MODSEC: A SECURE MODBUS PROTOCOL

A Thesis
Presented to
The Academic Faculty

By

Paul L. Wilson, III

In Partial Fulfillment
of the Requirements for the Degree
Masters of Science in the
School of Electrical and Computer Engineering

Georgia Institute of Technology

August 2018

Copyright © Paul L. Wilson, III 2018

MODSEC: A SECURE MODBUS PROTOCOL

Approved by:

Dr. Beyah, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Copeland
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Grijalva
School of Electrical Engineering
Georgia Institute of Technology

Date Approved: July 21, 2018

If we get some idea of how the past has shaped us, we may better understand what we see today and are likely to see tomorrow

UNKNOWN

”Now unto him who is able to keep you from falling”

ACKNOWLEDGEMENTS

There are many, many people who helped me through both my undergraduate and graduate experience at Georgia Tech, too many to name by name. However, a special token of appreciation goes to:

- All glory be to God!
- Dr. Raheem Beyah, my advisor and mentor, for taking me under his wing my freshman year of college and helping me to grow, learn, and develop into the person I am today.
- My family, who has been a constant support system for me throughout my entire life, has always encouraged me and been there for me.
- David Formy, Celine Irvene, Samuel Litchfield, Dominique Paster, and the other members of the Communications Assurance and Performance Group, for constant support through my thesis and research.
- My many friends that I refuse to name for so that I don't leave anyone out, but who also were a major support system in my life, both in and outside of the classroom, helping me to maintain sanity and always there to show some love.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Many of todays most critical infrastructures rely on the successful operation of Supervisory Control and Data Acquisition (SCADA) systems distributed all around the world. Infrastructures such as water treatment plants, gas stations, and transportation all rely on SCADA systems, and any form of disruption has the potential to cause grave harm to a society. As technology has continued to grow and evolve, networks have also been able to grow in both space and complexity while also allowing for system operators to more efficiently manage these systems. Despite this growth, many of the communication protocols that these systems use have failed to change, and systems that were never meant to be brought to an insecure environment like the Internet are being exposed, bringing forth a wide range of security vulnerabilities to these infrastructures.

Modbus, introduced in 1979, is one of the original communication protocols used in SCADA environments and, to this day, is still implemented in nearly all industrial and automation equipment. The protocol is popularly used by programmable logic controllers (PLCs) to control actuators and gates within a system through a master-slave architecture. Despite its popularity, the protocol lacks any form of security and exposes the ability for a nefarious actor to easily control devices in a network and cause chaos.

This thesis presents ModSec, a protocol that brings practical security enhancements to the Modbus protocol. The contribution can be separated into two separate goals: to add security to each of the protocols messages through a means of authentication and integrity, as well as a permission-based scheme to limit the effects that an unintended message can pose. ModSec is shown to prevent against many of the attacks that have already been proven against the Modbus protocol, while also taking into consideration the end systems. Many of the systems that are implements in SCADA environments are either low or lack processing power that would be necessary to fully implement common security mechanisms, like encryption. ModSec takes a novel approach to this problem, resulting in little

overhead to the systems or the messages, thus allowing for the protocol to continue to be used without being effected by a large amount of latency or stress on the system.

CHAPTER 1
INTRODUCTION

Today many of the worlds most critical infrastructures consist of facilities that are large and distributed. Infrastructures that include water plants, gas stations, oil refining industries, and transportation substations play critical roles in the success of a nations economy and have many moving parts that are necessary in order for it be successful. To maintain these systems, it is essential for operators to constantly monitor and control various components. While these key infrastructures are not new, the implementation of modern networking has eased the burden of managing these critical systems by allowing operators to have remote control of the environment. The earliest forms of these critical infrastructures were simple networks that connected monitoring and command devices with sensors and actuators. These simple networks have since advanced into complex networks that allow communication not only within a single system at a single location, but around the globe. These systems are commonly referred to as industrial control systems (ICSs). An ICS traditionally has various key components, including supervisory control and data acquisition (SCADA) systems, distributed control systems (DCS), and programmable logic controllers (PLCs).

As these systems continue to grow in size and complexity, as does the importance of modernizing these systems in an attempt to increase the overall efficiency while also reducing costs. It has become popular over the past two decades for these ICSs, and in particular, SCADA systems, to be connected online to allow remote operations, while also optimizing processes. This phenomenon, however, has introduced gaping security issues into networks that were originally designed to be closed off from the world. Where the problem of security was commonly solved by increasing physical security mechanisms, the various potential issues that lie within the Internet have now also been introduced on a much more

extreme scale. By introducing these processes and controls to the Internet, the threat of malicious attacks over these networks are not only possible, but can also cause significant amounts of damage to the infrastructure itself, as well as the safety of the public. Furthermore, many of the devices that are used in these environments are low-powered, focusing more on single tasks like changing an actuator or a sensor, thus making traditional network security practices impractical in these environments.

There have been numerous studies within academia that have discussed the increasing number of threats within critical infrastructures, as well as numerous examples of these threats coming to fruition with actual attacks that have caused large amounts of damage. While implementing traditional network security techniques like firewalls and intrusion detection systems are effective at addressing vulnerabilities within corporate networks [1], they fail to address attacks targeted specifically at process control networks. A key weakness within these systems are the communication protocols that are used within SCADA that both control and monitor devices. These protocols, designed for closed networks, fail to address common security issues like confidentiality, integrity, and authentication but are still largely used today by nearly all vendors within ICS and SCADA.

The primary focus of this work is to enhance the Modbus protocol, one of the most common protocols within ICS and SCADA. Modbus is a simple protocol that is used in a wide variety of environments and is used to control sensors, actuators, and many other key components within critical infrastructures. To add security, various cryptographic algorithms are used and deployed within the Modbus message protocol prior to and during transmission over the various transmission mechanisms that Modbus supports.

1.1 Primary Contributions and Thesis Organization

The primary contributions of this thesis are the following:

- A detailed overview of the current vulnerabilities and attacks available on the Modbus protocol.

- Several key additions to the Modbus protocol to address various security issues that are present due to the protocol being completely open, focusing on authentication, integrity, and non-repudiation.
- A permission scheme for the protocol to add authorization to the protocol to prevent the transmission and action of unauthorized commands.
- An implementation of the newly proposed protocol.
- An evaluation of how the new protocol, ModSec, affects usage and latency compared to a previous implementation.
- An evaluation of how ModSec addresses the various attacks and vulnerabilities that are present on Modbus.

The remainder of this thesis is presented as follows: Chapter 2 describes background of SCADA systems and the Modbus protocol, current vulnerabilities and attacks that have been proven, and proposed solutions to attempt to address and mitigate various Modbus vulnerabilities. Chapter 3 introduces ModSec, outlining the new protocol and how it addresses authentication, authorization, integrity, and non-repudiation. Chapter 4 discusses the implementation of ModSec and its affects on latency and processing, as well as how the proposed solution addresses many of the attacks that were presented in various other works. Finally, Chapter 5 presents the conclusion and potential avenues for future work.

CHAPTER 2
BACKGROUND AND RELATED WORK

2.1 SCADA Background

While commonly overlooked in industrialized societies, ICS and SCADA systems play a key role in providing citizens with many basic modern-day conveniences. Today, essentials like power and running water and everyday conveniences like public transit buses and train stations all rely on ICS (specifically SCADA systems). The disruption of these services can immediately cause chaos in a modern day society. Over the past few decades, the quick advancements in technology have allowed for industries to also advance critical aspects of SCADA, allowing for these systems to be made more stable and efficient. With these advancements come new challenges and threats that can lead to global chaos. The introduction of the Internet to these industrial systems has further progressed the threat to a level that, if left unaddressed, could be a disaster waiting to happen.

When first introduced, the threats to SCADA were limited and focused on physical access. The first generation of SCADA systems were completely isolated and connected to other systems. This allowed for the design of many of the initial protocols introduced to have open networks, completely ignoring security as whole. The largest threat would be an attacker that needed to be authorized and have physical access to the machine [2].

As technology continued to advance, the idea of using isolated machines began to transform into the use of distributed architectures. Instead of relying on a single system, the new paradigm focused on distributing the system to multiple stations that would be connected through a local area network (LAN). During this period, secure design was largely ignored and overly complicated. The threat model expanded from solely focusing on physical access to also considering remote execution. Furthermore, many vendors did not rely on a

standard network protocol, requiring security experts to have a deep understanding of each proprietary network protocol in order to configure and manage the security system. These complexities opened the door for what became known as one of the first reported attacks on a SCADA network in 1982. In the attack, a Trojan program was remotely inserted into a SCADA system's software, resulting in a massive natural gas explosion along the Trans-Siberian pipeline [2].

The next generation of SCADA architecture allowed for systems to not only be connected, but also allowed for the system to control devices that were both separated geographically and connected to multiple local area networks. This phenomenon became known as process control networks (PCN). Failing to successfully address the issues from the previous generation, this connection only furthered the complexity of the threat model, allowing for an attacker to now be physically located anywhere in the world with the potential to perform remote exploitation. Several attacks came about during this period of time that had major impacts. In 1999, a Trojan program in a Russian oil corporation disrupted the control of gas flow for several hours [3, 4]. In 2003, the Sobig worm [5] infected a railroad company, resulting in the shut down of signaling, dispatching, and other systems and causing major train delays. Another infamous worm, Slammer [6], attacked a nuclear power station in the same year. In 2004, the Sasser worm [7], targeted airlines and rail transportation companies, causing for delays and cancellations of both flights and trains. In 2009, one of the most infamous worms, Conflicker [8], resulted in the shutdown of an entire air fleet [2].

The fourth, and current, generation of industrial technologies that has become popular is known as the Internet of Things. This technology allows for organizations to reduce cost and improve maintenance management [2]. Similar to the other trends that progressed the connectivity of SCADA systems and devices, security continued to remain a stagnant afterthought. Stuxnet [9], one of the most infamous attacks on a SCADA system, began the influx of many more attacks on SCADA. Stuxnet, occurring in 2010, resulted in the

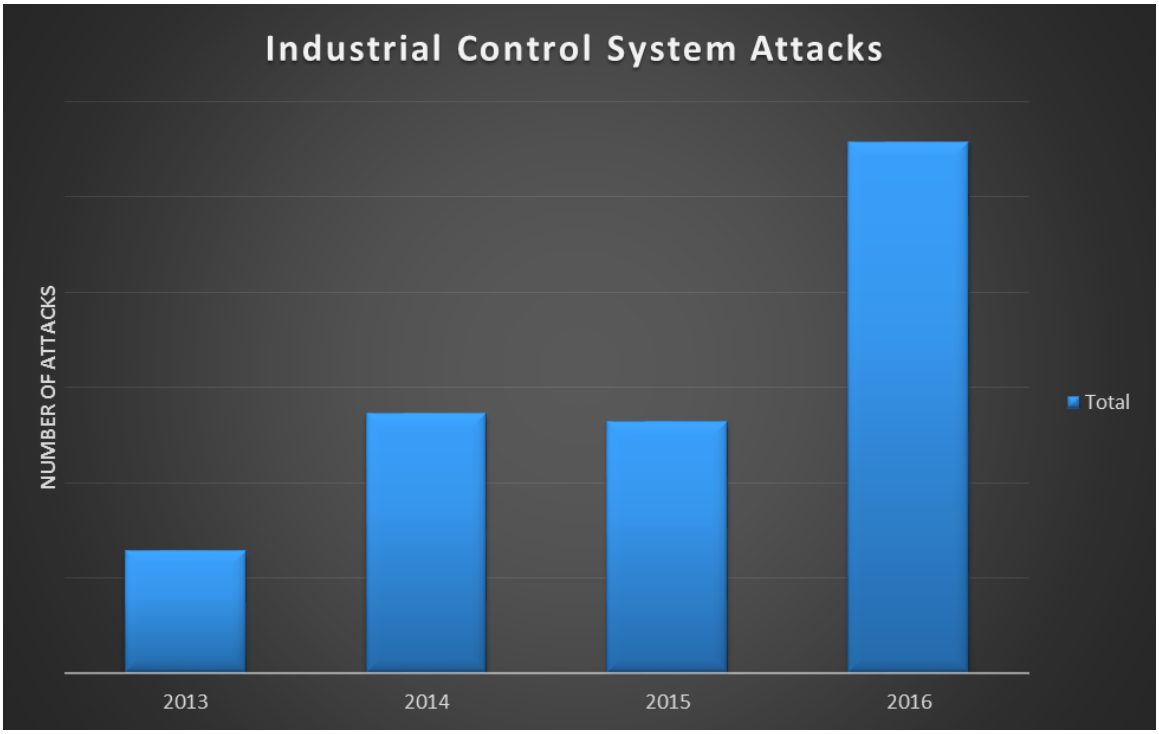

Figure 2.1: Industrial Control System Attacks [11]

destruction of approximate a fifth of Iran's nuclear centrifuges [10]. It has been reported that the number of attacks between the years of 2012 and 2014 raised by a factor of 636%. Additionally, the amount of security disclosures has also dramatically increased, showing that many of the vulnerabilities that were found resulted in buffer overflows. Thus, these vulnerabilities not only could potentially lead to a denial of service in the network, but also potentially allow for attackers to gain remote execution and completely take over the system [2].

A more recent report shows that the number of attacks has continued to spike, where there was an increase of over 100% between the years of 2015 and 2016. These results are evident in Figure 2.1. In 2016 alone, there were a significant number of attacks on ICS aimed at shutting down an energy grid in Europe, a New York Dam, and a power grid in Ukraine [12, 13, 14]. The number of attacks is directly correlated with the connectivity between SCADA devices and external networks [11]. By connecting these SCADA systems

on-line, the systems have the potential to be made visible to anyone around the world. The popular search engines ERIPP [15] and SHODAN [16] give anyone with interest the ability to search for these Internet-facing devices, noting key information about the devices that can be used to craft an attack [2]. With the constant rise in the number of attacks, as well as the potential consequences that can be a result, new solutions need to be implemented to address the lack of security that is inherent within SCADA as a whole. Many SCADA users assume that by simply implementing a virtual private network (VPN), the control systems are automatically protected. This, however, has been shown many times to not be the case [2]. While adding a VPN is an important step to securing SCADA devices a whole, it does not address vulnerabilities that specifically target process control networks.

Securing SCADA and ICS environments is critical to ensuring the safety of not only the systems themselves, but also to ensure the safety of the many people around the world that rely on these systems to live a safe and healthy life. While many research efforts are focused on addressing the vulnerabilities by means of adding end-to-end communication security and assurance, it is also critical to take a step deeper and address the protocols that these systems rely on for communication. There are several key protocols that are commonly used in SCADA and ICS environments that lack even the most basic security mechanisms due to the time period of when they were developed and the fact that they were designed for an isolated network. This thesis focuses on adding security mechanisms to the Modbus protocol, one of the most common protocols within SCADA that is commonly used the oil and gas industries, along within many other critical infrastructures. The security issues within Modbus are well known, as a large amount of research has been poured into identifying potential and actual attacks and vulnerabilities [17, 18, 19, 20, 21].

2.2 Modbus Background

Introduced in 1979 by the manufacturer Modicon, the Modbus communication interface is one of the most popular and widely used protocols in the area of industrial supervisory

controller and data acquisition (SCADA). To this day, nearly all industrial and automation equipment vendors continue to support the protocol in their new products [22]. The protocol itself was originally created to easily exchange data between programmable logic controllers (PLCs) and other devices on a production floor. The protocol is an open standard, playing the role of describing the messaging structure while leaving the other aspects of the communication flexible. This has allowed for the protocol to be leveraged in a wide variety of devices, ranging from microcontrollers and PLCs, to intelligent sensors and playing a large role in the Internet of Things (IoT) phenomenon [23]. Modbus can run over virtually all forms of communication media, including twisted pair wires where it was first leveraged, to wireless, fiber optic, Ethernet, telephone modems, cell phones, and microwaves [23, 22].

Traditionally, Modbus is implemented via a master-slave architecture, where a master device or server communicates with one, or multiple, slave(s) at a time. Within SCADA, the master device is typically a PLC, PC, Distributed Control System (DCS), Remote Terminal Unit (RTU), or Human Machine Interface (HMI). The slave device is typically a field device that is connected to the network in a multi-drop configuration. This device is typically connected to the physical world through sensors, actuators, or other forms of PLCs [22]. Within the SCADA environment, Modbus is typically used to provide request and response message services, allowing for the master to gain insight on the various slave devices. The four types of messages that are seen within Modbus fall under the categories of (1) request, (2) response, (3) message confirmation, and (4) message indication.

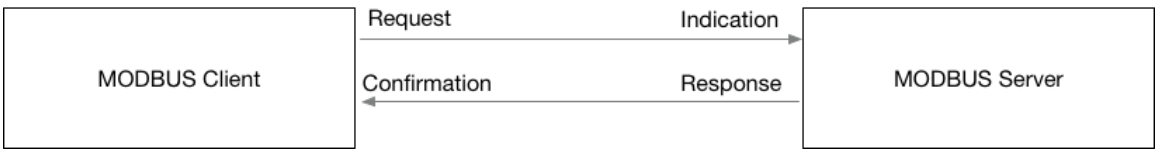

Figure 2.2: Client Server Communication

1. **Modbus Request Message:** Usually the master initiates the transmission and sends

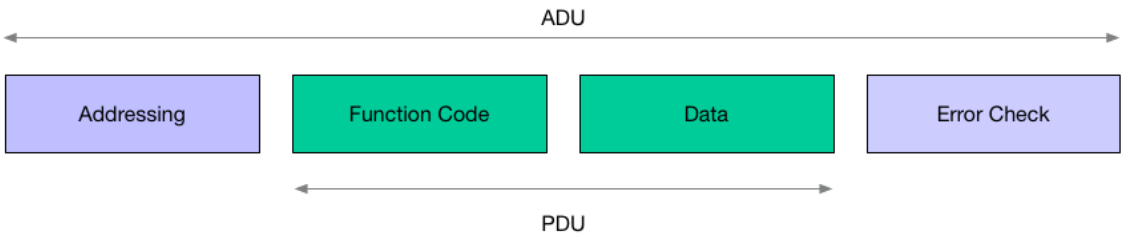

Figure 2.3: Modbus Frame

the request message to the slave

2. **Modbus Response Message:** Field devices are configured to generate and transmit a response back to the master that has requested information

3. **Modbus Message Confirmation:** Upon receiving a response, the master transmits a confirmation message to the sending slave device

4. **Modbus Message Indication:** Slave devices generate indication messages that show the request messages have been received

Within Modbus, only master devices are designed to initiate communication, and slave devices are designed to only respond to messages that they receive [23]. There are instances of hybrid devices that act as slaves, but also have write capability [22]. This communication is visualized in Figure 2.2.

The most common versions of Modbus are Modbus Serial, or RTU, and Modbus/TCP. Within Modbus Serial, data is transmitted over modem links like RS-232 or RS-485, and is the original implementation of the protocol. Modbus/TCP, however, has become ever more popular as it allows for communication over greater distances while taking advantage of TCP/IP protocols and Ethernet technology. This interaction makes it possible to travel over routed networks. Within both instances of Modbus, however, there is a common Protocol Data Unit (PDU) that includes necessary information for the sending and receiving devices to process the information, visualized in Figure 2.3. The PDU consists of two critical fields,

a (1) function code field and a (2) data field [23].

1. **Function Code Field:** The function code field commands the slave device to what action to perform. There are various functions within Modbus, such as read input status, read output status, write data, and others that are listed in Table 2.1. The function code is added in the request message by the master. Once the slave receives the message, it attempts to perform the requested action, and if unable, it will return the necessary information.

2. **Data Field:** The data field contains an arbitrary number of bytes depending on the function code that is specified and the necessary information for the target device to perform the requested action. In the case of a read action, the data field might include the devices memory map in order to read the necessary information. In the case of a write action, the data field may carry the actual data that needs to be written to the device.

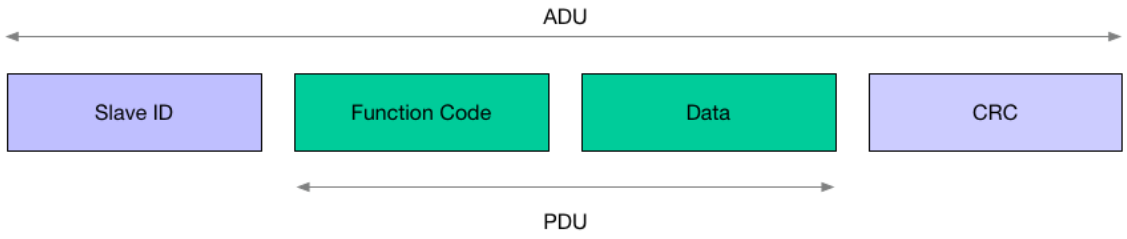

Figure 2.4: Modbus Frame using Modbus Serial

While the Application Data Unit (ADU) is the same over both forms of media, the Protocol Data Unit (PDU) differs between Modbus Serial and Modbus/TCP. Within Modbus Serial, a 1 byte address field is prepended to the PDU, and a 2 byte checksum is appended to the PDU, visualized in Figure 2.4.

- **Address Field:** The address field contains one byte of information specifying the identity for which the message is being directed to or from. The address ranges from

Table 2.1: Function Codes

Function Name	Function Code	Description
Read coil or digital output status	01	The field device responds to the logical coil(s) ON/OFF status.
Read digital input status	02	Read discrete inputs from the field device.
Read holding registers	03	Retrieves the contents of the holding register(s) from field device.
Read input registers	04	Retrieves the contents of input register(s) from the field device.
Force single coil	05	The ON/OFF status of single logic coil is changed from the field device.
Preset single register	06	To change the content of a single holding register.
Read exception status	07	To retrieve the status of eight digital points as a short message request from the field device.
Loopback test	08	Employs diagnostic features including CRC errors and reports according to exceptions to test the operation of the system.
Force multiple coils or digital outputs	0F	To manage the ON/OFF status of the coils (or group of coils).
Force multiple registers	10	To change the content of a single register and to manage a group of coils.

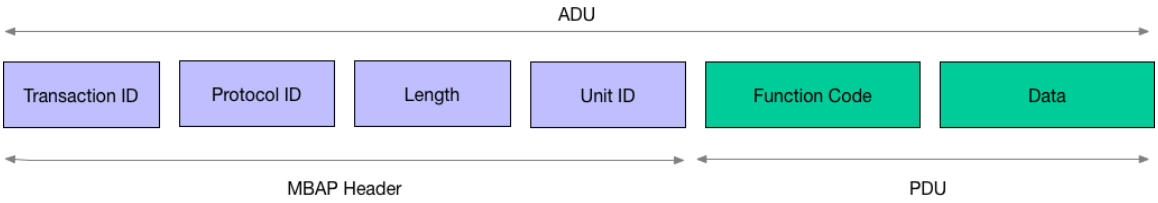

Figure 2.5: Modbus Frame using Modbus/TCP

1 to 247, limiting the total number of devices in a logical Modbus network to 246. In the case where the address is 0, the message is broadcast to all devices instead of a single device.

- **Checksum:** The checksum is used for error checking by employing a cyclic redundancy check (CRC) to compute the numeric value of the message. The numeric code detects errors and changes to the message during transmission. During the processing of the message, if the calculated value of the message and the transmitted value does not match, then the device asks for a retransmission.

Within Modbus/TCP, the ADU consists of a Modbus application protocol header (MBAP) to assist in identifying the ADU while the data is carried over the TCP/IP network, visualized in Figure 2.5. The MBAP consists of a 2 byte Transaction Identifier, 2 byte Protocol Identifier, 2 byte Length Field, and 1 byte Unite Identifier field.

- **Transaction Identifier:** The transaction identifier field contains two bytes that are set by the client to uniquely identify each request. The bytes are echoed by the server in the response to properly identify the response message.

- **Protocol Identifier:** The protocol identifier field was designed to be used for intra-system multiplexing. Within the Modbus protocol, the value is always 0.

- **Length:** The length field identifies the total number of bytes in the message following the MBAP.

- **Unit Identifier:** The unit identifier field is a single byte used to communicate via devices such as bridges, routers, and gateways that use a single IP address to support multiple independent Modbus end units.

Despite these key changes necessary for transmission over the two mediums, upon receipt, the message is processed in the clear by the end device through the use of the function code and exception codes, if necessary. The overall flow is visualized in Figure 2.6. Upon receipt of the message, the device performs the following operations:

1. **Validate Function Code:** The device first checks to see if a valid function code has been received, and if it has been configured to perform the requested action. In the case where it does not have the requested capability, an exception is returned to the sender in a response.

2. **Validate Data Address:** The device checks the data field of the PDU to see if the requested data is valid in respect to the mapping of the devices memory map. In the case where the data address has some error associated with it, an exception code is returned to the sender.

3. **Validate Data Value:** The device checks the validity of the data value to determine if the requested action can be performed. In the case of a write, it will check if the data value is something can be written before executing the requested action. Again, in the case where this check fails, an exception is returned to the sender.

4. **Execute Function:** Once all preliminary checks have been passed, the device executes the requested action using the data provided in the PDU. A number of different exceptions can be returned in the case that an error occurs during the execution.

5. **Send Modbus Response:** Once execution has been successfully completed, a response is generated and sent to the requesting device.

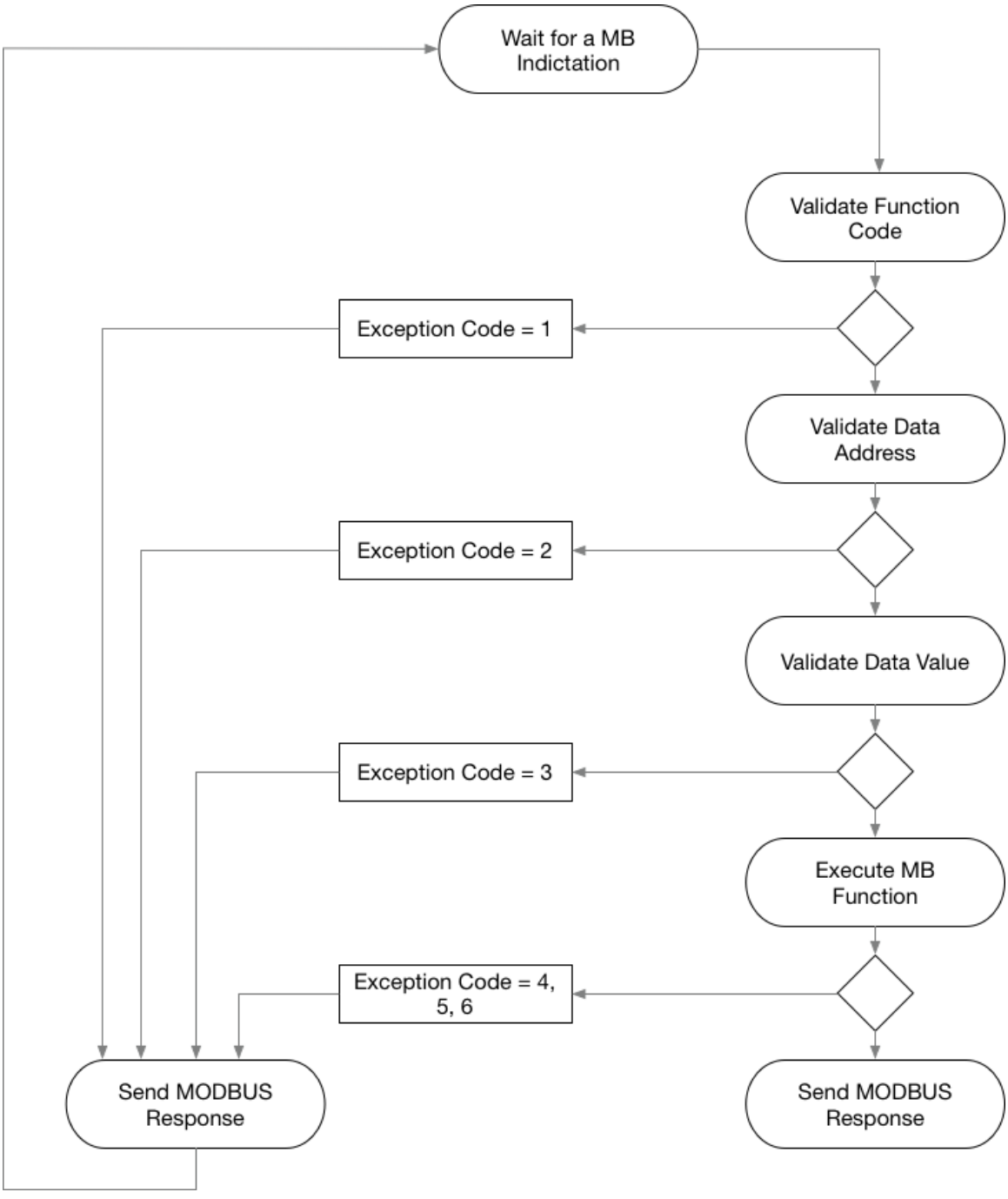

Figure 2.6: Modbus Transaction State Diagram

The simplicity in this architecture plays a key role in the overall popularity of this protocol. Setup and implementation are both easy and straightforward, and because it can be used in nearly any environment, many vendors and manufacturers do not hesitate at implementing the protocol in their products and systems [22, 23]. A key feature this is missing, however, is security. The next subsection outlines how the simplicity of the protocol has been, and will continue to be, used against it.

2.3 Modbus Attacks and Vulnerabilities

In general, attacks on Modbus can be grouped into three categories: (i) attacks that exploit the Modbus protocol specifications, (ii) attacks that exploit vendor implementations of the Modbus protocol, and (iii) attacks that target the infrastructure as a whole, ranging from the information technology, network, and telecommunication aspects. This work focuses on attacks are that against the protocol specifications, and thus are common to all Modbus systems and networks that conform to the protocol specifications. There has been a significant amount of research outlining many vulnerabilities against Modbus, both hypothetical and practical [17, 18, 19, 20, 21]. Additionally, commonly used hacking tools like Metasploit [24] contain a collection of attacks leveraging weaknesses against the protocol to perform malicious actions. Many of these attacks take advantage of the simplistic and open nature of the protocol, and lack of security inherently built in.

There is no way that a Modbus device can distinguish a malicious request from the proper request that comes from the true sender. As described by the California Energy Commission this one of Modbus's critical flaws. "When the master sends a message to the field device, it needs to first authenticate the device from which it obtained the packet and then process the packet. The Modbus protocol lacks this ability and hence middle man attacks can easily take place in Modbus [17]."

Due to Modbus being a completely open protocol, there are numerous works that show traditional attack vectors that are present in Modbus [17, 18, 19, 20, 21]. These attacks,

Table 2.2: Key Modbus Attack Goals

Attacker Goal [19]	Severity of Impact	Underlying Critical Vulnerabilities
Identify Modbus Device [19]	Very Low	Lack of Confidentiality
Disrupt Master-Slave Communications [19]	Moderate	Lack of authentication
Disable Slave [19]	Moderate	Lack of authentication
Read Data from Slave [19]	Moderate	Lack of Confidentiality, Lack of authentication
Write Data to Slave [19]	High	Lack of authentication, Lack of integrity
Program Slave [19]	High	Possible lack of authentication, Lack of Integrity
Compromise slave [19]	Very High	Lack of Integrity, Possible lack of authentication
Disable master [19]	Moderate	Lack of authentication
Write Data to Master [19]	High	Lack of authentication
Compromise Master [19]	Extreme	Lack of authentication

combined with Man-in-th-Middle (MITM) attacks and replay attacks, can give the attacker complete control of the entire network, which can lead to catastrophic results.

In 2004, Byres et al. made use of attack trees to further model the various types of attacks against Modbus, identifying several key goals that an attacker can go after, grouping these attacks into the categories of: (i) general attacks, (ii) attacks against the master, and (iii) attacks against slaves [19]. In 2008, Huitsing et al. further researched attacks on Modbus and successfully performed and categorized dozens of attacks, finding that a large proportion of the high-impact attacks involved a form of interception, interruption, modification, and/or fabrication of control system assets. The attacks were separated based on attacking either Modbus Serial or Modbus TCP, and 20 distinct attacks on Modbus Serial and 28 distinct attacks on Modbus TCP were found, with many more potential attack instances [25].

Based on previous research, Table 2.2 outlines key attack goals that are leveraged by

exploiting the Modbus protocol directly, though there are many others that are not listed. While many of these attacks require access to the system in order to be performed, gaining this necessary access has proven to be easier than expected. Tools like ERIPP [15] and SHODAN [16] show that there are many devices connected directly to the Internet and openly available to send commands to. Other scenarios have shown that companies tend to connect these networks with there enterprise networks, thus giving access indirectly through the Internet. The proposed protocol addresses many of these vulnerabilities and weaknesses, and will simulate several of these attacks to prove the validity of the solution.

2.4 Related Work

To address the lack of security built into the Modbus protocol, several studies have been presented that propose solutions by providing authentication and integrity while attempting to mitigate the various potential attacks. Many of these proposed solutions provide an adequate level of security by using practices that are common in traditional network security. These proposed solutions, however, fall short in practicality, in large part because they fail to take into consideration the end systems, which are commonly older and lack the necessary computing power to perform compute intensive cryptographic operations. Others take these considerations into account, but fall short due to the inability to implement their proposed solutions, where visibility and monitoring are essential in practice.

Fovino et al. proposed a solution that takes advantage of current security best practices by focusing on the traditional security requirements of confidentiality, integrity, and non-repudiation of the message. To provide these guarantees, the authors propose the implementation of a system that uses RSA to encrypt/decrypt messages by using public and private keys to verify authenticity of messages [1]. This approach, while adding security, falls short due to the amount of time and processing power that is needed to constantly perform RSA, making this solution impractical.

More recently, Shahzad et al. proposed a much stronger solution that aims to not only

improve the security in the communication of the Modbus protocol through the use of cryptographic tools, but also take into account the limitations on both ends of the protocol, where systems are much less powerful [26]. This improvement, however, does result in a shortcoming that would cause issues in practice, in that the secure protocol inhibits visibility in the system as a whole. In many systems currently deployed, visibility is key in that there are analytical tools employed to ensure the system is performing as intended. To circumvent this lack of visibility, it would be necessary to deploy even more keys to allow for the analytical tools to gain insight in the protocol, thus increasing the complexity of the overall system. Another drawback to the proposed solution is the requirement to change the overall architecture to support this improvement with the addition of its security development module. Without a fall-back solution for older systems which may not support this change, this solution would cause various issues in practice.

In 2017, Schneider Electric further proposed best practices to secure the Modbus protocol by taking advantage of TLS to employ tunneling. Beyond the loss of visibility that was previously mentioned, this solution also has a drawback by failing to accommodate end systems which may be unable to support the computation power necessary to perform the necessary encryption.

CHAPTER 3

MODSEC DESIGN

The proposed research introduces into Modbus a practical enhancement that provides security to the protocol. The contribution of the ModSec work can be split into two separate goals: to add security to the protocol itself by introducing a fast, efficient, and practical methodology for adding authentication and checking the integrity of messages within the protocol, and to introduce a permission-based scheme to prevent commands that can be introduced into the network by a nefarious actor.

3.1 Security Enhancements

Traditionally, a communications protocol is deemed to have a base level of security if it satisfies the requirements of confidentiality, integrity, and availability, commonly referred to as the CIA triad. These three objectives are traditionally provided through the implementation of various cryptographic schemes, each with a different impact on the transmission delay. Within ICS, however, confidentiality is often ranked as the least important [27]. In systems within ICS, incidents that take the highest priority include the following:

- Blocked or delayed flow of information through ICS networks, which could disrupt ICS operation.
- Unauthorized changes to instructions, commands, or alarm thresholds, which could damage, disable, or shut down equipment, create environmental impacts, and/or endanger human life.
- Inaccurate information sent to system operators, either to disguise unauthorized changes, or to cause the operators to initiate inappropriate actions, which could have various negative effects.

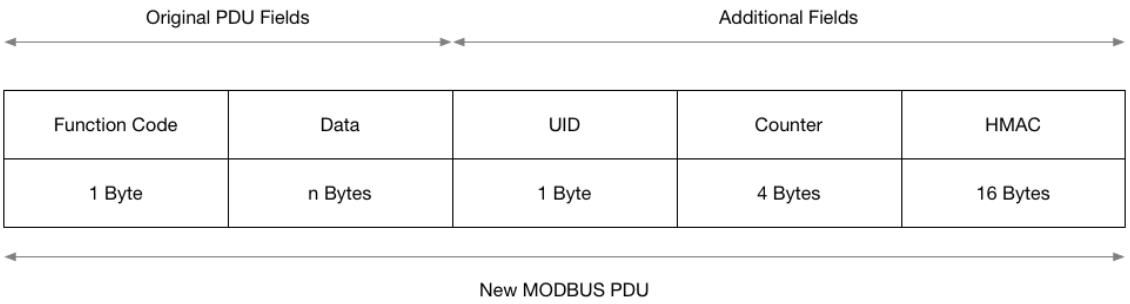

Figure 3.1: Modified Modbus PDU

- ICS software or configuration settings modified, or ICS software infected with malware, which could have various negative effects.
- Interference with the operation of equipment protections systems, which could endanger costly and difficult-to-replace equipment.
- Interference with operations of safety systems, which could endanger human life.

Based on these potentially critical incidents provided by NIST [27], the primary focus of this work is to redevelop the Modbus protocol to address potential issues, as well as the various vulnerabilities that have already been outlined by other research and discussed in detail in Section 2. In terms of Modbus, the protocol should guarantee: (i) integrity, (ii) authentication, (iii) non-repudiation, and (iv) replay protection.

3.1.1 Modbus PDU Modifications

To provide these guarantees, the proposed ModSec protocol modifies the PDU that is used in both Modbus over serial and TCP. The proposed new PDU is seen in Figure 3.1. Because Modbus is transmitted in an open network, there are numerous vulnerabilities that can change or corrupt the message. To add authentication and integrity to the communication channel, the following fields are added to the PDU:

- **UID:** The proposed protocol requires uniquely identifying the sender of the message. A one byte field is added to allow for uniquely identifying up to 255 devices, which

is greater than the current maximum of 247 field devices in a traditional Modbus configuration.

- **Counter:** A four byte counter field is added to uniquely identify each message that is originated from the master node to the slave node. The master node increments this value for each message that is sent to the slave device. The counter is also used to track the length of the time the secret key between the master and slave device lasts. To successfully prevent a potential replay attack, the master must generate a new secret key with the slave prior to the counter wrapping. If the counter value does wrap, the master terminates the key and performs the process to create a new secret key between the devices. Only the master increments this value.

- **HMAC**: A 16 byte keyed-hash message authentication code is appended to the end of each message. The HMAC serves the purpose of verifying both the integrity of the message and the authentication of the message. The HMAC uses the cryptographic key that is shared between the master and slave device to perform the cryptographic hash as defined in RFC 2104 [28].

The original size of the Modbus packet is not fixed, but the size of the PDU is limited due to the original implementation of Modbus using RS232 and RS485. The PDU is limited to 253 bytes in transmission, which results in the Modbus ADU having a maximum of 256 bytes when transmitting over serial and 260 bytes when transmitting over TCP [29]. The proposed solution uses 21 bytes of the Modbus PDU to add the necessary security parameters, thus limiting the data component of the PDU to a maximum of 232 bytes. Because the modifications are limited to the PDU, the implementation will not be affected by the medium, and is thus easily implemented over Modbus Serial and Modbus TCP.

Two separate security developments have been made to address the different forms of communication within Modbus, addressing unicasting and broadcasting separately. Unicasting occurs when a master sends a message to a specific slave device, which is the most

common usage of Modbus in production systems. Modbus also allows for message broadcasting, by which a master sends a message to all slave devices, though this is not very commonly used in practice. Due to the subtle nuances of the protocol, and the implementation of cryptographic functions, unicasting and broadcasting must be treated differently.

3.1.2 Modbus Device Certificates

The proposed protocol leverages several different cryptographic algorithms to secure and verify the Modbus PDU: AES, RSA, SHA-2, and public key cryptography. The algorithms are used to ensure message authentication, integrity, and non-repudiation. Prior to beginning communication, each device must be undergo an initialization scheme to be added to the Modbus ”network”. This initialization scheme involves the creation of public and private keys and X.509 certificates for each device in the network. When adding a new Modbus master device, the following steps are necessary:

1. Generate the device specific cryptographic keys (private and public key pair) using RSA.

2. Generate a unique device identifier that will be used as the UID in the protocol. This identifier needs to be unique in respect to the Modbus network of devices.

3. Identify the permissions this master device must hold to properly operate in the network.

4. Create a X.509 certificate, holding the devices public key, device ID, and permissions.

Similar to the master, each slave device must also undergo an initial setup to be added to the network of devices, outlined in the following steps:

1. Generate the device specific cryptographic keys (private and public key pair) using RSA.

2. Generate a unique device identifier that will be used as the UID in the protocol. This identifier needs to be unique in respect to the Modbus network of devices.

3. Identify the permissions this slave device must hold to properly operate in the network.

4. Create a X.509 certificate, holding the devices public key, device ID, and permissions.

The proposed initialization process can be done in the devices themselves, or by using a separate machine. In the case where the process is completed on a separate machine, the public and private key pairs as well as the X.509 certificate must be securely transfered to the target device prior to any further action. This secure transfer can be done using secure transfer protocols like SSH or SFTP, or through off-line measures such as the use of a USB device.

Following the certificate creation, the master and slave devices must interchange certificates, where the master would be able to access the certificate of the slave device(s), and the slave would have access to the master's certificate. The certificate is important for determining initial authenticity of the sender of the message, as well as for properly determining privilege.

Certificate Authority

To further expand the security provided by the use of X.509 certificates in the protocol would involve introducing a Certificate Authority. By implementing a Certificate Authority, the certificate for each device would be signed and authenticated by a verified entity. This protection would also prevent any attempts at modifying a certificate, as it would immediately cause for the certificate authority to become corrupt.

Leveraging a Certificate Authority, however, was not done in ModSec for various reasons. First, introducing and requiring a Certificate Authority would require further inter-

action with device vendors, as well as constant communication between device vendors and the user base in order to verify the validity of a certificate, as well as to both issue new certificates and revoke invalid certificates. The complexity of introducing a Certificate Authority also largely outweighs the benefits of not having one at all. Beyond relying on device vendors to correctly implement a system and causing processing delays, assuming the devices are physically secure, the self-signed certificates that are manually copied to each device provide the same level of security, as the certificates would not be able to be successfully modified without causing corruption.

3.1.3 Modbus Secret Key Sharing

In order to successfully ensure the authenticity and integrity of the messages between the sending and receiving device, the proposed protocol appends an HMAC, otherwise known as a keyed-hash message authentication code, to each message by hashing the contents of the message with a secret key shared between the two devices. This key, **K**, is shared between the two devices by using the Diffie-Hellman algorithm, which allows for the creation and sharing of a shared secret over an insecure channel. It additionally allows the devices to securely share this key without relying on encrypting and decrypting the message.

The key-sharing technique is implemented into the ModSec protocol by using function codes rather than during the initialization process. Function codes are used because the key-sharing process will occur multiple times through the life of the protocol, rather than just at one time during an initial setup. It is necessary to generate a new secret key **K** each time the four byte counter value wraps to prevent relay attacks. By allocating four bytes of data to the counter field, the master and slave devices are able to communicate 2^{32} times before it would be necessary to exchange keys, but it would also enforce the practice of exchanging keys.

To properly describe the key-sharing algorithm the following variables are used:

- *Alice* = Master device
- *Bob* = Slave device
- S_A = Alice's private key
- P_A = Alice's public key
- S_B = Bob's private key
- P_B = Bob's public key
- p = prime modulus shared between Alice and Bob
- g = prime base shared between Alice and Bob
- a = secret number chosen by Alice
- b = secret number chosen by Bob
- A = Public number calculated by Alice
- B = Public number calculated by Alice
- $K_{Alice,Bob}$ = Secret shared-key shared between Alice and Bob

To perform the Diffie-Hellman algorithm, Alice initiates a new command that encompasses both the prime modulus p and prime base g that will be used by both Alice and Bob to compute the secret shared key, $K_{Alice,Bob}$. Through the Diffie-Hellman algorithm, both of these values, p and g, can be transmitted publicly. To ensure the authenticity of the message, Alice signs the message using her private key, S_A, and the cryptographic hash function *SHA-256* to create a digital signature.

Upon the retrieval of the command to begin the Diffie-Hellman algorithm, Bob first must ensure the authenticity of the message by verifying the signed portion of the message using the Alice's public key, P_A, which was exchanged within the certificate sharing discussed in the previous section. Assuming the digital signature has been verified, Bob sends back a message response with the same prime modulus and base, p and g, but signs the message with his own private key, S_B, again using the cryptographic hash function *SHA-256* to create a digital signature.

$$A = g^a \bmod p \tag{3.1}$$

Upon the receipt of the command from Bob, Alice checks the authenticity and integrity of the message by verifying the signature using Bob public key, P_B. Assuming the message is verified, the master begins the next step of the Diffie-Hellman algorithm, which is to select a secret number, *a* and then calculate the public number *A* using equation 3.1. Following this calculation, *A* is sent to Bob, with the message contents again signed with S_A.

$$B = g^b \bmod p \tag{3.2}$$

Bob, following retrieval and verification of the message, also chooses a secret number, *b*, and uses it to calculate his public number *B* using equation 3.2. Following this calculation, *B* is sent back to the master. This message is again signed with S_B.

$$K_{Alice,Bob} = B^a \bmod p == A^b \bmod p \tag{3.3}$$

Following the successful transmission of these four messages, the Alice and Bob are now able to compute the secret key, $K_{Alice,Bob}$, using equation 3.3. K_{AB} is stored in memory and is to be used for operations moving forward. This key remains valid until the counter wraps, or the master sends a request to begin the key sharing process again.

3.1.4 Modbus Message Transmission

Following the successful creation of the shared secret key, the two devices are able to commence communication similar to the traditional Modbus protocol. The overall flow, however, has additional security measures prior to transmitting a new message as well as prior to processing a received message.

Unicasting

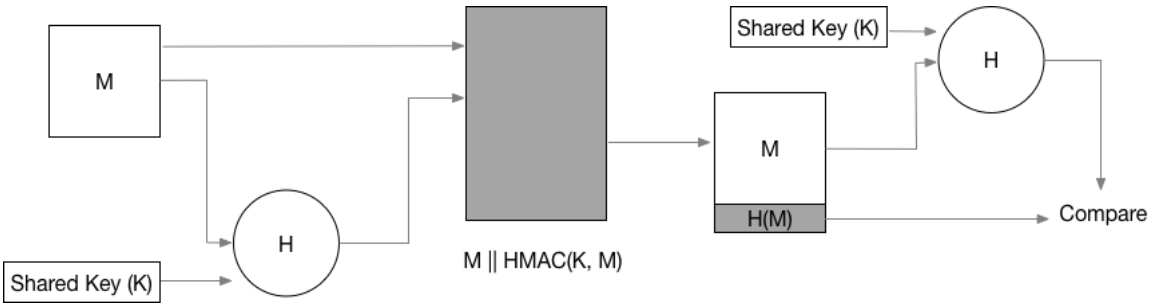

Figure 3.2: ModSec Cryptographic Model for Unicasting

Unicasting is the most common form of messaging within Modbus. The flow of unicasting, however, is different than broadcasting. In the case where the master device is unicasting its messages, or sending a message to a specific device, the new process follows a state diagram outlined in Figure 3.3. The cryptographic model that is followed is also visualized in Figure 3.2 with the following parameters:

- ***M*** = Contents of the message being transmitted. This information includes the Function Code, Data, ID, and Counter Value.
- ***K*** = Shared cryptographic key between the sending and receiving device, previously referenced as $K_{Alice,Bob}$
- ***H*** = HMAC Algorithm defined as:

$$HMAC(K, m) = H((K' \oplus opad)||H((K' \oplus ipad)||m)) \tag{3.4}$$

The specific implementation of the cryptographic functions is as follows:

1. The master prepares to send a new message, ***M***, to the slave device by selecting a function code, gathering the necessary data, retrieving the device identifier, and incrementing the message counter value.

2. The SHA-2 hashing function is deployed on the contents of the message ***M***, using the secret cryptographic key ***K*** that is pre-shared between the master and the anticipated slave device. The result is appended to the message.

3. Upon receiving the message, $\boldsymbol{M}$, at the target device, the device uses the SHA-2 hashing function with the secret key, $\boldsymbol{K}$, from the sender of the device to verify the authenticity and integrity of the message.

4. Assuming the message is correct, the device next checks the certificate of the sender to determine if it has permission to perform the requested action.

5. If all checks are passed, the function is executed, and the response is generated, following the steps 1 and 2.

The outlined parameters succeed in cryptographically ensuring both the authenticity of the sender and the integrity of the message by using the shared secret key between the two devices. Additionally, by leveraging the certificate that is shared between the two devices during the initial setup, authorization is added. In the case where these checks fail due to an invalid hash, the certificate does not exist, the secret key does not exist, or the message is invalid (can potentially occur if the device is not correctly configured to use the proposed protocol), the necessary error code is returned to the sender. These additional errors codes are outlined in Table 3.1.4.

Broadcasting

In scenarios where the master needs to broadcast a message to all devices on the network, the proposed solution does not follow the same process as in the unicasting scenario. Within unicasting, the sending and receiving devices use a symmetric key shares only between the two devices to perform the hashing and ensure authenticity and integrity. In broadcasting, however, it would be infeasible to share another key across all devices in the network. Instead, the master signs the message contents with its private key, and appends the contents with the message as seen in Figure 3.4. This solution continues to provide the same level of security. Authenticity is guaranteed because the master is the only entity able to sign with their private key, and the receiving slave is able to verify by the master through its

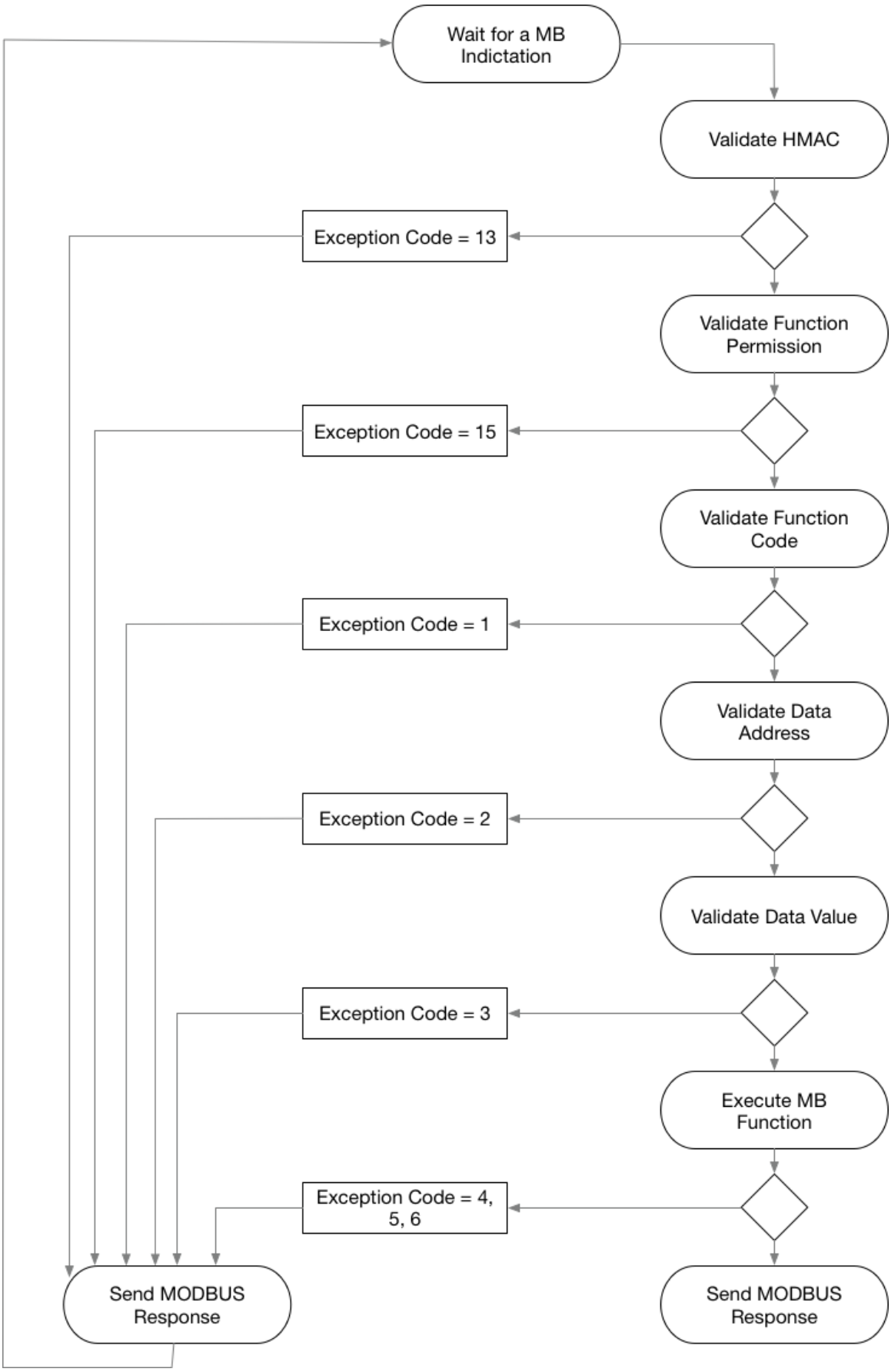

Figure 3.3: Modified Modbus Transaction State Diagram

Table 3.1: New Error Codes

Code	Name	Description
0C	Illegal Message	The query is missing parameters to make the message valid, potentially the the uid, counter, or HMAC that was introduced in the proposed protocol
0D	Invalid Hash	When the receiving device attempts to calculate recalculate the hash, it fails to match that which was sent. Error can occur if there was an error during transmission, a third party attempted to change the data sent, or the user is not who they claim to be.
0E	Invalid Certificate	When the receiving device tries to check the certificate of the sending device and it is either missing or invalid due to tampering. The initialization process will need to be completed again to address this error.
0F	Insufficient Permission	When the receiving device checks the permission of the sender before performing the action and it has been determined that the sender does not have permission to perform the requested action.

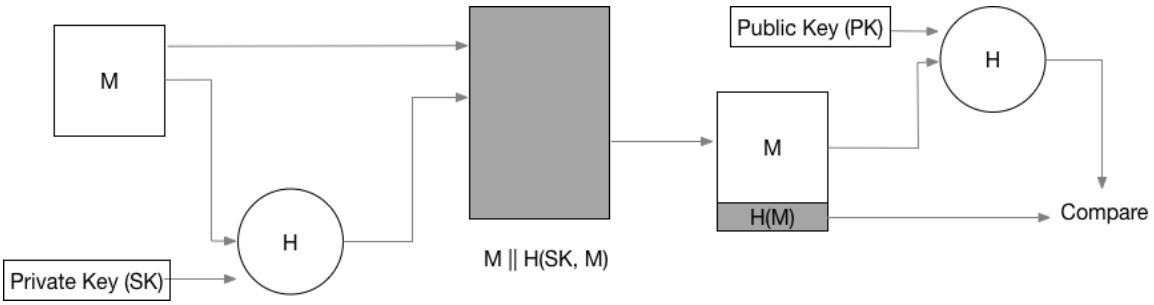

Figure 3.4: ModSec Cryptographic Model for Broadcasting

public key. Similarly, any attempted modifications to the message would discovered with the difference during hash comparison.

A return message from the slave device to the master device does not have to be created because slaves do not traditionally respond to broadcast messages.

In the case where a certificate authority is introduced, it could potentially become more practical to use the same solution provided for unicasting, except additional latency would be added by constantly communicating with the certificate authority.

3.2 Permission Schema

To further enhance the security of the Modbus protocol, ModSec also incorporates a permission scheme to limit devices in the actions that they can perform. As defined in the protocol description [29], function codes fall under the category of either public, user-defined, or reserved by a company or legacy product. Within these categories, the majority of the functions are for the operations of either reading data from a device or writing data to a device. In a Modbus network, only specific devices should have permission to do perform these actions, however. If a nefarious actor were to become apart of the network and send commands to devices, there would be unanticipated and potentially harmful results. If the requested command were to read data, critical information relating to the overall network, or the process that device is connected to, can easily be obtained. By leveraging the diagnostics commands, a third party could easily identify other devices that are apart of the

Modbus network, which is key to the reconnaissance phase of an attack. If the third party were to request a write command, the resulting device has the potential to close an actuator, or rewrite memory or data that is key to the system's overall success. This can also lead to the potential of completely rewriting a program on a PLC or the PLC's entire firmware to execute a remotely triggered attack, in which the effects could be monumental [9, 30].

To address these potential issues, each device is assigned their permission through the initialization process outlined in Section 3.1.2, and these permissions are stored through the lifetime of the device. Once a request has been received and verified, the end device would first check to see if the sender has permission to perform the requested action prior to performing the action and returning any data. In the case where the user does not have permission, a related error code is returned to the sender.

To implement the proposed schema, the permissions are stored on the device along side its public and private keys in a certificate. Introduced in version 3 of X.509, extension fields are allowed to be added to the certificate [31]. These extensions can be custom, as long as is follows the guidelines as prescribed in RFC 3280 [31]. The schema incorporates two custom fields to successfully save and permission information: a permission field and a device identifier field.

The permission field stores 3 bits that are translated to READ-WRITE-CUSTOM. In the first digit, a 0 or 1 is used define if the device has permission to perform read operations from the device. In the second digit, a 0 or 1 is used to define if the device has permission to preform write operations on the device. In the third bit, a 0 or 1 is used to determine if the device has permission to perform user defined, or reserved, functions.

The device identifier is used to store the identifier of the respective device, and is used to lookup the permissions of the requesting device. Storing the device identifier here makes it possible to easily share these certificates across devices and securely maintain the permissions that have been defined during the initialization process.

This extensions field is both a convenient and a secure place to store this information.

Any attempt to modify the certificate, whether the public key or the permissions, causes corruption in the certificate itself, and will return an error code as defined in Table 3.1.4. Additionally, when sharing the certificate with other devices, it is possible to retain these permissions without having to introduce a completely new storage solution for this critical information.

In order for a malicious actor to change these permissions, the actor not only has to sniff the network to learn about the various devices, she/he also must gain physical access to each machine in the network to modify the certificates, a task that is extremely difficult.

CHAPTER 4

MODSEC IMPLEMENTATION AND EVALUATION

The Modbus protocol has been implemented in hundreds of thousands of devices around the world, ranging from the open source community, to the Modicon group, to the vendors of SCADA devices themselves. While there are numerous attacks that can be potentially leveraged on the many different implementations, the focus of this thesis is aimed to address concerns within the protocol itself. To modify the protocol to meet the specifications described in the previous section, the open-source Modbus library libmodbus was modified to display the impact of the proposed protocol, as well as to evaluate the expected performance changes that have arisen. The key changes included the addition of new function codes to perform the cryptographic key sharing, changes in the protocol and flow to support the additional bytes of the PDU and checking the validity of data, and permission checking by leveraging the certificates that are generated on each machine.

4.1 Public Key and Certificates

Prior to implementing the protocol and beginning communication, it is necessary for each device to create their public and private keys, as well as create the certificate that is used to validate the communication. Following the creation of the certificates, the use of a secure channel is necessary to transfer the certificates to the device. The secure channel can be created by leveraging popular protocols like SSH or SCP, or by transferring the certificates off-line.

To perform the creation of the public key and certificate, the commonly used library OpenSSL is leveraged. OpenSSL is used commonly to secure communication in many communication schemes, and provides the ability to generate both public and private cryptographic keys as well as the necessary X.509 certificates. The OpenSSL library is openly

available to download and use on a system. The size of the library is large and the process of building OpenSSL requires a a lot of memory, so the creation of the certificates can be done on a machine that is separate from the target machine. Once the OpenSSL library has been installed on the machine, it is possible to create the certificate using a series of commands and a custom configuration file. Appendix A shows an example configuration file that is used to create the certificate. In the configuration file, two unique fields are created with an OID of ”1.2.3.4.5.6.7.8” and ”1.2.3.4.5.6.7.9”, which are both arbitrarily chosen for uniqueness. According to the standard, it is necessary that this value does not collide with other commonly used OID's, though because the certificate is used in a closed setting, it is not necessary to undergo the process of registering the value. These values are both defined using the ANS.1 format, where both values are stored as UTF8 strings that can be later parsed. The device identifier in the example is set as the value 12. The permissions are configured as 111, which translated to having the permission to perform the operations of READ, WRITE, and OTHER. The other information in the configuration file is added for completeness, but can be modified for accurate usage during implementation. When creating the certificate for the protocol, it is necessary to ensure that the device id is a unique value within the network.

To generate the certificate files, the following OpenSSL command can be issued, resulting in the certificate files:

Listing 4.1: OpenSSL Generate Certificate Files

```
openssl req -x509 -nodes -days 365 -newkey rsa:1024 -keyout
   cert.key -out cert.pem -config req.cnf -sha256
```

The command expects the configuration file to be named as ”req.cnf”, though this can be easily changed. The output of is a key file and a pem file that is generated using the RSA algorithm with a 1024 bit key and the SHA-256 algorithm. The certificate is also valid for 1 year, or 365 days, though this value can also be changed. To examine the

resulting certificate file, the commands differ depending on if the generating machine is a Unix machine or a Windows machine.

In the case where the machine is Unix-based, the following command prints out the certificate file in a readable format:

Listing 4.2: OpenSSL Visualize Certificate File (Unix)

```
openssl x509 -in cert.pem -noout -text
```

The command prints out all of the data in a readable format. The necessary protocol information is printed under the X509v3 extensions section, where under the section with our custom OID "1.2.3.4.5.6.7.8", both the device identifier and the permissions are visible.

In the case where the machine is Windows-based, it is first necessary to convert the PEM file into a format that is readable by Windows. The tool CertUtil [32] is a command line tool that is commonly pre-installed on Windows machines because it plays a critical role in Certificate Services. The tool can be leveraged to convert the format to the CER file format through the following command:

Listing 4.3: OpenSSL Visualize Certificate File (Windows)

```
CertUtil -decode cert.pem cert.cer
```

Following the conversion, it is possible to now simply open the file, where the details show all information of the certificate, including the necessary device identifier and permissions that were initially assigned.

Upon the completion of the generation of the certificate, it is necessary to move the files to the target machine through a secure method. Assuming devices are on the network, it is possible to use either secure copy (SCP) or the secure file transfer protocol (SFTP) to move the files from the primary machine to the target machine. In the case where the devices are not on the network, other secure methods, like using a USB key, can be leveraged to also move the files. Insecure methods like telnet, however, cannot be used due to the fact

that if a nefarious actor were to be in the network, they would now also be privy to the certificate file, thus giving them the necessary information to extract messages and pretend to be another device.

4.2 Key Exchange

Two of the key additions to the Modbus protocol rely on successful cryptographic hashing the data, and performing validation of the received data. To implement these forms of security, it was necessary to implement two new function codes to perform the cryptographic key exchange. By leveraging the logic that is involved in the Diffie-Hellman algorithm, it is possible to secretly share a key between two parties on an open network.

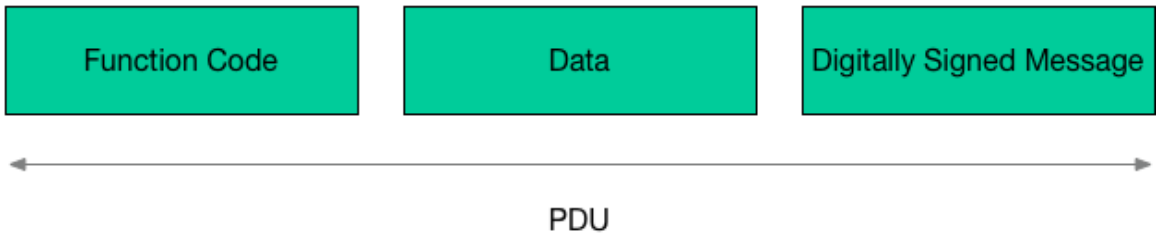

Figure 4.1: ModSec PDU for Key Exchange

Performing the Diffie-Hellman algorithm is completed by introducing two unique function codes, 100 and 101, to complete the two part exchange between the master and slave devices. Unlike the original PDU that is described in Figure 2.3, or the modified PDU described in Figure 3.1, function codes 100 and 101 will create a PDU that is described in Figure 4.1. Because the two devices have not already posses a valid secret key shared between them, the only way to validate the messages are both authentic and maintain integrity is for the sender to sign the message contents with its private key. The two devices have already shared their certificates, allowing for the device to use the senders public key to verify the contents of the received message. The following subsections further describe the two new function codes, including expected responses.

4.2.1 Function Code 100 (0x64): Begin Key Sharing

The function code 100 is used to begin the Diffie-Hellman algorithm between a master (Alice) and slave device (Bob). Alice must first generate, and then send a prime and a primitive root of the prime, along with the message digitally signed by the private key to Bob. A normal response from Bob is the same prime and primitive root sent by Alice, however signed with his own private key. Upon receipt, the two devices begin the first step of the Diffie-Hellman algorithm. A normal request and response scenario is outlined below.

Request

Function Code	1 byte	0x64
Public (Prime) Modulus	8 bytes	Calculated by the master
Public (Prime) Base	4 bytes	Calculated by the master
Signed Message	256 bytes	Calculated by master

Response

Function Code	1 byte	0x64
Public (Prime) Modulus	8 bytes	Calculated by the master
Public (Prime) Base	4 bytes	Calculated by the master
Signed Message	256 bytes	Calculated by slave

4.2.2 Function Code 101 (0x65); Complete Key Sharing

The function code 101 is used to complete the Diffie-Hellman algorithm between a master (Alice) and slave device (Bob). This function code is used in combination with the previous function code introduced to allow for the two devices to successfully share the secret cryptographic key. Assuming the first part of the key sharing algorithm has been completed, Alice originates the next message, which contains her newly calculated public key (in respect to the Diffie-Hellman algorithm) to Bob. A normal response from Bob is to send its newly calculated public key (in respect to the Diffie-Hellman algorithm). Upon receipt, the two devices are now able to use the received values to complete Diffie-Hellman with a

calculated secret key. This secret key is now stored in memory and used for all operations from this point on. A normal request and response scenario is outlined below.

Request

Function Code	1 byte	0x65
Public Number ***A***	8 bytes	0x0000 to 0xFFFF
Signed Message	256 bytes	Calculated by master

Response

Function Code	1 byte	0x65
Public Number ***B***	8 bytes	0x0000 to 0xFFFF
Signed Message	256 bytes	Calculated by slave

4.3 Modified Code Flow and Structure

To implement the ModSec protocol, the open source software libmodbus [33] was leveraged as a starting point. Libmodbus is an actively developed project that is freely available on Github [34] to modify, as well as contribute to. The implementation is based on the Modbus specifications [29], and has numerous pulls and stars, proving that the project is actively used in the open source community, potentially in commercial industries as well.

A large portion of ModSec relies on the presence of a shared key between two entities. To implement this process, two function codes were implemented as described in section 4.2. The function codes were added to the source code similar to the other function codes that are previously available in Modbus. To provide cryptographic security to the messages, OpenSSL was also leveraged. In the case of both function code 100 and 101, the necessary changes to the Modbus PDU occur at the end of the PDU, and by taking note of the requested function code, the necessary changes are simply appended.

In the case of function code 100, where the master (Alice) initiates the key sharing process, Alice leverages the Diffie Hellman portion of the OpenSSL library to generate the initial parameter necessary for the algorithm. Both of these values are stored internal to the

program as a part of the entire Modbus context, and both the numbers are also appended to the message. Following this, the Alice digitally signs the message by using another portion of the OpenSSL library and the parameters that were initially defined in the devices certificate file.

Upon the receipt of this message, the recipient (Bob) first verifies Alice's message through the same digital signature process and by using Alice's public key, which was shared in the certificate file. Bob then saves these two initial parameters to his Modbus context, and returns the data, except with his own signature. The two values must be the same in order for the Diffie Hellman algorithm to work as intended. With the use of the two OpenSSL methods, the first phase of the Diffie-Hellman algorithm is completed.

In the case of function code 101, where the master (Alice) intends to complete the Diffie-Hellman algorithm began with function code 100, the code flow is again similar to that of the initial Modbus protocol. The changes are again appended to the message after the function code, and thus no changes to the overall code flow are necessary to begin with. Prior to sending the message, however, the first check is to confirm that the generated Diffie Hellman parameters, the prime modulus and prime base, are stored internally. Next, the OpenSSL library included the method "EVP_PKEY_keygen" to create the public key that is to be shared with Bob. This key, again stored locally in the Modbus context, is appended to the message, followed by the digital signature of the message. This message is then sent to Bob.

Once the slave (Bob) receives function code 101 from the master, it verifies the message through the signature using OpenSSL as before, and then uses OpenSSL to generate its own public key to be shared. Along with signed and sending the message back to Alice, Bob also uses his newly generated public key and Alice's shared public key to create the private key using the OpenSSL function "DH_compute_key". Once Alice receives and verifies Bob's message, Alice also uses the function "DH_compute_key" to calculate the same key. This private key is then saved internally and used for all future messages between Alice

and Bob.

Upon the successful sharing of the secret key, the code flow changes to check the authenticity, integrity, and permission of each message prior to actions. Failure to pass these checks result in an error code being returned.

Throughout the lifetime of each Modbus server instance, their is a context that is used to saved certain key variables. In addition to public and private keys as described above, variables such as a the counter are saved and updated upon sending and receiving of messages from devices. The file ”modbus-private.h” is used define the protocol information including the header, ADU, and checksum that is implemented when using both Modbus RTU or Modbus/TCP.

To add security against various forms of integrity attacks, the counter plays a key role in ensuring messages below the current count are not replayed, thus acting as a window similar to that as described in the anti-replay sub-protocol that is used in IPsec. Upon the receipt of each message, a check to confirm the validity of the message is done by comparing the message's counter value with the current received counter value. If the value is lower than the last received message, then the message will be returned with an exception code, displaying that the message is invalid. This counter variable is only incremented by the master, and does not have the ability to decrement. It is, however, reset during the key exchange process. Because the size of the counter field in the ModSec protocol is 32 bits, once the value wraps, all messages will automatically be marked as invalid due to this check. This is, however, the intended nature by forcing a new key to be exchanged when this wrapping occurs, thus preventing potential replay attacks that can occur when the values wraps. This check occurs prior to the sending of any messages.

The newly modified code flow is best depicted in Figure 3.3, where each additional check is outlined in detail. ModSec introduces several key steps to verify the message, all of which occur prior to a successful read or write action. To successfully add these security features, several functions were created and added to the code flow. Upon the successful

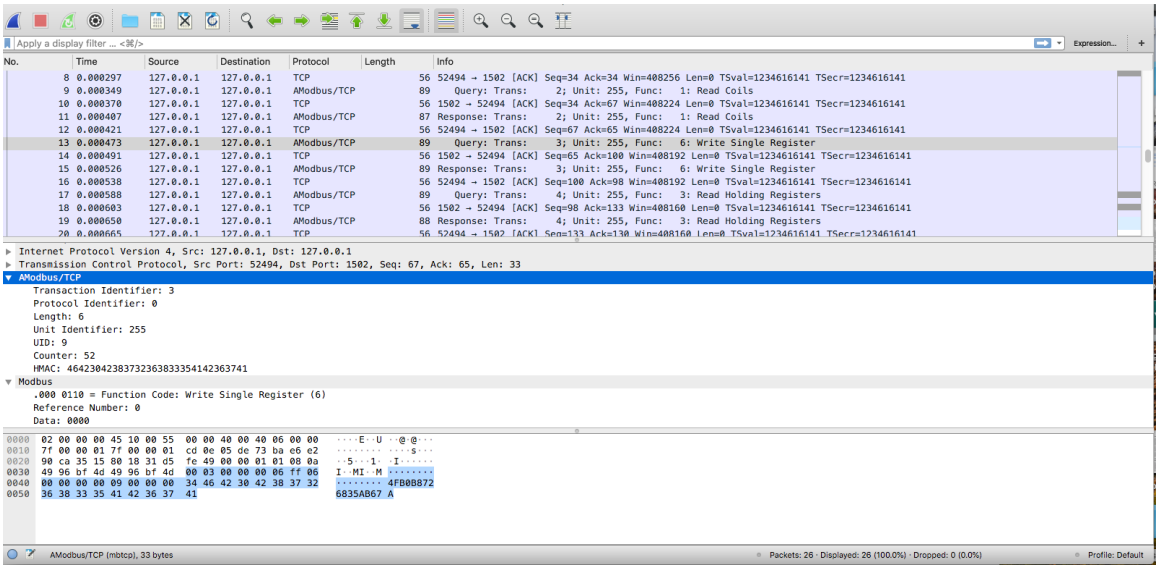

Figure 4.2: ModSec Write Register in Wireshark

validation of authentication, integrity, and permissions, the the resulting traffic resembles that of Figure 4.2 and Figure 4.3. In Figure 4.2, the master sends a command to the slave to ”write single register”, but appended to the data to write (00 00 00 00) is also the UID, counter, and HMAC that is appended to the original Modbus message. Similarly, in Figure 4.3, a slave received a command from the master, and upon checking and validating the messages authenticity, integrity, and permissions, it performs the ”read holding registers” command, and thus returns the requested data.

The first function added to the code base, entitled _modbus_calculate_counter, serves the purpose of ensuring the validity of the shared key. As described earlier, the secret key is only to be used for a limited number of iterations before needing to be expired and replaced with a new key. The counter variable is stored in the overall Modbus context and is updated prior to each message, thus easily accessible. In the case where the counter is too high, an error code is sent in place of the desired message, notifying the sender and receiver that it is necessary to create a new secret key.

The next function, entitled _modbus_calcaulate_hmac, calculates the Hashed Message Authentication code that is to be appended to the message prior to sending. To calculate the

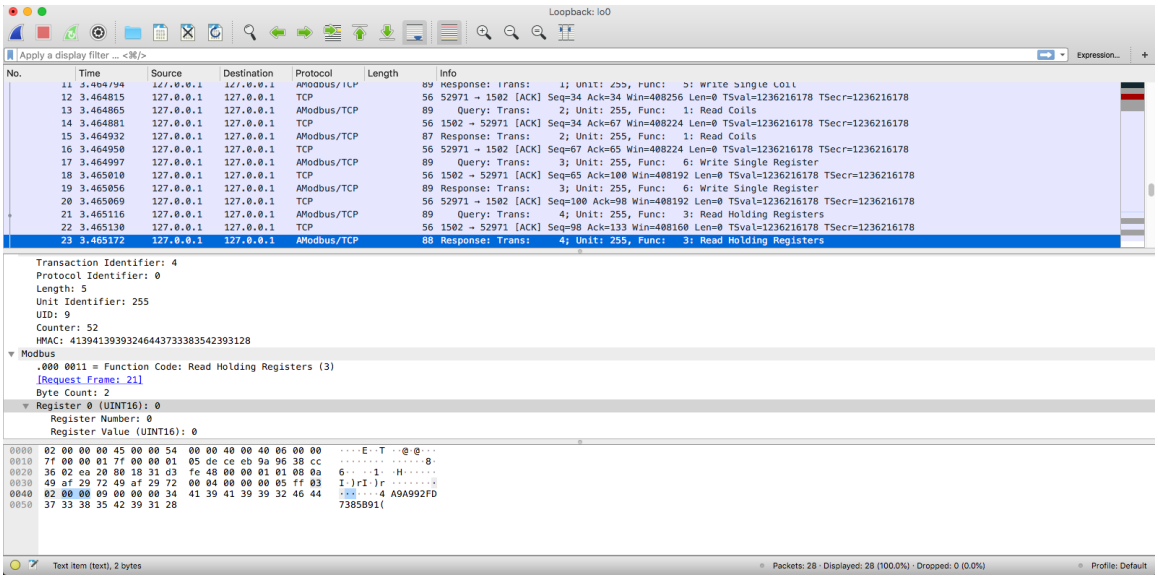

Figure 4.3: ModSec Read Register in Wireshark

HMAC, the open-source library OpenSSL is leveraged. Using the library, it is possible to both sign and verify the signature with the secret key that is shared between the sender and recipient. In the case where no key is shared, the error code OE is sent by the sender instead of the message, notifying the sender and recipient that the message cannot be sent due to an invalid secret key. In the situation where the key is valid however, it is used with the message to create a signature that is then appended to the message. The library leverages the SHA256 algorithm, resulting in a 32 byte hash digest. As noted in the Figure 3.1, ModSec uses 16 bytes for the HMAC, not 32, thus requiring for the digest to be truncated to 16 bytes. The HMAC RFC [28] defines that truncation of the digest is valid as long as the output length is not less than half of the hash output, as well as over 80 bits; thus the truncated hash digest is both valid and secure. The truncation is performed by using the 32 leftmost bits of the hash, also which is recommended in the RFC. The results of this function are used for both appending the hash digest to the message, as well as to verify the security of the message as whole.

Following the change in the structure, it was necessary to implement the HMAC code to both perform the hashing operations on the message structure, as well as to verify the

validity and authenticity of the structure. By leveraging the OpenSSL library, the HMAC portion of the code was used to perform the keyed-hashing operation on the message prior to sending the message, as well as upon receipt of a message to confirm validity.

While using an HMAC provides the security guarantees as described in Chapter 3, it was also discussed that in the case of sending a broadcast message, using an HMAC would become impractical due to the substantial number of keys that would be required. To add security to this form of messaging, the sending device signs the message with its private key, and the receiving device uses the sending devices public key to verify its authenticity.

Prior to sending the message, the UID of the device, the counter of the message, as well as the HMAC is added to the end of the message.

Upon the receipt of the message, the OpenSSL library is again leveraged to now verify the authenticity and integrity of the received message. By removing the appended HMAC from the received message and then calculating the hash digest, leftmost 16 bytes of the hash digest are compared to that which is transmitted. In the case where their is a mismatch, then it is known that an error occurred during the transmission of the message, and an error code is sent as a response. While the cause of the transmission error is unknown, any further actions are stopped, preventing any unintended or malicious events from occurring.

Prior to any execution of any action, it is necessary to check the device requesting said action has the proper permissions. Assuming the certificates were exchanged properly. the certificate can be parsed using the X.509 portion of the OpenSSL library.

When parsing through the certificate, it is necessary to validate the certificate. Because the protocol uses self-signed certificates instead of leveraging a certificate authority, the primary form of validation is the issuance and expiration dates.

Following this confirmation, the next goal is to validate the device has permission to perform the requested action. Within OpenSSL, it is possible to parse the stack of X.509 extensions. In the example certificate, the OID of ”1.2.3.4.5.6.7.8” was chosen to represent the device ID, and the OID of ”1.2.3.4.5.6.7.9” was chosen to represent the permission.

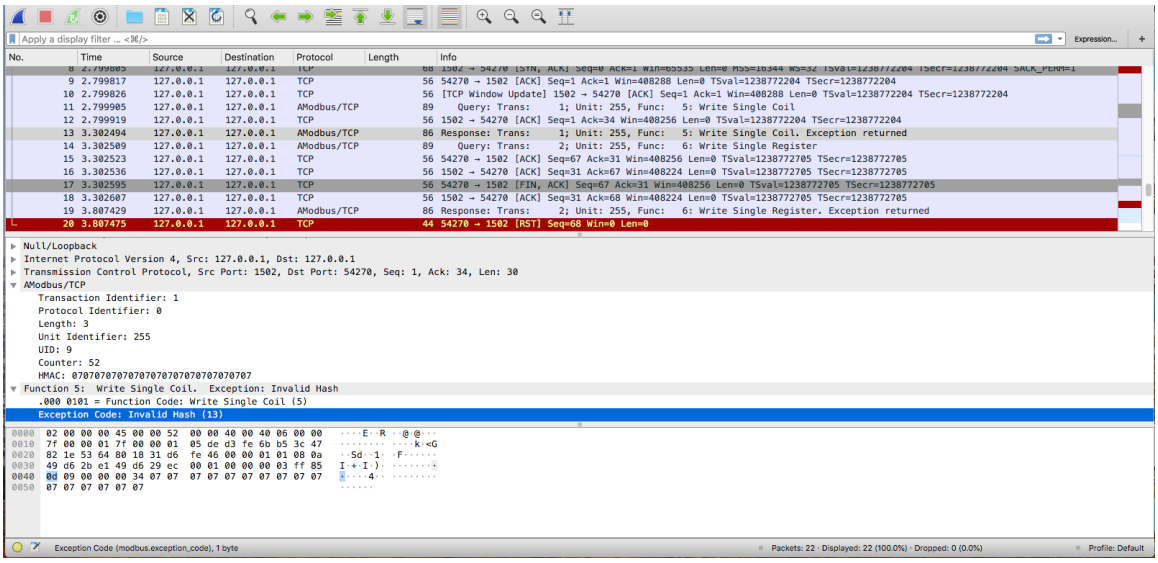

Figure 4.4: ModSec Invalid Hash

During this check, the value, which is stored in ANS.1 is parsed and converted to a string, which is then parsed to determine if permission is granted for the command.

Assuming all of the additional security parameters have passed successfully, the Modbus protocol will proceed as intended and uninterrupted, allowing for all of intended actions to be performed successfully. The overall code flow does not change greatly, and with the small footprint of authentication, integrity, and permission checks, the overall security of the protocol is increased greatly. In the case where one of these checks security checks fail, however, an error code is instead transmitted, preventing any incorrect or malicious activity from occurring. Figure 4.4 shows an example of Wireshark traffic where an error does occur. In the example, the transmitted hash was not incorrect, thus resulting the an error code and no further action. The modified source code of libmodbus can be found on Github in my public repository [35].

4.4 Wireshark Modifications

In order to properly display the packets in Wireshark, several modifications to Wireshark were necessary. Due to the open-source nature of the source code, it was possible to down-

load the code from GitHub and make modifications directly. In order to make the modification, it was necessary to create a custom dissector, through which the purpose is to accurately identify and display the protocol. Using the already provided source code as a guide, creating the dissector involved noting various protocol fields, lengths, and expected types and results. The source code is provided also in my public Github repository [35].

4.5 Performance Variances

To properly measure the performance of the proposed protocol, the following section evaluates both the changes in message size, as well as the efficiency of the proposed cryptographic algorithms in comparison to encryption the message as a whole, as suggested by other works.

4.5.1 PDU Size Changes

Function	Modbus TCP	ModSec	Overhead
Write Coil (0x05)	11 bytes	32 bytes	191%
Write Register (0x06)	12 bytes	33 bytes	175%
Write Multiple Coils (0x0F)	260 bytes	281 bytes	8%
Write Multiple Registers (0x10)	260 bytes	281 bytes	8%

Table 4.1: Comparison of packet size.

To add the necessary security measures to the protocol, several bytes were added to the PDU of the Modbus message, including a 1 byte UID, 4 byte counter, and a 16 byte HMAC. While adding these additional fields does not extend the maximum size of the PDU, it does increase the overall size of the message transmitted. Because the maximum PDU size is not changed, the transmission of the message will not cause for the packet to exceed the size of the maximum transmission unit (MTU). There will, however, be a longer transmission time based on the larger packet size, though this speed will be based on the transmission medium. Table 4.1 shows a comparison in packet size of commonly used Modbus function

codes, and shows that while there is a difference, the amount of communication latency should not be noticeable.

4.5.2 Processing Time and Cycle Count

By introducing cryptography to the Modbus protocol, it is necessary to measure the real-world performance of the cryptographic functions in order to properly evaluate the efficiency of the protocol. To measure the efficiencies of the algorithms chosen, speed benchmarks provided by [36] are used to display the overall affect on the transmission and handling of messages. The benchmarks provided were calculated using a Skylake Core-i5 with a CPU frequency of $2.7e^{09}$Hz.

Digital Signature

Function	**# of Bytes**	**Clock Cycles (Signing)**	**Processing Speed (Signing)**
Write Coil (0x05)	16 bytes	70,150 Cycles	1.03 milliseconds
Write Register (0x06)	17 bytes	70,159 Cycles	1.03 milliseconds
Write Multiple Coils (0x0F)	265 bytes	72,486 Cycles	1.04 milliseconds
Write Multiple Registers (0x10)	265 bytes	72,486 Cycles	1.04 milliseconds

Table 4.2: Digital Signature Performance on Common Modbus Functions.

To create a secure digital signature for authenticating the message through various portions of the protocol, the SHA-256 hashing algorithm was employed with the 2048-bit RSA signing algorithm. To evaluate the overall performance of both the signing and verifying processes, each algorithm is measured separately.

The SHA-256 hashing algorithm is incorporated to first perform a hash on the message contents. The cryptographic hash of the message is used for the signature creation to reduce the mutability risk for the RSA signature, as well as to creating the RSA signing process more efficient. The hashing process for SHA-256 takes an average of 9.38 cycles per byte, or 275 MiB/second.

The RSA signature scheme is then employed to create the secure digital secure on the hash of the contents of the message. For signing a message, RSA takes an average of 2.78 Megacycles for each operation, or approximately 1.03 milliseconds per operation. For verifying the signature of the message, RSA takes an average of 0.07 Megacycles, or approximately 0.03 Milliseconds per operation. Table 4.2 shows the estimated average number of clock cycles and transmission speed to create and verify the digital signature appended to the contents of the message.

The process of creating a digital signature for a message occurs in two instances: for performing the cryptographic key sharing and for signing messages that are broadcast from the master to all slave devices.

In the example of a master broadcasting a message to the slave devices, the processing time and speed for the write coil commands is as follows. To perform the SHA-256 hash on the 16 byte message would take approximately 55.49 nanoseconds. By adding this time to the 1.03 milliseconds for the signing of the message, it would take approximately 1.03 milliseconds. Once the device received a message, the signature must be verified as valid. As for the verification of the signature, the process would take approximately 0.0301 milliseconds for the 16 byte message. Table 4.2 shows the performance calculations for performing the digital signature process prior to the transmission of each message.

HMAC

Function	**# of Bytes**	**Clock Cycles**	**Processing Speed**
Write Coil (0x05)	16 bytes	150.72 Cycles	55.89 ns
Write Register (0x06)	17 bytes	160.14 Cycles	59.34 ns
Write Multiple Coils (0x0F)	265 bytes	2496.3 Cycles	925.72 ns
Write Multiple Registers (0x10)	265 bytes	2496.3 Cycles	925.72 ns

Table 4.3: HMAC Performance on Common Modbus Functions.

Implementation of the HMAC algorithm involves not only the hashing of message and the key, but also requires the setup of the Setup Key and Initialization Vector. In cryptog-

raphy, an initialization vector is a fixed-size input to a cryptographic primitive that is either random or pseudo-random. In the proposed protocol, the HMAC uses SHA-256 with a 128-bit key to ensure the necessary security measures. For this algorithm, the averages amount of clock cycles necessary to setup the initialization vector is 590, or approximately 0.219 microseconds. The hashing process takes an average of 9.42 cycles per byte, or 273 mebibytes/second.

The calculation of performing the HMAC on the message occurs prior to sending each message in the protocol, as well as on the receipt of each message to ensure the authenticity and integrity of the message. Table 4.3 shows the calculated average number of clock cycles and transmission speed to perform the HMAC on commonly used commands within the Modbus protocol. To use the Write Coil function code as an example, the message would contain a total of 16 bytes within ModSec, prior to appending the result of the hashed message. To perform the keyed hash on the message would take approximates 55.89 nanoseconds. Including the time it would take to setup the initialization vector, the average time for transmitting the message would be approximately 274.89 nanoseconds. Within the HMAC algorithm, however, the initialization vector is static, meaning that the setup of the initialization vector would only occur once. Thus, the average time for each message transmission is only 55.89 nanoseconds added, a small time sink for adding both authentication and integrity to the message.

Overall Impact

ModSec does not introduce encryption into the protocol because of the heavy dependencies that are required, as well as the heavy toll and delays that would be caused by constant encryption. Considering the age and processing power of many of the devices that use Modbus, avoiding encryption while adding necessary security measures were key to the design of ModSec.

4.6 Attacks on Modbus

In order to measure the performance of the new schema, several attack scenarios were designed to disrupt the normal flow of communication. Many of the attacks and vulnerabilities that were discussed in Section 2 are openly available using tools like Metasploit [24], or can be easily replicated using common open source networking and penetration testing tools. To evaluate the effects of these attacks, as well as to prove the validity of the proposed protocol, the basis of many of the common attacks on the Modbus protocol have been replicated using both the traditional Modbus protocol as well as ModSec. Attacks that are made by leveraging the lack of confidentiality are not addressed in the proposed protocol, though this is a calculated limitation that should be addressed by other security means. As described in NIST'S guide for ICS security [27], confidentiality is the least important aspect of security in respect to SCADA and ICS systems, and by implementing encryption to defend against this weakness, the solution can become impractical to systems that are in production.

4.6.1 Man-in-the-Middle Attacks

Based on the open nature of the Modbus protocol, once an attacker gains access to the network, viewing the traffic and gaining information on the network is extremely trivial. Using openly available Modbus scanning tools like ModScan [37], the protocol can be leveraged to quickly discover available slaves within the network, which is key to the reconnaissance phase of an attack. Due to the lack of security in the current protocol, particularly the lack of authentication, performing a man-in-the-middle attack is possible and one of the most potentially harmful potential on the Modbus protocol. In a man-in-the-middle attack, the attacker logically takes measures to trick the communication to flow through itself and then to the targeted device, rather than from the sender to the targeted device itself, as seen in Figure 4.5. Within Modbus, the target machine has no indication that the message came

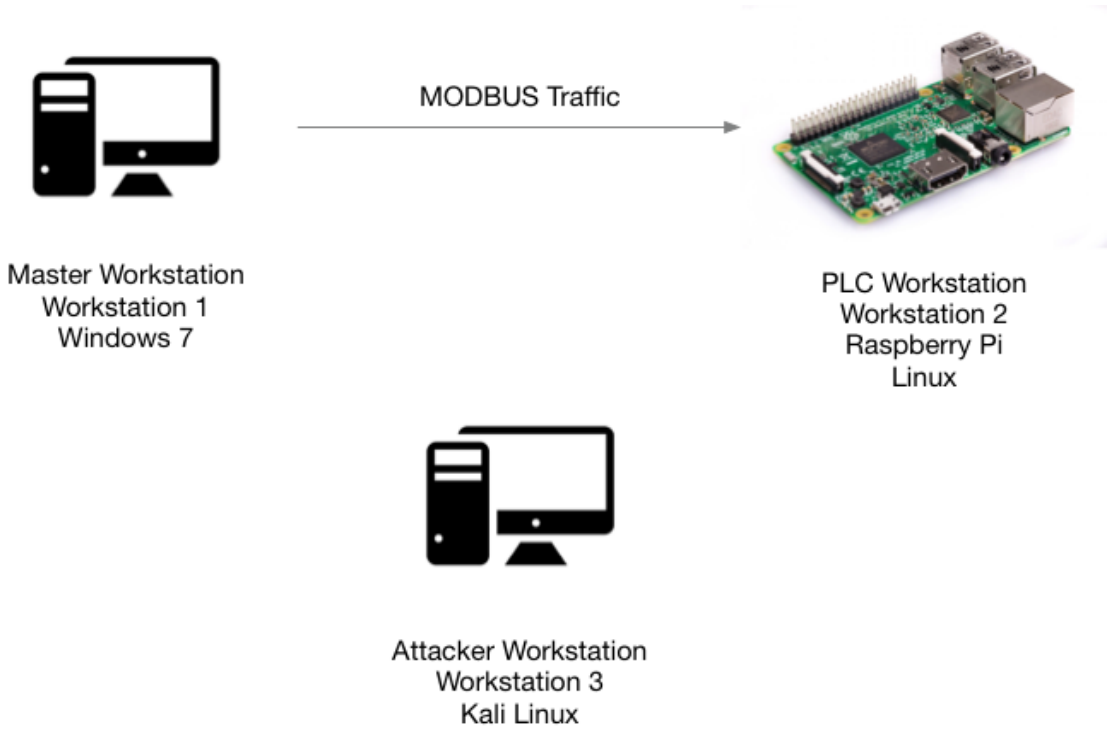

Figure 4.5: Modbus Server/Client Test Setup

from the master, or anyone else, and is configured to simply perform the task if possible, and return a response. Based on this nature, an attacker can intercept messages on the network, and send harmful messages to a slave by posing as a master.

To explain this scenario, 3 workstations were configured on a network to simulate the attack as follows: (1) a master workstation running Windows 7 and Modbus Poll [38]; (2) a PLC workstation running a standard Linux Debian Operation System and ModbusPal [39]; and (3) a simulated attacker workstation running Kali Linux. Using Modbus Poll, an open-source tool provided by Modicon to allow the simulation of a master device in a Modbus/TCP network, and ModbusPal, another open-source tool used to allow the machine to act as a slave and visually display the changes the of the internal data, machines 1 and 2 are able to communicate using the traditional Modbus protocol over TCP. The master is able to perform traditional actions in a Modbus environment like settings coils, reading and writing data, performing diagnostics, and various other tasks as defined by the Modbus

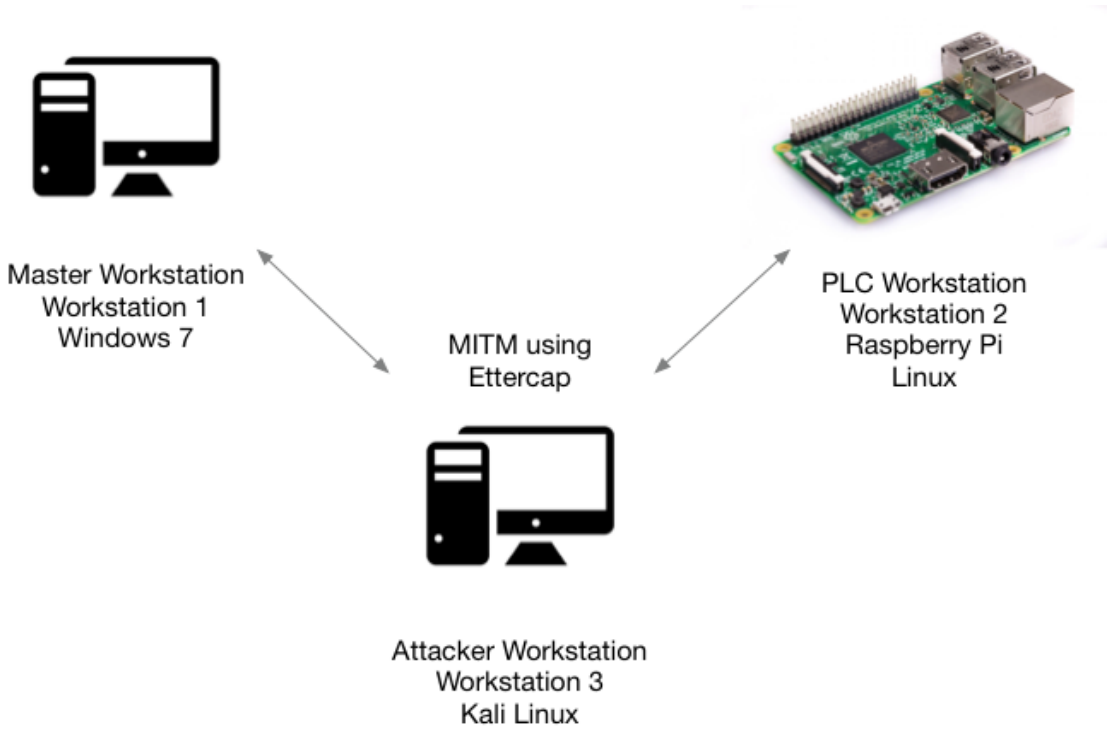

Figure 4.6: Modbus Server/Client MITM Traffic

protocol [29].

To demonstrate the man-in-the-middle attack, the third workstation is configured on the network with the intent of maliciously modifying the target machine. This workstation is configured using Kali Linux, which is a Debian-based Linux distribution configured with various, open-source networking tools aimed to assist in performing penetration testing and security auditing. Because the workstation being within the network, with the help of Wireshark, the it is able to passively view the network traffic between machines 1 and 2. To perform the MITM attack itself, the tool Ettercap was leveraged to allow the workstation to insert itself in the middle of the communication. Ettercap is an open-source networking tool that is focused on performing man-in-the-middle attacks on a LAN. Within Ettercap, the following steps were performed to exploit the open Modbus communication channel between workstation-1 and workstation-2:

1. Enter unified sniffing mode, allowing the tool to sniff network packets and gain in-

sight on the overall network.

2. Scan over hosts, allowing the tool to find both the master workstation and the PLC workstation. The results of the scan include the MAC addresses of the two devices on the network.

3. Use the retrieved MAC address to leverage ARP poisoning. The result of the ARP poising/spoofing is the linking of the attackers MAC address with the IP address of the legitimate computer on the network, evident in Figure 4.6

4. Create an Ettercap filter to modify Modbus TCP communications coming from the Master workstation with a destination of the PLC workstation.

Ettercap allows for the user to create a filter to further sort the data and modify the network packets. The example below in Listing 4.4 shows a small code snippet that was used to change a message from the master to the slave device, commanding for coil to be turned on. The code snippet captures this message, and modifies it forcing the slave to instead turn off the coil. In a real world scenario where the coil may be connected to an actuator, flipping this value obviously can lead to a detrimental situation.

Listing 4.4: Example Ettercap Filter

```
if(ip.proto == TCP && tcp.dst == 502) {
  if(search(DATA.data, "\xff\x00") {
    replace("\xff\x00", "\x00\x00");
   }
}
```

With the help of Ettercap, the attacker (workstation-3) is able to successfully modify commands sent from the master device to the slave device. A simple example is where the master sends a write command, aiming to set the value of a coil to a specific number, but the attacker is able to successfully change the response nefariously. With this setup, the master

is completely unaware that this extra activity is going on, and with further configuration, Ettercap would be able to respond to the master with the relevant information to further trick the device into believing all operations are performing as intended.

While straightforward, this simple example outlines one of the largest weaknesses of the Modbus protocol. MITM is the basis for many of the attacks that have been outlined in the numerous attack taxonomies on the protocol itself, and outlines the importance of introducing authentication to the protocol itself. While some research suggests relying on intrusion detection systems and firewalls, even these studies show that an advanced attacker can bypass the protections afforded by these solutions, and thus successfully modify commands sent to these critical systems. Once inside the network, creating firewalls to find and prevent this communication will be nearly impossible, as it trying to differentiate malicious traffic from legitimate traffic would be extremely difficult.

ModSec addresses this attack directly with its security implementation. Once configured as described in previous sections, the communication channel between workstation-1 and workstation-2 becomes secure and insusceptible to this form of MITM attacks. Following the same process described using Ettercap, performing ARP poisoning itself will continue to perform as intended because the attack leverages weaknesses of the TCP communication itself. Despite this fact, however, the attacker (workstation-3) only has the ability to forward the traffic from sender (workstation-1) to the intended recipient (workstation-2). By attempting to modify any information within the packet itself, the Modbus message is no longer valid due to the appended HMAC. The attacker, assuming physical access to workstation-1 and/or workstation-2 is impossible, is unable to attain the secret key shared between the two devices, and is thus unable to calculate the correct hash that is appended with each message. Due to the checks that are added to the protocols implementation and work-flow, messages will not be processed unless all checks have been passed, which the attacker is unable to forge.

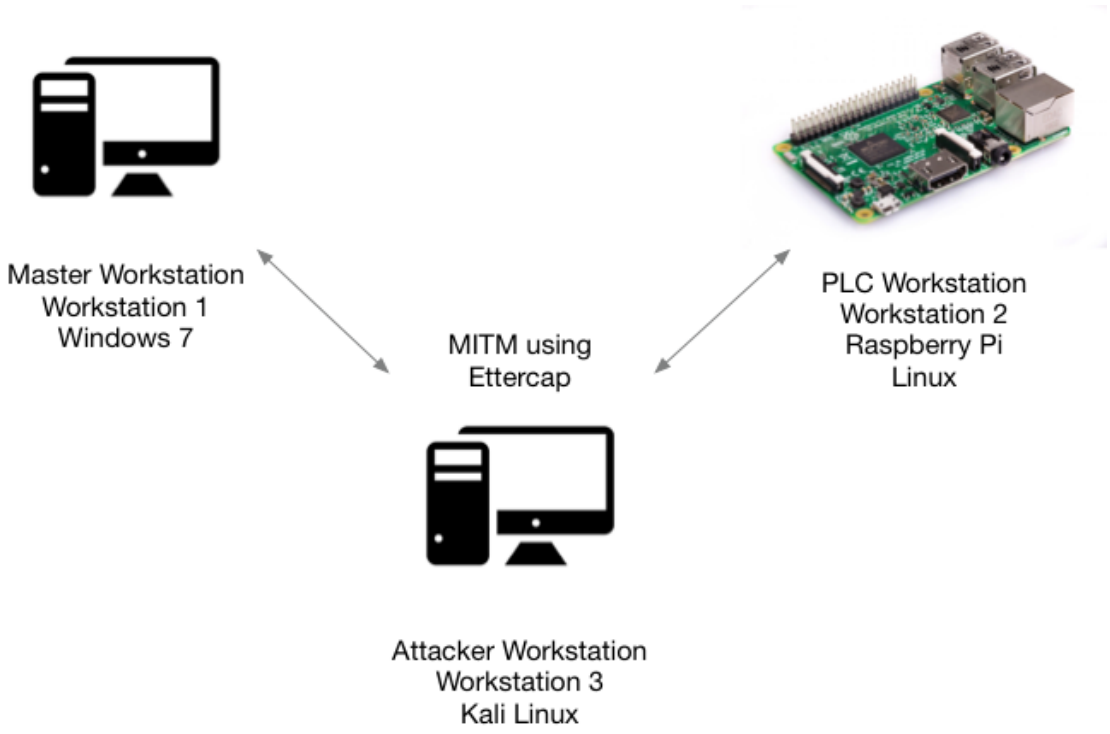

Figure 4.7: Modbus Server/Client Replay Attack Traffic

4.6.2 Replay Attacks

Similar to in the MITM attack, because Modbus occurs in an open network, once an attacker gains access to the network, they are able to easily view the traffic and gain information for an attack on a Modbus end device. By focusing on the lack of integrity within the Modbus protocol, an attacker is able to easily perform a replay attack, which can also be referred to as a playback attack. A replay attack occurs when the unauthorized actor captures the network traffic and then later sends the communication to its original destination, acting as the original sender. Unlike in the MITM attack where the attacker needs to take steps to sit in the middle of the connection and intercept the communication, within a replay attack, the attacker can simply listen to and capture the ongoing traffic with the ability to replay it to perform some malicious activity. Based on the current protocol design, there are no measures for checking the validity of the message or its sender, allowing for anyone in the network to execute potentially dangerous actions.

There are numerous tools at a potential attacker, or researchers, disposal for performing a replay attack. To perform a replay attack on the Modbus protocol, commonly used networking and open source tools were used, following the setup described in Figure 4.7. Performing the replay attack involves a nefarious party intercepting traffic between devices on the network. After gaining incite on the communication, the attacker has the ability resend the same traffic, with the idea that the receiving device does not have the ability to differentiate the malicious sender from a valid one.

To begin, the attacker first needs to view and capture network traffic between the targeted devices. With the use of Wireshark, all of the traffic in the network can be easily captured. With previous knowledge that Modbus is traditionally captured on port 502, it becomes even easier to filter out other traffic to focus strictly on the Modbus traffic. Wireshark also possesses the ability to parse Modbus traffic beyond the traditional TCP/IP traffic, further allowing the malicious actor to gain understanding of the traffic without needing to parse the data between the devices. With the ability to easily read the traffic, the actor can use the published documentation of the Modbus protocol [29] to determine the purpose of the function code, and as long as the protocol follows the guidelines, all traffic can easily be determined.

To perform the replay attack, a series of tools can be used to simply replay legitimate packets. In this attack scenario, the open-source library scapy was used to parse and replay the Modbus traffic. With the combination of Wireshark and scapy, the packet information can easily be saved as a PCAP (packet capture) file and replayed. While relatively straightforward and simple in its nature, this attack can cause detrimental effects in the receiving device. One example includes an action that requires for the device to either open or close an actuator. Replaying a packet that reverses this action could cause for significant damage in any environment. Within a replay attack, the master may never even be informed of this attack.

ModSec defends against potential replay attacks with the implementation of a non-

decreasing, 32-bit counter. This counter is designed similar to the anti-replay protocol, a sub-protocol that plays a key role in preventing replay attacks within IPsec [40]. The counter, initialized at 0, is designed to increment at each message sent by the master. The receiving node is designed to maintain a sliding window record of the counter values received from the master. Messages that are received with a lower counter value than the current window value are thus perceived as invalid, and thus returns an error exception. Unlike within IPsec, however, the sliding window is limited to as size of 1. By enforcing this window size, the attacker would be unable to use an earlier counter value. Comparatively, TCP traditionally uses Window Scaling, where the TCP Window buffer contains a copy of all packets sent out in case they are lost in transit. A window size too large could allow for an attacker to guess a value within this window and cause unexpected actions. By enforcing a window size of 1, however, ModSec prevents an attacker from this potential attack. The window value is related to the secret cryptographic key exchanged between the master and slave nodes in that it is initialized at the creation of a new key, and thus is reset when a new key is exchanged. This relationship also forces the devices to exchange a new key before, or when, the counter value wraps. Being as though the field size is 32 bits, when the value wraps to 0 again, the potential for another replay attack to occur presents itself, and this mechanism presents this scenario from occurring. Combined with the appended HMAC, the counter acts as a nonce in relation with the cryptographic key, and prevents repeated messages from occurring.

Using the same attack scenario described earlier, while Wireshark and scapy provide the ability to successfully watch and monitor the network, replaying an attack is prevented, as the target device has already encountered the counter value and prevents any further action. In combination with the HMAC, this attacker is also unable to change this counter value without recalculating the HMAC, as simply increasing the counter causes for the HMAC to be incorrect. As this attacker is not privy to the secret key, they are thus unable to recalculate the hash, thus rendering this attack useless.

The introduction of the counter field to the Modbus protocol does not prevent the attacker from snooping into the communication. It does, however, again prevent this attacker from being able to make potentially critical changes that can negatively impact the system as a whole.

4.6.3 Discussion on Lack of Authorization Vulnerabilities

Another commonly identified weakness of the Modbus protocol is its lack of authorization within messages. While the intention of the protocol is for messages to be limited between the master and slave devices, where the master only has permission to request actions from its slaves, there are no true checks to limit this action. Furthermore, in the case where a new device is introduced to the network, using Modbus TCP or Modbus Serial, this device can easily perform reconnaissance on the network, and begin commanding other slave devices to perform actions. As seen in the many attack scenarios discussed, this lack of security can cause detrimental harm to the overall network, and defense measures need to be put in place to prevent this from occurring.

ModSec introduces a permission scheme that is tied into the certificates that are issued out in the system. The permissions, tied to the devices public key, cannot be changed unless through a secure means during the initialization process, and are thus secure enough to rely on for authorizing communication.

CHAPTER 5
CONCLUSION AND FUTURE WORK

5.1 Conclusion

In this research, we produced ModSec, a secure Modbus protocol, that adds a much needed level security to the popular SCADA protocol Modbus. The proposed solution addresses many of the security issues that were left out of the initial design of Modbus due to the period of time when Modbus was introduced and implemented by implementing cryptographic functions. The issues of authentication, integrity, and non-repudiation are all addressed practically. The solution not only fills in various security holes, but also adds a permission scheme to further lock down various functions that can cause catastrophic results if executed by a nefarious actor. The proposed solution also shows that the additional security features have only a small amount of overhead in the protocol size as well as message transmission, allowing for the Modbus protocol to be implemented in scenarios where real time transmission is critical, and where end devices are limited in processing power. Therefore, according to these extreme transmission demands, device vendors are able to deploy security into the Modbus protocol as part of a true SCADA system. Finally, the proposed ModSec protocol proves to not only address many of the attacks that have already been shown to work against the protocol, both hypothetical and theoretical, but also helps lay the ground work for future implementations of adding security to open protocols in SCADA environments. While the proposed ModSec protocol is not a solution that can address all problems in SCADA and ICS environments, the solution does close the gap in one of the most common protocols used in systems all around the world.

5.2 Future Work

5.2.1 Certificate Authority

In the proposed ModSec protocol, public key cryptography was implemented in order to provide key sharing, as well as to store the permissions. These certificates, however, had to be manually installed and transferred over a secure channel, and any changes would require another manual initialization process. In a traditional public key infrastructure (PKI) setting, a certificate authority would help ease this burden and can additionally help expand the process of managing keys to allow for broadcasting of messages securely. To expand the work, a single-tiered certificate authority can be leveraged to begin to manage the certificates in the network of Modbus devices. There has been a significant amount of research describing the pros and cons of using a multi-tiered network, as well as how to properly setup certificate authorities in a PKI network [41, 42]. The CA, if added, would play a critical role in the overall success of the described protocol, and its successful deployment is critical in ensuring the security of the messages as a whole.

5.2.2 Privilege Management Infrastructure

This proposed research also leverages the extensions component of an X.509 certificate, introduced in version 3 of the certificate, to implement permissions within the protocol. In the current state, in order to change the permissions of a device, the initialization setup must occur again and certificates must be transferred over a secure means. In the case where a certificate authority were to be introduced, it would not only be responsible for managing the public key infrastructure, the certificate authority is also responsible for managing the permissions in proposed architecture. Research, however, has shown that it may be best to separate permissions from the certificate itself, especially if the permissions were subject to change often. In 2001, Privilege Management Infrastructures (PMI) were introduced to address this issue by making use of attribute certificates. By using a PMI, the responsibility

of privileges and authorizations can be managed separately from keys and authentication. Making use of a PMI would significantly raise the complexity of deploying the proposed solution, but could potentially raise both the security and usability of the system as a whole [43, 44, 45].

5.2.3 Cryptography

This works aims to provide a solution that would not only be practical in defending against the numerous potential attacks on the Modbus protocol, but also takes into account the wide range of end devices. In order to support systems that are of low-power, low-memory, the cryptographic methods that have been selected take into consideration both speed and efficiency. If a network were to be set up that uses modern equipment only and these limitations were not in place, stronger cryptographic solutions could replace the ones that are outlined in the earlier sections, and provide an even greater level of security.

Additionally, as industry continues to move to even greater levels of computing, older cryptographic methods begin to become antiquated and replaced with newer, stronger and more efficient algorithms. Examples like SHA-1 and MD-5 were originally cryptographically secure, but recent advancements allows for solutions to be implemented to bypass the security measures these algorithms were meant to protect [46, 47]. With that in mind, the cryptographic methods outlined can easily be replaced in the future with other cryptographic methods that may provide better efficiency, especially on devices that are low power. It is important, however, that any future solutions provide the same guarantees of authentication, integrity, and non-repudiation.

5.2.4 Multiple Masters

The work takes into account a traditional network where there is a single master and a wide number of slaves (ranging from 1 to 247). With the advent of Modbus/TCP, there are examples where a network could integrate multiple masters into the network, and potentially

even use more than the traditional number of slaves [48]. While the designed protocol does not take this situation into account, minor changes could be implemented to allow for this situation.

One approach to solving this problem would be for the certificate authority to be responsible for managing the additional devices. In this case, the network with multiple masters would could potentially have more the 247 devices in the network, which would also require a change in the uid field of the proposed protocol. The field length is designed to consider the traditional maximum device size of the network, and would need to be expanded to fit the new maximum. This change would negatively impact the number the maximum number of bytes transmitted for the commands, and would provide latency in the process of looking up the certificate from the CA as well as transmission.

A second approach to handling multiple masters would be to consider each master as its own network, and the slave devices would overlap in the various networks. This approach would allow for the proposed protocol to remain unchanged by assigning a certificate authority to each master device. This solution, however, adds complexity on the slave devices by requiring for these devices to speak with multiple certificate authorities. An additional parameter would need to be added to inform the slave which CA it must communicate with. This approach also has a weakness in having to further manage keys within each network, and can lead to significantly more keys within the network, as well as on each device.

REFERENCES

[1] I. N. Fovino, A. Carcano, M. Masera, and A. Trombetta, "Design and implementation of a secure modbus protocol," in *Critical Infrastructure Protection III: Third Annual IFIP WG 11.10 International Conference on Critical Infrastructure Protection, Hanover, New Hampshire, USA, March 23-25, 2009, Revised Selected Papers*, C. Palmer and S. Shenoi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 83–96, ISBN: 978-3-642-04798-5.

[2] IBM Managed Security Services Threat Research Group, "Security attacks on industrial control systems," IBM Corporation, Tech. Rep., 2015.

[3] Repository of Industrial Security Incidents, *Hacker takes over russian gas system*, `http://www.risidata.com/Database/Detail/hacker-takes-over-russian-gas-system`, 1999.

[4] B. Miller and D. Rowe, "A survey scada of and critical infrastructure incidents," in *Proceedings of the 1st Annual Conference on Research in Information Technology*, ser. RIIT '12, New York, NY, USA: ACM, 2012, pp. 51–56, ISBN: 978-1-4503-1643-9.

[5] N. Weaver, V. Paxson, S. Staniford, and R. Cunningham, "A taxonomy of computer worms," in *Proceedings of the 2003 ACM Workshop on Rapid Malcode*, ser. WORM '03, New York, NY, USA: ACM, 2003, pp. 11–18, ISBN: 1-58113-785-0.

[6] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver, "Inside the slammer worm," *IEEE Security Privacy*, vol. 1, no. 4, pp. 33–39, Jul. 2003.

[7] Trend Micro, "The SASSER Event: History and Implications," Trend Micro, Tech. Rep., Jun. 2004.

[8] *Conficker worm: Help protect windows from conficker*, `https://technet.microsoft.com/en-us/security/dd452420.aspx`, Feb. 2009.

[9] R. Langner, "Stuxnet: Dissecting a cyberwarfare weapon," *IEEE Security Privacy*, vol. 9, no. 3, pp. 49–51, May 2011.

[10] R. Nigam, *Known scada attacks over the years*, `https://blog.fortinet.com/2015/02/12/known-scada-attacks-over-the-years`, Feb. 2015.

[11] D. McMillen, *Attacks targeting industrial control systems (ics) up 110 percent*, https://securityintelligence.com/attacks-targeting-industrial-control-systems-ics-up-110-percent/, Dec. 2016.

[12] K. Zetter, *Inside the cunning, unprecedented hack of ukraine's power grid*, https://www.wired.com/2016/03/inside-cunning-unprecedented-hack-ukraines-power-grid/, Jun. 2017.

[13] D. E. Sanger, *U.s. indicts 7 iranians in cyberattacks on banks and a dam*, https://www.nytimes.com/2016/03/25/world/middleeast/us-indicts-iranians-in-cyberattacks-on-banks-and-a-dam.html, Mar. 2016.

[14] J. Dunietz, *Is the power grid getting more vulnerable to cyber attacks?* Aug. 2017.

[15] ERIPP, *Eripp*, 2017.

[16] Matherly, John, *Shodan*, 2017.

[17] I. Ghansah, "Best practices for handling smart grid cyber security," California Energy Commission, Tech. Rep., 2014.

[18] G. A. Cagalaban, Y. So, and S. Kim, "Scada network insecurity: Securing critical infrastructures through scada security exploitation," *Journal of Security Engineering*, 2010.

[19] E. Byres, M. Franz, and D. Miller, "The use of attack trees in assessing vulnerabilities in scada systems," Jan. 2004.

[20] T. H. Morris and W. Gao, "Industrial control system cyber attacks," in *Proceedings of the 1st International Symposium on ICS & SCADA Cyber Security Research 2013*, ser. ICS-CSR 2013, Leicester, UK: BCS, 2013, pp. 22–29, ISBN: 978-1-780172-32-3.

[21] W. Gao and T. H. Morris, "On cyber attacks and signature based intrusion detection for modbus based industrial control systems," *Journal of Digital Forensics, Security and Law: Vol. 9 : No. 1 , Article 3.*, 2014.

[22] Moore Industries Worldwide, "Using modbus for process control and automation," Moore Industries-International, Inc., Tech. Rep., 2007.

[23] Siemens USA, *History of the modbus protocol*, http://w3.usa.siemens.com/us/internet-dms/btlv/CircuitProtection/MoldedCaseBreakers/docs_MoldedCaseBreakers/Modbus%20Information.doc, Aug. 2017.

[24] Rapid7 LLC, *Metasploit project*, Feb. 24, 2017.

[25] P. Huitsing, R. Chandia, M. Papa, and S. Shenoi, "Attack taxonomies for the modbus protocols," *International Journal of Critical Infrastructure Protection*, vol. 1, no. Supplement C, pp. 37 –44, 2008.

[26] A. Shahzad, M. Lee, Y. K. Lee, S. Kim, N. Xiong, J.-Y. Choi, and Y.-H. Cho, "Real time modbus transmissions and cryptography security designs and enhancements of protocol sensitive information," *Symmetry*, vol. 7, pp. 1176–1210, 2015.

[27] K. Stouffer, V. Pillitteri, S. Lightman, M. Abrams, and A. Hahn, "Guide to industrial control systems (ics) security," National Institute of Standards and Technology, Tech. Rep., 2015.

[28] H. Krawczyk, IBM, M. Bellare, UCSD, R. Canetti, and IBM, "HMAC: Keyed-Hashing for Message Authentication," RFC Editor, RFC 2104, Feb. 1997.

[29] *Modbus application protocol specification v1.1b3*, Modbus Organization, 2012.

[30] C. Schuett, "Programmable logic controller modification attacks for use in detection analysis," PhD thesis, Air Force Institute of Technology, 2014.

[31] R Housley, R. Laboratories, W. Polk, NIST, W. Ford, VeriSign, D. Solo, and Citigroup, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile," RFC Editor, RFC 3280, May 2002.

[32] Microsoft, *Certutil*, 2012.

[33] S. Raimbault, *Libmodbus*.

[34] *Build software better, together*.

[35] CoolestNerdIII, *Coolestnerdiii/libmodbus*, Apr. 2018.

[36] Crypto++ Community, *Crypto++ benchmarks*, Dec. 2017.

[37] WinTECH Software Design, *Modscan 32*, 2017.

[38] Witte Software, *Modbus poll*, 2017.

[39] darkweb and nnovic, *Modbuspal - java modbus simulator*, 2017.

[40] S. Kent, "IP Authentication Header," RFC Editor, RFC 4302, Dec. 2005.

[41] W. Shanks, "Building and managing a pki solution for small and medium size business," SANS Institute, Tech. Rep., 2013.

[42] S. Hromberger and S. Pietrowicz, "Pki security considerations for ami, smart grid, and icss networks," National Electric Sector Cybersecurity Organization, Tech. Rep., 2012.

[43] D. W. Chadwick, "An x.509 role-based privilege management infrastructure," University of Salford, Tech. Rep., 2001.

[44] D. W. Chadwick and A. Otenko, "The permis x.509 role based privilege management infrastructure," *Future Generation Computer Systems*, vol. 19, no. 2, pp. 277 –289, 2003, Selected Papers from the TERENA Networking Conference 2002.

[45] D. Chadwick, "The x.509 privilege management infrastructure," 2004.

[46] X. Wang and H. Yu, "How to break md5 and other hash functions," 2005.

[47] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov, "The first collision for full sha-1," 2017.

[48] J. Rinaldi, *Multiple masters*, `https://www.rtaautomation.com/blog/multiple-masters/`, Aug. 2017.