*Proceedings of the 7th International Conference of Control Systems, and Robotics (CDSR'20)*
*~~Niagara Falls, Canada~~ Virtual Conference – November, 2020*
*Paper No. 127*
*DOI: 10.11159/cdsr20.127*

# Covert Communication Using MODBUS Protocol in IoT Devices

**Sashaa Nagrikar[1], Saeed Alshahrani[2], Daryl Johnson[3]**
[1]Rochester Institute of Technology
1 Lomb Memorial Drive, Rochester, US
sn1945@rit.edu; sa7762@rit.edu; daryl.johnson@rit.edu

***Abstract -*** Internet of Things (IoT) is a part of Cyber Science that has been gaining popularity exponentially. IoT are generally referred to as smart devices since the carry out their operations with minimal human intervention. The IoT devices are connected to each other via a device such as a centralized modem. Through this method, IoT helps provide an easier life for its consumers. Even so, these smart devices are flawed and face privacy challenges and can be exploited at the physical level to obscurely perform information exchange that they are not intended to do. This is known as a covert channel. By definition, a covert channel is some form of a medium which is used by exploiting its functionalities to secretly send and receive messages which they are not originally programmed to do so. Hence following the above definition, "MODBUS Protocol" was chosen to be used as a communication protocol in a Master-Slave model for a covert channel. The MODBUS protocol uses a Master and Slave system model where the Master sends functional instructions to the slaves and the slaves return the output corresponding to the instruction. By exploiting this feature of the Master-Slave architecture, we have built a covert channel wherein the receiver maps each character of the covert message into an instruction and sends it to the slave and the slave strips off the data in that instruction and sends it to the intended receiver, where the receiver maps the instruction back to the character and prints out the message.

***Keywords*: MODBUS, Master, Slave**

## 1. Introduction

According to an IoT tech news source, "Sales of IoT cellular devices will approach 350 million per year by 2025" [1]. This shows that the upcoming expectations of the IoT devices aim at convenience for people to use. However, there is a known fact in the world of Cyber Science that "Security and Convenience are related in indirect proportionality". More convenience, lesser is security. By taking advantage of this fact, we came up with a covert channel to abuse the functionalities of an IoT device by using MODBUS protocol since IoT is built using protocols that help communicate within centralized networks. A covert channel is a form of a communication medium that is kept hidden so that messages could be sent and received in secrecy. In order to do so, the original functionalities of an API are programmed in a way to send and receive messages thereby abusing it. For instance, if a person commands Amazon Echo to switch on a light bulb, it must complete only that particular functionality. However, this paper talks about how messages can be sent and received along with switching ON and OFF a light bulb in an IoT device.

Nowadays, IoT devices are operated over a wireless internet connection (WiFi) and these connections are established using a wireless protocol. For example, MQ Telemetry Transport (MQTT) helps in transmitting packets between IoT devices via remote locations [3]. Instead of MQTT, for our research we have used the MODBUS protocol to communicate with the IoT device. The purpose of using MODBUS protocol in an IoT device is that it is not easy to understand and even harder to implement. Supervisory Control and Data acquisition (SCADA) systems have adopted this protocol because of its easy operability but not yet implemented in IoT systems. Hence, such MODBUS requests could escape through packet analyzers in current times.

MODBUS has two types of communications which are query/response and broadcast [4]. The covert channel in this paper uses the first approach. In the query/response, the communication is between the Master (user client) and the Slave (server-IoT device). The Master initiates the communication by requesting for the current status of the light bulbs (read commands) and the Slave replies back with the status. The Master could also request to change the current status of the light bulbs from On to OFF or vice-versa (write commands) and the Slave follows the functionality and switches OFF or ON a light bulb accordingly. Another example would be that the user wants to turn the heater up; he would use the device application to initiate the request to the centralized modem which is (Master) to pass the request packets to the IoT heater (Slave). The full establishment will be via MODBUS functions.

## 1.1 MODBUS Protocol

The MODBUS packet is encapsulated inside a TCP/IP frame which consists of the IP address of the slave (the IoT device) and its standard MODBUS port number 502 with which a TCP connection is established. MODBUS protocol uses a connection-oriented communication which consists of the two most important functional parameters: the executable function number and the register data, in order to execute instructions

The most prominently used functions for communication are read coils (0x01), Write Single Coil (0x05) and Write Multiple Coils (0x0F)[4].

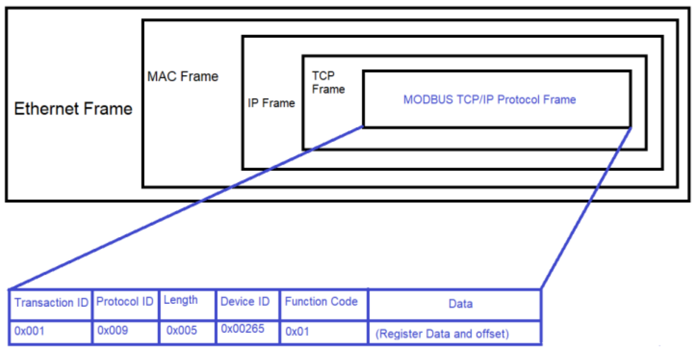


Fig 1: MODBUS TCP/IP Packet

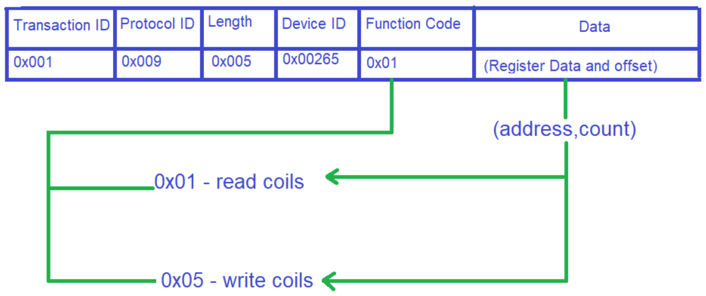


Fig 2: MODBUS TCP/IP Function

Based on these function numbers, an IoT device performs the operations. These functions are given (address, count) values whenever a function is required to be performed. The address parameter contains the number addressed to a specific light bulb that one wants to access. This parameter ranges from 0 to 9, since we have used 10 light bulbs for this research paper. The address value can be increased from more than 9 if more devices are added to the circuit. The count parameter only uses 2 values - 0 for OFF and 1 for ON.

For example:

pi.read_coils(1,1).bits: This command reads the value of the bulb number 2 to see if it is switched ON.

pi.write_coil(2,1): This command will switch ON the light bulb number 3.

Our covert channel has been developed to only and only use the "read" function for our commands list. The reason being that "read" instructions allow one to set the "count" value for more than 1. The slave uses this functionality to send out garbage values back to the master but the slave is still entitled to perform its functions. On the other hand, the "write" function does not accept any value beside "0" or "1". For any other value it sends out an error message; hence, to separate the legitimate overt channel from the covert channel, our working model depends on using "read" functions only.

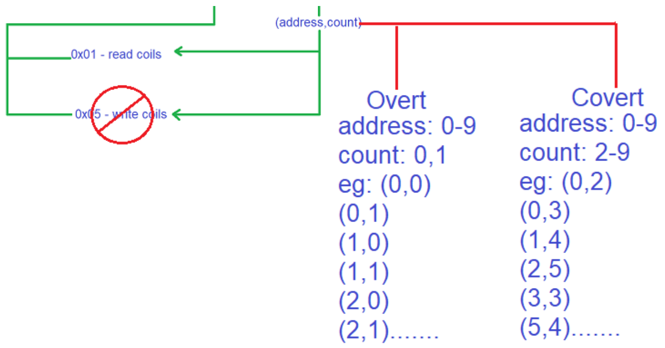


Fig 3: MODBUS TCP/IP Options

### 1.2 Working Model

The covert channel is built using 2 Masters and 1 Slave. To simplify, there will be 2 clients (covert sender and covert receiver) and one server (IoT device). For the IoT device, this working model uses Raspberry Pi 3 (RPi3). The code to process received instructions from the master to switch ON and OFF a bulb has been fed into the RPi3. The RPi3 is connected to the circuit according to Figure 5.

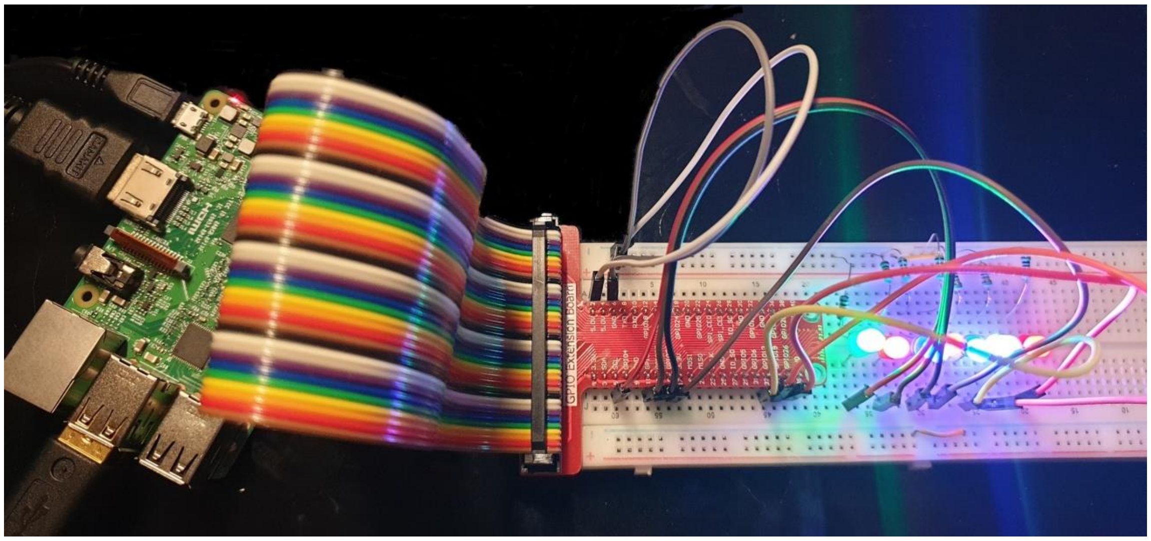


Fig 4: Raspberry Pi 3 Circuit

Under normal circumstances, if Master 1 wishes to communicate with its Slave, it will generate a MODBUS packet containing the requested function and (address, count) value and send it as a query to the RPi3. Using the above example: pi.read_coils (1,1).bits, the function number will be 0x01, the (address, count) value is (1,1). Once the query is received by the Slave, it will start peeling off the packet, layer-by-layer, to finally reach the MODBUS TCP/IP packet. It checks for all the fields in the MODBUS packet and fulfils the query request with an appropriate response sent back to Master 1. Figure 6 shows the process clearly.

## 2. Our Covert Channel

Based on the above working model, we made a few changes in the working code that programmed the Slave's functionalities.

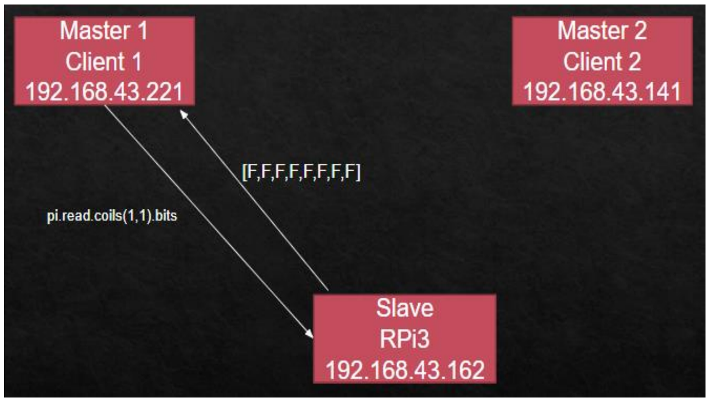


Fig 5: Working Model

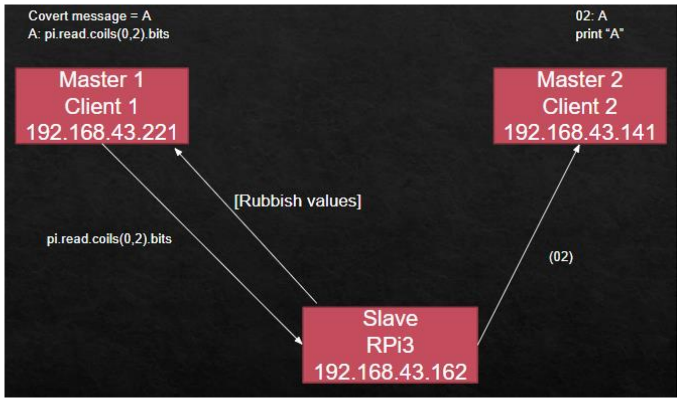


Fig 6: Covert Model

- **Step 1:** Enter the covert message: For this research paper, only block letters, 0-9 digits, space and period "." were fed into the system. The number of possible characters increases as the number of devices connected to the Slave increases.
- **Step 2:** Every character is mapped to a read function. This read function is fed with garbage (address, count) values. A combination of these values is the actual message here. The reason the "read" function was chosen is that RPi3 responded to garbage read requests but not garbage "write" requests. Hence, the entire mapping of characters is based on garbage "read" requests only.
- **Step 3:** A legitimate MODBUS TCP/IP packet with the embedded read function is sent to Slave the IP address of Master 1. After looking at the packet captures in Wireshark, they only seem to be legitimate read requests initiated from Master 1. Hence, nothing suspicious is noticeable.
- **Step 4:** After the MODBUS packet reaches the Slave, it analyses the packet structure and data. The Slave specifically computes the value of (address, count). Since the count is neither 0 nor 1, it sends back rubbish values to Master 1. Meanwhile, according to the modified code, the Slave strips off the values of (address, count) and sends those as a string to the IP address of Master 2 as shown in Figure 5.
- **Step 5:** Master 2 is already running the listener for the values sent by the Slave. The values reach Master 2 and are further computed. A similar version of the mapping is programmed at Master 2 which maps the received string into a character. And thus, the covert message is received.

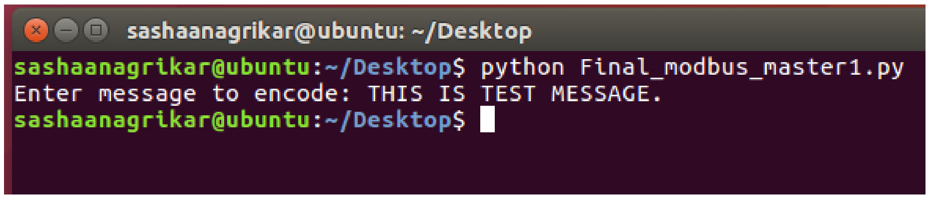
### 2.1 Elements of a Covert Channel

- **Medium:** the "read" function acts as a medium to generate the covert message.
- **Modulation:** the (address, count) values act as modulated values which represents each character
- **Encoding:** at the receiver's side, the covert message is encoded (mapped) into a series of "read" functions with the modulated values whereas on the receiver's side, the modulated values are encoded back into the corresponding characters.

## 3. Implementation

- **Master 1:** When the program is run, it asks for a covert message. The covert message is fed in by the covert sender as follows:

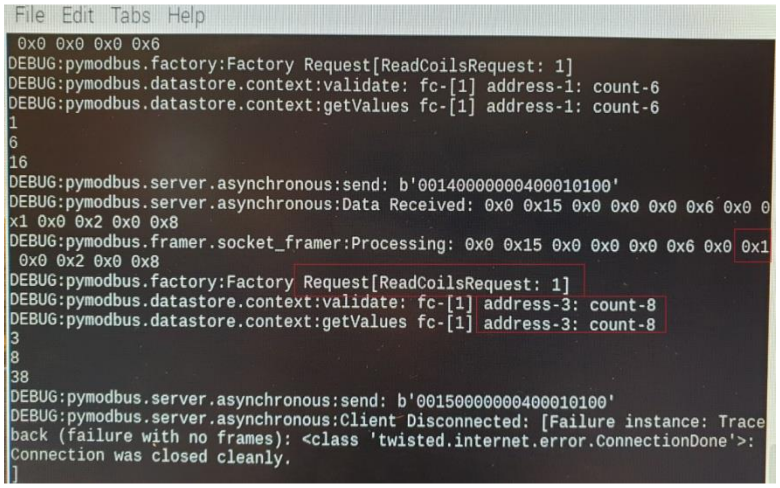
Once the message is entered, the Master 1 uses an interval of time to map the characters to read commands until the entire message is sent.



```
sashaanagrikar@ubuntu: ~/Desktop
sashaanagrikar@ubuntu:~/Desktop$ python Final_modbus_master1.py
Enter message to encode: THIS IS TEST MESSAGE.
sashaanagrikar@ubuntu:~/Desktop$
```

Fig 7: Covert Sender Master - 1

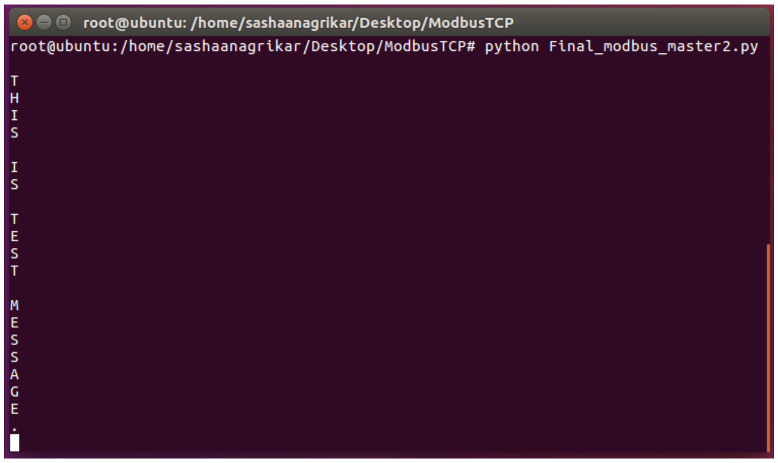
- **Slave:** At the server side, the Slave receives the read requests and generates packets accordingly. It also strips out the (address, count) value and sends it to Master 2. The image shows that a "read" function is requested having the function code 0x01. The (address, count) are stripped and shown separately and then concatenated into a string. When the message is complete, the connection is closed.



```
 0x0 0x0 0x0 0x6
DEBUG:pymodbus.factory:Factory Request[ReadCoilsRequest: 1]
DEBUG:pymodbus.datastore.context:validate: fc-[1] address-1: count-6
DEBUG:pymodbus.datastore.context:getValues fc-[1] address-1: count-6
1
6
16
DEBUG:pymodbus.server.asynchronous:send: b'00140000000400010100'
DEBUG:pymodbus.server.asynchronous:Data Received: 0x0 0x15 0x0 0x0 0x0 0x6 0x0 0
x1 0x0 0x2 0x0 0x8
DEBUG:pymodbus.framer.socket_framer:Processing: 0x0 0x15 0x0 0x0 0x0 0x6 0x0 0x1
 0x0 0x2 0x0 0x8
DEBUG:pymodbus.factory:Factory Request[ReadCoilsRequest: 1]
DEBUG:pymodbus.datastore.context:validate: fc-[1] address-3: count-8
DEBUG:pymodbus.datastore.context:getValues fc-[1] address-3: count-8
3
8
38
DEBUG:pymodbus.server.asynchronous:send: b'00150000000400010100'
DEBUG:pymodbus.server.asynchronous:Client Disconnected: [Failure instance: Trace
back (failure with no frames): <class 'twisted.internet.error.ConnectionDone'>:
Connection was closed cleanly.
]
```

Fig 8: RPi3

- **Master 2:** The Master 2 side that is currently in listener mode accepts the strings being sent from the slave and maps them back into characters and prints the output of every character as shown in figure 10.



```
root@ubuntu: /home/sashaanagrikar/Desktop/ModbusTCP
root@ubuntu:/home/sashaanagrikar/Desktop/ModbusTCP# python Final_modbus_master2.py

T
H
I
S

I
S

T
E
S
T

M
E
S
S
A
G
E
.
```

Fig 9: Covert Receiver Master - 2

## 4. Analysis

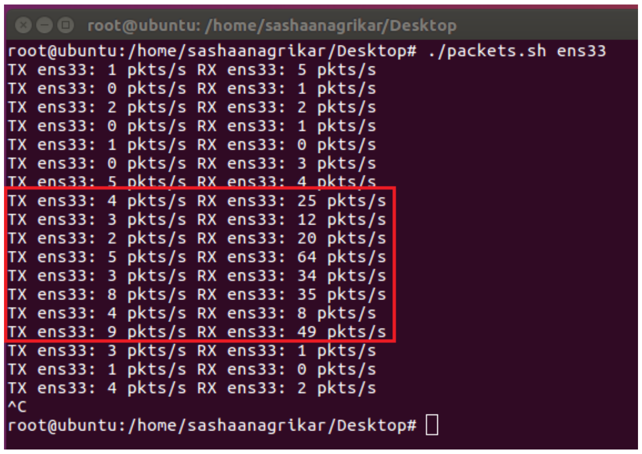
### 4.1 Packets per second

The above message consisted of 21 characters. In order to send and receive 21 characters, it took a total of 6-7 seconds. As shown in figure 11, the numbers of packets sent per second were 3-5 packets.

The above image shows the number of packets transmitted (TX) and received (RX). However, the maximum packets sent vary from 2 to 9 and the number of received packets varies from 12 to 64. Since every request requires a TCP 3 way handshake to be established, the number of transmitted and received packets increases.

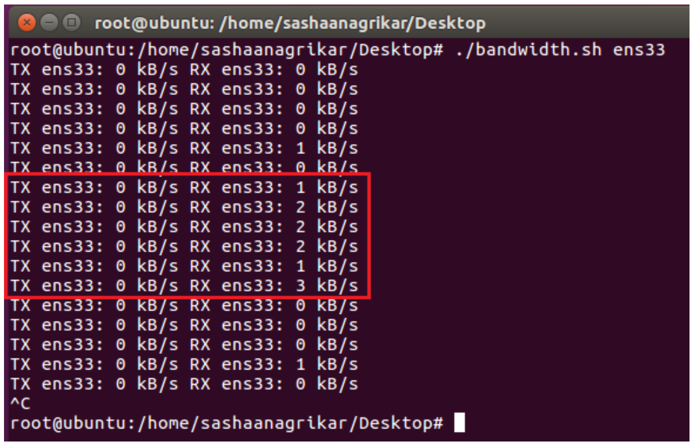
### 4.2 Bandwidth

Even with such a noisy channel/interface, the bandwidth required is very low. The analysis shows that a maximum of 3 kb/s of bandwidth is used to send the packets. It can be seen in figure 12.



```
root@ubuntu:/home/sashaanagrikar/Desktop# ./packets.sh ens33
TX ens33: 1 pkts/s RX ens33: 5 pkts/s
TX ens33: 0 pkts/s RX ens33: 1 pkts/s
TX ens33: 2 pkts/s RX ens33: 2 pkts/s
TX ens33: 0 pkts/s RX ens33: 1 pkts/s
TX ens33: 1 pkts/s RX ens33: 0 pkts/s
TX ens33: 0 pkts/s RX ens33: 3 pkts/s
TX ens33: 5 pkts/s RX ens33: 4 pkts/s
TX ens33: 4 pkts/s RX ens33: 25 pkts/s
TX ens33: 3 pkts/s RX ens33: 12 pkts/s
TX ens33: 2 pkts/s RX ens33: 20 pkts/s
TX ens33: 5 pkts/s RX ens33: 64 pkts/s
TX ens33: 3 pkts/s RX ens33: 34 pkts/s
TX ens33: 8 pkts/s RX ens33: 35 pkts/s
TX ens33: 4 pkts/s RX ens33: 8 pkts/s
TX ens33: 9 pkts/s RX ens33: 49 pkts/s
TX ens33: 3 pkts/s RX ens33: 1 pkts/s
TX ens33: 1 pkts/s RX ens33: 0 pkts/s
TX ens33: 4 pkts/s RX ens33: 2 pkts/s
^C
root@ubuntu:/home/sashaanagrikar/Desktop#
```

Fig 10: Number of “read” requests sent per second



```
root@ubuntu:/home/sashaanagrikar/Desktop# ./bandwidth.sh ens33
TX ens33: 0 kB/s RX ens33: 0 kB/s
TX ens33: 0 kB/s RX ens33: 0 kB/s
TX ens33: 0 kB/s RX ens33: 0 kB/s
TX ens33: 0 kB/s RX ens33: 0 kB/s
TX ens33: 0 kB/s RX ens33: 1 kB/s
TX ens33: 0 kB/s RX ens33: 0 kB/s
TX ens33: 0 kB/s RX ens33: 1 kB/s
TX ens33: 0 kB/s RX ens33: 2 kB/s
TX ens33: 0 kB/s RX ens33: 2 kB/s
TX ens33: 0 kB/s RX ens33: 2 kB/s
TX ens33: 0 kB/s RX ens33: 1 kB/s
TX ens33: 0 kB/s RX ens33: 3 kB/s
TX ens33: 0 kB/s RX ens33: 0 kB/s
TX ens33: 0 kB/s RX ens33: 0 kB/s
TX ens33: 0 kB/s RX ens33: 0 kB/s
TX ens33: 0 kB/s RX ens33: 1 kB/s
TX ens33: 0 kB/s RX ens33: 0 kB/s
^C
root@ubuntu:/home/sashaanagrikar/Desktop#
```

Fig 11: Bandwidth used to send the packets

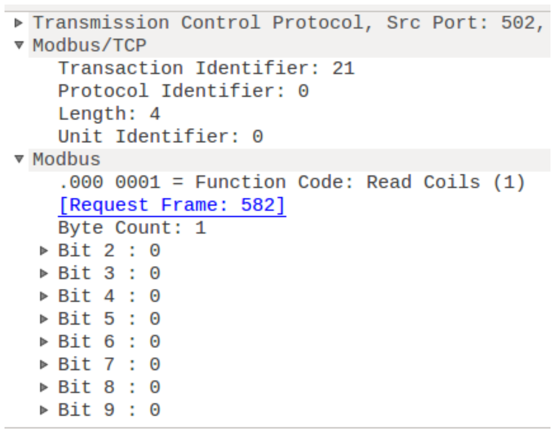
### 4.3 Detectability

The "read" requests are flooded into the network during character mapping; hence network packet analyzers like Wireshark are unable to capture all the read requests. Every MODBUS packet that is captured is very similar to the legitimate "read" request packet. Hence, it is difficult to decipher between legitimate and garbage "read" requests. However, if one analyzes the packets received by the Slave from the Slave side, one can easily differentiate garbage read requests from the legitimate ones since the "count" value in (address, count) would be more than 1. However, in order to analyze the packets, the packet analyzer must be physically connected to the Slave which is never the usual case.

Under normal circumstances, there are no packet captures physically connected to the Slave side. As a result, it is estimated that this covert channel escapes detectability by over 90% of the time.

### 4.4 Robustness

This covert channel uses the "read" functions to circulate covert data among the masters and slaves. Since, from the analysis, it was seen that that slave provides from the higher priority for the "read" functions over other functions, the robustness level is maintained from medium to high. Even after the detection of the covert channel, it is not likely to remove the malware infested configurations from the IoT device. Along the similar lines, the network packet analyzers process the read requests from the covert channel as legitimate requests, hence network filters also make little to no restrictions for the "read" requests.



```
▶ Transmission Control Protocol, Src Port: 502,
▼ Modbus/TCP
    Transaction Identifier: 21
    Protocol Identifier: 0
    Length: 4
    Unit Identifier: 0
▼ Modbus
    .000 0001 = Function Code: Read Coils (1)
    [Request Frame: 582]
    Byte Count: 1
  ▶ Bit 2 : 0
  ▶ Bit 3 : 0
  ▶ Bit 4 : 0
  ▶ Bit 5 : 0
  ▶ Bit 6 : 0
  ▶ Bit 7 : 0
  ▶ Bit 8 : 0
  ▶ Bit 9 : 0
```

Fig 12: Wireshark capture of Garbage "read" request same as legitimate "read" request from the sender PC

### 4.5 Prevention

The only way to prevent this covert channel is to buy a new device. Once the malware is installed into the slave, only through factory reset configurations the slave can be brought back to its original functionalities. Since, its detectability rate is less than 10%, it is very much likely to be ignored and hence prevention is averted.

## 5. Limitations

- The only characters used for this system model to generate a covert message are block letters from A to Z, digits from 0 to 9, space and period ".". For more characters and special symbols, more light bulbs must be added. Hence, more devices are connected to the IoT device; better crafted covert messages will be generated.
- If in any situation, if a legitimate user accidentally uses the count value as more than 1, it will get back rubbish values as its answer; however the covert message might be altered. This can be called a plausible human error which cannot be escaped at any point. In such cases, covert receivers might have to ask the sender to send the message again.
- The MODBUS protocol used in this research is the TCP/IP frame which does not include a CRC (message error check functionality) unlike MODBUS RTU model. Hence, there are no functions to check if the received covert message is correct.

## 6. Conclusion and Future Scope

MODBUS's mission is to establish a communication between devices working in the same network to secretly send information from one user to another. For our covert channel, we established communication between devices by abusing the basic "read" functionality of the system model. The reason this covert channel was implementable is because the functionality did not include any restrictions on the use of values outside their operating domain. The IoT device was not designed on how to respond to garbage requests. There are several IoT devices that are manufactured with such loopholes for the sole purpose of making them light on computing. Since IoT devices are not equipped with high computing capabilities, their programming is predominantly focused on "what to do" when a correct command is entered, rather including operations on a side note about "what to do" when garbage requests are received.

Since this covert channel was built on 10 LED bulbs on a breadboard which were being operated using Raspberry Pi 3, they are also implementable in worldly acceptable IoT devices such as a thermostat, IoT Christmas lights or even Amazon Echo. Once their functioning APIs are gained, their functionalities can be changed and inserted back into the devices very easily. This idea could also be extended to using the MODBUS RTU model for its functioning instead of the MODBUS TCP/IP model since RTU includes a CRC check in its packet. This would give more robustness to the covert message. One may also use different function numbers such as 0x02, 0x03, 0x04, 0x06, 0x07 and 0x08 other than the ones used in this research paper.

Creating a covert channel using the MODBUS protocol gives endless possibilities of escaping detection for a huge amount of time. From our analysis, we can state that MODBUS protocol shows a promising path towards covert communication.

## Acknowledgement

We would like to thank our guide Prof. Daryl Johnson for teaching us about Covert Channels and helping us in our articulating idea for a covert channel and providing us with the required hardware and Raspberry Pi3 to build our covert channel. We would also like to thank our Dean and Department Head for their financial support behind our conference publication.

## References

[1] News, IoT, and IoT News. "Iot Cellular Device Shipments To Approach 350M Per Year By 2025 – With Strong China Growth Noted". Iot Tech News, 2019, https://iottechnews.com/news/2019/nov/08/iot-cellular-device-shipments-approach-350m-year-2025-strong-china-growth-noted/
[2] "What Is Modbus? 14 Most Asked Questions - B&B Electronics". Bb-Elec.Com, 2019, https://www.bb-elec.com/Learning-Center/All-White-Papers/Modbus/The-Answer-to-the-14-Most-Frequently-Asked-Modbus.aspx
[3] "MQTT". Mqtt.Org, https://mqtt.org/.
[4] Dartmouth College, I3P, and University Tulsa. "Critical Infrastructure Protection III | Springerlink". https://link.springer.com/book/10.1007%2F978-3-642-04798-5
[5] Worldcomp-Proceedings.Com http://worldcomp-proceedings.com/proc/p2014/SAM4151.pdf
[6] Pymodbus Documentation: https://pymodbus.readthedocs.io/en/latest/index.html