACSI and GOOSE are more vulnerable to security attacks. The following section will give some description on ACSI and GSE: GOOSE and GSSE (Generic Substation State Events) communication profiles.

2.5.1 Abstract Communication Services Interface (ACSI)

This is a primary interface for IEC-61850 standard as it is used for communication between applications in the substation and the servers. An object oriented approach is adopted in designing ACSI. It consists of three basic components:

- A set of objects
- Set of services for manipulating and accessing those objects.
- A base set of data types for describing objects.

This model is just high level description of substation automation and it has to be mapped over real set of protocol that are practical to implement and that can operate over the computing environment generally found in the power industry. IEC-61850 maps ACSI to MMS (Manufacturing message specification). The reason for choosing MMS is that it is the only public standard which can implement the complex naming and service models of IEC-61850. Hence, each IEC-61850 object is mapped to MMS object. Each IEC-61850 service is mapped to MMS service/operation.

2.5.2 Generic Substation Events (GSE)

The main goal of GSE is to provide fast and reliable distribution of system wide input and output values. It is based on autonomous decentralization. This allows simultaneous delivery of the same generic substation event information to one or more physical devices by using multicast/ broadcast services.

GSSE (GENERIC SUBSTATION STATE EVENTS)

It is only used to exchange the status data using a status list (string of bits) instead of data set (GOOSE). Its format is simpler than GOOSE and hence it is handled faster in some devices.

GOOSE (GENERIC OBJECT ORIENTED SUBSTATION EVENTS)

It is used to transfer wide range of possible common data organized by DATA SET (status, value). GOOSE data is directly embedded into the Ethernet data packets.

Chapter 3 IEC-61850 Network Traffic Analysis

3.1 GOOSE Packets

GOOSE data is directly embedded into the Ethernet data packets. The GOOSE frame structure is as shown in Figure 5. GOOSE message frame can be divided into following three parts:

(1) Header MAC

(2) Priority Tagged

(3) Ethernet PDU (contains GOOSE PDU)

Header MAC

This is the first portion of the frame and it consist of the destination and source MAC address. The addresses are multicast and defined as 01-0C-CD-xx-xx-xx. Both the addresses are 6-byte long.

Priority Tagged

- **TPID:** Tag protocol identifier (2 bytes), TPID indicates Ether type assigned for 802.1Q Ethernet encoded frames and is given by 0x8100
- **TCI:** Tag control Information (2 bytes), It consist of Canonical Frame (1 byte) Indicator and VLAN identifier (if VLAN is not used this byte would be set to 0).

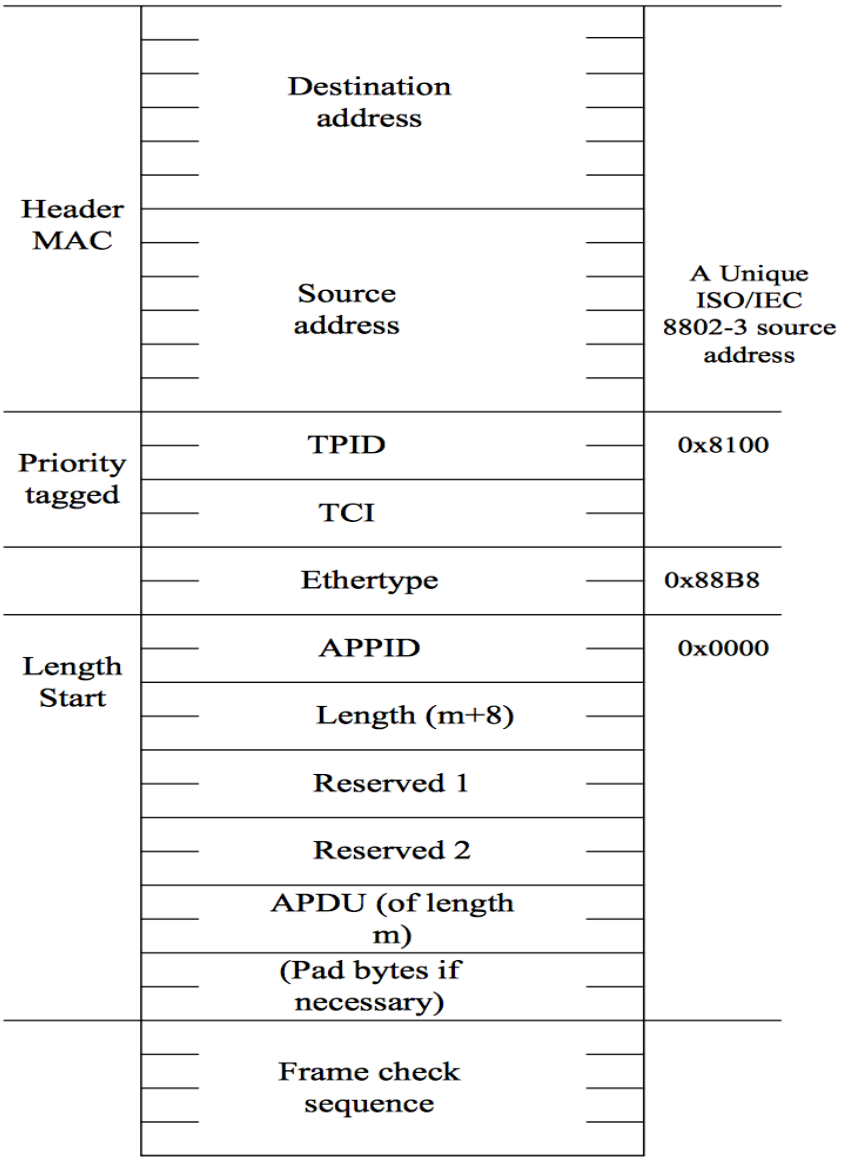


Figure 5: Goose Frame Structure [8]

Ethernet PDU

- **Ethernet type:** It is of 2 bytes and it indicates 'GOOSE' type 0x88B8.
- **APPID:** It is of 2 bytes and it is used to select the GOOSE messages from the frame. The MSB indicates the APPID type and the LSB indicates the actual ID.

- **Length:** It is the number of octet starting from the APPID and also includes Application Protocol Data Unit (APDU). Hence the value of Length is m + 8, where m is the length of APDU.

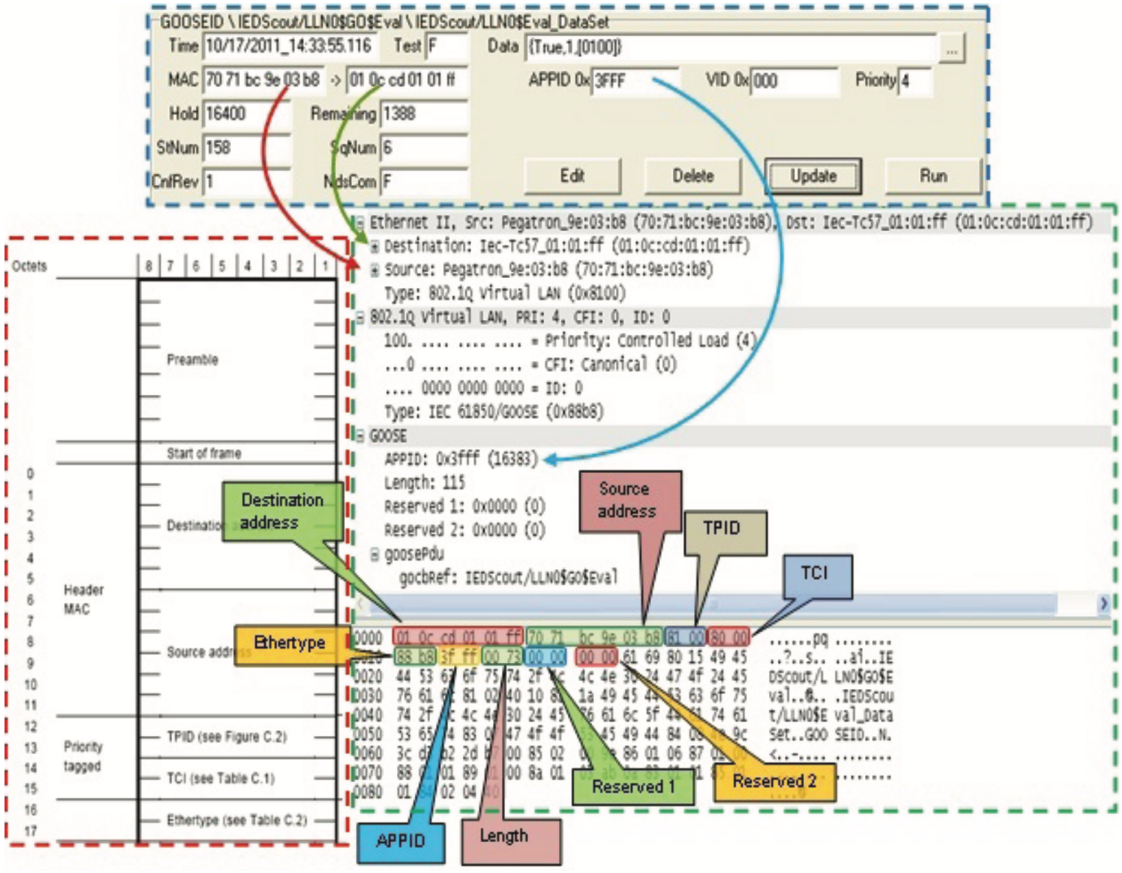


Figure 6: GOOSE Message frame captured in Wireshark [8]

Figure 6 above shows the GOOSE message frame captured in Wireshark. The figure indicates all three parts of the GOOSE message frame Header MAC, Priority Tagged and Ethernet PDU.

GOOSE PDU:

In GOOSE PDU all the elements occur in the following order: **TAG, LENGTH and followed by DATA** as shown in Figure 7. The data within the GOOSE PDU is encoded using the Abstract Syntax Notation ONE / Basic Encoding Rules (ASN.1/BER). TAG indicates the type of information represented by the data followed. LENGTH indicates the number of bytes of data.

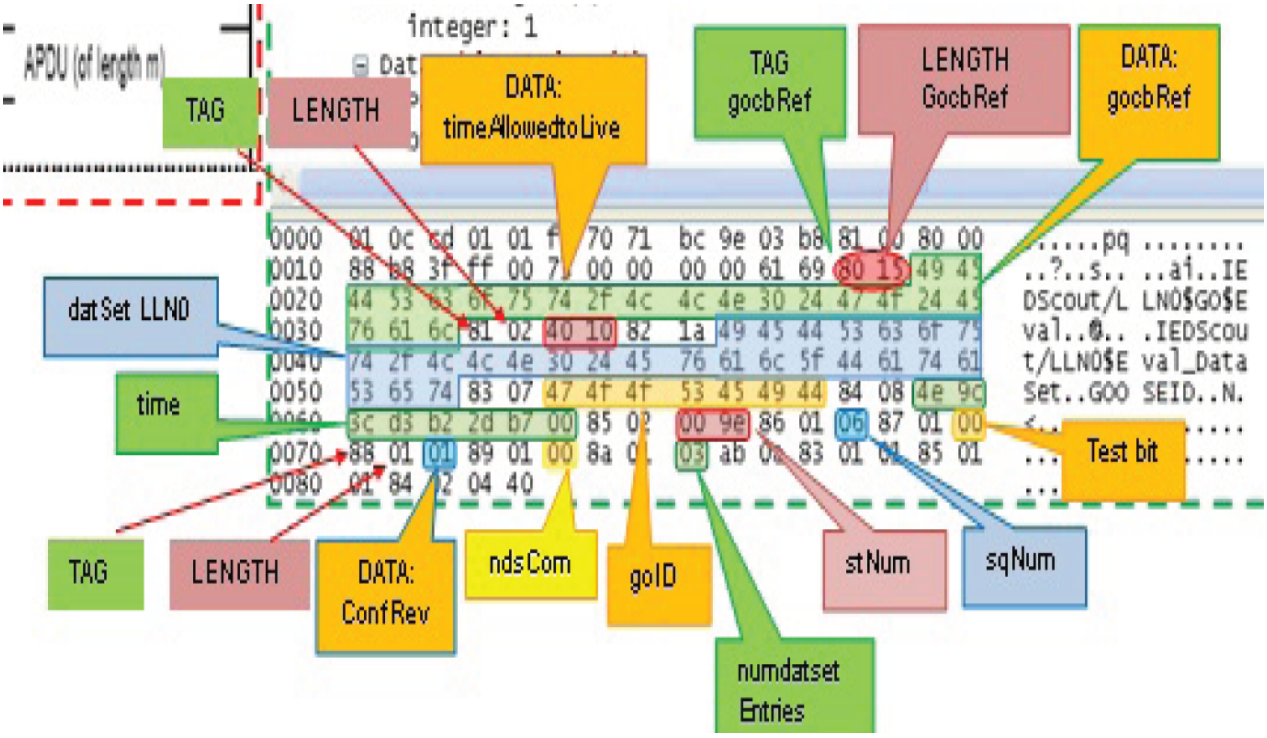


Figure 7: GOOSE PDU fields [8]

- **gocbRef:** It gives the name of GOOSE control block (27 bytes).
- **timeAllowedtoLive:** Indicates the maximum time the packet remains alive after being transmitted (2 bytes).
- **dataSet:** It references to the DATA-SET whose member's values are transmitted. The members of the data set are uniquely numbered starting from 1 (32 bytes).
- **goID:** It indicates the GOOSE ID and it is of 7 bytes.
- **timestamp:** Time stamp for each GOOSE message (8 bytes).
- **stNum:** State number is assigned whenever the GOOSE message is generated as a result of event change (2 bytes).
- **sqNum:** This number is assigned to the re-transmitted messages in increasing order (1 byte).
- **Test:** This bit is set if in the test mode.
- **ConfRev:** Indicates the version of Intelligent electronics device(IED).
- **ndsCom:** This is set when the data in the GOOSE message is invalid.
- **numdatasetEntries:** Indicates the number of data present in the received GOOSE message (1 byte).
- **alldata:** Actual data of all the members in the data set.

3.2 Manufacturing Message Specification (MMS)

As mentioned in 2.5.1 Abstract Communication Services Interface (ACSI), IEC-61850 maps ACSI to MMS for practical implementation. MMS uses ASN.1 as abstract syntax notation at the presentation layer. Abstract notation is used for defining data structure or set of values for messages and applications [9]. MMS encodes ASN.1 data using BER (refer APPENDIX A Basic Encoding Rules for details). The protocol stack for MMS is shown in Figure 8.

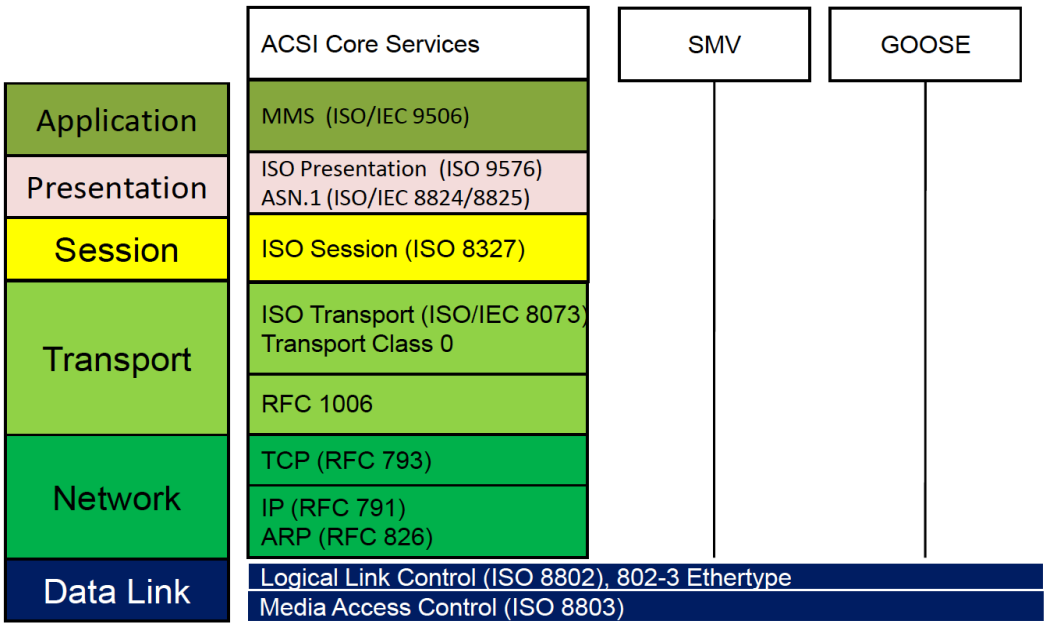


Figure 8: MMS Protocol Stack [7]

3.2.1 Decoding MMS PDUs

MMS model has following 14 types of PDU:

```
MMSpdu ::= CHOICE {
  confirmed-RequestPDU    [0] IMPLICIT Confirmed-RequestPDU,
  confirmed-ResponsePDU   [1] IMPLICIT Confirmed-ResponsePDU,
  confirmed-ErrorPDU      [2] IMPLICIT Confirmed-ErrorPDU,
  unconfirmed-PDU         [3] IMPLICIT Unconfirmed-PDU,
  rejectPDU               [4] IMPLICIT RejectPDU,
  cancel-RequestPDU       [5] IMPLICIT Cancel-RequestPDU,
  cancel-ResponsePDU      [6] IMPLICIT Cancel-ResponsePDU,
  cancel-ErrorPDU         [7] IMPLICIT Cancel-ErrorPDU,
  initiate-RequestPDU     [8] IMPLICIT Initiate-RequestPDU,
  initiate-ResponsePDU    [9] IMPLICIT Initiate-ResponsePDU,
  initiate-ErrorPDU      [10] IMPLICIT Initiate-ErrorPDU,
  conclude-RequestPDU    [11] IMPLICIT Conclude-RequestPDU,
  conclude-ResponsePDU   [12] IMPLICIT Conclude-ResponsePDU,
  conclude-ErrorPDU      [13] IMPLICIT Conclude-ErrorPDU}
```

Figure 9: MMS PDUs [10]

The traffic received from Fortinet has following four PDUs:

- Initiate Request PDU
- Confirmed Request PDU
- Conclude Request PDU
- Confirmed Response PDU

Conclude request and Confirmed response PDUs received in the traffic (pcap) file are incomplete i.e., they do not have sufficient data as defined in [10]. The following section

shows how to decode Initiate request and Confirmed request PDU. Same process can be followed to decode the rest of the PDUs.

Initiate Request PDU:

Initiate request received from the pcap file looks as follows:

a8 27 80 02 75 30 81 02 03 e8 82 02 03 e8 83 01 05 a4 16 80 01 02 81 03

05 fb 00 82 0c 03 ff ff ff ff ff ff ff ff ff 18

The package above is encoded using BER's TLV (Tag, Length, Value) format similar to GOOSE as shown in Figure 7. The detailed description of decoding the Tag value is described in APPENDIX A Basic Encoding Rules. Now let's go through the decoding process for this PDU and determine each field. For the sake of understanding the Tag, length and Value fields are indicated as follow:

Tag: Black

Length: Red

Value/Data: Blue

(1) **a8** 27 80 02 75 30 81 02 03 e8 82 02 03 e8 83 01 05 a4 16 80 01 02 81 03

05 fb 00 82 0c 03 ff ff ff ff ff ff ff ff ff 18

Tag: a8 (a: indicates Content specific constructed tag,

8: indicates "Initiate request PDU" (Refer Figure 9))

Length: 27

```
Initiate-RequestPDU ::= SEQUENCE {
  localDetailCalling                  [0] IMPLICIT Integer32 OPTIONAL,
  proposedMaxServOutstandingCalling   [1] IMPLICIT Integer16,
  proposedMaxServOutstandingCalled    [2] IMPLICIT Integer16,
  proposedDataStructureNestingLevel   [3] IMPLICIT Integer8 OPTIONAL,
  initRequestDetail
    [4] IMPLICIT SEQUENCE {proposedVersionNumber
                             [0] IMPLICIT Integer16,
                           proposedParameterCBB
                             [1] IMPLICIT ParameterSupportOptions,
                           servicesSupportedCalling
                             [2] IMPLICIT ServiceSupportOptions,
                           ...,
                           additionalSupportedCalling
                             [3] IMPLICIT AdditionalSupportOptions,
                           additionalCbbSupportedCalling
                             [4] IMPLICIT AdditionalCBBOptions,
                           privilegeClassIdentityCalling
                             [5] IMPLICIT VisibleString}}
```

Figure 10: Initiate request PDU [10]

(2) a8 27 **80** 02 75 30 81 02 03 e8 82 02 03 e8 83 01 05 a4 16 80 01 02 81 03

05 fb 00 82 0c 03 ff ff ff ff ff ff ff ff ff 18

Tag: 80 (8: indicates Content specific primitive, 0: indicates field

"localDetailCalling" (Refer Figure 10))

Length: 02,

Data: 0x7530 (i.e., 30000)

(3) a8 27 80 02 75 30 **81** 02 03 e8 82 02 03 e8 83 01 05 a4 16 80 01 02 81 03

05 fb 00 82 0c 03 ff ff ff ff ff ff ff ff ff 18

Tag: 81 (8: indicates Content specific primitive tag,

1: indicates field “proposedMaxServOutstandingCalling”)

Length: 02

Data: 03 e8 (1000)

(4) a8 27 80 02 75 30 81 02 03 e8 **82** 02 03 e8 83 01 05 a4 16 80 01 02 81 03

05 fb 00 82 0c 03 ff ff ff ff ff ff ff ff ff ff 18

Tag: 81 (8: indicates Content specific primitive tag,

2: indicates field “proposedMaxServOutstandingCalled”)

Length: 02

Data: 03 e8 (1000)

(5) a8 27 80 02 75 30 81 02 03 e8 82 02 03 e8 **83** 01 05 a4 16 80 01 02 81 03

05 fb 00 82 0c 03 ff ff ff ff ff ff ff ff ff ff 18

Tag: 81 (8: indicates Content specific primitive tag,

3: indicates field “proposedDataStructureNestingLevel”)

Length: 01

Data: 05 (5)

(6) a8 27 80 02 75 30 81 02 03 e8 82 02 03 e8 83 01 05 **a4** 16 80 01 02 81 03

05 fb 00 82 0c 03 ff ff ff ff ff ff ff ff ff ff 18

Tag: a4 (a: indicates Content specific constructed tag,

4: indicates field “initRequestDetail”)

Length: 16

(7) a8 27 80 02 75 30 81 02 03 e8 82 02 03 e8 83 01 05 a4 16 **80** 01 02 81 03

05 fb 00 82 0c 03 ff ff ff ff ff ff ff ff ff ff 18

Tag: 80 (8: indicates Content specific primitive tag,

0: indicates field “proposedVersionNumber”)

Length: 01

Data: 02 (2)

(8) a8 27 80 02 75 30 81 02 03 e8 82 02 03 e8 83 01 05 a4 16 80 01 02 **81** 03

05 fb 00 82 0c 03 ff ff ff ff ff ff ff ff ff ff 18

Tag: 80 (8: indicates Content specific primitive tag,

1: indicates field “proposedParameterCBB”)

Length: 03

Data: 05 fb 00

(05: padding,

fb00: indicates values of CBB parameters

1... .... = str1: True

.1.. .... = str2: True

..1. .... = vnam: True

...0 .... = valt: False

.... 1... = vadr: True

.... .0.. = vsca: False

.... ..0. = tpy: False

....0 = vlid: False

0... .... = real: False

..0. .... = cei: False)

(9) a8 27 80 02 75 30 81 02 03 e8 82 02 03 e8 83 01 05 a4 16 80 01 02 81 03

05 fb 00 **82** 0c 03 ff ff ff ff ff ff ff ff ff ff 18

Tag: 82 (8: indicates Content specific primitive tag,

0: indicates field “Service Support Options”)

Length: 0C (12), Padding: 03

Data: ffffffffffffffffffff18 (Refer to [9] for detailed list of Service Support Options)

Confirmed request PDU:

Received traffic:

a0 14 02 01 00 a5 0f a0 08 30 06 a0 04 80 02 6d 75 a0 03 83 01 00

```
Confirmed-RequestPDU ::= SEQUENCE {
   invokeID          Unsigned32,
   listOfModifiers   SEQUENCE OF Modifier OPTIONAL,
   service           ConfirmedServiceRequest,
   ...,
   service-ext       [79]  Request-Detail OPTIONAL
   -- shall not be transmitted if value is the value
   -- of a tagged type derived from NULL--}
```

Figure 11: Confirmed Request PDU [10]

(1) **a0** 14 02 01 00 a5 0f a0 08 30 06 a0 04 80 02 6d 75 a0 03 83 01 00

Tag: a4 (a: indicates Content specific constructed tag,

0: indicates "Confirmed request PDU" refer Figure 9)

Length: 14

(2) a0 14 **02** 01 00 a5 0f a0 08 30 06 a0 04 80 02 6d 75 a0 03 83 01 00

Tag: 02 (indicates Universal primitive tag)

Length: 01

Data: 00 (indicates “invokeID = 0”)

(3) a0 14 02 01 00 **a5** 0f a0 08 30 06 a0 04 80 02 6d 75 a0 03 83 01 00

Tag: a5 (a: indicates Content specific constructed tag,

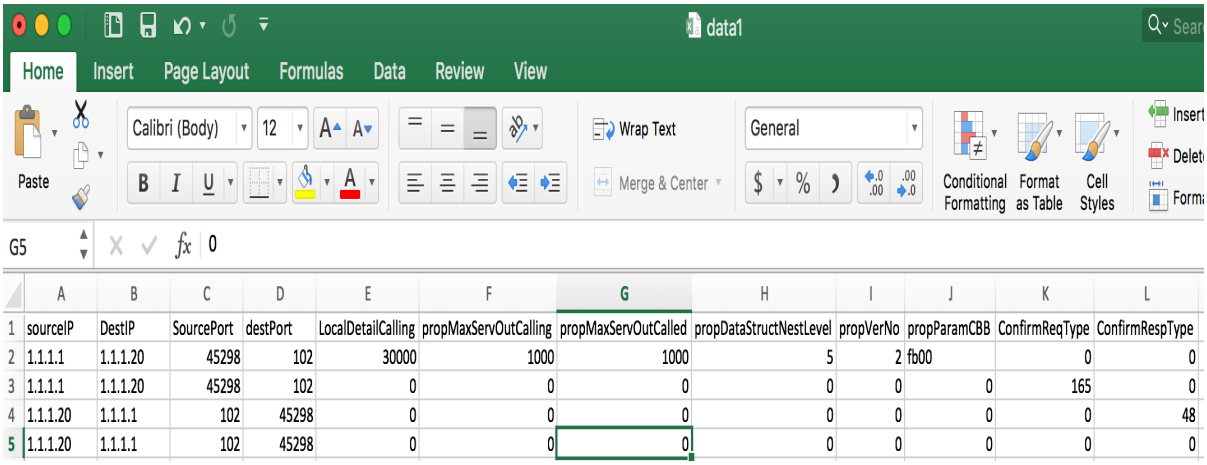
5: indicates Write service request, refer [9] for list of Confirmed request services)

Length: 0f

3.3 IEC-61850 Pcap Reader

IEC-61850 pcap reader is a java code which parses a pcap traffic file containing MMS traffic and extracts required fields and writes them to a CSV file. The code starts by parsing the Ethernet header, IPv4 header, TCP header and then follows the IEC-61850 protocol stack and parses corresponding headers until it reaches the MMS PDU. Then the type of MMS PDU is determined and PDU is decoded as shown in Section 3.2.1 Decoding MMS PDUs. At present, the application can parse/decode all four PDUs received from traffic given by Fortinet which are initiate request, confirmed request PDUs, confirmed response and conclude request PDU. The application can be easily extended to parse/decode the rest of the PDU’s. Figure 12 shows sample output of the application, the output is a CSV file (opened using Excel for better display) which contains several fields from IPv4 header, TCP header, initiate request PDU (LocalDetailCalling, propMaxServOutCalling and so on), confirmed request PDU (ConfirmReqType) and confirmed response PDU

(ConfirmRespType).



	A	B	C	D	E	F	G	H	I	J	K	L
1	sourceIP	DestIP	SourcePort	destPort	LocalDetailCalling	propMaxServOutCalling	propMaxServOutCalled	propDataStructNestLevel	propVerNo	propParamCBB	ConfirmReqType	ConfirmRespType
2	1.1.1.1	1.1.1.20	45298	102	30000	1000	1000	5	2	fb00	0	0
3	1.1.1.1	1.1.1.20	45298	102	0	0	0	0	0	0	165	0
4	1.1.1.20	1.1.1.1	102	45298	0	0	0	0	0	0	0	48
5	1.1.1.20	1.1.1.1	102	45298	0	0	0	0	0	0	0	0

Figure 12: IEC-61850 pcap reader sample CSV output

Chapter 4 SCADA Networks and MODBUS Protocol

4.1 SCADA Networks

Supervisory control and data acquisition (SCADA) is a software system used to automate and/or monitor industrial processes in various vertical markets: manufacturing, transportation, energy management, building automation, and any other field where real time operational data is used to make decisions [2]. SCADA systems integrate the data transmission with data acquisition systems and use Human Machine Interface (HMI) to allow centralized monitoring and controlling of various processes. SCADA networks are very different from conventional networks (e.g. campus or enterprise networks). The SCADA networks generally consist of Programmable Logic Controllers (PLC), Intelligent Electronic Device (IED), HMI, Remote Terminal Unit (RTU) and so on. On the other hand, the conventional networks consist of routers, switches and stations. Second difference is the protocol used by the networks, conventional network uses HTTP, FTP, while SCADA networks uses industrial protocols like MODBUS. The last difference is the network topology, SCADA networks are highly sophisticated and organized because they have a specific process and the protocols are related to that process. Also, this makes SCADA network more predictable compared to the traditional networks. Hence, it becomes easier for the hacker to predict the behaviour of SCADA network if he knows what process is executed by the network. It is necessary to identify these differences in order to provide network security for SCADA networks.

SCADA network in a power plant usually contains five zones as follow:

1. Internet Zone
2. Datacenter Zone
3. Plant Network Zone
4. Control Network Zone
5. Field I/O Zone

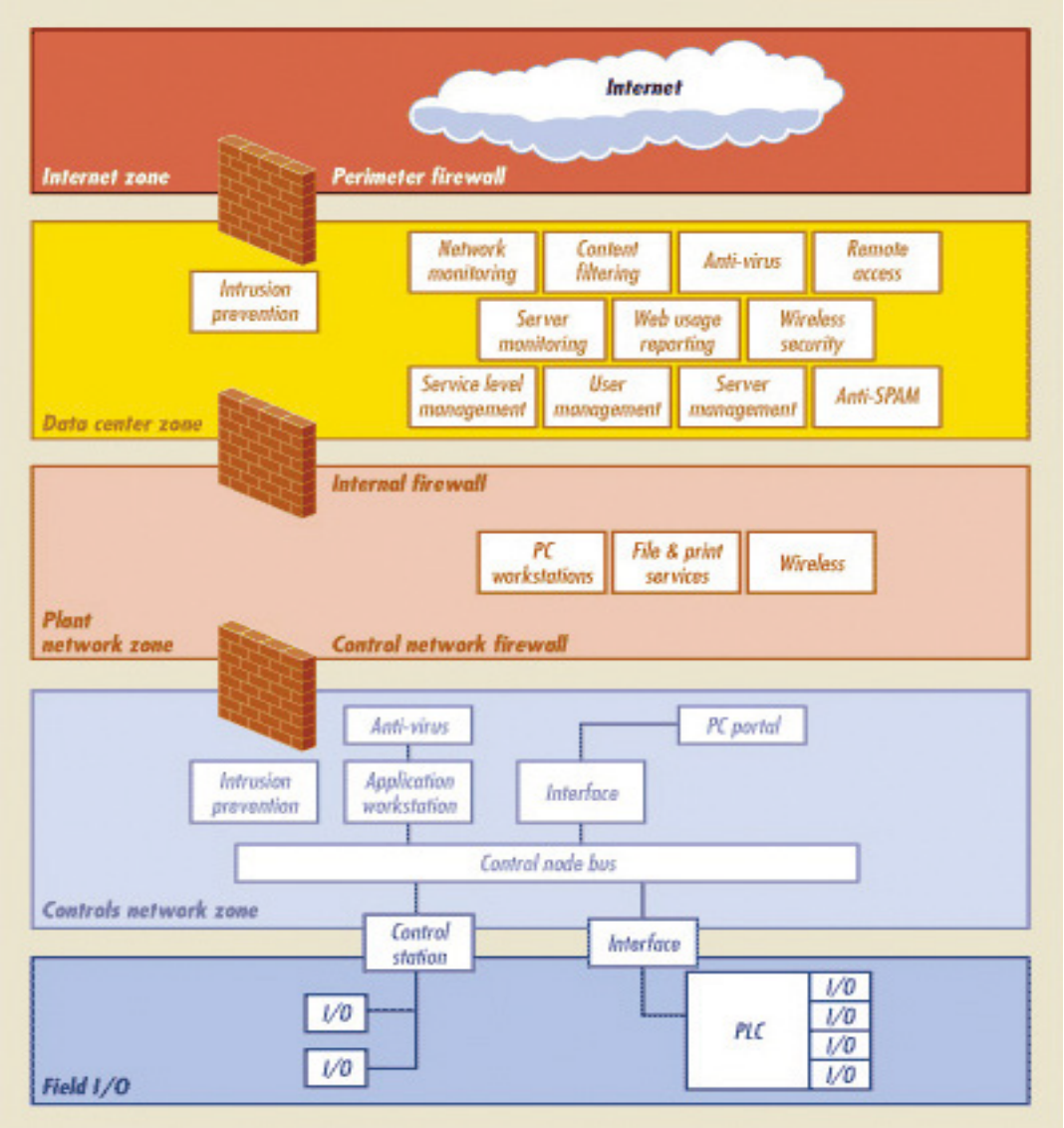


Figure 13: SCADA Zones of Power Plant [11]

Internet Zone: This zone is used for transmission of data/information between the vendors. As internet has many protocol vulnerabilities, this zone is the primary target in most cases. Some of the attacks are Denial of Service, packet sniffing (unencrypted data can be sniffed using Wireshark and many other open source tools) and so on.

Datacenter Zone: This zone collects the production related data from control zone and plant network zone. This zone is mostly running TCP/IP protocols and it contains typical management information systems and enterprise resource planning systems.

Plant Network Zone: This zone connects the control and field I/O zones as they lie in a single substation. If a hacker gains the control of this zone, it is possible for him to cause a blackout by taking the substation offline.

Control Network Zone: The data generated in the field I/O zone is transmitted to this zone. This zone performs the supervisory control of the plant i.e., an operator uses the data to monitor and control the plant production from this zone.

Field I/O Zone: This is the zone were actual industrial devices are deployed which runs the industrial protocols like MODBUS. All the equipments like reactors, controllers, pumps and PLCs are located in this area, hence this is the most important area of the industry network.

Before, the SCADA networks were considered to be safe as it was believed that these networks are isolated and hackers do not have enough information on the protocols and services used by them. But, nowadays Internet is a part of these networks and it is required for many business purposes like billing for electric services. Hence, SCADA networks are no longer isolated networks and are prone to attacks. It is also possible to infect an isolated SCADA network with the help of a portable storage devices like USB.

The incident happened in Iran (mentioned in Chapter 1) is an example of attacking isolated SCADA networks with the help of infected USB.

According to HMS (Industrial network company which manufactures and markets industrial communication products), Ethernet/IP is growing at very fast pace and it accounts for 38% of the market in 2016. Ethernet/IP is mostly used Ethernet network with 9%, followed by PROFINET (8%), Ether CAT (6%), MODBUS/TCP (4%) and Powerlink (3%).

4.2 MODBUS Protocol

MODBUS is a serial communication protocol developed by MODICON (Schneider Electric) in 1979 for their PLCs. It is published openly and it is royalty free. MODBUS has become very popular and a de facto communication standard in the industry because of its high real time performance and low deployment cost. MODBUS can be accessed by the internet community at a reserved system port 502 on the TCP/IP stack. It offers services based on the function codes and it is a request/response type of protocol.

MODBUS protocol can be classified into three types based on its implementation as follow:

1. MODBUS TCP/IP (over Ethernet).
2. Asynchronous serial transmission (This includes different media like wire: EIA/TIA-232-E, EIA-422, EIA/TIA-485-A, fiber, radio etc.)
3. MODBUS PLUS: This network requires dedicated co-processor for handling fast token rotations.

Figure 14 shows the communication stack for MODBUS protocol. MODBUS is an application layer messaging protocol i.e., 7th layer in OSI model. Devices like PLC, HMI, I/O devices and so on can use MODBUS protocol to initiate the operation. An example MODBUS architecture using different mediums (mentioned above) is shown in Figure 15 below.

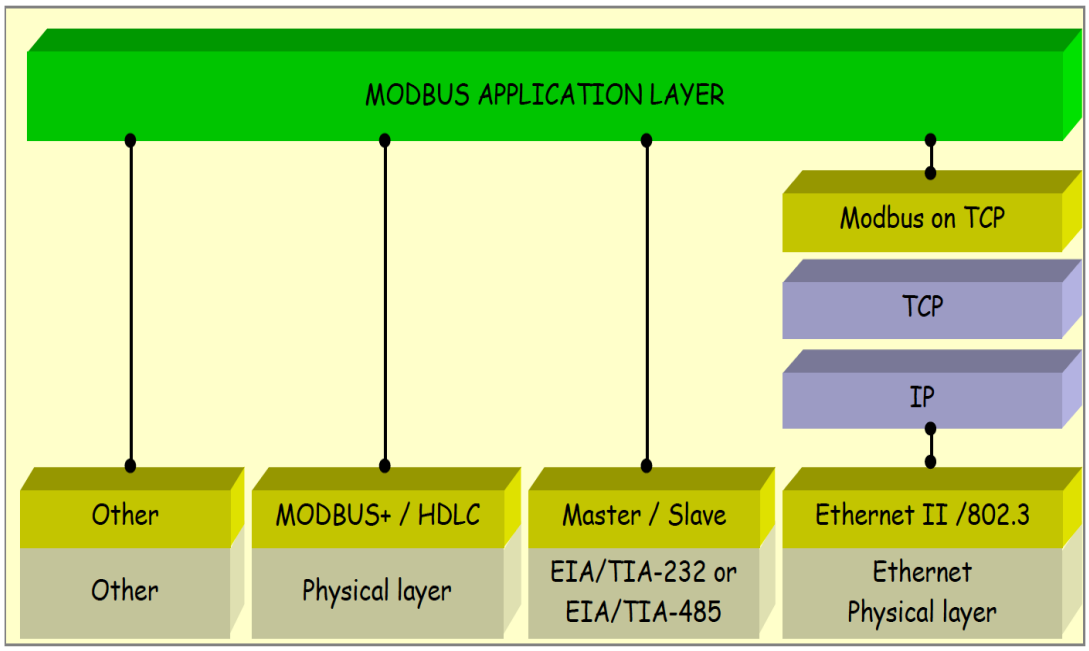


Figure 14: MODBUS Communication Stack [12]

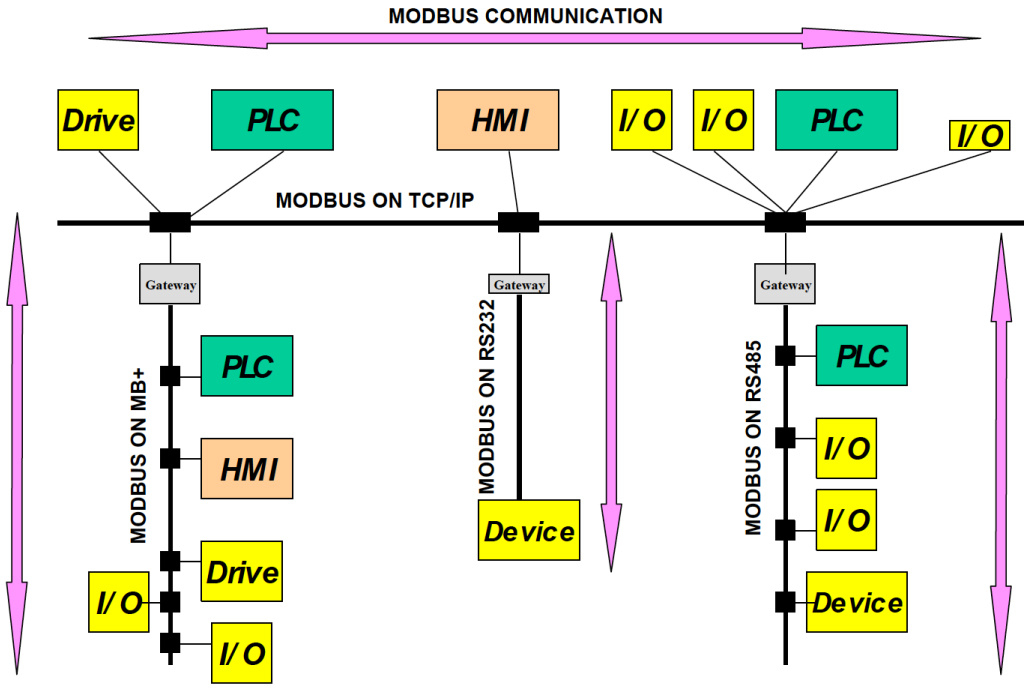


Figure 15: Example MODBUS Architecture [12]

4.2.1 MODBUS Frame Structure

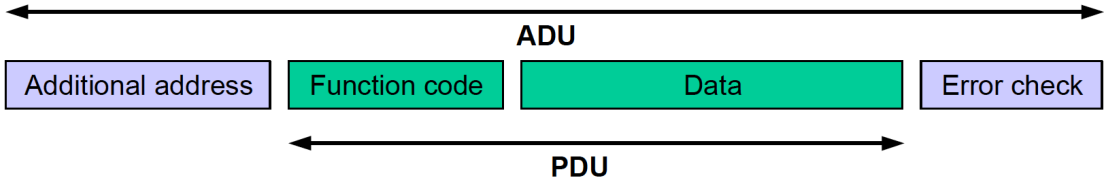


Figure 16: MODBUS Frame [12]

- **PDU**: Protocol data unit is defined by Modbus independent of underlying communication layers and it consists of function code and data bits.
- **Function code:** It is of 1byte and tells the server what action to perform. It ranges from 1 to 255 (decimal). 128 to 255 is reserved for exception responses.

- **Data:** This field contains extra information needed by the server in order to perform the action requested by the function code. For example, register addresses, number of items to be handled etc. It can be empty
- **Error Check:** This field provides a method for both master/slave to validate the integrity of the received message.
- **Additional Address:** This field provides the recipient device's address. The address can range from 1 to 247 decimal. 0 is used as broadcast address.

4.2.2 MODBUS Transactions/ Query- Response Cycles

As mentioned before, MODBUS is a request/response protocol. A typical MODBUS transaction (communication cycle) involves client sending an initiate request. This request consists of a function code and data along with the other parts of the MODBUS frame. The function code will specify the action that server has to perform and the data field would contain any extra information required to perform the requested action. If the server can perform the requested action without any error, then it is called error-free transaction (shown in Figure 17). In this case the server would send back the requested data in the data field along with the function code and the transaction is completed as shown in Figure 17. On the other hand, if the server fails to perform the requested action then it sends back the error code/exception code along with the requested function code as shown in Figure 18. The exception code is then used to determine further actions to be taken.

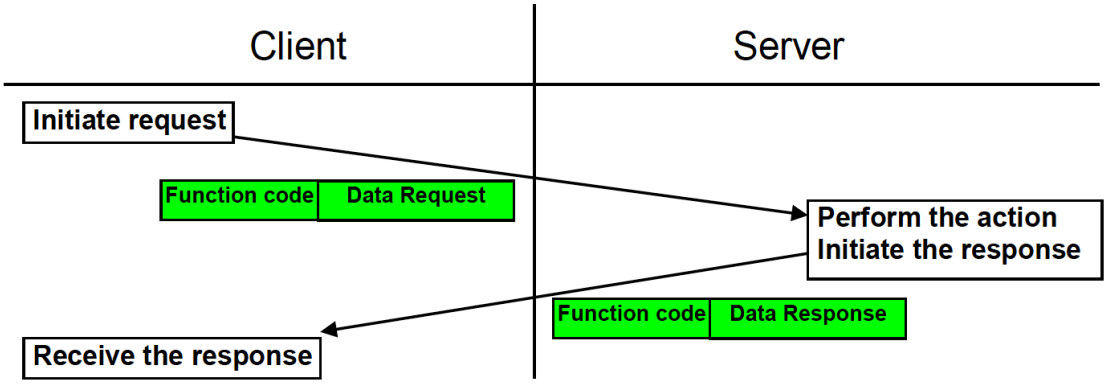


Figure 17: MODBUS Transaction (Error Free) [12]

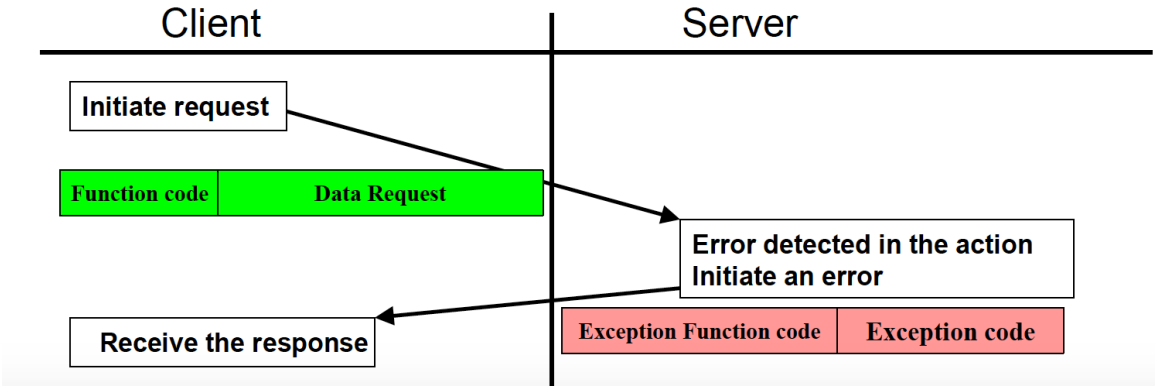


Figure 18: MODBUS Transaction (Error) [12]

4.2.3 MODBUS Data Model

In MODBUS the slave device provides the master device with the following object types:

(1) Coils: These are 1 bit read only boolean values which are mostly used to represent the outputs.

(2) Discrete Inputs: These are 1 bit read only boolean values which are typically used to represent the sensor inputs.

(3) Input registers: These are 16 bit read only registers which are used to represent analogue input values.

(4) Holding registers: These are 16 bit read and write registers which are used to represent the analogue output values.

There is no difference in the application behavior due to the distinctions between the inputs, outputs and bit addressable or word addressable. All four object types can be considered as overlaying on one another.

Chapter 5 Online Intrusion Detection System (OIDS)

5.1 OIDS Architecture

As mentioned earlier, the OIDS developed as a part of this project uses the testbed developed by Liao Zhang in [2]. In a real world scenario, two most important things needed to carry out the attacks are attack target and attacker (hacker). Also, for detecting or

preventing these type of attacks a network intrusion detection system is required along with firewalls and prevention system. Hence, the OIDS consists of the following components:

- Attack targets (Developed in [2])
- Attack toolkit (Developed in [2])
- Detection system (Developed as a part of this project)
- Offline Training Module (Developed as a part of this project)

The testbed developed in [2] consists of the attackers, attack targets and defenders deployed on virtual machines and they connect with each other on LAN. The private IPs begin with 10, 172.16- 172.31 or 192.168. In the testbed all the attackers are in 192.168.100.0/24 network and the targets are in 10.0.0.0/24 network. Here attackers and targets are kept in two different networks in order to simulate a real world scenario. The administration network shown in the Figure 19 is in 172.16.1.0/24 segment and people can use it by logging in through ssh or by web console. The detection system works in a bridge mode to route and transmit packets between both networks. The detection system is capable of logging all the packets for further analysis in the offline module.

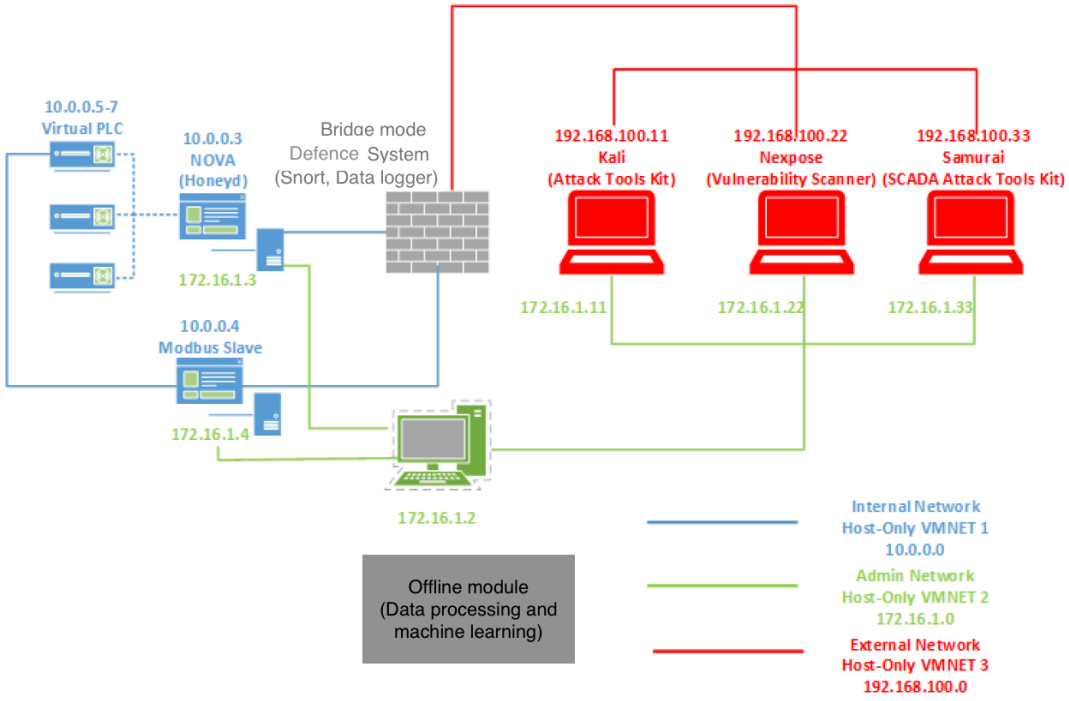


Figure 19: Online Intrusion Detection System Architecture (After [2])

5.2 Attack Targets

In an industrial network the attack target is usually the network itself. As PLC is the most used device in a SCADA network, it was chosen to be an specific attack target in [2]. All the PLCs developed in [2] are simulated with the help of software (shown with blue part in Figure 19). Now, considering hacker's perspective the testbed requires an industrial process with the target which would help to simulate a more realistic SCADA network. Hence, a system with two tanks using the MODBUS TCP was built to demonstrate a simple industrial process as shown in Figure 20. MBLogic's HMI builder was used to develop this tank system. Now this tank system has following two parts:

1. **HMI:** It is shown by Nova in Figure 19, it can also be referred as Modbus master. The main function of HMI is to pull/query the liquid level of the two tanks from the sensors and send the desired pump speed to the sensors/motor.
2. **Modbus Slave (sensors):** This is shown as Modbus slave in Figure 19. The main function of the sensors is to poll the data (tank liquid level, motor speed).

In terms of TCP/IP, HMI is a client who sends the request and sensor/motor is a server that processes the request and sends the response back to the client. In terms of Modbus, HMI is the Modbus master and sensor/motor is the Modbus slave.

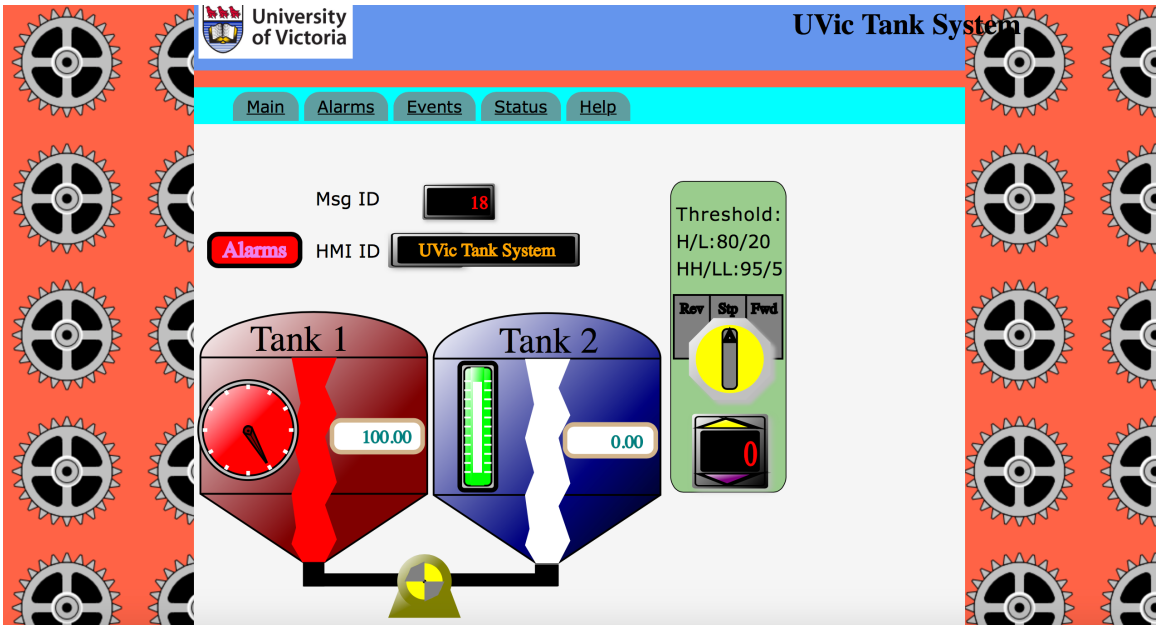


Figure 20: Tank System HMI [2]

As shown in the above Figure both tanks have water or some kind of liquid and the liquid level is shown on the column bar of each tank e.g. it shows 100 for tank 1 in above figure. The liquid level value can range from 0 to 100. The pump used for pumping the liquid from one tank to another can be turned on by setting the knob on the right to a particular position

(Forward or reverse). ‘Fwd’ pumps liquid from tank 1 to tank 2, ‘Rev’ pumps in opposite direction and ‘Stp’ is used to stop the pump. The speed at which the liquid is pumped can be changed with the help of buttons located below the knob. The speed range is -9 to 9, for instance -9 means pump liquid from tank 2 to tank 1 at the speed of 9 units per second. Also, the tank system has 4 threshold levels set. HH: If liquid level is above 95 then the system would generate an alarm, LL: If liquid level is below 5 then system would generate an alarm, H: If liquid level goes above 80 the system would generate a warning and L: If liquid level goes below 20 the system would generate a warning.

5.2.1 Honeypot

The tank system is configured as honeypot in order to entrap the hackers. Honeypot can be defined as a decoy computer system that is used to lure hackers and in turn detect, prevent or study the attempts made by hackers to gain unauthorized access of the system. There are three types of honeypots.

1. Pure Honeypot: It is a fully operational production system and it is used very less due to its high implementation cost.
2. High Interaction Honeypot: It behaves/imitates like the production system by using software approach. The tank system mentioned above is a high interaction honeypot.
3. Low interaction Honeypot: In this honeypot, only most frequently used services are simulated by software. This type of honeypot is deployed using honeyd (a daemon which can build any number of fake systems e.g. Linux servers, PLCs and so on).

The tank system developed in [2] is small and when hacker scans the network, he would either lose interest or realize that this is a honeypot. Hence honeyd is used to configure a complex network. Now honeyd is a command line tool that uses a configuration file to create the complex networks. Nova, a web console that provides many ready to use services and scripts to automatically create a honeyd configuration is used in [2]. As shown in Figure 21, a profile called Schneider PLC is created and four nodes are added just by assigning IP range 10.0.0.5-8 using Nova. By using Nova and honeyd one can easily expand the high interaction honeypot (i.e., the tank system) into a large scale network and entrap the hackers.

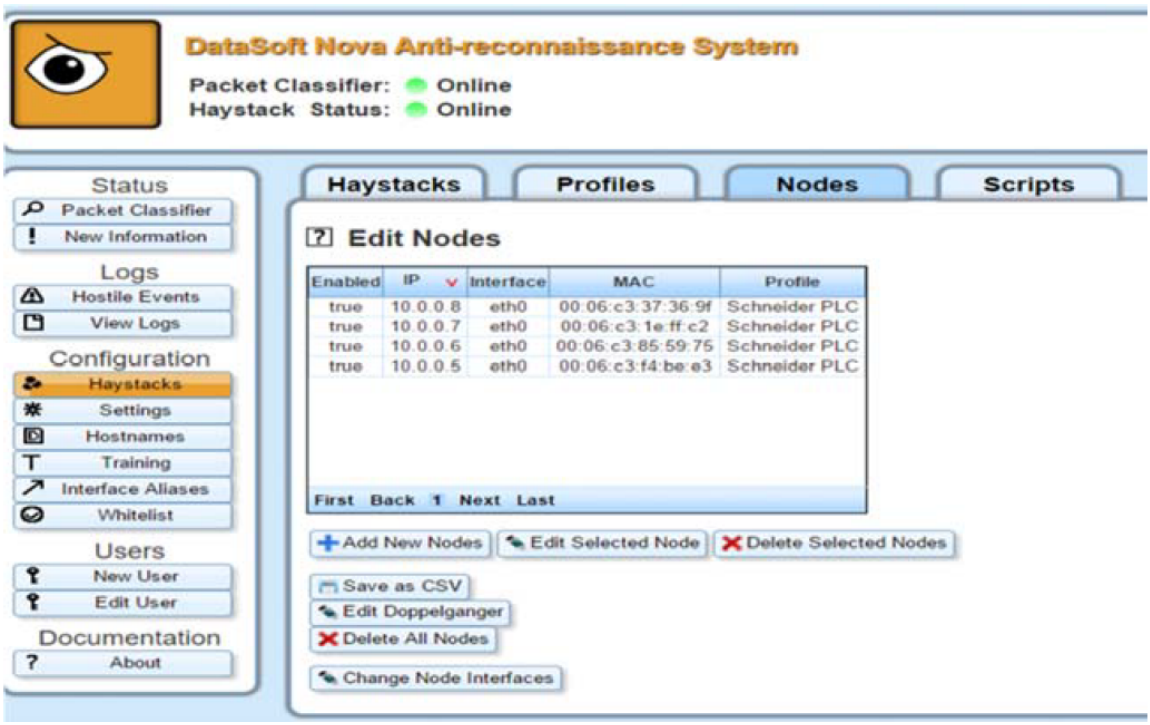


Figure 21: Nova web console [2]

5.3 Attackers

OIDS has three different attack toolkits as shown in Figure 19.

5.3.1 Kali

Kali [13] Linux is a linux distribution specifically designed for digital forensics and penetration testing. Kali Linux is pre-installed with more than 300 penetration testing tools. The two tools that are used for attacking the tank system in this project are Nmap and Modpoll. Nmap is used to hack into a network and find out its topology which includes the device IP addresses, open ports, protocols, services running on different ports and so on. Modpoll is a tool that is used to send instruction to MODBUS master and slave. One can set the liquid level and control the motor speed of the tank system using this tool. Most of the attacks launched in this project (detailed description in chapter 6) are done using Modpoll. This tool is not pre-installed in kali it has to be downloaded from its website.

5.3.2 Nexpose

Nexpose is an attack tool that focuses on vulnerability scan just like Nessus scanner. Nexpose has a web console (Figure 22) that can be used to conduct all round vulnerability scan. The main goal of incorporating this tool in the attack toolkit is to identify the vulnerabilities of the attack targets in the OIDS. This tool is not used in conducting the attacks on the tank system.

5.3.3 Samurai

Samurai is also a Linux distribution like Kali with plenty of attack tools. The main difference between kali and Samurai is that the tools in Samurai are industrial oriented e.g. modscan, it only scans Modbus devices. Figure 23 shows Samurai's tool menu.

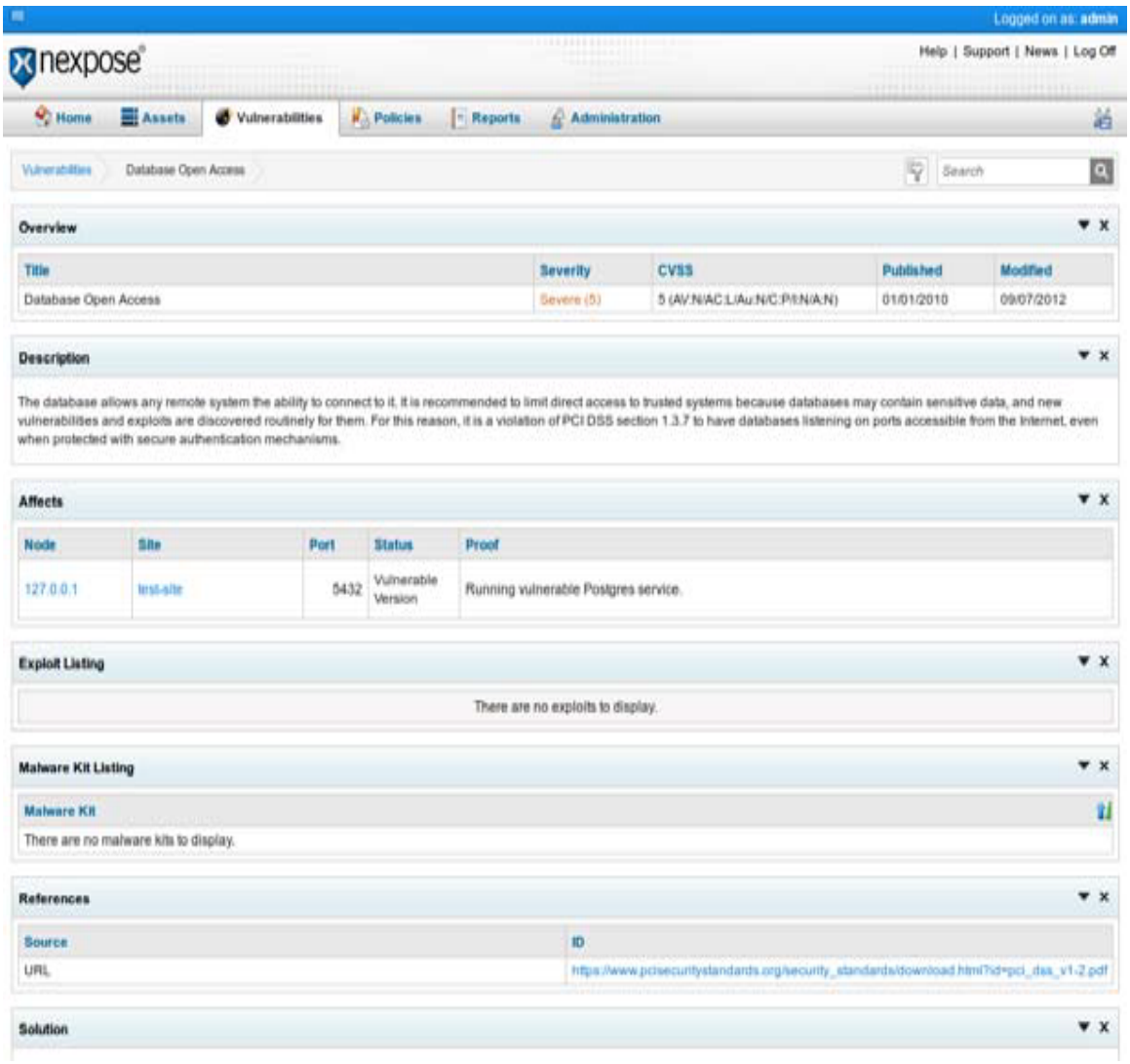


Figure 22: Nexpose Web Console [2]

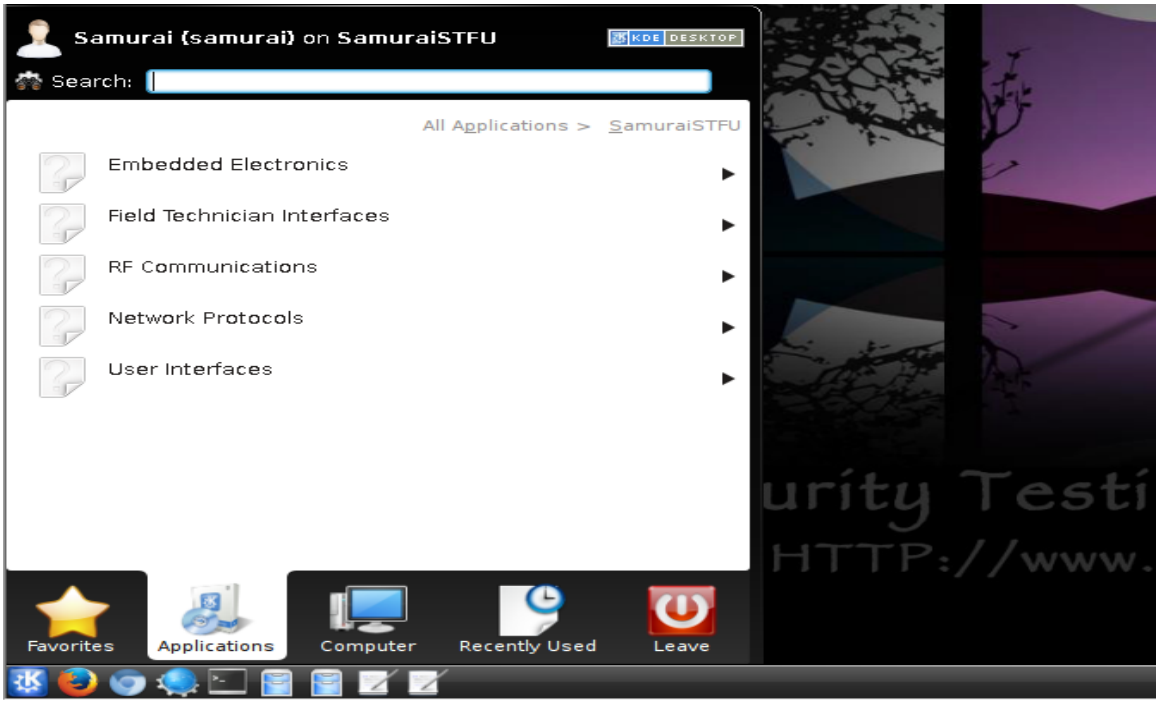


Figure 23: Samurai Tool menu [2]

5.4 Defense (Detection) System

The main objective of the defense system is to detect, alert and terminate any malicious session/traffic. The defense system is configured to be in bridge mode (refer Chapter 8 for configuration details) and as a result the iptables packets transfer in bridge mode. This makes the defense system work like a gateway and the hackers would not even know about its existence. In this project the defense system shown in grey part in Figure 19 has two main functions:

1. Capture all the traffic that goes through it.
2. Detect all the malicious/attack traffic going through it and generate alerts.

As the main goal of OIDS is to provide intrusion detection based on logistic regression machine learning algorithm, it is necessary to capture all the traffic that is going through the defense system. Wireshark is used to capture the traffic as shown in Figure 24. This captured traffic (Pcap file) is then given to the offline module where MODBUS reader parses the pcap file and generates a CSV file with desired features and label. After which, this CSV file is given to MATLAB code [5] which uses logistic regression algorithm to train the data and generate weights for each feature in the CSV file. This whole process taking place in offline module is demonstrated in Chapter 6.

Intrusion Detection System (IDS) are used to detect any attempt for unauthorized access to a computer network by analyzing the traffic on the network. Snort, the most popular network IDS is used in this project for providing the detection (refer Chapter 7 for details). IDS uses predefined rules to analyze the network packets and detect malicious behaviour. Mostly the IDS is deployed between the network router and the Internet and the firewall is deployed behind the router. The main reason for this is you want IDS to capture as much information as it can. If the firewall is placed ahead of IDS, then firewall would filter a lot of traffic and this would hinder the IDS' operation.

Also, one important thing to note here is that the defense system in OIDS is only capable of detecting the malicious traffic based on machine learning weights. The defense system does not cut off the malicious session or drop the attack packets i.e., the system does not provide intrusion prevention.

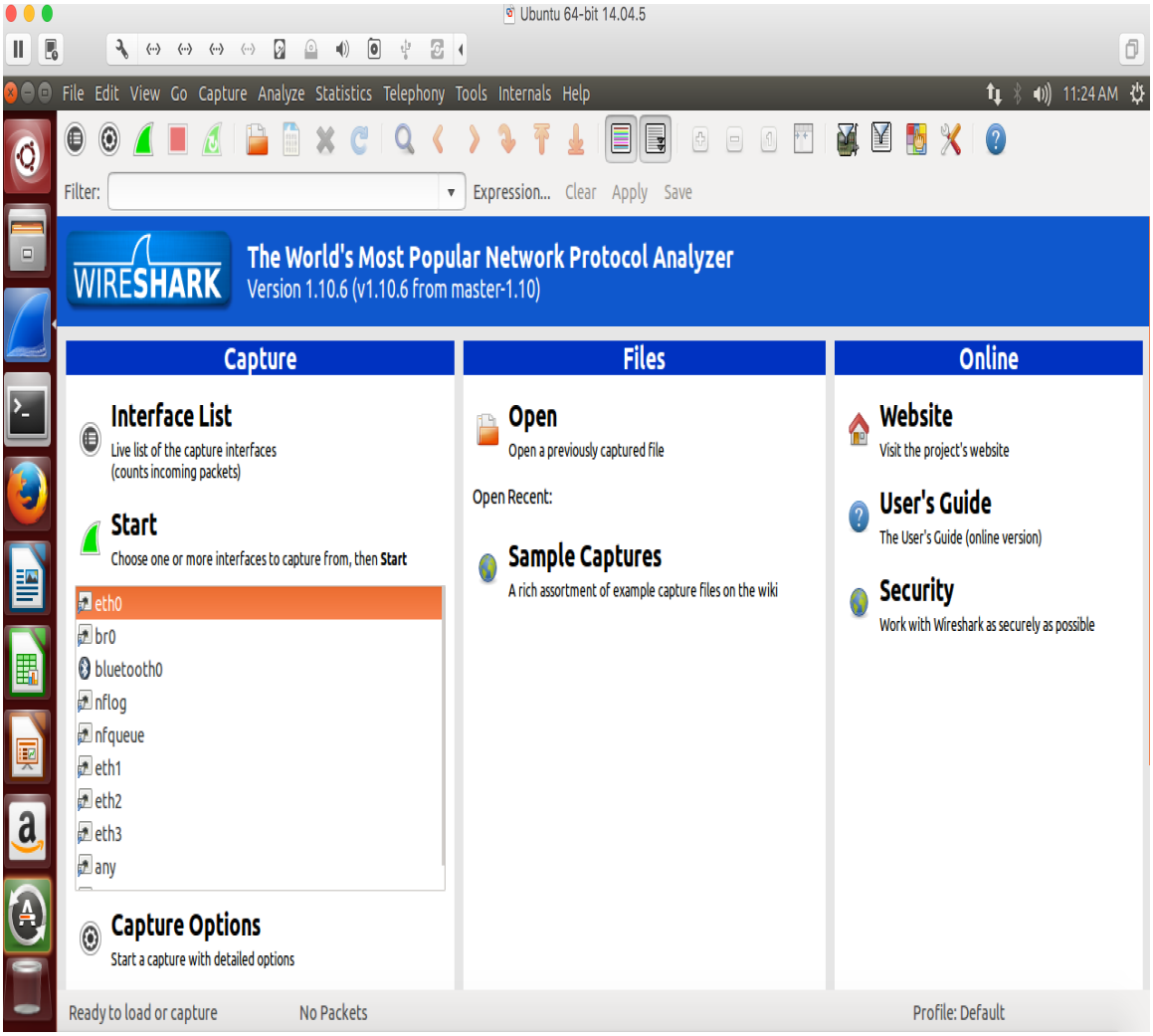


Figure 24: Wireshark

Chapter 6 Traffic Generation and Dataset Processing

This chapter will discuss how to generate the attack and normal traffic with the help of the attack tool kit and gather the data. This chapter also discusses how to use the MODBUS reader for generating the CSV files containing the desired features and how to obtain machine learning weights corresponding to each feature in the CSV file.

6.1 Traffic Generation

As mentioned earlier, Kali linux is installed with a command line tool called Modpoll. This tool is basically used to issue MODBUS commands to the tank system (attack target). One can easily change the liquid levels in the tanks and also modify the speed of the pumps used in the tank system (Figure 20) with the help of Modpoll. But before going into this details, it is necessary to know how to setup OIDS.

6.1.1 OIDS Setup

(1) Setup the attack targets: The attack target comprises of Modbus slave (server) and Modbus master (client). Start the server first (run mod_slave.sh script), which is implemented on Mod slave virtual machine in [2] as shown in Figure 25. After this, start the client i.e., HMI/ Modbus master as shown in Figure 26. This is implemented on the Nova virtual machine and also start the honeyd daemon in order to use the additional configured PLCs (shown in Figure 21).

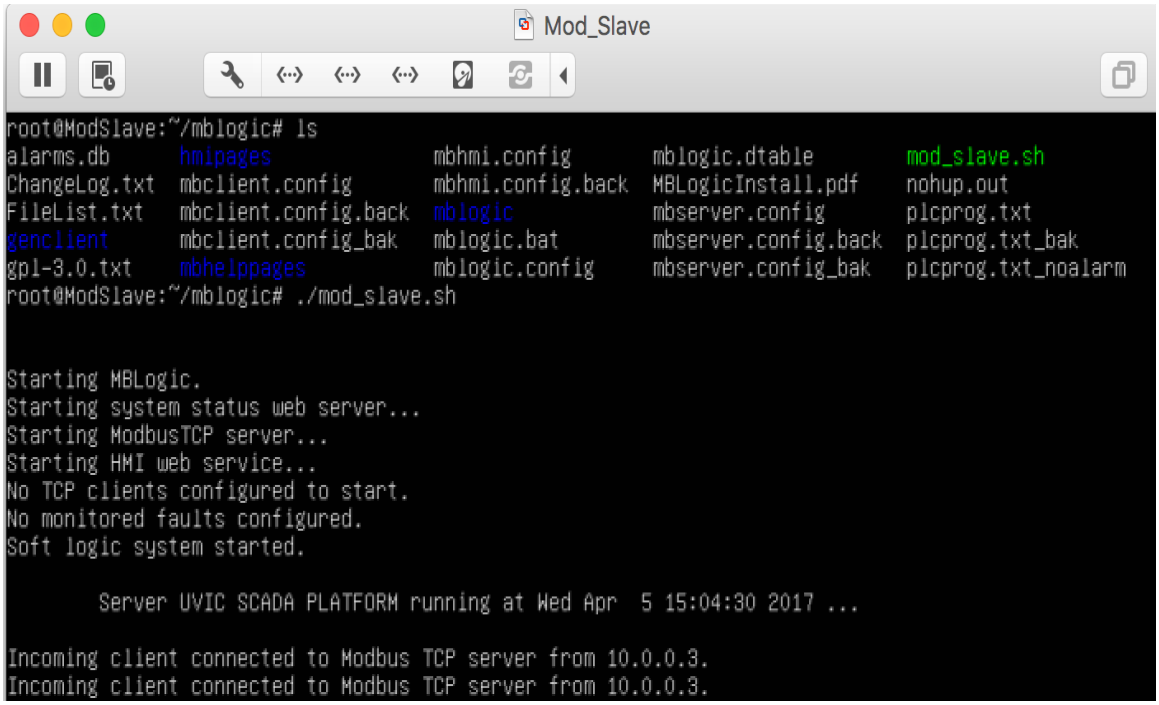


Figure 25: Starting Mod Slave server

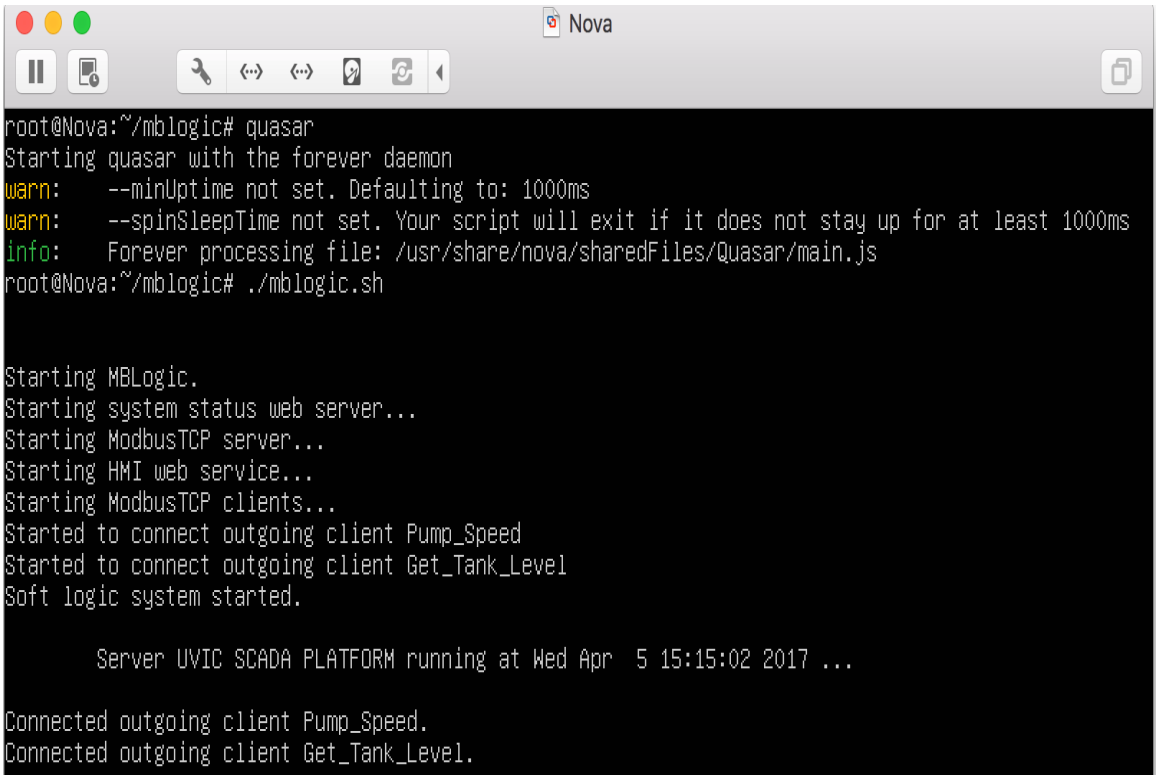


Figure 26: Starting the HMI and honeyd

(2) Setup Defense system: As mentioned earlier, the defence system is configured to be in bridge mode so that the hacker does not realize its presence. Start the virtual machine called Defense wall and setup Wireshark to listen on br0 (bridge) port as shown in Figure 27. Wireshark will be used to capture all the traffic going through the defense wall.

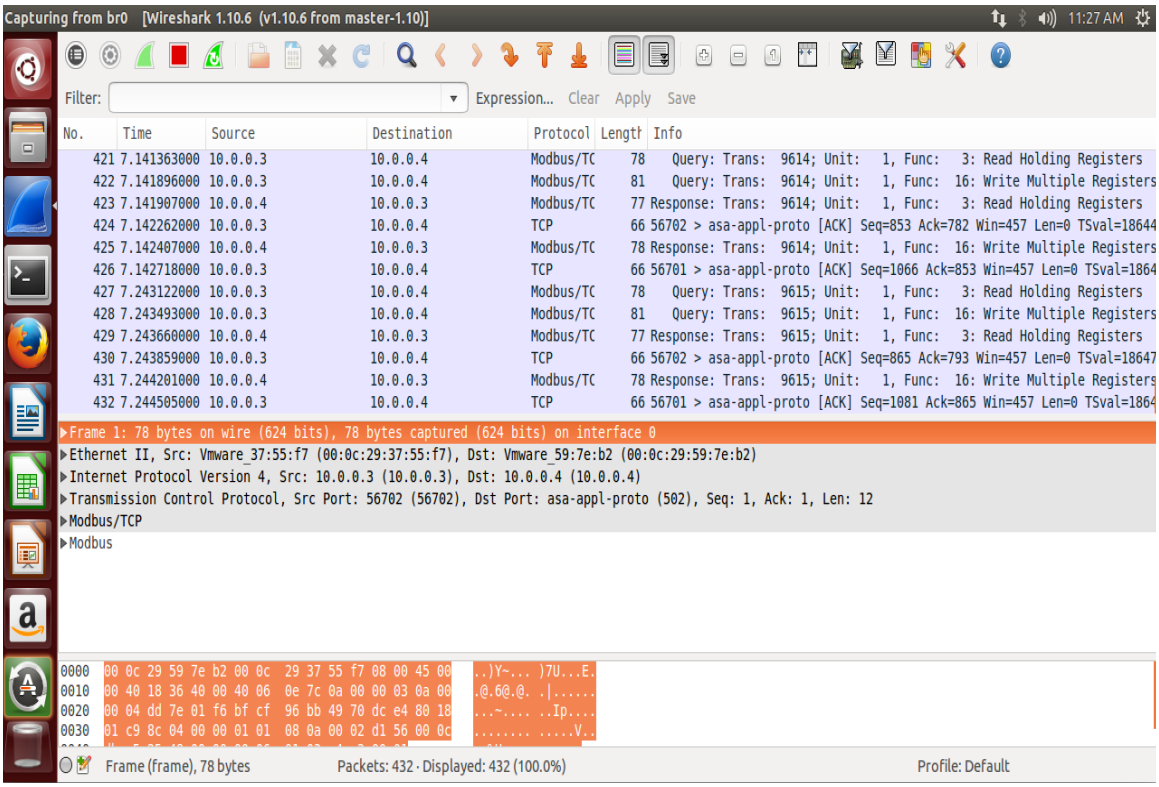


Figure 27: Wireshark capturing packets on Defense wall

(3) Start the attack machine: Now that everything is setup, start the attack machine which is named as kali. This machine consists of simple shell scripts that generates both attack and normal traffic using Modpoll commands. Following section gives the details on this scripts.

6.1.2 Traffic Generation Scripts

The traffic generation scripts are divided into two parts:

(1) Normal traffic generation: This includes the scripts that will issue the Modbus command where the tank liquid level and motor speed are kept within the allowed limits (as mentioned in 5.2 Attack Targets). That is no Alarm will be generated. There are four scripts for generating normal traffic as follow:

- pump_speed_reg.sh: This script sets the pump speed to different values and all the values are within the valid range i.e., +9 to -9 as shown in Figure 28 below.

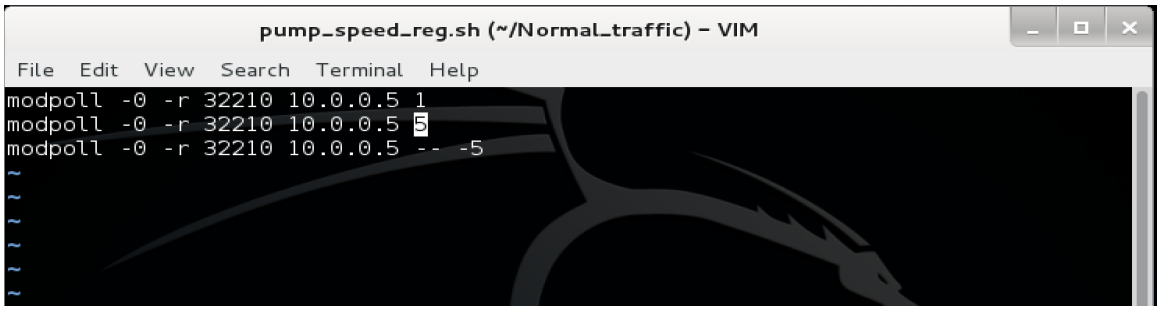


Figure 28: pump_speed_reg.sh

- tank_level_normal.sh: This script sets the liquid level in both the tanks within the valid range of 5 to 95 as shown in Figure 29 below.

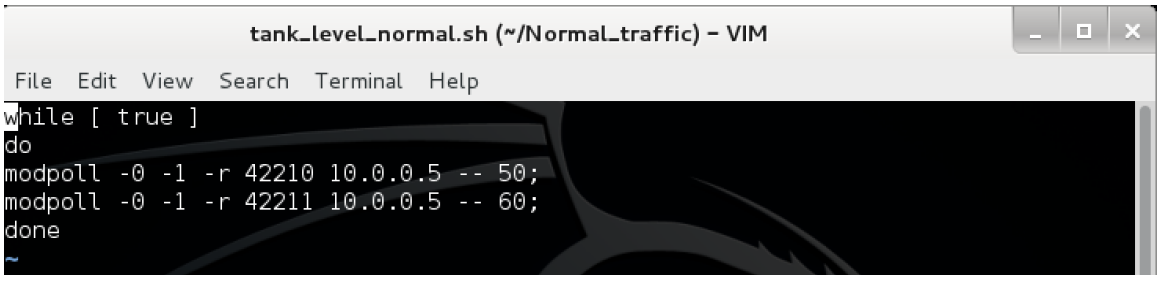


Figure 29: tank_level_normal.sh

- modify_threshold_normal.sh: This script sets the alarm threshold levels for liquid in the tank to the normal values i.e., 95 (HH) and 05 (LL) as shown in Figure 30 below.

Figure 30: modify_threshold_normal.sh

- pump_speed_conti.sh: This script changes the pump speed continuously within the normal range of -9 to +9. This ensures that no alarm is raised as the liquid level always stays within the normal range.

Figure 31: pump_speed_conti.sh

(2) Attack traffic Generation: There are four scripts that allows you to generate the attack traffic by changing the pump speed and water level to abnormal values and generate alarm.

- pump_speed_attack.sh: This script sets the value of pump speed to abnormal values i.e., +200 and -200. This causes the liquid level within the tanks change rapidly and generates the alarm.

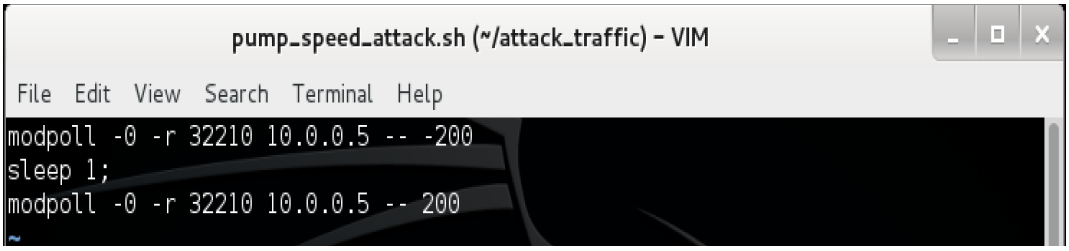


Figure 32: pump_speed_attack.sh

- tank_level_attack.sh: This script directly sets the level of the liquid in the tanks to abnormal values. Because of this the alarm will be generated as the liquid levels in the tanks are out of normal range.

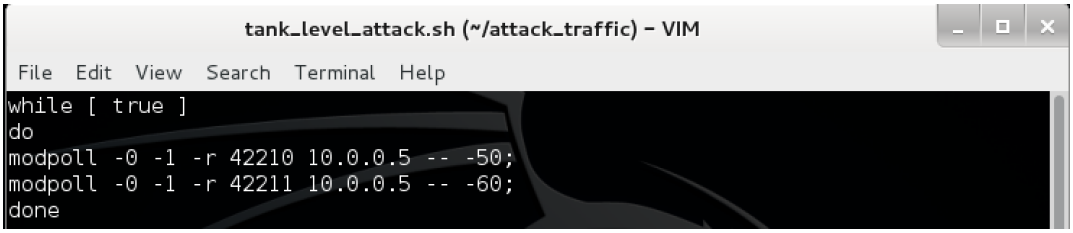


Figure 33: tank_level_attack.sh

- modify_threshold_attack.sh: This script changes the alarm threshold levels for the liquid in the tank (i.e., HH and LL) to different values. Hence, the alarm would not be generated even if the liquid level is more than allowed value.

Figure 34: modify_threshold_attack.sh

- dos.sh: This script sends massive Modbus instruction with incorrect CRC (cyclic redundancy check) and in turn causes the PLC to enter Denial of Service.

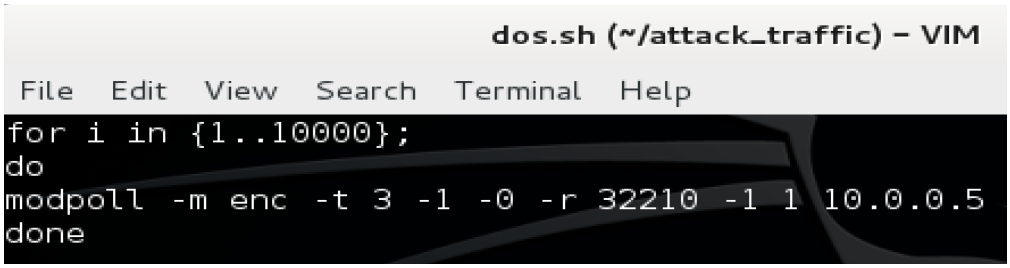


Figure 35: dos.sh

6.2 Dataset Processing

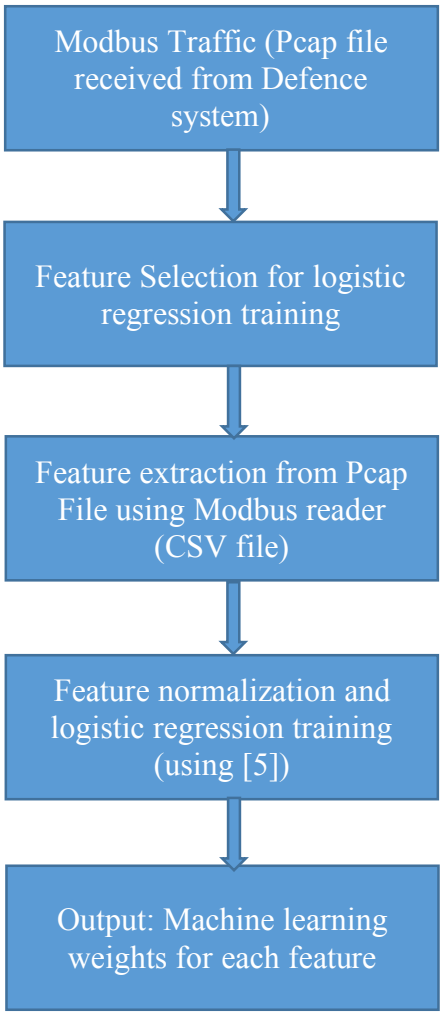


Figure 36: Dataset processing in offline module

Figure 36 shows the data processing steps that are carried out in the offline module (local machine) in order to obtain the machine learning weights for online intrusion detection.

1. **Modbus traffic**: Over 5000000 packets were captured by following the steps mentioned in Section 6.1 above. The captured traffic includes both attack and normal packets. The captured Modbus packets consisted either one of the two following two function codes:

   Write Multiple Registers (Function code 16): This function code is used to set the pump speed and liquid level of the tank system to some value. As explained in 4.2.2 MODBUS Transactions/ Query- Response Cycles ,each Modbus transaction consist of a query and response packet. Figure 37 below shows the query/request packet for function code 16.

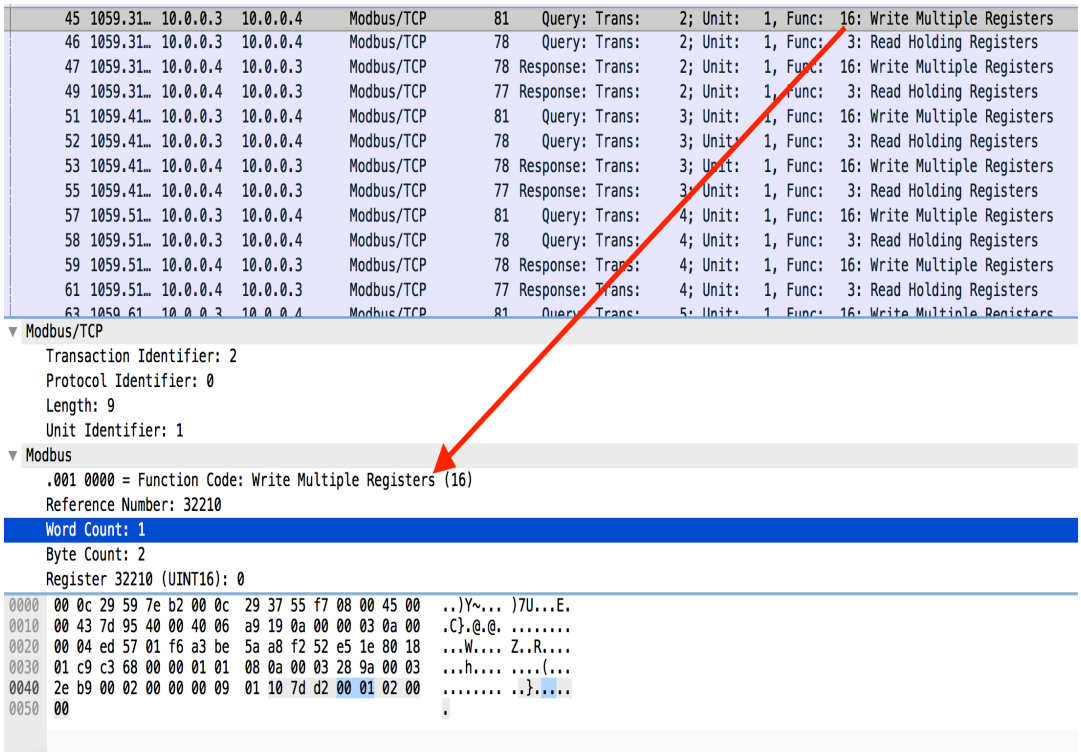


Figure 37: Function code 16 request packet

Logic Address	HMI Objects	System Address
YS10	Tank1 Level	42210[HoldingReg]
YS11	Tank2 Level	42211[HoldingReg]
XS10	Pump Speed	32210[HoldingReg]
YS12	Threshold HH	42212[HoldingReg]
YS13	Threshold LL	42213[HoldingReg]
YS14	Threshold H	42214[HoldingReg]
YS15	Threshold L	42215[HoldingReg]

Modbus Master (Client)
10.0.0.5 (Honeyd)
10.0.0.3 (Physical IP)

Modbus Poll

Logic Address	HMI Objects	System Address
YS10	Tank1 Sensor	42210[HoldingReg]
YS11	Tank2 Sensor	42211[HoldingReg]
XS10	Pump Valve	32210[HoldingReg]

Modbus Slave (Server)
10.0.0.4 (Physical IP)

Figure 38: Address of the tank system [2]

The reference number field in the packet's PDU (Protocol Data Unit) shows value 32210 which is the system address for pump speed holding register (see Figure 38). In the request packet shown in Figure 37, the Register 32210 (UINT16) has value '0' which means in this packet Modbus master (10.0.0.3) is requesting Modbus slave (10.0.0.4) to set the pump speed value to 0. If the value of reference number is 42210 or 42211 in the packet, then it means that packet is requesting to set the liquid level in tank 1 (42210) or tank 2 (42211) to a certain value as indicated by the last field in request packet PDU. Figure 39 shows the response packet corresponding to the above request packet.

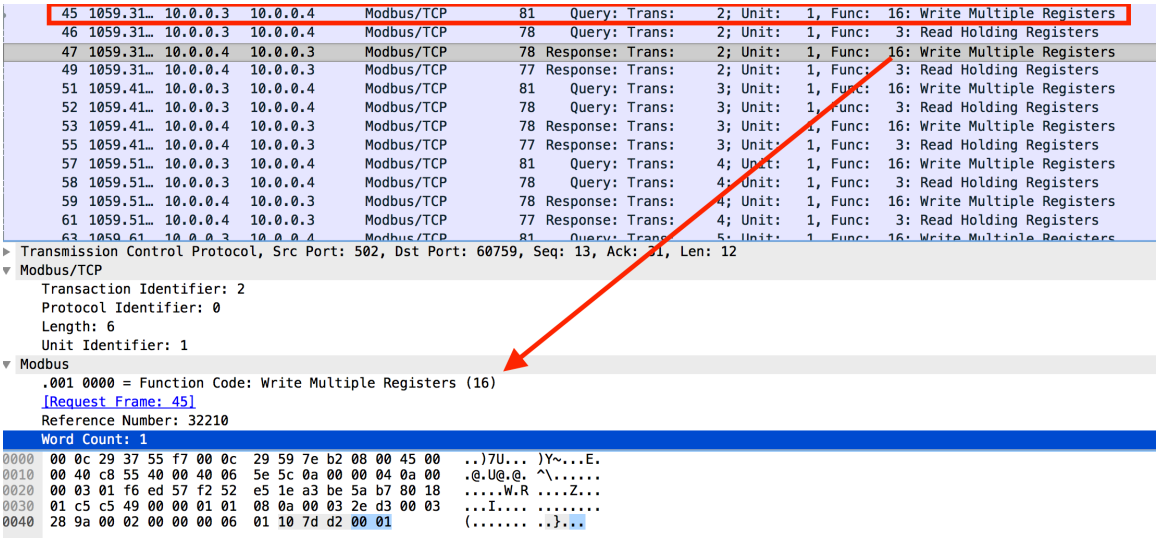


Figure 39: Function code 16 response packet

Read Holding Registers (Function code 3): Figure 40 shows the request packet for function code 3, it is requesting to read the liquid level in tank 2 (as reference number field's value 42211 corresponds to tank 2 level). The response packet for this gives the liquid level of tank in the field 'Register 42211 (UINT 16)' which is 100 as shown in Figure 41.

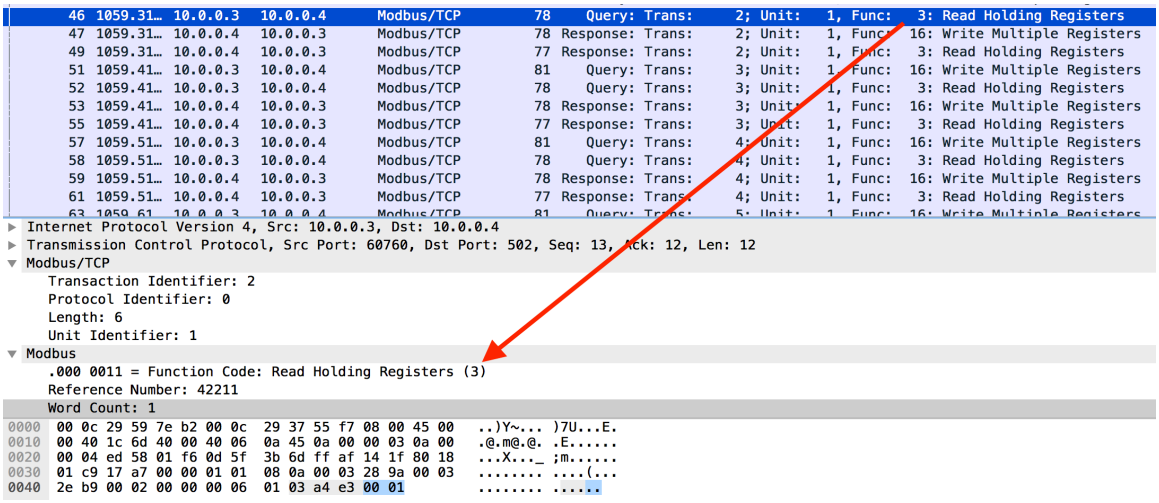


Figure 40: Function code 3 request packet

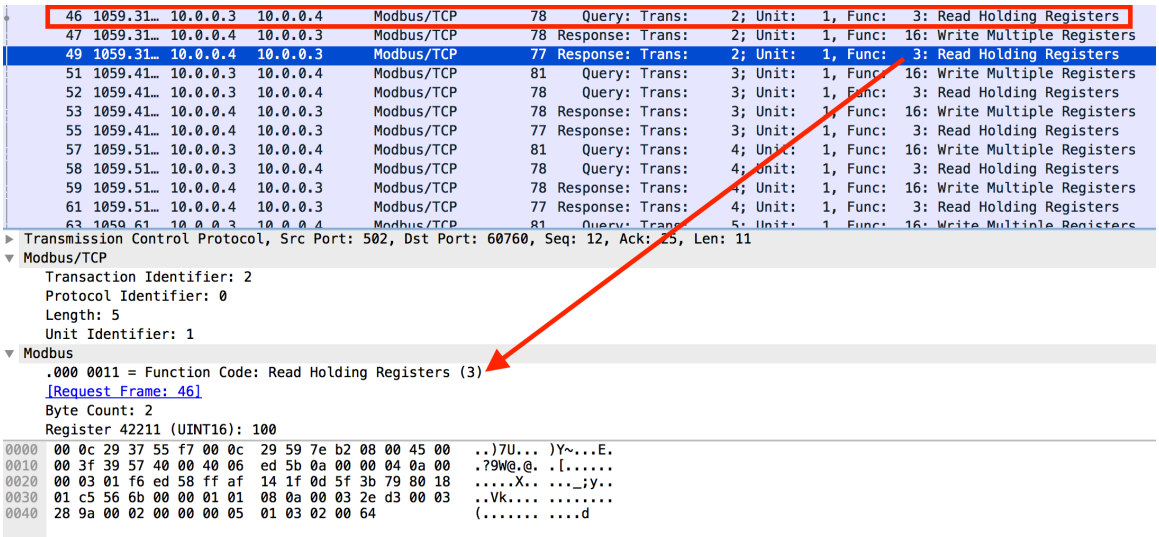


Figure 41: Function code 3 response packet

2. **Feature Selection for logistic regression training:** Feature selection also referred as variable selection is a process for selecting subset of relevant features/variables required to construct a machine learning model (logistic regression model in this case). The data consists of many features, but many of the features would be either redundant or irrelevant. Based on the traffic received following features were considered for training:

   - Source IP address
   - Destination IP address
   - Source Port
   - Destination Port
   - Protocol Identifier
   - Unit Identifier
   - Transaction Identifier
   - Function Code
   - Reference number
   - Write Data (Register 32210 field of function code 16 request packet in Figure 37)
   - Resp Data (Register 42211/42210 field of function code 3 response packet)

   Out of the above 11 features, Protocol identifier and Unit identifier fields were redundant i.e., their values were same for each packet throughout the data. Hence, those two features were removed from the data set and are not included in the training. Also, timestamp of the packet can be included as a feature in order to calculate the volume or frequency of the packets arrived. This parameter would be helpful in detecting Denial of Service attack as we can differentiate between normal

packet rate and abnormal rate based on the frequency obtained from the timestamps. In this project we are using supervised machine learning approach which means all the packets would be labelled to a particular class/type. As our main goal is to identify whether a packet is attack or normal, 'Alarm' is selected as a classification label because the tank system generates an alarm when the liquid level in the tanks are above HH (95) or below LL (5).

3. **Feature extraction using Modbus reader:** As mentioned earlier, Modbus reader is a Java code which takes pcap file as input, parses the file and extracts the above mentioned features into a CSV file (shown in Figure 42). This code can be modified to extract any feature from a packet in the file. It reads through Ethernet header, IP header (extracts Source and destination IP), TCP header (extracts source and destination port), MBAP header (Modbus application header, extracts transaction identifier) and finally reads the Modbus PDU where it extracts function code, reference number, read data / write data (based on function code). The source and destination IP are hash coded to a particular integer value in the CSV file. This makes it easier for the Matlab code [5] to parse the IP values while training the data. The IP address 10.0.0.3 is represented by value '511552168', IP address 10.0.0.4 is represented by '511552169' and attack machine kali's IP address 192.168.100.11 is represented by 836853820.

   The value of label 'Alarm' which determines whether the received packet is attack or not is decided by the value of Write data (pump speed, liquid level or threshold level requested in the packet). Table 2 gives the list of cases where the

packet is considered to be attack by the Modbus reader based on the values of WriteData. As mentioned earlier, this project uses supervised machine learning approach hence it is necessary to classify the data into a particular type. The table below does not include the classification for DOS attack because timestamp would have to be added to the feature list and based on the timestamp one has to determine the packet frequency i.e. how many packets per second (or some time) are considered to be normal and if the packet frequency is greater than that, it would be classified as DOS attack.

Reference number	**Write Data**	**Alarm**	**Description**
32210	Greater than 9 or less than -9	1	In this case the packet sets pump speed (32210) outside the valid range hence alarm is set to 1.
42210 or 42211	Greater than 95 or less than 5	1	In this case the packet sets the liquid level in either tank 1 (42210) or tank2 (42211) outside the valid range. Hence alarm is set to 1.
42212	Not equal to 95	1	In this case the packet sets the value of HH other than the normal threshold, hence alarm is set to 1.

42215	Not equal to 5	1	In this case the packet sets the value of LL other than the normal threshold value, hence alarm is set to 1.

Table 2: Cases where Alarm is set to '1' by Modbus reader

	A	B	C	D	E	F	G	H	I	J
1	sourceIP	DestIP	SourcePort	destPort	TransIdentif	FuncCode	Refno	WriteData	RespData	Alarm
2	511552168	511552169	58783	502	6298	16	32210	0	0	0
3	511552169	511552168	502	58783	6298	16	0	0	0	0
4	511552168	511552169	58784	502	6298	3	42211	0	0	0
5	511552169	511552168	502	58784	6298	3	0	0	100	0
6	511552168	511552169	58783	502	6299	16	32210	0	0	0
7	511552168	511552169	58784	502	6299	3	42210	0	0	0
8	511552169	511552168	502	58783	6299	16	0	0	0	0
9	511552169	511552168	502	58784	6299	3	0	0	0	0
10	511552168	511552169	58783	502	6300	16	32210	0	0	0
11	511552168	511552169	58784	502	6300	3	42211	0	0	0
12	511552169	511552168	502	58783	6300	16	0	0	0	0
13	511552169	511552168	502	58784	6300	3	0	0	100	0
14	511552168	511552169	58783	502	6301	16	32210	0	0	0
15	511552168	511552169	58784	502	6301	3	42210	0	0	0
16	511552169	511552168	502	58783	6301	16	0	0	0	0
17	511552169	511552168	502	58784	6301	3	0	0	0	0
18	511552168	511552169	58783	502	6302	16	32210	0	0	0
19	511552168	511552169	58784	502	6302	3	42211	0	0	0
20	511552169	511552168	502	58783	6302	16	0	0	0	0
21	511552169	511552168	502	58784	6302	3	0	0	100	0
22	511552168	511552169	58783	502	6303	16	32210	0	0	0
23	511552168	511552169	58784	502	6303	3	42210	0	0	0
24	511552169	511552168	502	58783	6303	16	0	0	0	0
25	511552169	511552168	502	58784	6303	3	0	0	0	0
26	511552168	511552169	58783	502	6304	16	32210	0	0	0
27	511552168	511552169	58784	502	6304	3	42211	0	0	0
28	511552169	511552168	502	58783	6304	16	0	0	0	0
29	511552169	511552168	502	58784	6304	3	0	0	100	0
30	511552168	511552169	58783	502	6305	16	32210	0	0	0
31	511552168	511552169	58784	502	6305	3	42210	0	0	0
32	511552169	511552168	502	58783	6305	16	0	0	0	0

Figure 42: Modbus reader output CSV file

4. **Normalization and logistic regression training:** The above CSV file generated is given to Matlab code developed by Dr. Tao Lu in [5]. The data is first normalized to ensure that all the data is in the same scale. If the values of different features are widely different then this would affect the ability to learn. Hence, the data is normalized and then it undergoes training using logistic regression algorithm. The result of this is weight corresponding to each feature shown in Figure 42 except the label (Alarm). These weights are used in Snort for detection which is explained in following chapter.

Chapter 7 Intrusion Detection using Snort

7.1 SNORT

Snort is an open source network intrusion detection and intrusion prevention system, originally developed by Martin Roesch in 1998. Snort is not used in the intrusion prevention mode for this project as the main goal is to provide detection based on machine learning weights derived from logistic regression training. Snort is capable of performing real time traffic/protocol analysis, packet logging and content matching and searching. Snort can be configured to run into following three different modes:

- Sniffer mode: In this mode Snort continuously reads the packets off a network and displays them on the screen.
- Packet logger mode: In this mode the packets are logged on to the disk.
- Network Intrusion Detection System (NIDS) mode: In this mode Snort performs detection and analysis on network traffic based on the ruleset defined by the user. Then it performs the specified action based on what has been detected.

7.1.1 Snort Workflow

Snort's workflow is shown in Figure 43.

- Packet decoder: The network packets on the wire are decode by the 'Packet Decoder'. The packet decoder determines the packet's protocol and then matches the packet data with its allowable behaviour based on the type of protocol. It also generates alerts in case of malformed packet headers, over lengthy packets, TCP options set in the header are incorrect and other similar behaviours. It is possible

to enable or disable more verbose alerting for all these fields in Snort.conf (configuration file for Snort) file as shown in Figure 44. After this, the packets are sent to the preprocessors if they are enabled in the Snort.conf file.

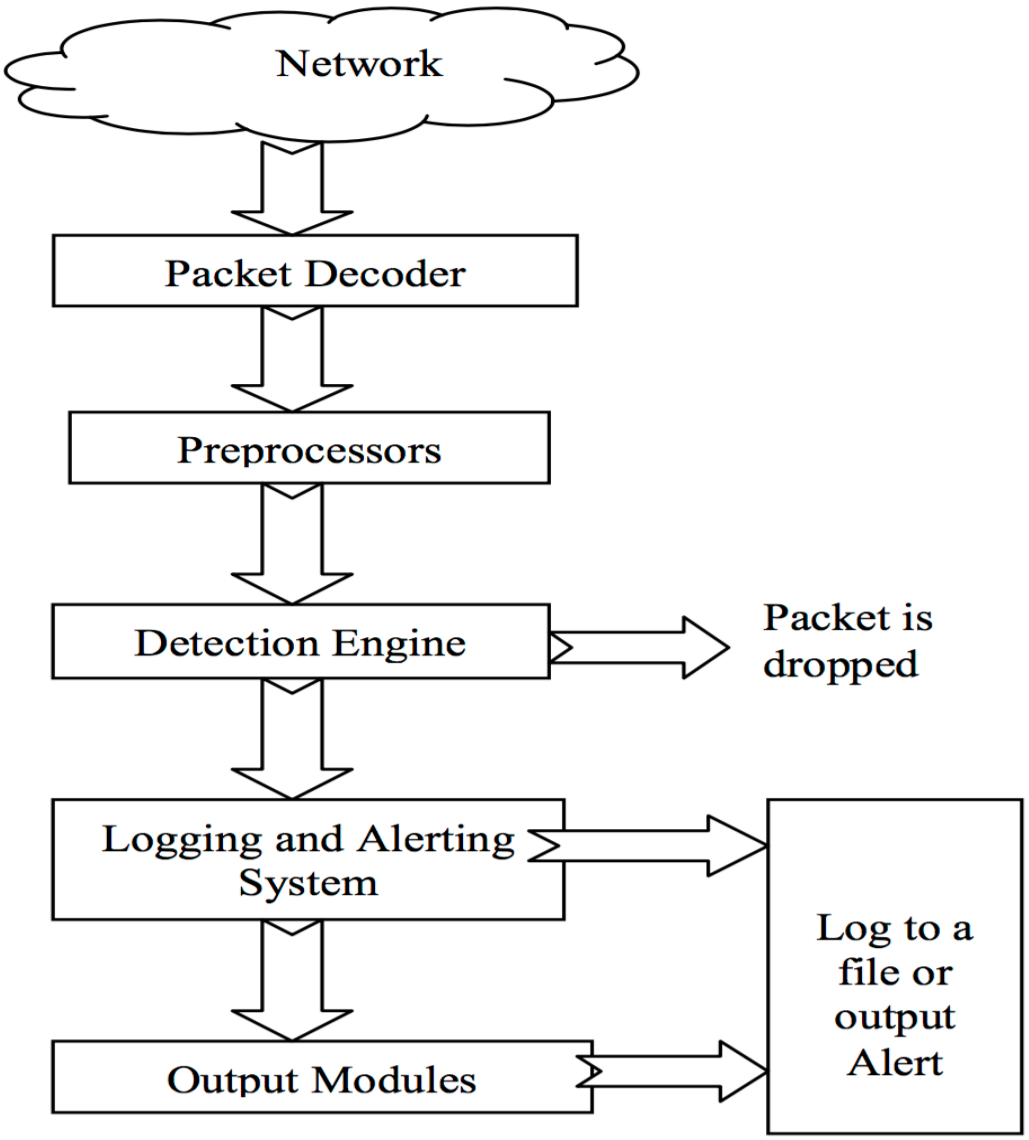


Figure 43: Snort Workflow [14]

```
##################################################
# Step #2: Configure the decoder.  For more information, see README.decode
##################################################

# Stop generic decode events:
#config disable_decode_alerts

# Stop Alerts on experimental TCP options
config disable_tcpopt_experimental_alerts

# Stop Alerts on obsolete TCP options
config disable_tcpopt_obsolete_alerts

# Stop Alerts on T/TCP alerts
config disable_tcpopt_ttcp_alerts

# Stop Alerts on all other TCPOption type events:
config disable_tcpopt_alerts

# Stop Alerts on invalid ip options
config disable_ipopt_alerts

# Alert if value in length field (IP, TCP, UDP) is greater th elength of the packet
# config enable_decode_oversized_alerts

# Same as above, but drop packet if in Inline mode (requires enable_decode_oversized_alerts)
# config enable_decode_oversized_drops

# Configure IP / TCP checksum mode
config checksum_mode: all

# Configure maximum number of flowbit references.  For more information, see README.flowbits
# config flowbits_size: 64
```

Figure 44: Snort Decoder configurations

- Preprocessors: Preprocessors are the Snort plug-ins which allows the user to manipulate the incoming packets in many different ways. If no preprocessor is enabled in the snort.conf file, then packet would be given to the detection engine as it is received on the wire. This may be dangerous as Snort provides variety of preprocessors that are capable of detecting port scans (portscan and portscan2 preprocessor), reassemble TCP fragments (frag2 preprocessor) and so on. The Snort manual [15] provides description of 24 such inbuilt snort preprocessors.

- Detection Engine: This engine takes the data from the decoder and/or preprocessor if they are enabled and then performs the rule matching as defined in the snort.conf file.

- Logging and alerting: This part generates appropriate alerts and logs the messages depending on the detection engine output.

- Output module: Snort provides different types of output plugins (sys log, csv, unified and so on) which can be used based on your requirements. These plugins process the alerts and generates the final output.

7.2 Detection using logistic regression weights

7.2.1 Detection Algorithm

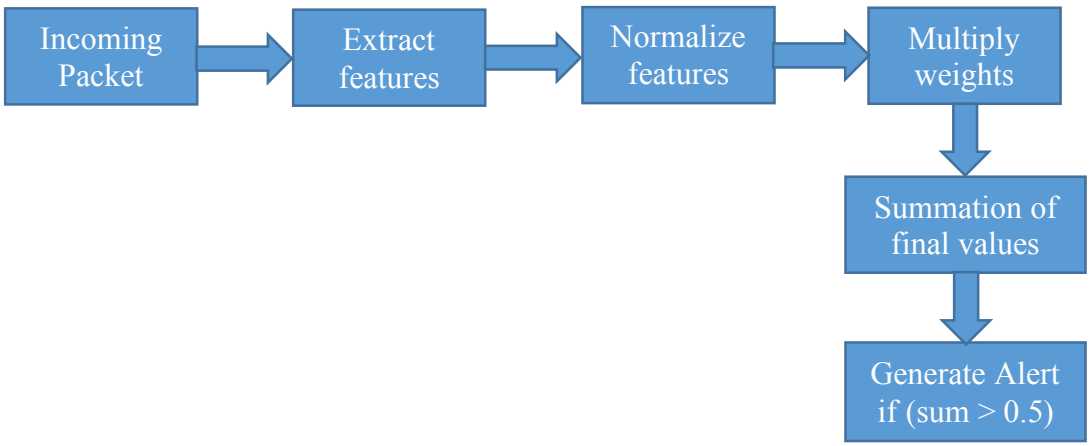


Figure 45: Intrusion Detection Algorithm

The intrusion detection algorithm shown in Figure 45 developed as a part of this project is used to detect whether the incoming packet is attack or normal. It is not possible to implement the above algorithm with the help of Snort rules because it would increase the complexity to a great extent. Also, one cannot access all the selected features mentioned in Section 6.2 Dataset Processing with the help of Snort rules. The preprocessor module of Snort is used to implement the above algorithm as it allows us to manipulate the incoming packet data as mentioned in Section 7.1.1 Snort Workflow.

Modbus Preprocessor: Snort provides a dynamic preprocessor that decodes the packets with Modbus protocol. Hence, this project uses the Modbus preprocessor in order to implement the detection algorithm. The Modbus preprocessor is located at snort-x.x.x.x/src/dynamic-preprocessor/Modbus inside the snort package as shown in Figure 46 below.

The modbus_decode.c file decodes the Modbus payload and hence the intrusion detection algorithm is implemented in this file (Refer Appendix C for the code). First of all, the packet decoder module of Snort decodes the incoming packet and the decoded information is used to assign the values to the SFSnortPacket data structure located in src/decode.h file. Now, given that the Modbus preprocessor is enabled in the snort.conf file and the incoming packet is Modbus the flow would be transferred to the Modbus preprocessor. After this the preprocessor has been configured to extract all the selected features and then the features are normalized using the Equation 1 (mean and standard deviation of a feature are derived from the training data), then the normalized features are multiplied with their corresponding weights derived from logistic regression training and

all the final values are added. If the final sum is greater than 0.5 then the packet is classified as attack packet as shown in Figure 47.

In logistic regression algorithm the predicted values are probabilities of belonging to a particular class. In this case, we are using binary classification i.e. the packet is classified as attack or normal (1 or 0). The detection threshold was chosen to be 0.5 because the probability can range from 0 to 1 and 0.5 is the center value which can detect most of the attacks. This threshold can be changed to a different value based on the overall goal of the system. If the main goal of the detection system is to get higher detection rate, then the threshold should be kept lower than 0.5. This would increase the number of false positives in the detection but, the detection rate would be much higher. On the other hand, if the goal of the system is to provide more accurate detection then the threshold value should be kept higher than 0.5. But this would increase the number of dales negatives in the detection. This trade-off can be reduced by testing various threshold values in the detection algorithm and finding the suitable value keeping the overall goal of the system in mind.

$$\boldsymbol{Feature(Normalized)} = \frac{\boldsymbol{Original\ Value - Mean\ value}}{\boldsymbol{Standard\ deviation}}$$

Equation 1: Feature Normalization

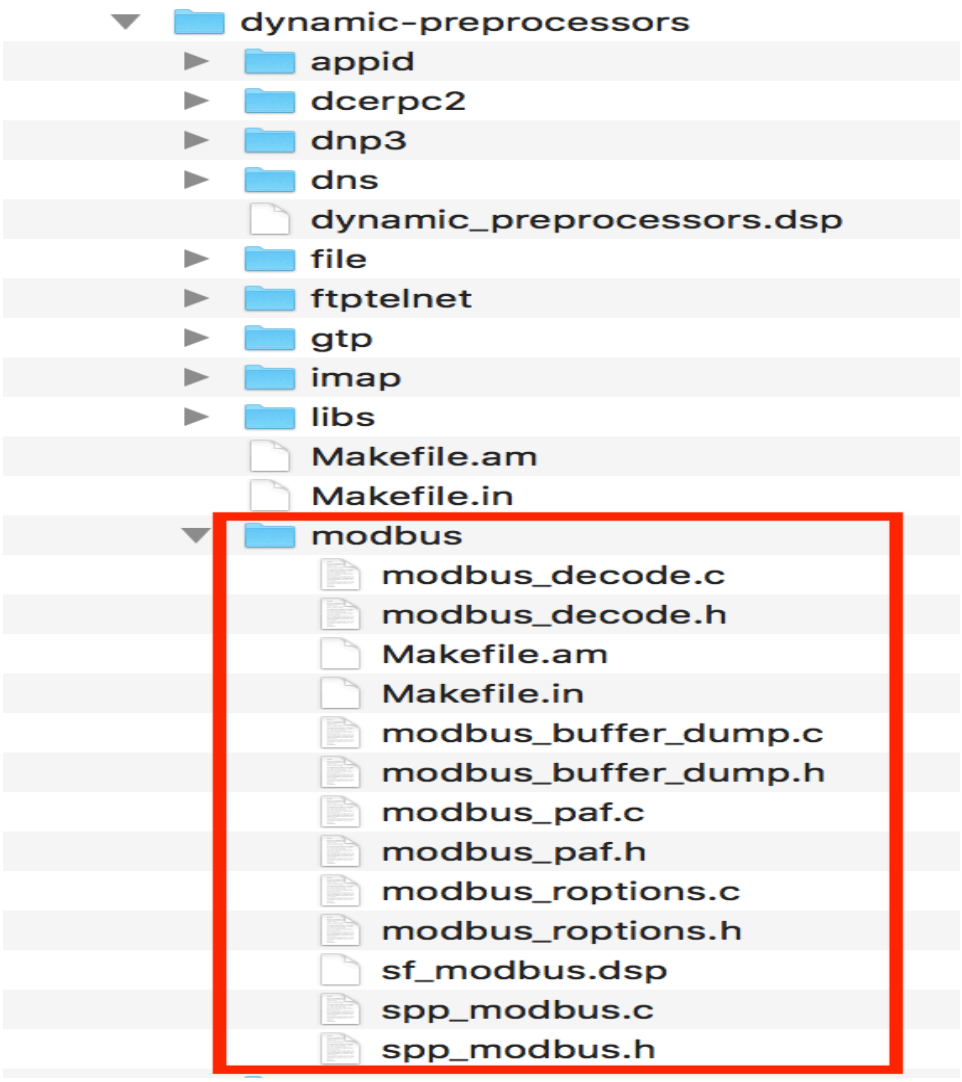


Figure 46: Modbus preprocessor

Snort provides a library function **_dpd.alertAdd** which can be used to generate built in alerts directly from the preprocessor. This function takes parameters like Generator ID, SID, Revision number of the rule, classification number, priority, a message and rule info as input and calls the Snort's alert engine to generate the alerts.

```
double final_sum = (SrcIphash_final + DstIphash_final + src_port_final + dst_port_final + trans_id_final + func_code_final +
                    Refno_final + Write_data_final + Resp_data_final);

_dpd.logMsg("\nFinal sum: %f", final_sum);

if (final_sum > 0.5)
{

        _dpd.logMsg("\n######### This is a attack packet based on final sum ########\n");
        _dpd.alertAdd(GENERATOR_SPP_MODBUS, MODBUS_ATTACK_PACKET, 1, 0, 3,
             MODBUS_ATTACK_PACKET_STR, 0);

}
```

Figure 47: Alert Generation

Once this is done the rule has to be enabled by adding it to the preprocessor.rules file located at **/etc/snort/preproc_rules/** as shown in Figure 48.

```
alert (msg:"MODBUS_ATTACK_PACKET"; gid: 144; sid: 4; rev: 1;)
alert ( msg: "TAG_LOG_PKT"; sid: 1; gid: 2; rev: 1; metadata: rule-type preproc ; classtype:not-suspicious; )
alert ( msg: "BO_TRAFFIC_DETECT"; sid: 1; gid: 105; rev: 1; metadata: rule-type preproc, policy balanced-ips drop, policy security-ips
 drop ; classtype:trojan-activity; reference:cve,1999-0660; )
alert ( msg: "BO_CLIENT_TRAFFIC_DETECT"; sid: 2; gid: 105; rev: 1; metadata: rule-type preproc, policy balanced-ips drop, policy secur
ity-ips drop ; classtype:trojan-activity; reference:cve,1999-0660; )
alert ( msg: "BO_SERVER_TRAFFIC_DETECT"; sid: 3; gid: 105; rev: 1; metadata: rule-type preproc, policy balanced-ips drop, policy secur
ity-ips drop ; classtype:trojan-activity; reference:cve,1999-0660;)
alert ( msg: "BO_SNORT_BUFFER_ATTACK"; sid: 4; gid: 105; rev: 1; metadata: rule-type preproc, policy balanced-ips drop, policy securit
y-ips drop ; classtype:trojan-activity; reference:cve,2005-3252; )
alert ( msg: "RPC_FRAG_TRAFFIC"; sid: 1; gid: 106; rev: 1; metadata: rule-type preproc, service sunrpc ; classtype:protocol-command-de
code; )
alert ( msg: "RPC_MULTIPLE_RECORD"; sid: 2; gid: 106; rev: 1; metadata: rule-type preproc, service sunrpc ; classtype:protocol-command
-decode; )
alert ( msg: "RPC_LARGE_FRAGSIZE"; sid: 3; gid: 106; rev: 1; metadata: rule-type preproc, service sunrpc, policy security-ips alert ;
classtype:bad-unknown; )
alert ( msg: "RPC_INCOMPLETE_SEGMENT"; sid: 4; gid: 106; rev: 1; metadata: rule-type preproc, service sunrpc, policy security-ips aler
t ; classtype:bad-unknown; )
alert ( msg: "RPC_ZERO_LENGTH_FRAGMENT"; sid: 5; gid: 106; rev: 1; metadata: rule-type preproc, service sunrpc, policy security-ips al
ert ; classtype:bad-unknown; )
alert ( msg: "ARPSPOOF_UNICAST_ARP_REQUEST"; sid: 1; gid: 112; rev: 1; metadata: rule-type preproc ; classtype:protocol-command-decode
; )
alert ( msg: "ARPSPOOF_ETHERFRAME_ARP_MISMATCH_SRC"; sid: 2; gid: 112; rev: 1; metadata: rule-type preproc ; classtype:bad-unknown; )
alert ( msg: "ARPSPOOF_ETHERFRAME_ARP_MISMATCH_DST"; sid: 3; gid: 112; rev: 1; metadata: rule-type preproc ; classtype:bad-unknown; )
alert ( msg: "ARPSPOOF_ARP_CACHE_OVERWRITE_ATTACK"; sid: 4; gid: 112; rev: 1; metadata: rule-type preproc ; classtype:bad-unknown; )
alert ( msg: "HI_CLIENT_ASCII"; sid: 1; gid: 119; rev: 1; metadata: rule-type preproc, service http ; classtype:not-suspicious; refere
nce:cve,2009-1535; reference:url,www.microsoft.com/technet/security/bulletin/ms09-020.mspx; reference:url,docs.idsresearch.org/http_id
s_evasions.pdf; )
alert ( msg: "HI_CLIENT_DOUBLE_DECODE"; sid: 2; gid: 119; rev: 1; metadata: rule-type preproc, service http ; classtype:not-suspicious
; reference:cve,2009-1122; reference:url,www.microsoft.com/technet/security/bulletin/ms09-020.mspx; )
alert ( msg: "HI_CLIENT_U_ENCODE"; sid: 3; gid: 119; rev: 1; metadata: rule-type preproc, service http ; classtype:not-suspicious; )
alert ( msg: "HI_CLIENT_BARE_BYTE"; sid: 4; gid: 119; rev: 1; metadata: rule-type preproc, service http ; classtype:not-suspicious; )
alert ( msg: "HI_CLIENT_UTF_8"; sid: 6; gid: 119; rev: 1; metadata: rule-type preproc, service http ; classtype:not-suspicious; refere
nce:cve,2008-2938; reference:cve,2009-1535; reference:url,www.microsoft.com/technet/security/bulletin/ms09-020.mspx; )
"preprocessor.rules" [readonly] 269L, 42023C                                                                  1,1           Top
```

Figure 48: Enabling rule in preprocessor.rules

This would generate the alert in the specified directory as shown in Figure 49 below.

```
[**] [144:4:1] MODBUS_ATTACK_PACKET [**]
[Priority: 0]
04/25-14:35:55.547575 00:0C:29:59:7E:B2 -> 00:0C:29:9E:CF:1B type:0x800 len:0x4E
10.0.0.4:502 -> 192.168.100.11:44127 TCP TTL:64 TOS:0x0 ID:52628 IpLen:20 DgmLen:64 DF
***A**** Seq: 0xA5033D9B  Ack: 0xB73B9D5A  Win: 0x7280  TcpLen: 32

[**] [144:4:1] MODBUS_ATTACK_PACKET [**]
[Priority: 0]
04/25-14:35:55.550626 00:0C:29:59:7E:B2 -> 00:0C:29:9E:CF:1B type:0x800 len:0x4E
10.0.0.4:502 -> 192.168.100.11:44128 TCP TTL:64 TOS:0x0 ID:8127 IpLen:20 DgmLen:64 DF
***A**** Seq: 0x8F221C05  Ack: 0x7AC54F70  Win: 0x7280  TcpLen: 32

[**] [144:4:1] MODBUS_ATTACK_PACKET [**]
[Priority: 0]
04/25-14:35:55.554087 00:0C:29:59:7E:B2 -> 00:0C:29:9E:CF:1B type:0x800 len:0x4E
10.0.0.4:502 -> 192.168.100.11:44129 TCP TTL:64 TOS:0x0 ID:27476 IpLen:20 DgmLen:64 DF
***A**** Seq: 0x1278C665  Ack: 0x6F683B9D  Win: 0x7280  TcpLen: 32

[**] [144:4:1] MODBUS_ATTACK_PACKET [**]
[Priority: 0]
04/25-14:35:55.557000 00:0C:29:59:7E:B2 -> 00:0C:29:9E:CF:1B type:0x800 len:0x4E
10.0.0.4:502 -> 192.168.100.11:44130 TCP TTL:64 TOS:0x0 ID:51899 IpLen:20 DgmLen:64 DF
***A**** Seq: 0x28228203  Ack: 0x3F2B9A75  Win: 0x7280  TcpLen: 32

[**] [144:4:1] MODBUS_ATTACK_PACKET [**]
[Priority: 0]
04/25-14:35:55.559871 00:0C:29:59:7E:B2 -> 00:0C:29:9E:CF:1B type:0x800 len:0x4E
10.0.0.4:502 -> 192.168.100.11:44131 TCP TTL:64 TOS:0x0 ID:38968 IpLen:20 DgmLen:64 DF
***A**** Seq: 0xED81F887  Ack: 0xA70AC8FB  Win: 0x7280  TcpLen: 32

[**] [144:4:1] MODBUS_ATTACK_PACKET [**]
[Priority: 0]
04/25-14:35:55.562640 00:0C:29:59:7E:B2 -> 00:0C:29:9E:CF:1B type:0x800 len:0x4E
10.0.0.4:502 -> 192.168.100.11:44132 TCP TTL:64 TOS:0x0 ID:31686 IpLen:20 DgmLen:64 DF
```

Figure 49: Generated Alerts

Chapter 8 OIDS Configurations

8.1 Virtual Machine

In this project the attack targets, defense system and attack toolkit are implemented using virtual machines with Linux Operating System. VMware Fusion which is the Mac version of VMware virtual machine software is used.

The two virtual machines that forms the attack targets Nova (10.0.0.3) and Mod_slave (10.0.0.4) along with kali (192.168.100.11) are taken from the testbed developed by Liao Zhang in [2]. The Defense Wall virtual machine is developed as a part of this project and its configurations are similar to Honey wall virtual machine in [2]. Both of them are configured in bridge mode so that the attackers are unaware of its presence.

All the virtual machines mentioned above are connected by the 4 virtual networks shown in Figure 50. The internal production network for tank system and honeyd is implemented by VMnet1 (10.0.0.0/24), the administration network is VMnet2 (172.16.1.0/24) and the external attack network where kali and other attack toolkits are deployed is implemented using VMnet3 (192.168.100.0/24). Defense wall routes the packets between VMnet1 and VMnet3. The internet connection for downloading the software packages is provided by VMnet8 (NAT network). Refer [2] to get the installation details for Nova, Mod_slave and attack virtual machines.

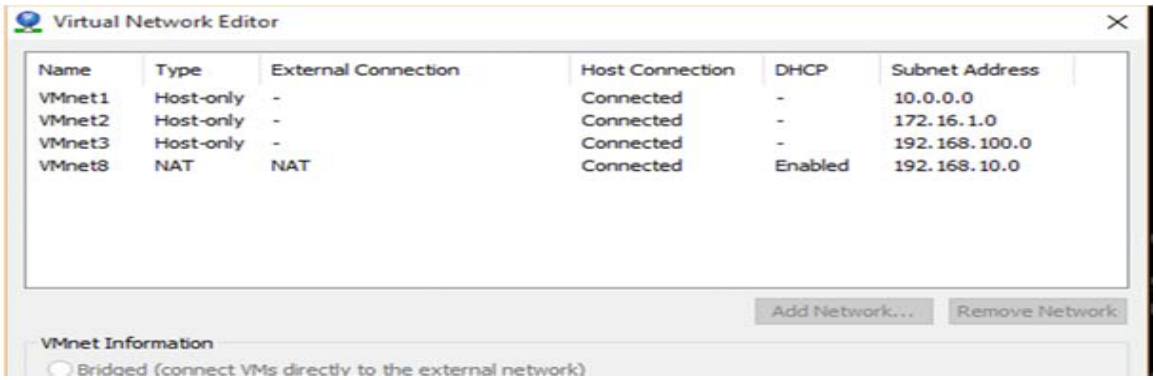


Figure 50: Virtual Network Settings [2]

8.2 Defense Wall Installation

Virtual Machine Host Name	Defense Wall
Interface and IP address	Br0: N/A (Bridge mode) Eth0: N/A (attack network) Eth1: N/A (production network) Eth2: N/A (administration network)
Extra Route	N/A
OS Information	Ubuntu 64 bit (14.04.5)
Virtual Machine Configuration	Memory 1 GB Processor 1 Hard Disk 20 GB Network Adapter 1 VMnet3 Network Adapter 2 VMnet1 Network Adapter 3 VMnet2

Installed Software	Snort (https://snort.org) Wireshark (https://wireshark.org)
Account/Password	Shivam Patel/ root
Run method	Start Wireshark and make it listen on br0 interface Run Snort in NIDS mode using following command ./sudo snort –vde –l ./log –c snort.conf –i br0
Updating the Detection file	In order to make any changes to the detection file e.g. updating weights, mean and standard deviation values and so on: 1. cd /home/Shivam/snort/snort-2.9.9.0/src/dynamic-preprocessors/Modbus 2. Open the modbus_decode.c file where the detection is performed. 3. Make the changes. 4. Run following commands to compile the changes: sudo make sudo make install

Table 3 : Defense Wall Configurations

Currently the defense wall is configured in a way that snort has to be started manually by running the command shown under ‘Run method’ field in Table 3. In order to start Snort when the virtual machine boots up the same command has to be added to **rc.local** file located at **/etc/rc.local**. This would execute the command on machine start-up.

Chapter 9 Future Work and Conclusion

9.1 Future Work

This project analyses the IEC-61850 protocol and its traffic data. It also implements an online intrusion detection system, following work can be done in the future:

1. Develop a test bed that can simulate the IEC-61850 network. This testbed should be able to generate attack and normal traffic for both MMS and GOOSE data.
2. Add functionality of reading GOOSE packets in IEC-61850 reader.
3. Implement intrusion detection on the IEC-61850 testbed based on machine learning techniques, similar to the detection done for Modbus in this project.
4. Deploy the testbed in the Internet and assign a public IP address to attract and analyze real IEC-61850 attacks.
5. Currently the supervised machine learning is implemented and next implement unsupervised machine learning approach to detect the attacks.
6. Also, the implemented detection algorithm has to be tested more. The testing phase would reveal proper detection threshold.
7. Add timestamp to the feature list of detection in order to detect Denial of Service attacks.
8. Implement intrusion prevention system on the defense wall which would be capable of stopping the Modbus attacks.
9. OIDS in this project is deployed using pure software approach. Some real PLC and slaves could be integrated into it.

10. Deploy the OIDS in the internet and assign public IP address to attract and analyze real Modbus attacks.

9.2 Conclusion

The research done on IEC-61850 protocol provides useful insights in the protocol. The traffic analysis for the protocol and IEC-61850 reader could be used in developing intrusion detection system for IEC-61850 protocol. The OIDS system developed in this project can capture the all the traffic needed for machine learning training. Modbus reader can easily read the Modbus traffic and extract necessary features in a CSV file for machine learning training. Also, the detection algorithm implemented in Snort's Modbus preprocessor can successfully detect various Modbus attacks based on machine learning weights.

Bibliography

[1] M. T. A. Rashid, S. Yussof, Y. Yusoff and I. Roslan, "A Review of Security Attacks on IEC61850 Substation Automation System Network," in *International Conference on Information Technology and Multimedia (ICIMU)*, Putrajaya, Malaysia, 2014.

[2] L. Zhang, "An Implementation of SCADA Network Security Testbed," University of Victoria, Victoria, 2015.

[3] S. Yasakethu and J. Jhiang, "Intrusion Detection Via Machine Learning for SCADA System Protection," in *1st International Symposium for ICS & SCADA Cyber Security Research 2013 (ICS-CSR 2013)*, Leicester, UK, 2013.

[4] H. Wang, T. Lu, X. Dong, P. Li and M. Xie, "Hierarchial Online Intrusion Detection for SCADA Networks," *CoRR,* vol. abs / 1611.09418, 2016.

[5] D. T. Lu, *MATLAB code for training data set using logistic regression algorithm,* Victoria: Dr. Tao Lu, 2016.

[6] Y. L. R. H. Campbell, "Understanding and Simulating the IEC 61850 Standard," University of Illinois at Urbana-Champaign, Illinois.

[7] J. Zhang and C. A. Gunter, "IEC 61850 - Communication Networks and Systems in Substations: An Overview of Computer Science," Illinois Secuirty Lab, Illinois, 2011.

[8] S. B. J. R.-M. C. Kriger, "A Detailed Analysis of the GOOSE Message Structure in an IEC 61850 Standard-Based Substation Automation System," *International Journal of Computers, Communication and Control,* vol. 8, no. 5, p. 708, Oct 2013.

[9] M. G. J. Jan Tore Sørensen, "A Description of the Manufacturing Message Specification (MMS)," SINTEF, Norway, 2007.

[10] H. Kirrmann, "ASN.1 encodings for MMS 9506-2003, plus the additions for IEC 61850 8.1," ABB Switzerland Corporate Research, Switzerland, 2005.

[11] P. D. Ernest Rakaczky, "Cyber Security Power Industry Locks Down," Invensys, Plano, Texas, 2011.

[12] The MODBUS Organization, "www.modbus.org," 28 December 2006. [Online]. Available: http://www.modbus.org/docs/Modbus_Application_Protocol_V1_1b.pdf. [Accessed 20 September 2016].

[13] Offensive Security, "Kali Homepage," Offensive Security, 20 September 2016. [Online]. Available: https://www.kali.org. [Accessed 20 March 2017].

[14] P. Mehra, "A brief study and comparison of Snort and Bro open source network intrusion detection systems," *International Journal of Advanced Research in Computer and Communication Engineering,* vol. 1, no. 6, August 2012.

[15] The Snort Organization, "Snort Users manual," Cisco and.or its affiliates, 2014-2015.

[16] J. Beale, A. R. Baker, B. Caswell and M. Poor, Snort 2.1 Intrusion Detection, Rockland: Syngress, 2004.

APPENDIX A Basic Encoding Rules [9]

The Basic Encoding Rules (BER) is one of the original sets of encoding rules specified by the ASN.1 standard for encoding abstract information into a concrete data stream. The rules, collectively referred to as a transfer syntax in ASN.1 parlance, specify the exact octet sequences which are used to encode any given data item before it is transmitted over a network. The BER syntax is defined by the ITU-T's X.690 standards document, which is part of the ASN.1 document series3. In addition to BER there are three alternative encodings, Canonical Encoding Rules (CER), Distinguished Encoding Rules (DER) and Packet Encoding Rules (PER), but as these are not relevant for this report we will not discuss them further.

Bit number	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**Implication**
	0 0 1 1	0 1 0 1							Universal Application specific Context specific Private
			0 1						Primitive data type Constructed data type
				X	X	X	X	X	Numeric identifier

Table 4: Description of the BER Identifier

The BER identifier is described in Table 1. The seventh and sixth bits are combined to denote the class of the ASN.1 tag. The sixth bit of the identifier indicates whether the represented data type is a primitive or constructed one. The remaining X'ed bits of the identifier represent a class number which is associated with a specific data type. BER is a self-identifying and self-delimiting encoding scheme, which means that each data value

can be identified, extracted and decoded individually. Each data element is encoded using a triplet consisting of a type identifier (tag), a length description and the actual data element1 The use of such a triplet for encoding is commonly referred to as a Tag-Length-Value (TLV) encoding.

[identifier (tag)] [length (of the contents)] [content]

The use of TLV encoding allows any receiver to decode the ASN.1 information from an incomplete information stream, without any requiring any pre-knowledge of the size, content or semantic meaning of the data, assuming that the communicating parties share the same context specific module definitions.

BER uses the unique code assigned to an ASN.1 data type as an identifier for a data type. This identifier is encoded as one or more bytes of every data type and creates the tag. We can distinguish between two data types using these identifiers. The identifier is well-structured to allow the representation of three levels of information within one such code. All information encoded into an identifier is illustrated in Table 1.

On the highest level, represented by the highest-order two bits of the tag octet(s), the class of the data type is encoded. The third highest bit of the identifier indicates whether the represented data type is a primitive or constructed one. A constructed data type can be seen as a complex or compound data type hierarchically based on one or more primitive data types. The remainder of the identifier is a numeric tag associated with a data type within a class. Tags ranging from 0 to 30 can be associated with the remaining 5 bits of the octets. For larger tags, these 5 bits are set to 111111, and one or more subsequent octets are used to encode the tag.

APPENDIX B MODBUS Function Codes

The Following table gives the list of various function codes available in MODBUS:

				Function Codes			
				code	*Sub code*	*(hex)*	*Section*
Data Access	**Bit access**	Physical Discrete Inputs	Read Discrete Inputs	**02**		**02**	6.2
		Internal Bits Or Physical coils	Read Coils	**01**		**01**	6.1
			Write Single Coil	**05**		**05**	6.5
			Write Multiple Coils	**15**		**0F**	6.11
	16 bits access	Physical Input Registers	Read Input Register	**04**		**04**	6.4
		Internal Registers Or Physical Output Registers	Read Holding Registers	**03**		**03**	6.3
			Write Single Register	**06**		**06**	6.6
			Write Multiple Registers	**16**		**10**	6.12
			Read/Write Multiple Registers	**23**		**17**	6.17
			Mask Write Register	**22**		**16**	6.16
			Read FIFO queue	**24**		**18**	6.18
	File record access		Read File record	**20**		**14**	6.14
			Write File record	**21**		**15**	6.15
Diagnostics			Read Exception status	**07**		**07**	6.7
			Diagnostic	**08**	**00-18,20**	**08**	6.8
			Get Com event counter	**11**		**OB**	6.9
			Get Com Event Log	**12**		**0C**	6.10
			Report Server ID	**17**		**11**	6.13
			Read device Identification	**43**	**14**	**2B**	6.21
Other			Encapsulated Interface Transport	**43**	**13,14**	**2B**	6.19
			CANopen General Reference	**43**	**13**	**2B**	6.20

Figure 51: MODBUS Function Codes [12]

APPENDIX C Intrusion Detection Algorithm Implementation

The following code shows implementation of the intrusion detection algorithm shown in Figure 45. The algorithm is implemented inside the Modbus_decode.c file.

```
/* Get all the fields and perform normalization */


        int ipV4_src[4];
        char stringSrcIp[16];
        int SrcIphash = 0;
        int i;

        uint32_t src_addr = packet->ip4_header->source.s_addr;
        ipV4_src[0] = ((src_addr & 0x0FF000000) >> 24) & 0xFF; /* Get Source Ip */
        ipV4_src[1] = ((src_addr & 0x0FF0000) >> 16) & 0xFF;
        ipV4_src[2] = ((src_addr & 0x0FF00) >> 8) & 0xFF;
        ipV4_src[3] = ((src_addr & 0x0FF)) & 0xFF;

           _dpd.logMsg("\tSource Ip address:
        %d.%d.%d.%d",ipV4_src[3],ipV4_src[2],ipV4_src[1],ipV4_src[0]);

        sprintf(stringSrcIp,"%d.%d.%d.%d",ipV4_src[3],ipV4_src[2],ipV4_src[1],ipV4_src[0]);

        SrcIphash = 0;

        for (i = 0; i < strlen(stringSrcIp); i++) {
           SrcIphash = 31 * SrcIphash + stringSrcIp[i];
        }
        _dpd.logMsg("Src Ip Hash code: %d", SrcIphash);

        double SrcIphash_norm = (SrcIphash - SrcIp_mean) / SrcIp_dev;
        double SrcIphash_final = SrcIphash_norm * SrcIp_wt;

int ipV4_dst[4];
```

```
        char stringDstIp[16];
        int DstIphash = 0;
        int i1;

        uint32_t dst_addr = packet->ip4_header->destination.s_addr;
        ipV4_dst[0] = ((dst_addr & 0x0FF000000) >> 24) & 0xFF; /* Get Dest Ip */
        ipV4_dst[1] = ((dst_addr & 0x0FF0000) >> 16) & 0xFF;
        ipV4_dst[2] = ((dst_addr & 0x0FF00) >> 8) & 0xFF;
        ipV4_dst[3] = ((dst_addr & 0x0FF)) & 0xFF;

        _dpd.logMsg("\tDestination Ip address:
%d.%d.%d.%d",ipV4_dst[3],ipV4_dst[2],ipV4_dst[1],ipV4_dst[0]);
        sprintf(stringDstIp,"%d.%d.%d.%d",ipV4_dst[3],ipV4_dst[2],ipV4_dst[1],ipV4_dst[0]);
        DstIphash = 0;
        for (i1 = 0; i1 < strlen(stringDstIp); i1++) {
           DstIphash = 31 * DstIphash + stringDstIp[i1];
        }
        _dpd.logMsg("\nDest Ip Hash code: %d", DstIphash);

        double DstIphash_norm = (DstIphash - SrcIp_mean) / DstIp_dev;
        double DstIphash_final = DstIphash_norm * DstIp_wt;

        int Srcport = packet->src_port;
        double src_port_norm = (packet->src_port - SrcPort_mean) / SrcPort_dev;
        double src_port_final = src_port_norm * SrcPort_wt;
        _dpd.logMsg("\nSrc Port: %d", Srcport);

        int Dstport = packet->dst_port;
        double dst_port_norm = (packet->dst_port - DstPort_mean) / DstPort_dev;
        double dst_port_final = dst_port_norm * DstPort_wt;
        _dpd.logMsg("\nDst port:%d", Dstport);

        int TransID = header->transaction_id;
        double trans_id_norm = (header->transaction_id - TransIdentif_mean) / TransIdentif_dev;
        double trans_id_final = trans_id_norm * TransIdentif_wt;
        _dpd.logMsg("\nTransaction ID: %d", TransID);

        int FuncCode = header->function_code;
        double funcCode_norm = (header->function_code - FuncCode_mean)/ FuncCode_dev;
        double func_code_final = funcCode_norm * FuncCode_wt;
```

```
        _dpd.logMsg("\nFunction code: %d", FuncCode);

        double Refno_final = 0.0;
        double Resp_data_final = 0.0;
        double Write_data_final = 0.0;

    if(header->function_code == 0x03)   /* Check if Function Code is 3 */
        {
            if (packet->flags & FLAG_FROM_CLIENT)/* Check if it is a request packet or not */
             {
                    ReadReqheader = (ReadReg_Reqheader_t *) packet->payload;

                    uint16_t Refno = ReadReqheader->Refno;
                    Refno = ((Refno & 0xFF00) >> 8) | ((Refno & 0xFF) << 8);
                    double Refno_norm = (Refno - Refno_mean)/ Refno_dev;
                    Refno_final = Refno_norm * Refno_wt;
                    _dpd.logMsg("\nFC 3 Req, Refno: %d", Refno);

                    Resp_data_final = 0.000;
                    _dpd.logMsg("\nFC 3 Req, Resp data final:%f ",Resp_data_final);

                    Write_data_final = 0.000;
                    _dpd.logMsg("\nFC 3 Req, Write data final: %f",Write_data_final);

             }
             else
             {
                    ReadRespheader = (ReadReg_Respheader_t *) packet->payload;

                    uint16_t Resp_data = ReadRespheader->Resp_data;
             //     Resp_data = ((Resp_data & 0xFF00) >> 8) | ((Resp_data & 0xFF) << 8);
                    double Resp_data_norm = (Resp_data - RespData_mean) / RespData_dev;
                    Resp_data_final = Resp_data_norm * RespData_wt;
                    _dpd.logMsg("\nFC 3 Resp, Resp_data : %d", Resp_data);

                    Refno_final = 0.000;
                    _dpd.logMsg("\nFC 3 Resp, Refno final: %f", Refno_final);
                    Write_data_final = 0.000;
             }
        }
```

```
   else if (header->function_code == 0x10) /* Func code 16 */
         {

               if (packet->flags & FLAG_FROM_CLIENT)   /* Check if it is a request packet or
not */
               {
                     WriteReqheader = (WriteReg_Reqheader_t *) packet->payload;

                     uint16_t  Refno = WriteReqheader->Refno;
                     Refno = ((Refno & 0x0FF00) >> 8) | ((Refno & 0x0FF) << 8);
                     double Refno_norm = (Refno - Refno_mean) / Refno_dev;
                     Refno_final = Refno_norm * Refno_wt;
                     _dpd.logMsg("\nFC 16 Req, Refno: %d", Refno);

                     uint16_t Write_data = WriteReqheader->Write_data;
               //    Write_data = ((Write_data & 0xFF00) >> 8) | ((Write_data & 0xFF) << 8);
                     double Write_data_norm = (Write_data - WriteData_mean) / WriteData_dev;
                     Write_data_final = Write_data_norm * WriteData_wt;
                     _dpd.logMsg("\nFC 16 Req, Write data: %d", Write_data);

                     Resp_data_final = 0.000;
                     _dpd.logMsg("\nFC 16 Req, Resp_data_final : %f",Resp_data_final);
               }
   else
               {
                     WriteRespheader = (WriteReg_Respheader_t *) packet->payload;

                     uint16_t Refno = WriteRespheader->Refno;
                     Refno = ((Refno & 0x0FF00) >> 8) | ((Refno & 0x0FF) << 8);
                     double Refno_norm = (Refno - Refno_mean) / Refno_dev;
                     Refno_final = Refno_norm * Refno_wt;
                     _dpd.logMsg("\nFC 16 Resp, Refno: %d", Refno);

                     Write_data_final = 0.000;
                     _dpd.logMsg("\nFC 16 Resp, Write_data_final: %f", Write_data_final);

                     Resp_data_final = 0.000;
                     _dpd.logMsg("\nFC 16 Resp, Resp_data_final: %f", Resp_data_final);
```

```
                }
        }

        double final_sum = (SrcIphash_final + DstIphash_final + src_port_final + dst_port_final +
trans_id_final + func_code_final + Refno_final + Write_data_final + Resp_data_final);

        _dpd.logMsg("\nFinal sum: %f", final_sum);


   if (final_sum > 0.5)
        {
                _dpd.logMsg("\n######### This is a attack packet based on final sum
########\n");
                _dpd.alertAdd(GENERATOR_SPP_MODBUS, MODBUS_ATTACK_PACKET, 1,
0, 3,
                    MODBUS_ATTACK_PACKET_STR, 0);

        }
```