PERFORMANCE EVALUATION OF MODBUS TCP IN NORMAL OPERATION AND UNDER A DISTRIBUTED DENIAL OF SERVICE ATTACK

Eric Gamess[1], Brody Smith[1], and Guillermo Francia III[2]

[1]MCIS Department, Jacksonville State University, Jacksonville, AL, USA
[2]Center for Cybersecurity, University of West Florida, Pensacola, FL, USA

ABSTRACT

Modbus is the de facto standard communication protocol for the industrial world. It was initially designed to be used in serial communications (Modbus RTU/ASCII). However, not long ago, it was adapted to TCP due to the increasing popularity of the TCP/IP stack. Since it was originally designed for controlled serial lines, Modbus does not have any security features. In this paper, we wrote several benchmarks to evaluate the performance of networking devices that run Modbus TCP. Parameters reported by our benchmarks include: (1) response time for Modbus requests, (2) maximum number of requests successfully handled by Modbus devices in a specific amount of time, and (3) monitoring of Modbus devices when suffering a Distributed Denial of Service attack. Due to the growing adoption of IoT technologies, we also selected two widely known and inexpensive development boards (ESP8266 and Raspberry Pi 3 B+/OpenPLC) to realize a performance evaluation of Modbus TCP.

KEYWORDS

Modbus, Internet of Things, Programmable Logic Controllers, Security, DDoS, Network Evaluation, Benchmark.

1. INTRODUCTION

Modbus [1-5] is the most commonly used protocol for industrial control systems. It was initially designed for serial connections (RS232 and RS485), with no security features. The original specification included two possible transmission modes: Remote Terminal Unit (RTU) and ASCII. Modbus TCP [6][8] is a much more recent development, created to allow Modbus RTU/ASCII protocols to be carried over TCP/IP-based networks.

Modbus RTU/ASCII was not developed with security in mind. Since the initial idea was a protocol for serial links, only a few people would have access to the devices and links; and in general, they would be technicians working on the project. This setup essentially enables the security weakness to withstand malicious attacks. Modbus TCP mimics as close as possible Modbus RTU/ASCII. Hence, no security elements were added to the protocol, leaving Modbus more vulnerable since it is now accessible remotely over TCP/IP.

Huitsing, Chandia, Papa, and Shenoi [21] presented a compilation of attacks for the Modbus RTU/ASCII and Modbus TCP protocols. A total of 33 attacks were identified: 5 focused on the serial versions, 13 for the TCP version, and 15 for both versions. Several research groups have been working on evaluating and improving the security of Modbus RTU/ASCII [22][23] and Modbus TCP [10][20][24]. In this paper, we introduce some benchmarks to evaluate Modbus TCP devices in normal operation and when they are under a Distributed Denial-of-Service (DDoS) attack. Our benchmarks report results on the response time for a Modbus request, the maximum number of requests handled by a Modbus device within a given time interval, and the

behavior of a Modbus device when facing a DDoS attack. Also, we chose two Modbus devices (ESP8266 [26-28] and Raspberry Pi 3 B+ [46] with OpenPLC [29-31]) and made a performance evaluation.

The rest of the paper is structured as follows. Section 2 discusses the related work. An introduction to Modbus is made in Section 3. We present our benchmarking tools for the evaluation of the Modbus protocol in Section 4. The description of the testbeds and the results of the evaluation of different Modbus devices in normal operation and under a DDoS attack are in Section 5. Finally, Section 6 concludes the paper and gives directions for future work.

2. RELATED WORK

The evaluation of network protocols has been well covered by the community. Several techniques are used for the assessment, such as modeling, simulations, and running benchmarks in a controlled environment. Sankaran [14] developed an analytical framework to model the performance of the network in the Internet of Things (IoTs) using Markov chains. This work helps in the detection of bottlenecks, the design of optimal sleep/wake-up schedules, and the performance tuning of applications. In [15], Shah and Mustari assessed the performance of the Enhanced Distributed Channel Access Function (EDCAF) of IEEE 802.11p for vehicular networks with an analytical model. The authors considered the major factors that could alter performance, including the four access categories (ACs). Salehi, Boukerche, and Darehshoorzadeh [16] studied and modeled the behavior of malicious nodes in a wireless mesh network based on unicast opportunistic routing protocols, using Discrete-Time Markov Chain (DTMC). They also proposed a new approach for packet drop ratio estimation, through which the negative impacts of uncooperative nodes can be quantified.

Ahmed, Mustafa, and Ibrahim [11] used OPNET Modeler to evaluate the performance of native IPv4, native IPv6, and 6to4 tunnels, in terms of response time, throughput, and packet loss. In [12], the authors evaluated the performance of three well-known routing protocols (ADV, AODV, and DSDV) in vehicular networks, using NCTUns [32], a popular network simulator. They varied different metrics such as the number of nodes, traffic patterns, and number of Road-Side Units (RSUs) to analyze their impact on the selected routing protocols. They reported results such as the Packet Delivery Ratio (PDR), average end-to-end delay, average number of hops, and Normalized Routing Load (NRL). Jameel and Shafiei [13] did a QoS performance evaluation of voice over LTE networks, using OMNeT++ [33] and SimuLTE [34], in three different scenarios. For all the scenarios, they reported results for the Mean Opinion Score (MOS), end-to-end delay, jitter, and packet loss rate.

Abdullah, Al-awad, and Hussein [17] did a performance evaluation of different open-source SDN controllers (libfluid, ONOS, Open Daylight, POX, and Ryu) in a linear topology that was built with the Mininet emulator, with different numbers of switches. The performance evaluation was done using Iperf (a well-known network benchmarking tool) and the “ping” command. In [18], the authors conducted a performance evaluation of three VPNs (PPTP, IPSec, and SSTP) in a Client/Server network environment over wired (Ethernet) and wireless media (IEEE 802.11ac) using both IPv4 and IPv6. To generate packet flows, the authors used Iperf in that study.

As we can see, a lot of work has been done in different areas of networks. However, there are just a few studies focused on protocols for industrial control systems, such as Modbus. Kim, Lee, and Choi [9] evaluated the performance of Modbus TCP and focused their work on the response time when changing the number of nodes and topology. They used simulations in their study, with ns-3

[35] as the network simulator. In the area of security for Modbus TCP, the authors of [19] used Snort [36], a well-known network intrusion detection and prevention system, to detect a flooding attack. Chen, Pattanaik, Goulart, Butler-Purry, and Kundur [20] studied a Man-in-the-Middle (MITM) attack and a DoS attack over the Modbus TCP protocol. To do so, they developed a testbed with a real-time power system simulator and a communication system simulator. The power system simulation was based on the power grid simulator of RTDS Technologies with LabVIEW. The communication system simulation was implemented using OPNET's System-in-the-Loop (SITL) simulator and open-source Linux tools. In [10], the authors proposed a model for Modbus TCP for intrusion detection based on Deterministic Finite Automatons (DFAs).

To the best of our knowledge, there is no benchmarking tool to assess the performance of Modbus TCP neither under normal operation nor during a DDoS attack.

3. AN OVERVIEW OF MODBUS

Modbus [1-5] is a communications protocol developed by Modicon Systems in 1979. Originally, Modbus was intended to be used with Programmable Logic Controllers (PLCs) via serial transitions and has since become the communication standard for electronic industrial machines. More recently, Modbus was extended to be used over TCP/IP connections also [6]. It is openly published; therefore, Modbus can be used freely by anyone.

The serial version of Modbus has two modes of transmission. The first mode, also known as ASCII mode, is not very popular and requires the transmission of two ASCII characters for each byte of data. The second mode, or RTU mode, has a greater character density, allowing better data throughput than ASCII for the same baud rate. Hence, it is more frequently used than ASCII. Modbus is based on a master-slave principle in its serial version, where only one master and up to 247 slaves are allowed on a network. A master can initiate a request via unicast or broadcast. The requests can be a query of information (e.g., read input register 5) or an action to be performed (e.g., write '1' to coil 8). When a slave receives a unicast request from the master, the slave will either respond with the information requested or by performing an action, then sending a confirmation back to the master. Most of the time, the slave's confirmation is exactly the same as the master's request. Slaves ignore malformed requests and do not send confirmations for broadcast requests. This prevents the master from potentially being flooded with 247 confirmations.

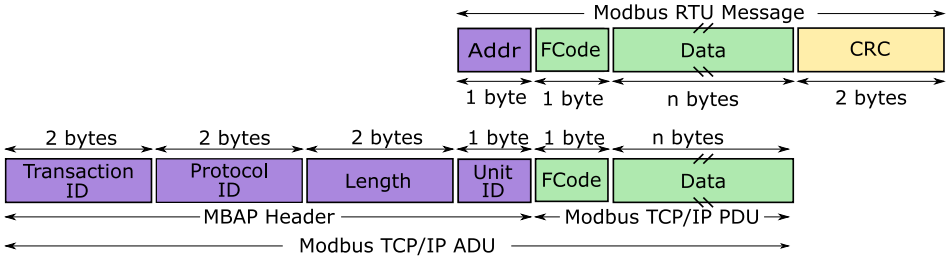

Figure 1. Modbus RTU Message vs. Modbus TCP/IP ADU

Figure 1 depicts a Modbus RTU message for serial communications and a Modbus TCP/IP Application Data Unit (ADU). There are four fields in a Modbus RTU message: (1) Address, (2) Function Code, (3) Data, and (4) CRC. The address field is 1-byte long and it is where the master specifies the address of the slave to be queried. The range for the address of a single device is 1-247. A slave that is replying to the master will respond with its own address as a way of telling

the master which slave sent the response. If the address field contains 0, this request will be a broadcast. As mentioned before, slaves may perform the required action; however, they do not respond with a confirmation. Addresses in the range 248-255 are reserved. The function code (FCode) field is 1-byte long and it is where the master specifies what function the slave should perform. The function code ranges from 1-255, though not all 255 values are used. The most useful function codes are 1-6, 15, and 16 (see Table 1). A responding slave will copy the function code of the action performed in the confirmation to the master. This validates that the slave understood the function. Note that the most significant bit of the function code is used to report errors. That is, when a problem occurs in the slave while executing the required action, the function code is echoed with this bit set to a logic 1, as a way to signal a problem to the master.

Table 1. Common Modbus Operations

FCode	**Operation**	**Reference**
0x01	Read Coils (Input/Output Bits)	0xxxx
0x02	Read Discrete Inputs (Input Bits)	1xxxx
0x03	Read Holding Registers (Input/Output Registers)	4xxxx
0x04	Read Input Registers (Input Registers)	3xxxx
0x05	Write Single Coil (Input/Output Bit)	0xxxx
0x06	Write Single Holding Register (Input/Output Register)	4xxxx
0x0F	Write Multiple Coils (Input/Output Bits)	0xxxx
0x10	Write Multiple Holding Registers (Input/Output Registers)	4xxxx
0x16	Mask Write Holding Register (Input/Output Register)	4xxxx
0x17	Read/Write Holding Registers (Input/Output Registers)	4xxxx

The data field is of variable length. When a master sends a request, the data field contains additional information that the slave must use to take the proper action as specified in the function code. This can include items like bit and register addresses, the number of items to be handled, and the value to be written in those bits and registers (if a “write” operation is performed). In a response to a “read” request, the data field usually consists of the value obtained from those bits and registers. Lastly, the Cyclical Redundancy Check (CRC) field is 2-byte long and ensures that the Modbus packet was not corrupted during the transmission. Notice that an ASCII message is similar to an RTU message, but it has a longer length and the CRC field is called Longitudinal Redundancy Check (LRC).

When communicating over TCP/IP [6], the Modbus Application Protocol (MBAP) header provides four additional fields that are: (1) transaction identifier, (2) protocol identifier, (3) length, and (4) unit identifier. The transaction identifier is 2-byte long and it is used for the pairing of requests with their responses. It is established by the master and many times implemented as a counter that is incremented every time the master sends a new request. The protocol identifier is 2-byte long and it is always set to 0 when using Modbus. The field “length” is 2-byte long and refers to the number of bytes required for the unit identifier, function code, and data fields. The address field of a Modbus RTU message was renamed “Unit Identifier” or “Unit ID” in Modbus TCP. It is not really used in the TCP version of Modbus, since the destination IP address is the way to select the slave to interrogate. However, it is useful when having a hybrid topology, where a gateway links together a Modbus TCP network with a Modbus serial environment. When a master in a TCP/IP network wants to interrogate a Modbus serial slave, the master puts the IP address of the gateway as the destination IP address, and the serial address of the slave in the “Unit ID” field. Once the request reaches the gateway, it is translated from the IP world to the serial world.

The function code and data fields are the same as they would be when communicating over serial lines. Notice that the CRC of Modbus RTU is not present in Modbus TCP. It is not required since most of the data-link layer technology (e.g., Ethernet, WiFi) has its own CRC mechanism.

4. BENCHMARKING TOOLS FOR MODBUS

We developed several benchmarking tools to assess the Modbus protocol in different aspects. These benchmarking tools can be used to compare different software implementations of the Modbus protocol, to assess the performance of Modbus devices, and to evaluate the behavior of the protocol under DDoS attack.

For development, several Modbus libraries are available. For the Python programming language, PyModbus [41] (Modbus RTU/ASCII/TCP), MinimalModbus (Modbus RTU/ASCII), Modbus-tk (Modbus RTU/TCP), and uModbus (Modbus RTU/TCP) are available. For Java programmers, JLibModbus (Modbus RTU/ASCII/TCP), jModbus (Modbus RTU/ASCII/TCP), and Modbus4J (Modbus RTU/ASCII/TCP) are good choices. libmodbus [43] is the library used by C programmers. It is open-source and permits to send/receive data through the Modbus protocol. This library can use a serial line (Modbus RTU/ASCII) or an Ethernet connection (Modbus TCP). We chose libmodbus to implement our benchmarks since it is multiplatform (Unix, Windows, and Mac), well documented, and in constant development.

A small example on how to use PyModbus is shown in Figure 2. The connection with the slave is done in Line 03. Lines 04, 06, 07, and 09 read data from the remote Modbus device using functions 1 (Read Coils), 2 (Read Discrete Inputs), 3 (Read Holding Registers), and 4 (Read Input Registers), respectively. Lines 10, 11, 12, and 13 write data in the remote Modbus device using functions 5 (Write Single Coil), 6 (Write Single Holding Register), 15 (Write Multiple Coils), and 16 (Write Multiple Holding Registers), respectively.

```
01: from pymodbus.client.sync import ModbusTcpClient
02: help(ModbusTcpClient)
03: client = ModbusTcpClient("10.0.0.20")
04: result = client.read_coils(address=5, count=3, unit=1)
05: print(result.bits[0])
06: result = client.read_discrete_inputs(address=2, count=4, unit=1)
07: result = client.read_holding_registers(address=5, count=4, unit=1)
08: print(result.registers[2])
09: result = client.read_input_registers(address=10, count=2, unit=1)
10: client.write_coil(address=7, value=True, unit=1)
11: client.write_register(address=7, value=2000, unit=1)
12: client.write_coils(address=0, values=[True, False, True, True], unit=1)
13: client.write_registers(address=20, values=[727, 90, 524], unit=1)
14: client.close( )
```

Figure 2. Using PyModbus to Generate Modbus Requests

For the command line, “modpoll” [42] is very popular. It is open-source and multiplatform. Figure 3 depicts some modpoll commands to replicate the steps done in Figure 2, as close as possible. Notice that modpoll does not support functions 5 (Write Single Coil) and 6 (Write Single Holding Register).

```
01: modpoll -h
02: modpoll  -1  -m tcp  -r 6  -c 3  -a 1  -t 0  10.0.0.20
03: modpoll  -1  -m tcp  -r 3  -c 4  -a 1  -t 1  10.0.0.20
04: modpoll  -1  -m tcp  -r 6  -c 4  -a 1  -t 4  10.0.0.20
05: modpoll  -1  -m tcp  -r 11  -c 2  -a 1  -t 3  10.0.0.20
06: modpoll  -1  -m tcp  -r 1  -a 1  -t 0  10.0.0.20  1  0  1  1
07: modpoll  -1  -m tcp  -r 21  -a 1  -t 4  10.0.0.20  727  90  524
```

Figure 3. Using Modpoll to Generate Modbus Requests

Metasploit [44][45] also has support for Modbus. Figure 4 shows some Metasploit commands to replicate the steps done in Figure 2, as close as possible. Notice that Metasploit does not support functions 2 (Read Discrete Inputs) and 4 (Read Input Registers).

```
01: msfconsole
02: use auxiliary/scanner/scada/modbusclient
03: set RHOST 10.0.0.20
04: set UNIT_NUMBER 1
05: set DATA_ADDRESS 5
06: set NUMBER 3
07: set ACTION READ_COILS
08: run
09: set DATA_ADDRESS 5
10: set NUMBER 4
11: set ACTION READ_REGISTERS
12: run
13: set DATA_ADDRESS 7
14: set DATA 1
15: set ACTION WRITE_COIL
16: run
17: set DATA_ADDRESS 7
18: set DATA 2000
19: set ACTION WRITE_REGISTER
20: run
21: set DATA_ADDRESS 0
22: set DATA_COILS 1011
23: set ACTION WRITE_COILS
24: run
25: set DATA_ADDRESS 20
26: set DATA_REGISTERS 727,90,524
27: set ACTION WRITE_REGISTERS
28: run
```

Figure 4. Using Metasploit to Generate Modbus Requests

4.1. Response Time

We developed the first benchmark to assert the response time when reading or writing coils or registers. The code snippet for this benchmark is shown in Figure 5 for coils. Initially, we create a Modbus context, set the response timeout, and connect to the Modbus device to be assessed. We take a timestamp before (timerStart) and after (timerEnd) the loop. The difference of the timestamps divided by the number of iterations will be recorded as the response time, when the delay between requests (delayBtwRequests) is 0.0 seconds. Otherwise, the delay between requests will be subtracted from the previous results. For each iteration, we read or write the coils. The response timeout for the read/write operation is controlled by the parameter “timeout” (response timeout). After each request, we update our statistics about the number of fail and successful requests. Additionally, the process sleeps as specified by parameter “delayBtwRequests”. Finally, after the loop, we close the connection, free the allocated memory, and display the statistics.

```
ctx = modbus_new_tcp(targetHost, 502);
modbus_set_response_timeout(ctx, &timeout);
modbus_connect(ctx);

gettimeofday(&timerStart, NULL);  /* Get the starting time */
for(int i = 0; i < numIterations; i++) {
  if(operation == READ) {
      rslt = modbus_read_bits(ctx, modbusAddr, numBits, array);
      /* Update statistics */
  } else if(operation == WRITE) {
      rslt = modbus_write_bits(ctx, modbusAddr, numBits, array);
      /* Update statistics */
  }
  usleep(delayBtwRequests);
}
gettimeofday(&timerEnd, NULL);    /* Get the ending time */

modbus_close(ctx);
modbus_free(ctx);
/* Display statistics */
```

Figure 5. Code Snippet for the Benchmark to Evaluate the Response Time

4.2. Maximum Number of Successful Queries in One Second

We developed the second benchmark to evaluate the maximum number of successful operations processed by a Modbus device in one second, when handling several flows of requests from processes running a single computer. The code snippet for this benchmark is shown in Figure 6 and Figure 7 for the parent and child processes, respectively. As shown in Figure 6, the parent process initially takes a timestamp (timerStart). Then, it creates the child processes and communicates with them through pipes. Per se, the parent does not send any Modbus request to the target. Hence, it waits for the completion of all the children before taking the final timestamp (timerEnd). Once done, the parent retrieves the statistics of failed requests from the pipes, aggregates them, and displays the final results.

```
gettimeofday(&timerStart, NULL);   /* Get the starting time */

for(int i = 0; i < numChildren; i++) {
  /* Create a pipe to communicate with child i */
  /* Create child i */
}

for(int i = 0; i < numChildren; i++)
  wait(NULL); /* Wait for the completion of all the children */
gettimeofday(&timerEnd, NULL);     /* Get the ending time */

for(int i = 0; i < numChildren; i++)
  /* Receive statistics from child i through pipe */

/* Aggregate statistics from all the children */
/* Display statistics */
```

Figure 6. Code Snippet for the Benchmark to Assess the Maximum Number of Successful Requests per Second – Parent Process

In Figure 7, we have two possibilities. On the one hand, a connection to the Modbus device is established only once, at the beginning of the benchmark and finalized at the end (i.e., when Lines 04 and 17 are uncommented, and Lines 06 and 14 are commented out). On the other hand, for some devices, it is better to establish the connection just before the Modbus TCP request, and finalize it just after receiving the answer (i.e., when Lines 04 and 17 are commented out, and Lines 06 and 14 are uncommented). In Section 5, we will come back over this difference.

```
01: ctx = modbus_new_tcp(targetHost, 502);
02: modbus_set_response_timeout(ctx, &timeout);
03:
04: modbus_connect(ctx);              /* Connecting at the beginning */
05: for(int i = 0; i < numIterations; i++) {
06: //   modbus_connect(ctx);         /* Connecting in each iteration    */
07:    if(operation == READ) {
08:         rslt = modbus_read_bits(ctx, modbusAddr, numBits, array);
09:         /* Update statistics */
10:    } else if(operation == WRITE) {
11:         rslt = modbus_write_bits(ctx, modbusAddr, numBits, array);
12:         /* Update statistics */
13:    }
14: //   modbus_close(ctx);           /* Disconnecting in each iteration */
15:    usleep(delayBtwRequests);
16: }
17: modbus_close(ctx);               /* Disconnecting at the end */
18:
19: modbus_free(ctx);
20: /* Send statistics to parent through pipe */
```

Figure 7. Code snippet for the Benchmark to Assess the Maximum Number of Successful Requests per Second – Child Process

4.3. Number of Successful Queries During a DDoS Attack

The objective of the third benchmark is to evaluate a Modbus device when facing a DDoS attack. Hence, this test requires several computers as shown in Figure 8. The idea is to generate a heavy load by flooding the target Modbus device with requests, so it is under stress, and to report the number of processed requests over time, in the legal querier. To do so, we defined two types of computers: (1) attackers and (2) the controller or legal querier. To make it simple and save resources, the computer that will control the attackers will also act as the legal querier. At least one attacker is required. The attackers flood the targets with Modbus requests, according to the parameters received from the controller. The controller makes the legal requests, and periodically reports the number of processed requests at its level (number of received Modbus responses), per period of time.

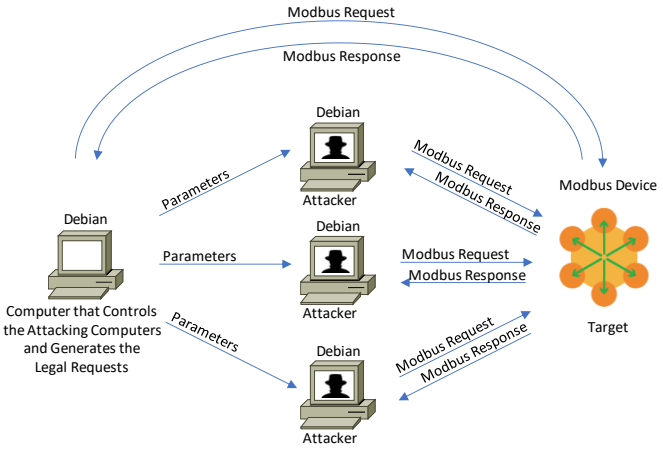

Figure 8. Testbed for the Number of Processed Requests in a Specific Time Period during a DDoS Attack

The code snippet for this benchmark is shown in Figure 9 and Figure 10 for the attackers and the legal querier, respectively. For each attack, an attacker accepts a connection from the controller and receives its parameters, which include: target Host, start Time, flooding Period, timeoutAtk, delay Btw Requests Atk, and operation. target Host is the IP address or hostname of the Modbus device to be attacked. The attacker will start its attack at start Time, for the period of time specified in flooding Period. The response timeout for the requests is given in time out Atk, and the delay between consecutive requests is delay Btw Requests Atk. When the time comes for the attacker to flood the target, it first establishes the connection, then for each iteration, it will send its request (read or write, according to parameter operation) and sleep the specified amount of time (delay Btw Requests Alk). At the end of the attack, the attacker closes the connection and releases the assigned resources and gets ready for the next one.

```
while(true) {
  /* Accept a new connection and receive parameters */
  ctx = modbus_new_tcp(targetHost, 502);
  modbus_set_response_timeout(ctx, &timeoutAtk);
  usleep(startTime);
  modbus_connect(ctx);
  while(currentTime < startTime+floodingPeriod) {
    if(operation == READ)
      rslt = modbus_read_bits(ctx, modbusAddr, numBits, array);
    else if(operation == WRITE)
      rslt = modbus_write_bits(ctx, modbusAddr, numBits, array);
    usleep(delayBtwRequestsAtk);
  }
  modbus_close(ctx);
  modbus_free(ctx);
}
```

Figure 9. Code Snippet for the DDoS Attack Benchmark – Attacker Computer

As shown in Figure 10, the controller starts its execution by making a connection to each attacker and sending the required parameters. Then, the controller creates a Modbus context for the communication with the target, sets the response timeout, and connects to the target Modbus device. While the experiment lasts, the controller repeats the following sequence: (1) send a request, (2) update the statistics, and (3) sleep the specified amount of time. At the end of the experiment, it frees the allocated resources and displays the statistics.

```
for(int i = 0; i < numAttackers; i++) {
  /* Compute parameters for attacker i */
  /* Connect to attacker i and send parameters */
}

ctx = modbus_new_tcp(targetHost, 502);
modbus_set_response_timeout(ctx, &timeoutLegalQuerier);

modbus_connect(ctx);
while(currentTime < endOfExperiment) {
  if(operation == READ) {
    rslt = modbus_read_bits(ctx, modbusAddr, numBits, array);
    /* Update statistics */
  } else if(operation == WRITE) {
    rslt = modbus_write_bits(ctx, modbusAddr, numBits, array);
    /* Update statistics */
  }
  sleep(delayBtwRequestsLegalQuerier);
}
modbus_close(ctx);
modbus_free(ctx);
/* Display statistics */
```

Figure 10. Code Snippet for the DDoS Attack Benchmark – Controller or Legal Querier

Similar to the second benchmark (see Section 4.2), this benchmark also has a version where the establishment and finalization of the TCP communication with the victim (target Modbus device) are done in the loop, for each iteration.

5. PERFORMANCE TESTS AND RESULTS

All the computers used in these tests have the following characteristics: Dell OptiPlex 3050 with an Intel Core i5-7500T CPU at 2.70 GHz, 8 GiB of RAM (1x8 GiB DDR4 2400MHz Memory), a 1024 GiB SSD, and an Intel dual-band wireless adapter (AC 3165 with support to IEEE 802.11 a/b/n/g/ac). We installed Debian 9.9 in all the computers.

Figure 11 depicts an open-source Internet of Thing (IoT) development board, called NodeMCU. NodeMCU Development Board is based on the ESP8266 [26-28], a cheap System on Chip (SoC) made by Espressif Systems to develop IoT applications. The ESP8266 consists of a WiFi-enabled microcontroller (Tensilica Xtensa® 32-bit LX106) operating at 80 or 160 MHz that supports IEEE 802.11b/g/n, a RAM of 128 kiB, and an external flash of 4 MiB. The board has 13 pins used for General Purpose Input/Output (GPIO) and 4 of those pins support pulse-width modulation. One pin supports SAR ADC, a type of analog-to-digital converter. For loading the firmware to the ESP8266, the Arduino Editor and a serial communication were used. The board includes a USB-to-UART controller from Silicon Labs (CP2102), so only a Micro-USB cable is necessary.

The ESP8266 has a TCP/IP stack with Modbus support. We selected this device for evaluation since it is very cheap (less than US$5 for the NodeMCU Development Board) and is now very popular in the IoT community. Moreover, it is now used in many academic courses related to IoT, and even in courses of Udemy (e.g., Learn ESP8266 for IoT [37], Home Automation Projects with NodeMCU ESP8266 & iOS 11 [38], 3-in-1 IoT Bundle: Arduino Uno, ESP8266, and Raspberry Pi [39], and ESP8266 for Absolute Beginner - Arduino Alternative 2019 [40]).

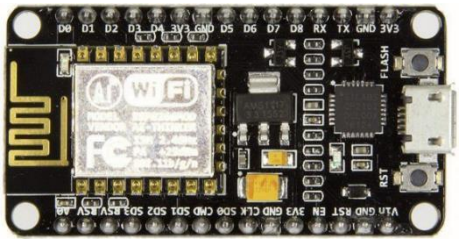

Figure 11. NodeMCU Development Board based on ESP8266

Raspberry Pi [46] is a series of single-board computers for teaching computer science at a low cost. The availability, low cost, and relatively high performance of the Raspberry Pi computers led them to become a popular choice in education, automation projects, and some commercial use. For less than US$40, a Raspberry Pi 3 B+ has a Broadcom BCM2837B0 SoC with a 1.4 GHz 64-bit quad-core ARM Cortex-A53 processor, 1 GiB LPDDR2 SDRAM, a dual-band 2.4 GHz and 5 GHz IEEE 802.11 a/b/g/n/ac wireless adapter, and an Ethernet port (300 Mbps). During the tests, the Raspberry Pi 3 B+ was running Raspbian 9.3 "Stretch". To simulate a PLC, OpenPLC [29-31] was installed to the Pi 3 B+.

A major difference between the implementation of the Modbus TCP protocol in the two devices is that the ESP8266 is constrained to one querier at a time, while OpenPLC allows an unlimited number of them simultaneously. Hence, for the ESP8266, we used the version of the benchmarks that establishes and closes the TCP connection just before and after the Modbus query, while we used the other version for the Raspberry Pi 3 B+/OpenPLC.

5.1. Results for the ESP8266

We used the testbed of Figure 12 for the experiments to assess the Modbus response time for the ESP8266. The connection between devices was made through a WiFi router (NETGEAR N300 model WNR2020v2), configured with WPA2-CCMP as the security protocol.

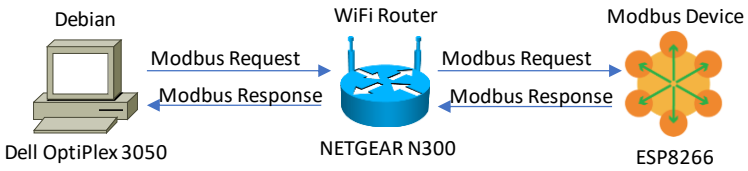

Figure 12. Testbed for the Response Time Experiments

Figure 13 depicts the response time that we obtained by running our first benchmark and varying the delay between requests (0, 25, 50, 75, and 100 ms). For each value of the delay, we reported the results of reading one coil (function code 1), writing one coil (function code 15), reading one holding register (function code 3), and writing one holding register (function code 16), for the ESP8266. We ran the benchmark thirty times and reported average results. This experiment seems to indicate that there is no significant difference in the response time for reading or writing. Furthermore, accessing a single coil or a whole 16-bit register also gave similar response times, around 12 ms. It is also worth mentioning that the delay between requests did not affect the results.

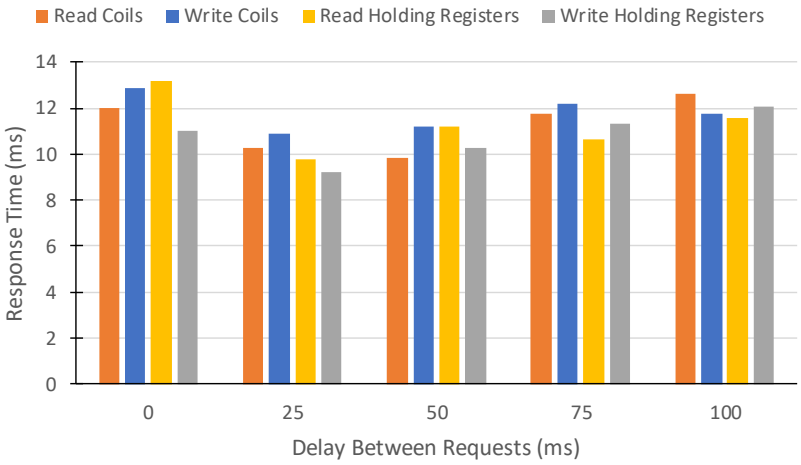

Figure 13. Response Time for the ESP8266 when Varying the Delay Between Requests

We used our second benchmark to flood the ESP8266 with requests, and report the maximum load supported by this Modbus device. The testbed was the same as the one depicted in Figure 12, with an underlying WiFi network. Figure 14 shows the total number of responses per second received from the ESP8266, which is equivalent to the maximum number of successful requests per second, or the maximum number of requests that were completed per second, before the response timeout. Figure 15 depicts the total number of failed requests per second during the same experiments. We varied the response timeout (20, 25, 30, 35, 40, 45, 50, 55, and 60 ms) and the number of child processes (1, 2, 3, 5, 7, and 9 child processes). For these experiments, we set the delay between requests in 0 ms to get the maximum possible numbers of requests successfully processed per second. Figures 14 and 15 correspond to Function Code 1 of Modbus (read coils), with one coil. We also did experiments with Function Code 15 (write multiple coils), 3 (read holding registers), and 16 (write multiple holding registers), and got similar results to the ones reported here, hence we are limiting this paper to "read coils" (Function Code 1).

When there is only one child process (CN=1), the ESP8266 can successfully handle about 75 requests/second (Figure 14). With two children (CN=2), it can respond to about 105 requests/second. According to those results, it looks like that the ESP8266 can process requests from several masters in parallel. As was already mentioned, it is not the case.

To deeper investigate this better performance with two queriers, we made some captures with Wireshark [7][25] for both experiments (CN=1 and CN=2). With one child, we clearly had the establishment of the TCP connection (three-way handshake), the Modbus request and response, and the finalization of the TCP connection. This sequence of events (TCP connection open, Modbus exchange, and TCP connection close) was repeated in time, without overlaps between one sequence and the next one. When having two children (A and B), one of the children will complete the TCP three-way handshake first (let's assume that child A did first). Hence, the ESP8266 will handle its requests, up to the finalization of the TCP connection. Nevertheless, Child B will try to establish a TCP connection, by sending a SYN segment. The ESP8266 will not answer this segment until the finalization of the other ongoing TCP connection. However, as soon as the current TCP connection with child A ends, the ESP8266 will continue with the pending TCP connection establishment, by sending a SYN+ACK segment to child B, followed by the ACK segment from child B to the ESP8266, thus, completing the three-way handshake in less time than in the experiment with one child (CN=1). We call this phenomenon the pipelining effect.

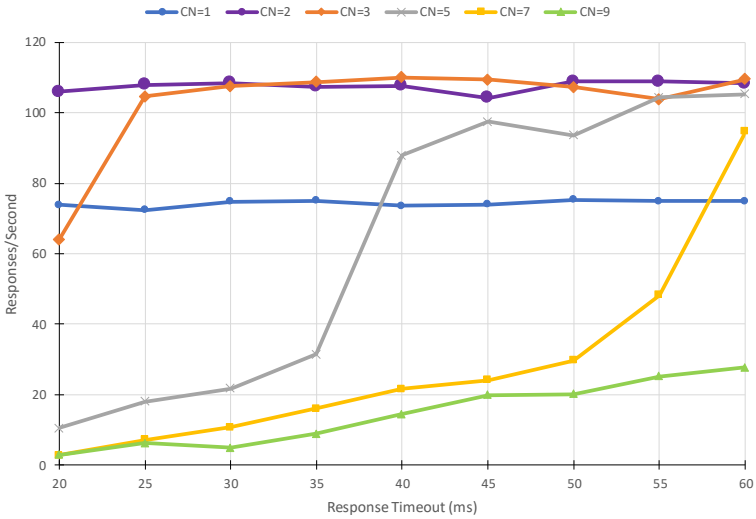

Figure 14. Maximum Number of Successful Requests per Second for the ESP8266 when Varying the Response Timeout

Some requests will fail to be completed in 20 ms with three children (CN=3). However, for a response timeout greater than or equal to 25 ms, all the requests are completed successfully in time, and the total number of successful requests per second is also around 105. This tendency of having failures for low values of the response timeout is getting more noticeable as the number of children grows. However, the total number of successful requests per second tends to rise toward 105 with the increasing value of the response timeout.

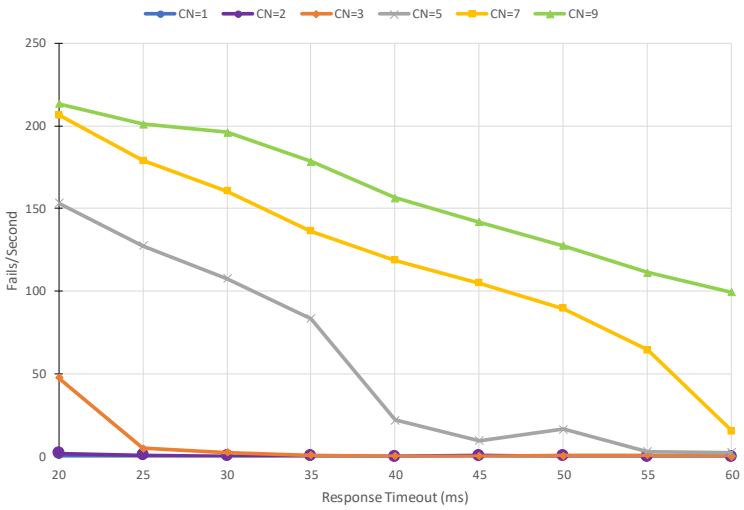

Figure 15. Total Number of Failed Requests per Second for the ESP8266 when Varying the Response Timeout

To study the behavior of the ESP8266 during a DDoS attack, we used the testbed of Figure 8 and the third benchmark. The communication between devices was done through WiFi, with the same wireless router of the previous experiment (NETGEAR N300 model WNR2020v2). Figure 16 shows the obtained results, with three attackers, a delay between requests of 0 s for both the legal querier and attackers, a flooding period of 20 s for the attackers, a response timeout of 500 ms for the attackers, and different values of the response timeout for the legal querier (20, 40, and 60 ms,

as the gray, orange, and blue curves, respectively). The attackers were activated at 4, 8, and 12 s. Since the flooding period of the attackers was 20 s, they ended their attack at times 24, 28, and 32 s, respectively. That is, the first attacker started and ended its flooding at times 4 and 24 s, the second attacker at times 8 and 28 s, and the last attacker at times 12 and 32 s.

In the initial part, the curves are like a stair going down, with a step for the starting of each attacker. In the final part, the curves are similar to a stair going up, with a step for the ending of each attacker. From 0 to 4 s, the legal querier was able to do 75 successful requests per second, confirming the results of Figure 14 (with one child process). Between 4 and 8 s, just one attacker was active, and the number of legal requests per second dropped down to 50. Between 8 and 12 s, there were two active attackers. On the one hand, for a response timeout of 40 and 60 ms at the level of the legal querier (orange and blue curves), the number of successful requests per second was around 33. On the other hand, the value was around 20 completed requests per second, for a response timeout of 20 ms (gray curve). When the three attackers were active (from 12 to 24 s), Figure 16 shows that the DDoS attack had a devastating effect over the legal traffic, especially for a response timeout of 20 ms (gray curve), since the legal querier was almost not able to poll the EMS8266.

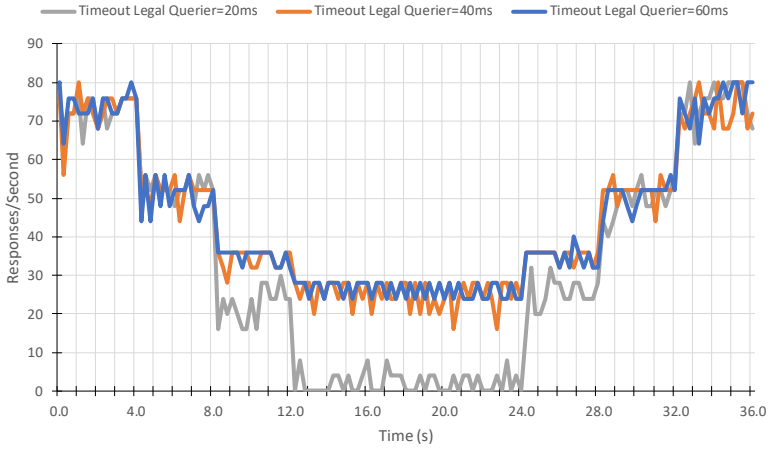

Figure 16. Number of Successful Requests per Second at the Level of the Legal Querier with Three Attackers for the ESP8266

Figure 17 shows the obtained results, with four attackers, a delay between requests of 0 s for both the legal querier and attackers, a flooding period of 24 s for the attackers, a response timeout of 500 ms for the attackers, and different values of the response timeout for the legal querier (20, 40, and 60 ms, as the gray, orange, and blue curves, respectively). The attackers were activated at 4, 8, 12, and 16 s.

Figure 17 evidences how a DDoS attack can impact legal traffic. For a response timeout of 20 ms (gray curve), when three attackers were active (on the one hand between 12 and 16 s, and on the other hand between 28 and 32 s), the legal querier can barely poll the ESP8266 a few times per second. With four active attackers, between 16 and 28 s, the legal querier was totally denied services.

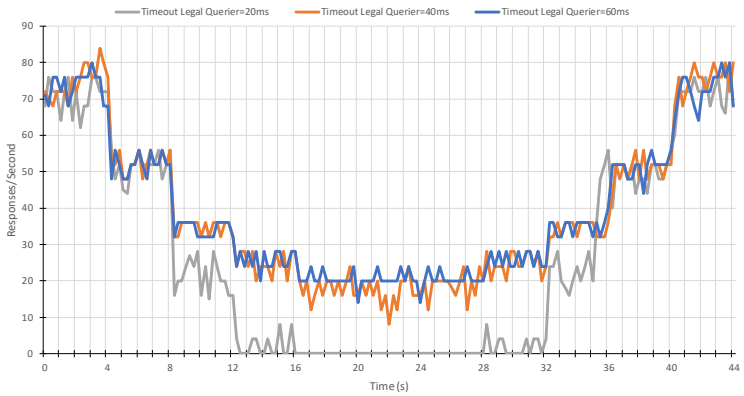

Figure 17. Number of Successful Requests per Second at the Level of the Legal Querier with Four Attackers for the ESP8266

5.2. Results for the Raspberry Pi/OpenPLC

We used a testbed similar to the one of Figure 12 for the experiments to assess the Modbus response time for the Raspberry Pi/OpenPLC, where the ESP8266 was replaced by a Raspberry Pi 3 B+ that was running OpenPLC.

Figure 18 depicts the response time that we obtained by running our first benchmark and varying the delay between requests (0, 25, 50, 75, and 100 ms). The response time fluctuates around 20 ms; it is bigger than the one of the ESP8266 (around 12 ms). The Raspberry Pi runs a full operating system (Raspbian 9.3) with a full implementation of the TCP/IP stack, resulting in a higher overload than the ESP8266, that runs a very specific firmware.

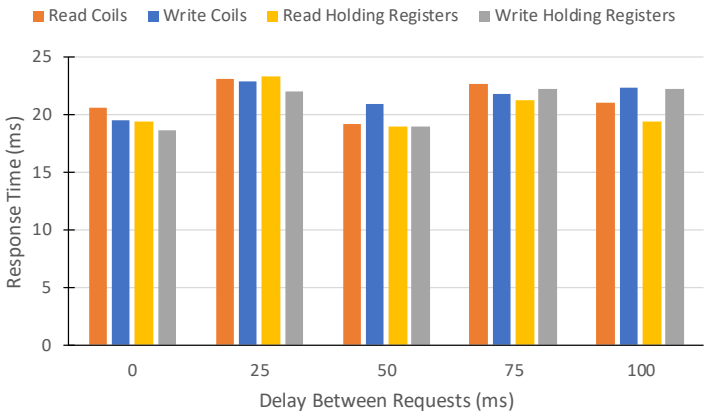

Figure 18. Response Time for the Raspberry Pi/OpenPLC when Varying the Delay Between Requests

Figure 19 depicts the total number of responses per second received from the Raspberry Pi/OpenPLC, which is equivalent to the maximum number of successful requests per second. Figure 20 shows the total number of failed requests per second during the same experiments. We varied the response timeout (20, 25, 30, 35, 40, 45, 50, 55, and 60 ms) and the number of child processes (1, 2, 3, 5, 7, and 9 child processes). For these experiments, we set the delay between requests in 0 ms to get the maximum possible numbers of requests successfully processed per second.

Here, we can see how a longer response time affects the results. Unlike the ESP8266 (see Figure 14) that has almost no lost when the response timeout is around 20 ms, the Raspberry Pi/OpenPLC lost most of its requests for this value, and starts performing much better when the response timeout is greater than or equal to 45 ms. However, these results also confirm that the Raspberry Pi has a much powerful processor, since it has a higher maximum number of responses per second, and it increases with the number of child processes.

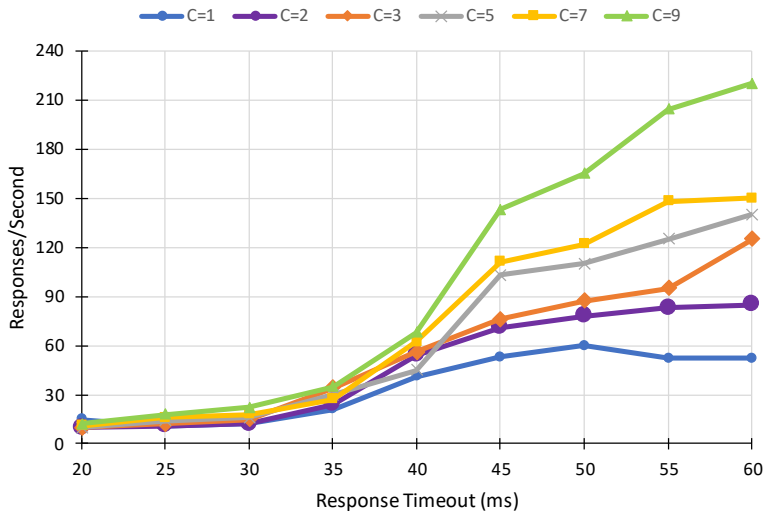

Figure 19. Maximum Number of Successful Requests per Second for the Raspberry Pi/OpenPLC when Varying the Response Timeout

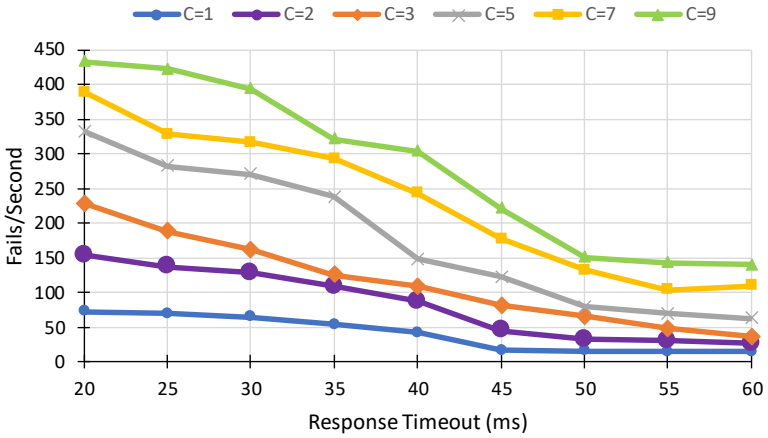

Figure 20. Total Number of Fails/Second for the Raspberry Pi/OpenPLC when Varying the Response Timeout

We also used the testbed depicted in Figure 8 and the third benchmark to study the behavior of the Raspberry Pi/OpenPLC during a DDoS attack. Figure 21 shows the obtained results, with three attackers, a delay between requests of 0 s for both the legal querier and attackers, a flooding period of 60 s for the attackers, a response timeout of 500 ms for the attackers, and different values of the response timeout for the legal querier (20, 40, and 60 ms, as the gray, orange, and blue curves, respectively). The attackers were activated at 15, 30, and 45 s. Since the flooding period of the attackers was 60 s, they ended their attack at times 75, 90, and 105 s, respectively.
Figure 21 is similar to Figure 16, like a stair going down with the activation of attackers, and then going up with the deactivation. Since the Raspberry Pi/OpenPLC has a longer response time, the number of successive legal queries is lower. It is noticeable that for a response timeout of 20 ms, the legal querier is totally denied services when the thee attackers are active.

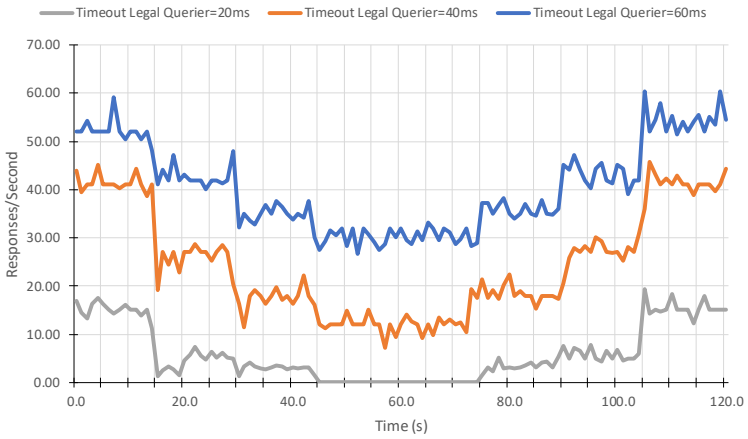

Figure 21. Number of Successful Requests per Second at the Level of the Legal Querier with Three Attackers for the Raspberry Pi/OpenPLC

6. CONCLUSIONS AND FUTURE WORK

The development of some benchmarks to assess the performance of Modbus TCP devices in normal operation and under a DDoS attack was addressed in this work. To the best of our knowledge, this is the first benchmarking tool to evaluate the performance of the Modbus TCP protocol. This tool should be very useful for developers that write Modbus TCP implementations, for network administrators that will have to choose between several implementations of this protocol, and for researchers that want to compare the performance of several industrial protocols (e.g., Modbus, Fieldbus, HART). We also did some experiments with two prominent and inexpensive development boards that are often used to implement IoT applications: ESP8266 and Raspberry Pi/OpenPLC. According to our experiments, the response time of the Raspberry Pi/OpenPLC is higher than the one shown by the ESP8266. However, the Raspberry Pi/OpenPLC outperforms the ESP8266 in the maximum number of successful queries in a specific amount of time.

As future work, we plan to extend our benchmarking tool to support IPv6 and apply it to industrial-grade Programmable Logic Controllers (PLCs). Also, we are interested in doing a performance evaluation at the level of IPv4/TCP, IPv6/TCP, IPv4/UDP, and IPv6/UDP, for widespread IoT devices under DDoS attacks.

ACKNOWLEDGMENTS

This work was partially supported by the National Security Agency, under grant H98230-18-1-0302 (DoD Cyber Scholarship Program).

REFERENCES

[1] Modbus Application Protocol Specification v1.1b3, April 2012.

[2] Modicon Inc., Modicon Modbus Protocol Reference Guide – PI-MBUS-300 Rev. J, June 1996.

[3] Modbus over Serial Line Specification and Implementation Guide v1.02, December 2006.

[4] M-System Co. Modbus Protocol Reference Guide. EM-5650 Rev. 10.

[5] R. Hulsebos, Modbus The Manual: Definitive Guide on Modbus, Mijnbestseller.nl, April 2019.

[6] Modbus Messaging on TCP/IP Implementation Guide v1.0b, October 2006.

[7] J. Bullock, Wireshark for Security Professionals: Using Wireshark and the Metasploit Framework, Wiley, 1st edition, March 2017.

[8] J. Johnson, Modbus Programming in C# (TCP/RTU): Full Example Projects, Independently published. April 2019.

[9] B. Kim, D. Lee, and T. Choi. Performance Evaluation for Modbus/TCP using Network Simulator ns-3. In Proceedings of the IEEE Region 10 International Conference (TENCON 2015), Macao, November 2015.

[10] N. Goldenberg and A. Wool, Accurate Modeling of Modbus/TCP for Intrusion Detection in SCADA Systems, International Journal of Critical Infrastructure Protection, vol. 6, no. 2, pp. 63-75, 2013.

[11] A. Ahmed, A. Mustafa, and G. Ibrahim, Performance Evaluation of IPv4 vs. IPv6 and Tunnelling Techniques Using Optimized Network Engineering Tools, IOSR Journal of Computer Engineering, Vol. 17, No. 1, February 2015.

[12] E. Gamess and M. Chachati. Analyzing Routing Protocol Performance with NCTUns for Vehicular Networks. Indian Journal of Science & Technology, Vol. 7, No. 9, pp. 1391-1402, September 2014.

[13] A. Jameel and M. Shafiei, QoS Performance Evaluation of Voice over LTE Network, Journal of Electrical & Electronic Systems, Vol. 6, No. 1, 2017.

[14] S. Sankaran, Modeling the Performance of IoT Networks, in Proceedings of the 2016 IEEE International Conference on Advanced Networks and Telecommunications Systems (ANTS 2016), Bangalore, India, November 2016.

[15] S. Shah and N. Mustari, Modeling and Performance Analysis of the IEEE 802.11p Enhanced Distributed Channel Access Function for Vehicular Network, in Proceedings of the 2016 Future Technologies Conference (FTC 2016), San Francisco, CA, USA, December 2016.

[16] M. Salehi, A. Boukerche, and A. Darehshoorzadeh, Modeling and Performance Evaluation of Security Attacks on Opportunistic Routing Protocols for Multihop Wireless Networks, Ad Hoc Network, Elsevier, Vol. 50, pp. 88-101.

[17] M. Abdullah, N. Al-awad, and F. Hussein, Performance Comparison and Evaluation of Different Software Defined Networks Controllers, International Journal of Computing and Network Technology, Vol. 6, No. 2, May 2018.

[18] S. Narayan, C. Williams, D. Hart, and M. Qualtrough, Network Performance Comparison of VPN Protocols on Wired and Wireless Networks, in Proceedings of the 2015 International Conference on Computer Communication and Informatics (ICCCI 2015), Coimbatore, India, January 2015.

[19] S. Bhatia, N. Kush, C. Djamaludin, J. Akande, and E. Foo, Practical Modbus Flooding Attack and Detection, in Proceedings of the Twelfth Australasian Information Security Conference (AISC 2014), Auckland, New Zealand, January 2014.

[20] B. Chen, N. Pattanaik, A. Goulart, K. Butler-Purry, and D. Kundur, Implementing Attacks for Modbus/TCP Protocol in a Real-Time Cyber Physical System Test Bed, in Proceedings of the 2015 IEEE International Workshop Technical Committee on Communications Quality and Reliability (CQR 2015), Charleston, SC, USA, May 2015.

[21] P. Huitsing, R. Chandia, M. Papa, and S. Shenoi, Attack Taxonomies for the Modbus Protocols, International Journal of Critical Infrastructure Protection, Vol. 1, pp. 37-44, December 2008.

[22] T. Morris, R. Vaughn, and Y. Dandass, A Retrofit Network Intrusion Detection System for Modbus RTU and ASCII Industrial Control Systems, in Proceedings of the 2012 45th Hawaii International Conference on System Sciences (HICSS 2012), Maui, Hawaii, USA, January 2012.

[23] É. Ádámkó, G. Jakabóczki, and P. Szemes, Proposal of a Secure Modbus RTU Communication with Adi Shamir's Secret Sharing Method, International Journal of Electronics and Telecommunications, Vol. 64, No. 2, pp. 107-114, April 2018.

[24] G. Sanchez, Man-In-The-Middle Attack Against Modbus TCP Illustrated with Wireshark, SANS Institute, 2019.

[25] N. Kumar Nainar, Y. Ramdoss, and Y. Orzach, Network Analysis Using Wireshark 2 Cookbook: Practical Recipes to Analyze and Secure your Network using Wireshark 2, Packt Publishing, 2nd edition, March 2018.

[26] C. Batrinu, ESP8266 Home Automation Projects: Leverage the Power of this Tiny WiFi Chip to Build Exciting Smart Home Projects, Packt Publishing, November 2017.

[27] M. Schwartz, Internet of Things with ESP8266: Build Amazing Internet of Things Project using the ESP8266 WiFi Chip, Packt Publishing, July 2016.

[28] P. Seneviratne, ESP8266 Robotics Projects: DIY WiFi Controlled Robots, Packt Publishing, November 2017.

[29] T. Alves, M. Buratto, F. de Souza, and T. Rodrigues, OpenPLC: An Open Source Alternative to Automation, in Proceedings of the 2014 IEEE Global Humanitarian Technology Conference (GHTC 2014), San Jose, CA, USA, October 2014.

[30] T. Alves and T. Morris, OpenPLC: An IEC 61,131–3 Compliant Open Source Industrial Controller for Cybersecurity Research, Computers & Security, Vol. 78, pp. 364-379, 2018.

[31] T. Alves, T. Morris, and S.-M. Yoo, Securing SCADA Applications using OpenPLC with End-to-End Encryption, in Proceedings of the 3rd Annual Industrial Control System Security Workshop (ICSS 2017), San Juan, PR, USA, December 2017.

[32] S.-Y. Wang and C.-C. Lin, NCTUns 6.0: A Simulator for Advanced Wireless Vehicular Network Research, in Proceedings of the 2010 IEEE 71st Vehicular Technology Conference (VTC 2010), Taipei, Taiwan, May 2010.

[33] S. Rajput, Basics of OMNeT++, LAP LAMBERT Academic Publishing, January 2019.

[34] A. Virdis and M. Kirsche, Recent Advances in Network Simulation: The OMNeT++ Environment and its Ecosystem, Springer, 1st edition, May 2019.

[35] R. Jha and P. Kharga, Advanced Open Source Simulator: ns-3, International Journal of Computer Sciences and Engineering (IJCSE), Vol. 3, No. 12, pp. 67-74, 2015.

[36] A. Thomas, Snort Primer: A FAQ Based Introduction to the Most Popular Open-Source IDS/IPS Program, CreateSpace Independent Publishing Platform, November 2015.

[37] A. Nayak, Learn ESP8266 for IoT: Build Your Home Automation with ESP8266 and Control Devices from Anywhere in the World, Udemy. https://www.udemy.com/learn-esp8266.

[38] J. Huang, Home Automation Projects with NodeMCU ESP8266 & iOS 11: Learn How to Easily Make Home Automation Projects with Basic Electronic Components, Udemy. https://www.udemy.com/making-home-automation-iot-projects-with-nodemcu-ios-11.

[39] A. Nayak and V. Khatiyan. 3-in-1 IoT Bundle: Arduino Uno, ESP8266, and Raspberry Pi: Learn and Understand the Inner working of Most Popular Devices and Hardware for Making IoT Usecases Come to Life. Udemy. https://www.udemy.com/learn-iot.

[40] Confiny, ESP8266 for Absolute Beginner - Arduino Alternative 2019: Build IoT Web Application, LCD Pattern, Physical Input, GPIO Interfacing, Firmware Installation, ESP12E, NodeMCU & more. https://www.udemy.com/getting-started-with-nodemcu-step-by-step.

[41] PyModbus – A Python Modbus Stack. https://pymodbus.readthedocs.io/en/latest/readme.html.

[42] Modpoll Modbus Master Simulator (modpoll). https://www.modbusdriver.com/modpoll.html.

[43] libmodbus. A Modbus Library for Linux, Mac OS X, FreeBSD, QNX and Win32. https://libmodbus.org.

[44] N. Jaswal. Mastering Metasploit, Packt Publishing, 2nd edition, September 2016.

[45] S. Rahalkar and N. Jaswal, The Complete Metasploit Guide: Explore Effective Penetration Testing Techniques with Metasploit, Packt Publishing, June 2019.

[46] D. Dieterle, Security Testing with Raspberry Pi, Independently published, June 2019.

AUTHORS

Eric Gamess received a M.S. in Industrial Computation from the National Institute of Applied Sciences of Toulouse (INSA de Toulouse), France, in 1989, and a Ph.D. in Computer Science from the Central University of Venezuela, Venezuela, in 2010. He is currently working as a professor at Jacksonville State University, Jacksonville, AL, USA. Previously, he worked as a professor at the University of Puerto Rico, Puerto Rico, the Central University of Venezuela, Venezuela, and "Universidad del Valle", Colombia. His research interests include Security, Vehicular Adhoc Networks, Networking, and the Internet of Things. He is one of the co-founders of the Venezuelan Society of Computing and has been in the organizing and program committees of several national and international conferences.

Brody Smith received a Bachelor in Computer Science from Jacksonville State University, Jacksonville, AL, USA, in 2019. He is currently working as a Cybersecurity specialist at Fort Gordon in Augusta, Georgia, USA. His research interests include Cybersecurity and Computer Networks.

Guillermo A. Francia, III received his Ph.D. in Computer Science from New Mexico Tech. Before joining JSU in 1994, he was the chairman of the Computer Science department at Kansas Wesleyan University. In 1996, Dr. Francia received one of the five national awards for Innovators in Higher Education from Microsoft Corporation. His research interests include information security and assurance, embedded and industrial control systems security, vehicular network security, machine learning, unmanned aerial vehicle security, and risk management. He is a two-time recipient of a Fulbright award (UK, 2017 and Malta, 2007) related to cybersecurity projects and is the 2018 winner of the National Cyber Watch Center Innovations in Cybersecurity Education award. He has successfully managed research projects worth over $2.5M that were funded by the National Science Foundation, Department of Defense, Department of Energy, and Department of Homeland Security. He held Distinguished Professor and Director of the Center for Information Security and Assurance positions at Jacksonville State University prior to joining the University of West Florida (UWF) in 2018. In April 2019, he received an appointment as Commissioner of the Computing Accreditation Commission of ABET. Currently, Dr. Francia is serving as Faculty Scholar and Professor at the Center for Cybersecurity.