NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

A SECURITY-CENTRIC APPLICATION OF PRECISION TIME PROTOCOL WITHIN ICS/SCADA SYSTEMS

by

Charles A. Allen

December 2021

Thesis Advisor:	Chad A. Bollmann
Co-Advisor:	George W. Dinolt
Second Reader:	Darren J. Rogers

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** December 2021	**3. REPORT TYPE AND DATES COVERED** Master's thesis

4. TITLE AND SUBTITLE
A SECURITY-CENTRIC APPLICATION OF PRECISION TIME PROTOCOL WITHIN ICS/SCADA SYSTEMS

5. FUNDING NUMBERS
REPQN

6. AUTHOR(S) Charles A. Allen

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Navy Cyber Warfare Development Group, Suitland, MD

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release. Distribution is unlimited.

12b. DISTRIBUTION CODE
A

13. ABSTRACT (maximum 200 words)

Industrial Control System and Supervisory Control and Data Acquisition (ICS/SCADA) systems are key pieces of larger infrastructure that are responsible for safely operating transportation, industrial operations, and military equipment, among many other applications. ICS/SCADA systems rely on precise timing and clear communication paths between control elements and sensors. Because ICS/SCADA system designs place a premium on timeliness and availability of data, security ended up as an afterthought, stacked on top of existing (insecure) protocols. As precise timing is already resident and inherent in most ICS/SCADA systems, a unique opportunity is presented to leverage existing technology to potentially enhance the security of these systems. This research seeks to evaluate the utility of timing as a mechanism to mitigate certain types of malicious cyber-based operations such as a man-on-the-side (MotS) attack. By building a functioning ICS/SCADA system and communication loop that incorporates precise timing strategies in the reporting and control loop, specifically the precision time protocol (PTP), it was shown that certain kinds of MotS attacks can be mitigated by leveraging precise timing.

14. SUBJECT TERMS
ICS, SCADA, cyber security, PTP, man in the middle, man on the side, MotS

15. NUMBER OF PAGES
109

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

A SECURITY-CENTRIC APPLICATION OF PRECISION TIME PROTOCOL WITHIN ICS/SCADA SYSTEMS

Charles A. Allen
Lieutenant, United States Navy
BA, University of Kansas, 2013

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 2021

Approved by: Chad A. Bollmann
Advisor

George W. Dinolt
Co-Advisor

Darren J. Rogers
Second Reader

Gurminder Singh
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Industrial Control System and Supervisory Control and Data Acquisition (ICS/SCADA) systems are key pieces of larger infrastructure that are responsible for safely operating transportation, industrial operations, and military equipment, among many other applications. ICS/SCADA systems rely on precise timing and clear communication paths between control elements and sensors. Because ICS/SCADA system designs place a premium on timeliness and availability of data, security ended up as an afterthought, stacked on top of existing (insecure) protocols. As precise timing is already resident and inherent in most ICS/SCADA systems, a unique opportunity is presented to leverage existing technology to potentially enhance the security of these systems. This research seeks to evaluate the utility of timing as a mechanism to mitigate certain types of malicious cyber-based operations such as a man-on-the-side (MotS) attack. By building a functioning ICS/SCADA system and communication loop that incorporates precise timing strategies in the reporting and control loop, specifically the precision time protocol (PTP), it was shown that certain kinds of MotS attacks can be mitigated by leveraging precise timing.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

ADC	analog to digital converter
AI	artificial intelligence
BMCA	best master clock algorithm
COTS	commercial-off-the-shelf
CPSA	Cyber-Physical Security Assessment
CPU	central processing unit
DOD	Department of Defense
DoS	denial of service
E-ISAC	Electricity Information Sharing and Analysis Center
EEMS	Engineering Enclave for Maritime Security
FCI	false command injection
FDI	false data injection
GNSS	global navigation satellite system
GPIO	general purpose input/output
HMI	human machine interface
ICS	industrial control system
ICS/SCADA	industrial control system (ICS)/supervisory control and data acquisition (SCADA)
IDE	integrated development environment
IEEE	Institute of Electrical and Electronics Engineers

IP	internet protocol
IoT	Internet of Things
IT	information technology
LAN	local area network
PTP	precision time protocol
MitM	man in the middle
ML	machine learning
MotS	man on the side
MQTT	message queuing telemetry transport
NIDS	network intrusion detection systems
NIC	network interface card
NTP	network time protocol
NPS	Naval Postgraduate School
OS	operating system
OT	operational technology
PLC	programmable logic controller
PKI	public key infrastructure
RF	radio frequency
RTU	remote terminal unit
RFC	Request for Comment
RPM	revolutions per minute
SANS	SysAdmin, Audit, Network, and Security

SCADA	supervisory control and data acquisition
SSH	secure shell
TCP	transmission control protocol
TLS	transport layer security
UDP	User Datagram Protocol
USN	U.S. Navy
VFD	variable frequency drives
VPN	virtual private network
WAN	wide area network
WLAN	wireless local area network

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgments

I would like to thank Commander Chad Bollmann for taking me on as an advisee, offering tremendous support and amazing opportunities, and encouraging me to explore the topics and subject matter in which I was most interested. The depth of learning that this endeavor drove in me is difficult to specifically enumerate. Digging into both the hardware and software aspects was enlightening, challenging, and a terrific experience. I can only hope that my work here furthers the important efforts of the CCW and the EEMS lab.

Thank you to Dr. George Dinolt for co-advising in this research endeavor. I relied on his depth of expertise in the field to get the ball over the goal line. His steadfast support and encouragement throughout this process was additionally critical. The advantage of having a recognized subject matter expert in your corner cannot be understated.

Thank you to Darren Rogers. Darren was consistently a sounding board, cooperative troubleshooter, and the glue guy, running down anything we needed for our project, despite having so many irons of his own in the fire. He's relied upon because he is reliable and I had an exceptional time working with him throughout this whole process.

Thank you to Vikram Kanth. With his wealth of knowledge and expertise, he was a key sounding board in clearing more than a few conceptual log jams. His selfless assistance in coding and data visualization, while in the midst of his own Ph.D. work, was a game changer.

I would also like to especially thank Micky Hall for his friendship, expertise, and support throughout the entirety of this program. Despite the rigors of a growing family and his own academic pursuits, he was relentlessly helpful and integral to the learning that occurred here. A father, a friend, and a tutor (for me and countless others) — Micky has it all.

Finally, endless and tremendous gratitude to my family. To my father, the intellectually and academically curious heavyweight encouraging me throughout this and every other undertaking I have ever started. To my mother, whose infinite support in everything that I do I am so blessed to have and not take for granted. And to my two role model brothers and their exploding families, I can't thank you guys enough!

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

1.1 State of Play

Industrial Control System and Supervisory Control and Data Acquisition (ICS/SCADA) systems are the bedrock of modern society. By monitoring, managing, and controlling routine yet vital functions across many key parts of industry such as manufacturing, energy, and transportation, ICS/SCADA systems are the invisible connective tissue on which mankind has come to rely. Together, ICS/SCADA systems represent concentric subsets of a broader operational technology (OT) systems, but as the foundation that underlies OT, failure of ICS/SCADA systems for any reason can have direct and pervasive adverse impacts on the broader system and beyond. Those impacts and interruptions are significant as they reach down to the most granular level of the systems that safely enable power generation and transmission, natural gas delivery, dams, water treatment, nuclear power, telecommunications, and much more. For simplicity, use of the term ICS/SCADA will hereafter be encapsulated by the overarching term "OT."

Targeting of these systems has been well documented. In 2011, Guan et al. [1] described a few important cases such as a worm that infected the Davis-Besse nuclear power plant in Ohio, sewage system release into public waterways in Australia, and water treatment system penetration in Pennsylvania. Since Guan et al., however, there have been more significant events with serious implications. While discussions regarding specific attribution are not the purpose of this paper, the 2009-10 Stuxnet computer worm gained international attention and fundamentally changed the discussion of cyberspace as a domain for warfare. Industry experts and analysts described Stuxnet to be a "cyber-weapon of mass destruction," very sophisticated, and extremely narrow in target scope, aiming squarely at the Iranian government's nuclear material enrichment program [2] In 2016, SysAdmin, Audit, Network, and Security (SANS) industrial control system (ICS) conducted a thorough analysis of a cyber-attack on the Ukrainian power grid through the Electricity Information Sharing and Analysis Center (E-ISAC). The cyber-attack that targeted SCADA-controlled power distribution and management resulted in electrical outages for approximately 225,000 customers [3]. While

still foregoing an attribution discussion, it is important to note that many organizations attributed the Ukrainian power grid attack to Russia which, if true, represents the risk that powerful nation states are capable of targeting OT systems around the world to serious and potentially disastrous ends.

Two key factors have resulted in the current status quo of OT systems [1]. First, the maturation of automation and its industrial applications was immediately recognized and adopted by virtually all players in critical infrastructure around the world. Second, potential cost-saving and convenience led to the interconnection of these industrial networks via widespread and cheap accesses, the internet chief among them. As is the case with many innovations that incrementally develop over a long period of time, security was not a feature baked into early supervisory control and data acquisition (SCADA) systems. Restricting physical access to management buildings or substations was once considered sufficient mitigation, but that control alone is clearly no longer enough to mitigate risk of attack [4]. Additionally, proprietary protocols and connections had in the past served as a barrier to interference by would-be interlopers, but the transition to standardized communication protocol standards like Ethernet (Institute of Electrical and Electronics Engineers (IEEE) 802.3) and expanded use of public communication media like the internet simplified possible attack vectors [2].

This is not to suggest that standards are a problem; rather, the opposite. Well-designed and well-implemented standards are integral to the protection of OT systems. Systems of systems, like both the internet and OT networks, rely on stacks or layers of communication protocols, with each layer either serving a different function or obfuscating some details of a lower layer. The power of layering and stacking protocols is that security functions can be built into systems by adding or changing different layers of the communication stack. The shift to reliance on the internet to manage geographically disparate industrial systems will continue to invite cyber intrusions of varying degrees. It is therefore exceedingly important that communications protocols are secure in design and implementation.

1.1.1 Current Approaches

Despite the low priority typically ascribed to security within OT designs and implementations, there are a number of techniques that have been utilized to varying degrees of success. As noted previously, implementing secure systems does normally incur trade-offs in other

aspects such as speed, efficiency, reliability, or accessibility. In most applications of OT systems, those trade-offs are not acceptable from a risk-tolerance standpoint. The classic security "CIA" triad of confidentiality, integrity, and availability (normally including authentication as well), is a model that certainly fits OT. In these systems generally, but most importantly in critical infrastructure or transportation, availability is the dominant piece. Malicious activity was not typically considered in the original design of the first systems, so confidentiality and integrity were assumed to a certain degree. However, as operators of OT systems have found cost-effective the benefits of connecting their systems to the internet, confidentiality and especially integrity have surged in importance — yet never to overtake availability [5].

Encryption A common solution to many cyber-related vulnerabilities, encryption has many useful and successful applications in commercial, industry, and government/military infor-mation systems. By encoding information using a cryptographic key, the information that is transmitted over an otherwise insecure medium becomes unintelligible to a would-be snooper or malicious actor. The information can only become useful again once it is decoded by the recipient using the correct decryption key. The obvious benefit to applying encryption within OT systems is that a malicious actor that has access to a communication medium between two nodes would be unable to decipher the information being exchanged on that medium. While useful in principle, in an OT system encryption could result in undesirable side effects. Fauri et al. described a few of the major downsides to encryption within an ICS system [5]. First, encryption may not actually result in enhanced security of the system, as the case studies of ICS attacks they reviewed would not have been prevented by encryption. Second, encryption could hamper ICS operators and system administra-tors from proper surveillance of their internal traffic because network intrusion detection systems (NIDS) may not be able to make sense of the encrypted traffic without adding substantial costs to the overall system. Finally, and drawing from their second conclusion, encryption can also hamper troubleshooting and recovery efforts. This last conclusion is a particularly important consideration because "uptime," or reliability, of an OT system is of paramount importance. The Colonial Pipeline ransomware attack of May 2021 is a salient example of this pitfall of encryption. Even after paying the ransom to receive the decryption keys in order to restore service as quickly as possible, the decryption process itself took an extended period of time, incurring continued damages to the victim organization and their customers [6].

Authentication Authentication, or the process of positively validating the correct identity of an information handler, is closely related to integrity. Although it is used to great effect in more common information technology (IT) environments, public key infrastructure (PKI) implementations in OT environments tend to be haphazard and cumbersome. PKI is a very effective method to distribute asymmetric cryptographic keys, but it typically requires a significant amount of overhead and management. Commonly within OT, only certain subsets of the system will utilize secure methods, while other parts closer to the control plane tend to still communicate in plain, unauthenticated traffic, leaving them very vulnerable to surveillance and even attacks like spoofing [7]. Other issues with PKI implementation in ICS systems are rooted in prevalent hardware issues: memory and processing power. The computational power required to effectively implement a system like this at multiple levels is typically greater than the individual nodes can support, which may have additional ramifications if the primary control function of the node is compromised. Finally, even with expenses paid towards building and implementing a PKI system, the issue of securely storing the various private keys persists [7].

Physical Controls Physical access controls are an important assumption in most studies of OT security, but they will not be explored in this research. Effective bypassing of physical security can in some cases render other cyber-based measures useless depending on where, physically or logically, access is achieved. Other physical controls, such as human intervention or a "man in the loop" are typically advisory and rely on accurate information within the OT system to be presented to the human monitoring the system, a key design consideration of the malware that damaged the Iranian nuclear program in 2010 [8].

1.1.2 Timing Approach

Each of the previously described approaches tends to result in a trade-off with reliability or safety that is not normally tolerated in OT systems. Therefore, this research sought to leverage existing safety and reliability aspects, specifically precise timing, to improve the security of the overall system without requiring a sacrifice of reliability or safety. Precise timing is typically required by most ICS systems within a certain tolerance. The network time protocol (NTP) is a computer network-based timing standard that is in wide use and

satisfies most requirements of IT environments [9]. For more time-critical systems like OT, especially in critical infrastructure or transportation environments, NTP is not accurate enough. In these cases, absent a proprietary solution, the precision time protocol (PTP) can be utilized to ensure proper synchronization across the nodes of the network.

1.2 Engineering Enclave for Maritime Security

The Engineering Enclave for Maritime Security (EEMS) laboratory at Naval Postgraduate School (NPS) provides a wide assortment of capabilities to build and simulate maritime information systems pursuant to academic research endeavors. The lab provided the equipment and oversight in building the simulation OT network that reflects basic maritime control mechanisms. The equipment provided by EEMS includes but is not limited to the three-phase induction motors, A/C drive components, programmable logic controller (PLC)s, Raspberry Pi and Arduino modules, as well as the necessary steering and throttle hardware.

1.3 Objectives

1.3.1 Goals

This scope of this thesis is to evaluate the specific utility of PTP as a security mechanism to guard against man on the side (MotS) attacks by a malicious actor with physical access to the communication network. This work specifically looks at the application of this strategy on the EEMS lab's OT simulation rack, whose assembly and configuration was a large part of the work herein. The thesis does not explicitly seek to evaluate potential issues with PTP such as denial of service (DoS) or other attack vectors beyond those described at the outset.

1.3.2 Contributions

The primary contributions of our research were:

- Assembling an OT subsystem physical testbed
- Using commercial-off-the-shelf (COTS) equipment to create a timing-synchronized control loop in the subsystem
- Demonstrating the use of timing as a component in securing an OT system

- Providing recommendations for further work leveraging PTP and the developed test bed

1.4 Chapter Organization

The organization of the remainder of this thesis is as follows. Chapter 2 provides background context, conducts a review of existing related work in this field, and discusses the specific application of that related work in regards to this research endeavor. Chapter 3 describes the methodology, design, hardware and software requirements as well as the implementation of the approach. Chapter 4 reviews the results of each phase of testing, with thorough consideration of both advantages and disadvantages as uncovered during testing, as well as a discussion of potential use cases. Chapter 5 is a comprehensive conclusion of all aspects of research. Finally, leveraging lessons learned from the aforedescribed research endeavors, Chapter 6 provides recommendations for future work with specific applicability to both government and private industry.

CHAPTER 2:
Background and Related Work

This chapter provides an overview of the timing protocol evaluated through our research in Section 2.1. In 2.1.1, the specific mechanics of the PTP protocol are discussed in more detail. Section 2.1.2 covers the very important Request for Comment (RFC) that defines key security improvements for the standard. Sections 2.2 and 2.3 review recent research into this topic area and discuss its applicability to our research endeavors.

2.1 Precision Time Protocol

The IEEE 1588 protocol standard for PTP was designed as a method to increase the timing accuracy of networked communications. The standard was designed to synchronize real-time clocks within a distributed network to a very high degree of precision. Originally published as IEEE 1588-2002 on November 8, 2002, the standard has been incorporated into other standards and has also been revised and improved since its original inception. The standard offers a targeted timing accuracy to less than a microsecond, a significant improvement on NTP's accuracy of a few milliseconds. PTP is constrained to operate within a small number of subnets, unlike NTP which can cross many subnets, wide area network (WAN)s, and even the internet. A trade-off to the benefit of an OT system is that the actual on-network constraints and resource requirements of PTP are comparatively less than that of an NTP network. Offering a greater degree of flexibility, PTP can be implemented either via hardware or software, though hardware implementations offer the highest degree of accuracy. Although the original design of PTP did not include a security specification, improvements have been recommended including RFC 7384, which will be discussed in greater detail.

The goals set out by the IEEE 1588 standard are as follows: sub-microsecond synchronization of real-time clocks in a distributed network, deployability within localized systems typical of industrial automation, and applicability within local area networks that support multicast communications (including but not limited to Ethernet) [10]. All three of these goals align to the intentions of this research endeavor: achieving a high degree of synchro-

nization within the OT network, deploying the protocol within a highly localized industrial application, and leveraging the Ethernet communication standard. Additional benefits include simple and free installation due to the open source nature as well as support for systems and networks with varying degrees of clock precision and/or reliability. As previously stated, the standard has been revised twice, with the most recent revision promulgated in 2019. However, the IEEE1588-2008 version is discussed in greater detail here because *linuxptp*, the Linux daemon used in this research to manage the PTP implementation, is based upon the 2008 revision.

2.1.1 Timing Methodology

The timing methodology of IEEE 1588 sought to address the problem of creating a highly accurate and consistent time base across the clocks of all devices within a system. IEEE1588-2008 is arranged into 19 clauses or subsections, with special attention paid to clauses 6 (PTP overview) and 13 (inter-clock message format).

Importantly, according to clause 6 of 1588-2008, "PTP is tolerant of an occasional missed message, duplicated message, or message that arrived out of order. However, PTP assumes that such impairments are relatively rare" [11].

PTP was designed with a multicast communication model in mind, lending itself well to the internet protocol (IP), User Datagram Protocol (UDP), and ethernet implementations. Because network components like bridges can introduce timing jitter, their deployment in our experimental design is limited. To the maximum extent possible, only essential non-PTP networking equipment (i.e. routers and switches) are utilized in order to minimize jitter.

PTP employs a master/slave architecture with a grandmaster clock atop the hierarchy transmitting timing synchronization to the subordinate slave clocks, which use the received timestamp information to compute an appropriate offset from the master and adjust their own clocks accordingly. Accurate timestamp data underpins the entire PTP protocol. A key guiding principle for this research was to leverage COTS equipment to the maximum extent possible. As a result, the design relied on software timestamping, a capability natively supported by the Raspberry Pi 4 with Linux kernel 3.0 or later [12]. Discussed in greater detail in subsequent sections, hardware timestamping can be utilized, perhaps in future endeavors, but would require additional hardware such as a PTP-compliant Raspberry Pi

"shield" or an alternate network interface card (NIC). The Raspberry Pi 4 units utilized software timestamping via the *linuxptp* Linux daemon.

The IEEE 1588-2008 standard defines two types of messages: event and general. Event message types for PTP are *sync*, *delay_req*, *pdelay_req*, and *pdelay_resp*. General message types are *announce*, *follow_up*, *delay_resp*, *pdelay_resp_follow_up*, *management*, and *signaling*. *Announce*, transmitted from the grandmaster, is used to establish the hierarchy and inform the slave clock(s). The messages *sync*, *delay_req*, *follow_up*, and *delay_resp* handle the transmission of timing data, while the remaining messages of *pdelay_req*, *pdelay_resp*, and *pdelay_resp_follow_up* measure the delay between two clocks that implement the peer delay mechanism.

As described in the establishing standard and corresponding to Figure 2.1, the following message exchange process occurs in PTP. First, the grandmaster sends a *sync* message to the slave and records this time as t_1. Second, when the slave receives the *sync* message, it records time of receipt as t_2. The grandmaster can, with proper hardware, embed the timestamp in the *sync* message, or it can embed the timestamp t_1 in a *follow_up* message, as is the case here. Third, the slave sends a *delay_req* message to the master, recording the time of transmission as t_3. Fourth, the master receives the slave's *delay_req* message, recorded at time t_4. Finally, the master sends a *delay_resp* message with the t_4 timestamp embedded back to the slave, which now has all necessary timestamp information in order to compute its own clock offset as well as the mean propagation time.

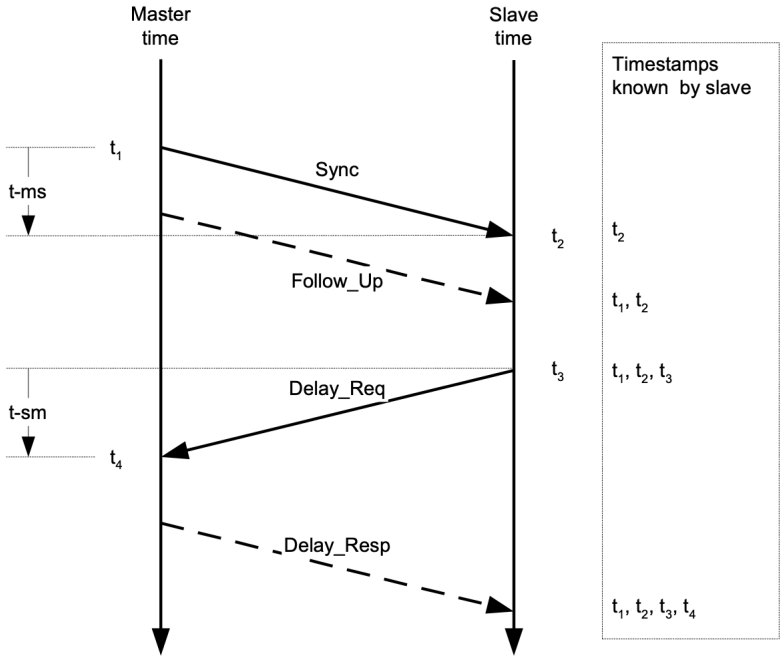

Figure 2.1. Basic PTP synchronization message exchange between a master and a slave clock, denoting the messages and corresponding timestamps. Source: [11, figure 12].

This transmission and computation process assumes transmission symmetry, that the transmission time from master to slave and from slave to master are the same. If asymmetry is inherent in the system, then the calculated clock offset times will be subject to a high degree of error.

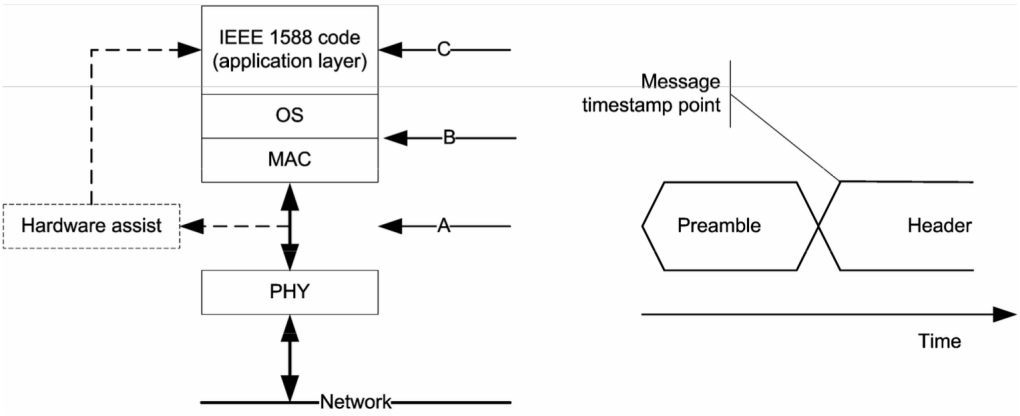

Figure 2.2. Example of hardware timestamp generation scheme employed by a PTP node. Source: [11, figure 14].

Figure 2.2 shows a timestamp generation model that is based upon hardware timestamping. As the Raspberry Pi 4 units used here were not (at the time) capable of hardware timestamping, the software timestamping that is relied upon occurs at level C of Figure 2.2, rather than at level A. A kernel-level implementation of PTP could implement software timestamping at level B of 2.2. The hardware assist shown at level A is lower in the protocol stack and provides a more accurate and more reliable aperture to generate a timestamp, which is why this method is preferred in most applications of PTP [11].

2.1.2 RFC 7384

Submitted in 2014, RFC 7384 is seminal in its definition of a set of security requirements for timing protocols in general, with special consideration for NTP and PTP. RFC 7384 states that while NTPv4 incorporated specific security mechanisms, PTP only included a thinly defined and "experimental" security protocol. Although the IEEE 1588-2019 revision provides options for enhanced security including cryptographic messaging, source authentication, and the addition of an informative security-focused annex, RFC 7384 has not yet been superseded and is referenced in the IEEE 1588-2019 revision [13].

Of particular interest to this project, RFC 7384 specifically addresses man in the middle

(MitM) attacks as part of its security model. A key finding from the RFC is the utility of confidentiality in mitigating MitM attacks by protecting the individual protocol packets. Despite the advantages conferred by confidentiality, likely through encryption, encryption alone only complicates MitM attacks and is not sufficient to outright prevent these attacks. For that reason, the RFC describes encryption as a "may," rather than a "must." Protection against packet delay and intercept attacks is, however, described as a "must," given the potentially severe implications to the system if clock accuracy is degraded, especially in a PTP system. Though described as a "must," this RFC does not define a specific approach to mitigating packet delay and intercept attacks, as it will depend on architecture and system requirements [14]. Despite being cut from the same cloth, a MotS attack varies from MitM primarily in the sense that a MotS does not "control a node but can still inject data" [15]. The defense against MotS attacks that is the focus of our research can also improve its effectiveness by leveraging a multitude of techniques, encryption among them.

Another very important consideration illuminated by RFC 7384 is that of a rogue master. A rogue master is a malicious node that convinces the slave nodes that it is the trusted grand-master clock. This can take a few different shapes in a PTP network such as a transparent clock or boundary clock, but the concept is very relevant to this research endeavor. The MotS attacker in this research design strongly resembles a rogue master.

2.2 Related Research

There is a significant body of research work in the realm of cyber-physical systems that runs the spectrum from critical infrastructure OT to Internet of Things (IoT) devices making their way into homes around the world. The research discussed in this section covers the cyber-physical systems that comprise OT networks and discusses known vulnerabilities in PTP.

2.2.1 “Impact of Malicious Control Commands in Cyber-Physical Smart Grids”

As an important recent case study, the 2015 cyber attack on the Ukrainian power grid was a prime motivator for the research conducted by Saxena et al resulting in a useful cyber risk mitigation tool called Cyber-Physical Security Assessment (CPSA) [16]. This tool was

specifically tailored against known Trojans like BlackEnergy that was used in the Ukrainian attack that accessed SCADA networks through hijacked virtual private network (VPN) connections into the utility provider's corporate network.

The assumed capability that the authors describe is one adopted in this research: The adversary has the technical and if necessary, physical access capabilities, to perform a MitM attack, thereby enabling the injection of illegitimate and/or malicious commands to the network. A key distinction in assumptions, however, is that the research did not assume the attacker to be an insider with legitimate access to properly functioning systems that were used improperly or maliciously. Of the three use cases Saxena et al described, Use Case 1 is the nearest comparison to our work, whereby the adversary "impersonates the network and sends a false (unwanted) but legitimate command outside of the control center to breaker of the largest generator" [16]. The implications of malicious commands are insecure operations of the power system as well as the possibility of shedding electrical load from the substations, resulting in power loss not unlike the 2015 Ukrainian attack. Potential future work could involve adapting the CPSA to model the shipboard environment.

2.2.2 "Performance Evaluation of IEEE 1588 Protocol Using Raspberry Pi over WLAN"

Allahi et al contributed significantly to the design and implementation considerations in this research, especially via their performance evaluation of PTP over a wireless local area network (WLAN) utilizing a Raspberry Pi [17]. Their work in both implementing and evaluating the performance of a readily-available Raspberry Pi Model 3 to handle PTP communications sufficiently over a wireless medium was a key validation of hardware selection for this research. Their success implementing IEEE 1588 over WLAN has important implications for industry by providing another method to achieve precise timing in their systems using COTS and open-source equipment. Allahi et al were able to demonstrate a wireless system with precise timing on par with the wired local area network (LAN) standard. The work of Allahi et al helped validate the hardware selection for this research. Additionally, their consideration of real-time network traffic that is distinct and in addition to the PTP traffic was useful to our work. Many of the complicating factors they had to consider for the WLAN PTP implementation, such as packet size constraints, wireless transmission delays, and additional external radio frequency (RF) interference, are not applicable to our work as

we utilized a wired medium.

2.3 Consideration of Current PTP Vulnerabilities

2.3.1 "Precision Time Protocol Attack Strategies and Their Resistance to Existing Security Extensions"

In one of the most significant and recent studies on PTP security, Alghamdi and Schukat evaluated and experimented with a variety of attack approaches that were outlined in RFC 7384, in addition to adding their own increased specificity to a few attack approaches [18].

Importantly, this research incorporated key design considerations from Annex P of IEEE 1588-2019. Their work demonstrated that adversaries can compromise any and all measure of PTP components (i.e., from the grandmaster to slave) and any boundary or transparent clocks in between. Because all key elements of PTP architecture were shown to be vulnerable, the difficulty of comprehensively mitigating threats to PTP networks is greatly increased. Alghamdi and Schukat showed that the internal attacker has the widest array of possible attack vectors to utilize to corrupt the timing in a system. The problem persists and while there are certainly security provisions that can and should be implemented to mitigate risk, there still does not exist any single mechanism to provide sufficient protection to key systems on its own.

2.3.2 "Impact of Cyberattacks on PTP"

By developing an effective test bed and using it to examine key vulnerabilities in PTP, DeCusatis et al highlighted what they referred to as a critical interdependence between security and timing [19]. Their findings regarding a severe vulnerability due to lack of sufficient authentication inside PTP networks are significant and should be leveraged in future revisions of the standard. For example, in a master spoof DoS attack they evaluated, the attack was roundly successful because the slave blindly accepted packets spoofed by the attacker. DeCusatis et al postulate a better method of providing enhanced identification in the system – one that allows the slave to generate the master's valid address from the traffic [19]. The threat model they designed within the construct of their test bed was based on the aforementioned RFC 7384, with special emphasis on MitM-style masquerade attacks

as well as DoS attacks across different portions of the network stack.

Another key revelation of DeCusatis et al is the counterproductive impact that hardware timestamping, an important piece of PTP precision, may have on the application of cryptographic standards. As the authors described in [19], there is a distinct disadvantage to the use of hardware timestamping because it occurs at the lowest level of the communication stack, as shown in Figure 2.2. This behavior is not necessarily compatible with higher-level encryption of data that occurs in the application layer. It is specifically that difference in timing between layers that defeats the advantage of hardware timestamping closer to the physical layer.

DeCusatis et al raised another consideration relevant to our work, one that would necessarily prohibit the integration of certain kinds of security mechanisms. As some security measures like signatures or certificates rely upon an accurate time of day to ascertain whether or not they are expired, a system that uses an arbitrary time might preclude their use [19].

One of the many important takeaways from their work is the value of authentication in critical networks, especially networks that provide important functionality such as precise timing. The multicast communication approach for PTP is effective in a trusted network, but the lack of authentication schemes in these networks renders them incredibly vulnerable. The multicast approach allows for an insider to glean necessary information about the master/slave communication scheme in order to duplicate the master itself. During this period of time, there would be an extra "slave" node as the attacker emulates a slave in order to collect the necessary information. The task is further simplified for the attacker because the return communications from the slave(s) back to the master can be entirely ignored [19]. Without a clear and effective mechanism for slaves to validate that received packets are from the legitimate master, the slave will accept anything that has the basic, easily observable characteristics of the master.

The master clock takeover attack described by the authors has key hallmarks that help the attack avoid detection by normal means and enhance the attack's effectiveness in elbowing out a legitimate master. First, the attacker can use fields in the PTP standard to suggest that it is more of a trustworthy time source than the legitimate node. More inherently-accurate precision timing sources are weighted more heavily and better sources are preferred by the nodes in the network, so an attacker advertising the highest "atomic clock" precision will be

preferred. The two stages of the attack they describe, sniffing and spoofing, can be readily performed by an insider attacker [19].

There is a large body of additional interesting and related work that examines PTP security mechanisms in practice, such as [20], [21], and [22]. We consider the works previously discussed here to be the most pertinent to our work.

2.3.3 Distinguishing Factors

The work we conducted differs from the aforementioned works in a number of respects. First, our work focused on the use of readily available COTS equipment, such as a Raspberry Pi, to enable a low-cost, easily-deployable, and open source precise timing command and control environment. Second, though the accuracy boost afforded by hardware timestamping is certain, we focused on the use of software timestamping to evaluate its security utility for a wider population of potential systems. Third, we sought to add this control layer on top of an existing OT system as a method of securing it against a MotS attack. Finally, we focused on the use of an arbitrary time, selected by the master in the network, in order to mitigate the attacker's false command injection (FCI).

CHAPTER 3:
Methodology

This chapter discusses the design and build phases of our research. This project had two primary parts. The first part was to complete the EEMS lab's OT rack so that it could be used as a basic simulation test bed for a ship's control system. The second part of the research was to add a supervisory control network, synchronized via PTP, and evaluate its ability to protect the subordinate control system against certain kinds of malicious activity.

Section 3.1 provides an overview of the hardware acquired by the EEMS lab that enabled the build. Section 3.2 describes the software associated with the various hardware components, the capabilities of the software, and the utility of each to our research. Section 3.3 specifically describes the build phase, whereby the constituent components discussed in section 3.1 were connected to provide the desired functionality. Section 3.4 describes the basic configuration of each of the components after assembly was completed. Section 3.5 details the governing approach to using our test bed to test and evaluate the MotS attacks by the malicious insider node, Eve. Finally, section 3.6 describes the command and control logic employed by the primary nodes Alice and Bob.

Due to three different programmatic contexts, we use three different groupings of naming conventions, depending on the context. In the context of the overall system, we use the names Alice, Bob, and Eve as originally described by [23] to refer to two legitimate nodes and a malicious node, respectively. In the context of PTP, Alice is the grandmaster, Bob is the PTP slave, and Eve ultimately acts as a rogue master. Finally, in the context of message queuing telemetry transport (MQTT) and the command and control communications the Alice node is referred to as human machine interface (HMI), the Bob node is referred to as remote terminal unit (RTU), and the Eve node attempts to conduct FCI.

First, we describe the hardware.

3.1 Hardware

The majority of the simulation hardware was acquired by the NPS EEMS lab from AutomationDirect, a popular North American vendor for industrial automation equipment. Additional equipment was acquired separately via SparkFun or other suppliers. Figure 3.1 below shows the front of the EEMS lab's OT simulation rack.

Figure 3.1. EEMS Lab OT Simulation Rack

Each of the constituent components, including the PTP control architecture, is described in greater detail in the following sections.

3.1.1 Motors

The two motors at the bottom of Figure 3.1 are two similar models of IronHorse general purpose, 1800 revolutions per minute (RPM) three-phase motors from AutomationDirect. The MTR2-P33 provides 1/3 horse power and the MTR2-P50 provides 1/2 horse power. Both are sufficient for simulation and demonstration, especially considering they are not intended to support an actual thrust load.

3.1.2 AC Drive

The AC variable frequency drives (VFD), on the bottom mounted rung in Figure 3.1, are GS1 series AC VFD from AutomationDirect. The drives are used to vary the speed of the aforementioned three-phase motors by converting AC power to DC, which is then synthesized back into three phase output power. The GS1-10P5 drives on the simulation rack support up to 1/2 horse power and take 100-120 volts on input with a maximum output voltage of 200-240 volts. The GS1 drive was configured with baseline settings according to Table 3.1. Of specific note in Table 3.1 are P3.00 and P4.00, which govern the source of operation and source of frequency and are both set via commands over RS-485 [24]. The possible alternative options for configuring the drive with different communication media or standards are described in [24].

Parameter	Description	Setting
P0.00	Motor Nameplate Voltage	230
P0.01	Motor Nameplate Amps	1.4
P0.02	Motor Base Frequency	60
P0.03	Motor Base RPM	1725
P0.04	Motor Max RPM	2000
P1.00	Stop Method	0 (Ramp to stop)
P1.01	Acceleration Time (seconds)	5.0
P1.02	Deceleration Time (seconds)	5.0
P2.00	Volt/Hertz Settings	1
P3.00	Source of Operation Command	3 (RS-485)
P4.00	Source of Frequency Command	5 (RS-485)
P8.00	User Defined Display Function	1
P9.01	Transmission Speed (baud)	2 (19200)
P9.02	Communication Protocol	5 (Modbus RTU)

Table 3.1. GS1 drive settings. Adapted from [24].

3.1.3 Programmable Logic Controllers

The PLC units are Koyo CLICK! PLCs, one of which has an ethernet-capable CPU. The primary PLC in use is the ethernet-capable C0-12-series that has four discrete input terminals, four discrete output terminals, four analog input terminals, and two analog output terminals, along with three options for communication ports: ethernet, Modbus, or RS-485 serial. They are managed using the provided CLICK! PLC software and initial setup was conducted using the ethernet communication interface [25].

3.1.4 Arduino Uno

The Arduino Uno prototyping board was used to manage the manual override and control signaling from the physical steering column (helm) and throttle (lee helm), enabling the

bypassing of the HMI push button interface when the emergency manual to override push button is pressed.

Due to challenges implementing RS-485 Modbus across the three different controller platforms (PLC, Arduino, and Raspberry Pi), we were not able to complete a full-path solution for the test bed. By the term "full-path", we mean that an ON command, by way of a button press on the HMI, traverses the entirety of the OT test bed network and actuates the motor by way of the VFD. Therefore, in addition to the aforementioned Arduino used to control the emergency manual override push button signaling, we added an additional Arduino to act as a VFD emulator and receive commands from the RTU over a similar RS-485 serial connection.

3.1.5 Raspberry Pi 4

The Raspberry Pi 4 units provide the backbone RTU to HMI communication loop using the MQTT lightweight and open-source publish/subscribe messaging protocol. They also enable the PTP master/slave software architecture. Additionally, a touchscreen HMI interface driven by a Raspberry Pi 4 can be provided in future work, but physical buttons and LEDs were installed in lieu of the touchscreen due to processing overhead.

3.1.6 Networking

We utilized a Linksys EA8500 router with gigabit ethernet interfaces. Gigabit speeds were maintained throughout the network, including corresponding network cabling (i.e., category 6 ethernet). In addition to the router, we used a Netgear GS105 gigabit switch. The router and switch were used to facilitate the MQTT and PTP communication as well as provide the aperture for Wireshark packet captures via an Apple MacBook Pro, also on the same network and subnet. Figure 3.2 shows this basic transmission control protocol (TCP)/IP networking configuration with all lines representing category 6 ethernet cables. The green lines represent the valid traffic paths for legitimate nodes Alice and Bob while the red line represents the attack vector of Eve in our test bed. It is shown later that removing the router by severing the connection between switch and router in Figure 3.2, and using statically-assigned IP addressing, can improve overall timing performance.

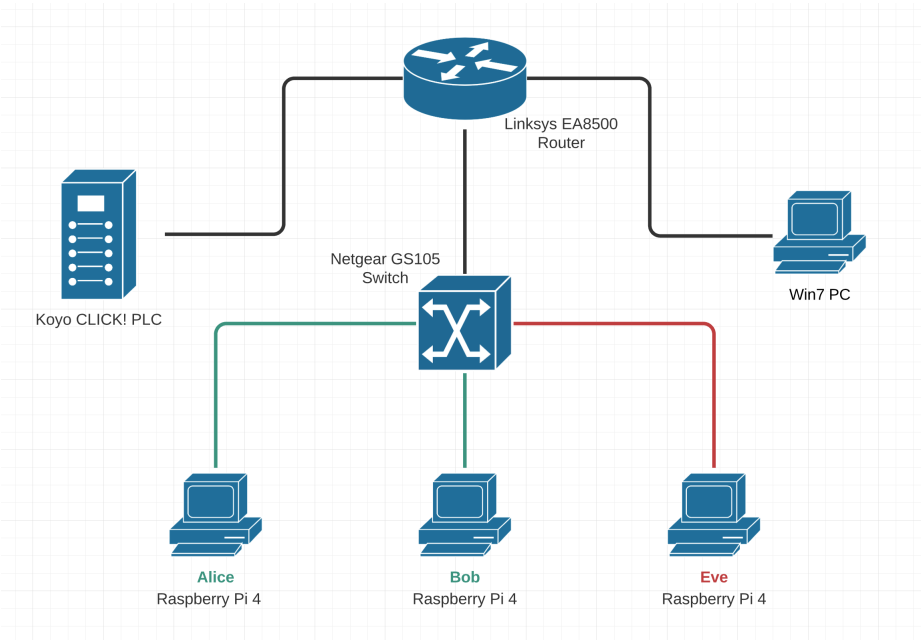

Figure 3.2. Test bed TCP/IP network map

Next, we describe the software used to enable key functionality of the aforementioned hardware.

3.2 Software

3.2.1 CLICK!

The Koyo CLICK! PLC comes with a robust software suite that is capable of communicating with the hardware over the TCP/IP, RS-232, or RS-485 protocols. It uses those protocols to read and write ladder logic-based programs to and from the PLC. Ladder logic is a written method of describing control circuitry that is based on relay contacts or switches, using Boolean expressions [26]. The PLC can use the TCP/IP connection to pass data back and forth from the connected PC. It can also receive a ladder logic program and then disconnect from the PC to operate in standalone mode. Figure 3.3 shows the software at startup, as it displays a list of connected and available PLCs, basic information about them, their current operating mode (RUN/STOP), and their current status.

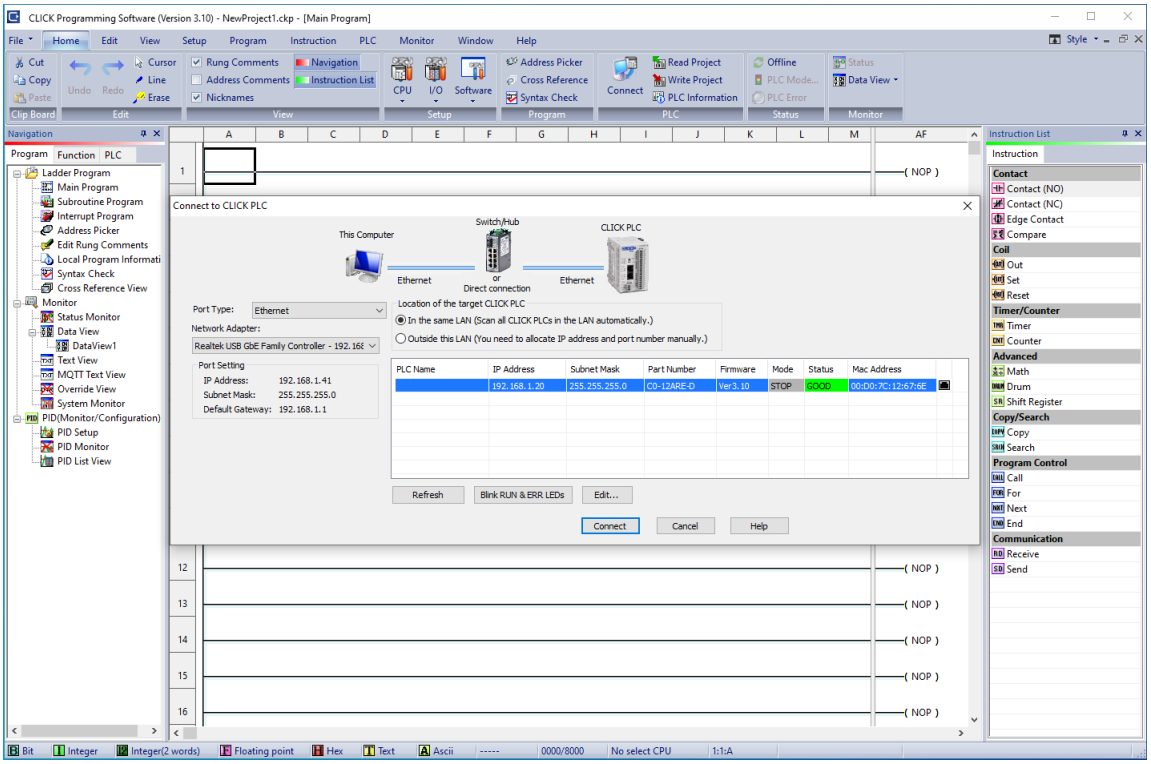

Figure 3.3. PLC communication software

3.2.2 Arduino

The Arduino integrated development environment (IDE) provides a simple, yet capable development environment for the units used in our work. It allows for specification of multiple COMM ports to talk to different boards simultaneously. Additionally, the IDE provides useful compilation and error handling in addition to the serial monitor, used to output serial data from Arduino input pins such as the potentiometers in the helm or lee helm. Figure 3.4 shows the Arduino IDE on the left and the serial monitor window on the right.

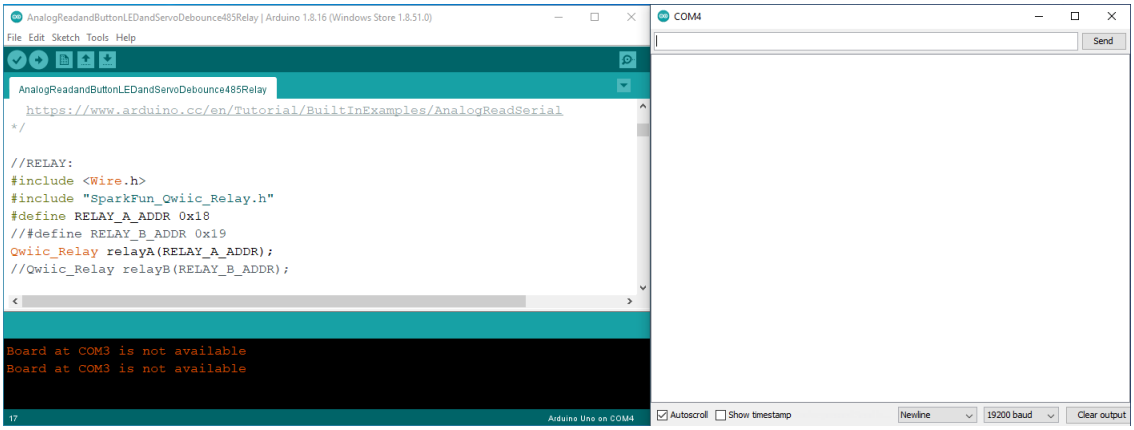

Figure 3.4. Arduino IDE software

3.2.3 RaspbianOS

The Raspberry Pis were loaded with the standard, most recent version of Raspbian OS, "buster," a powerful yet lightweight iteration of the Debian operating system (OS) that is optimized for the Raspberry Pi ARM architecture. The version we used is v5.10.17, which has baseline support for PTP included. It comes with configuration options that were useful here such as setting host names, enabling SSH, and enabling access to the Raspberry Pi's onboard general purpose input/output (GPIO) pins. Figure 3.5 shows the Raspberry Pi configuration window, accessed via SSH using `sudo raspi-config` (after enabling SSH during initial local configuration).

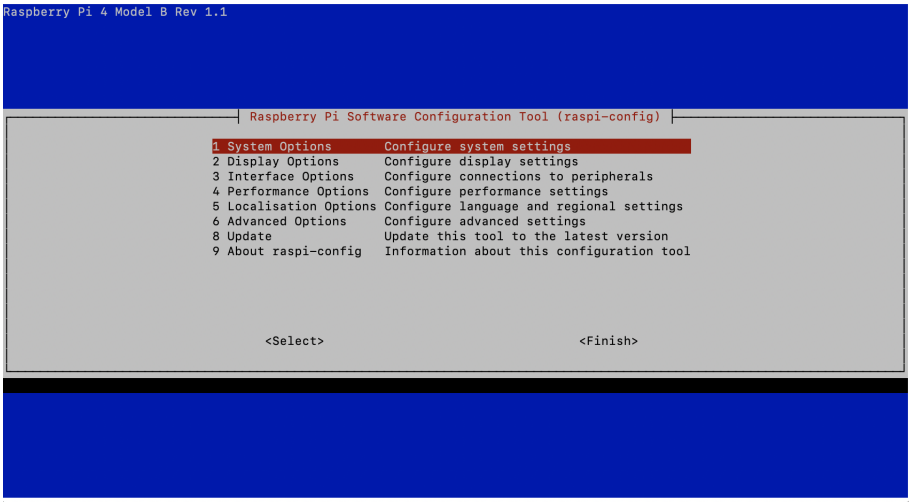

Figure 3.5. Raspberry Pi 4 configuration, accessed via SSH.

3.2.4 MQTT

MQTT, the message delivery protocol, uses a publish/subscribe messaging transport protocol. This protocol was selected for a few reasons. MQTT is lightweight, open source, reliable, and, importantly for ICS systems, very scalable. Per the OASIS standard and the MQTT governing body, MQTT is ideal for constrained environments and for machine-to-machine communications [27]. It is noted that shipboard environments do not typically use MQTT for vital ICS-type controls. MQTT afforded the advantages described above and provided the basic command and control necessary in this system. In order to leverage the advantages of MQTT in a small network configuration, we used a non-standard MQTT implementation with two clients communicating directly, rather than through a server as described by [27]. Figure 3.6 shows the basic communication loop between the two primary nodes. Solid lines indicate MQTT publish to the topic and dotted lines indicate MQTT subscribe to the topic.

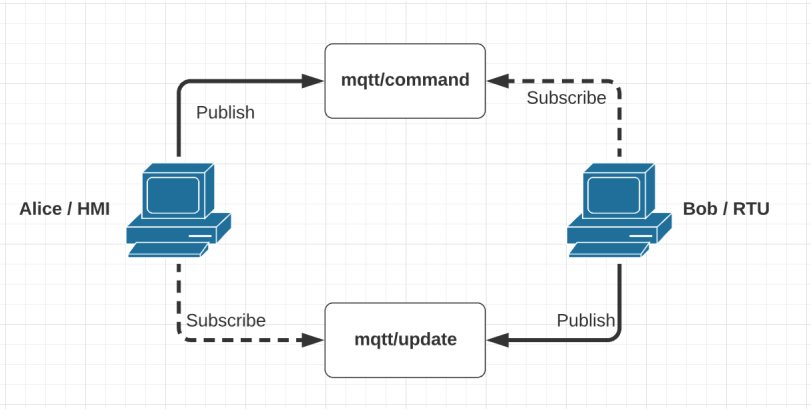

Figure 3.6. MQTT communication paths.

The utilized MQTT version supports transport layer security (TLS) encryption for connections between message brokers and subscribers, but in order to reduce potential errors, TLS was not employed.

3.2.5 PTP

There are a few different approaches to implement PTP on COTS equipment like Raspberry Pi units. Two different approaches to a software daemon, ptpd [28] and linuxptp [12], as well as a purely Python-based implementation of the IEEE 1588 2008 standard [29], were considered as possible methods of implementing PTP in this system. At the time of writing, there were no existing implementations of PTP based on the newer 2018 revision.

linuxptp The Linux PTP Project is an open source approach at implementing the IEEE 1588 standard for Linux kernels. It is licensed under the GNU General Public License. In 2010 Richard Cochran et al proposed an improvement on an earlier Linux kernel implementation of hardware timestamping in PTP, *ptpd* [30]. Cochran et al proposed subsequent improvements to the problem of synchronizing a Linux system clock with the PTP hardware clock at the NIC level [31]. As they stated in [31], they anticipated the necessary kernel patches that appeared in Linux Kernel version 3.0.0. Those patches and their work with PTP in a Linux kernel were part of the aforementioned Linux PTP Project and therefore the basis of *linuxptp*. Richard Cochran (and other contributors) continue to update and improve the tool,

which is available on his GitHub [12]. The *linuxptp* software package is easy to install and configure, and although our research only leveraged the software timestamping aspect, *linuxptp* is easily reconfigurable to support hardware timestamping with a compatible NIC. As the authors predicted in [31], Linux kernel version 3.0 does provide the necessary support for both hardware and software timestamping [12]. The *linuxptp* install package includes two subordinate packages: *ptp4l*, the actual daemon implementation, and *phc2sys*, which is the PTP Hardware Clock to System service. The *phc2sys* is only needed for hardware-level implementations that utilize hardware timestamping and require timing adjustments at the NIC level [12].

In the next section, we discuss the assembly and interconnection of the various hardware components.

3.3 Assembly

3.3.1 Arduino

Two Arduinos were utilized for this project. Arduino 1 serves as the emergency manual to override pushbutton and the physical helm and lee helm controls, as seen on the top of Figure 3.1. The position of the helm and lee helm is governed by a 5kΩ potentiometer inside each unit. The analog output terminal of both potentiometers are fed into the A0 and A1 analog input terminals on the Arduino, each supplied with 5V DC and grounded at the Arduino GND terminal. The voltage-interrupting pushbutton on the lee helm throttle also draws 5V DC, grounded through the board's GND terminal, and the button position is passed as a digital input via Arduino digital pin 2.

Two SparkFun Qwiic relays are used to switch the communication path from "normal operation" via the HMI, RTU, and PLC to the "emergency operation" condition directly to the VFD via a MAX485 module available to the Arduino. Additionally, after the push button is pressed, a red LED attached to pin 13 is used to visually display that the system is in the emergency condition by energizing the light in addition to the relays switching to the Normally Open position. The left image in Figure 3.7 shows the setup for Arduino 1 and the right image in Figure 3.7 is the full schematic diagram for the circuitry.

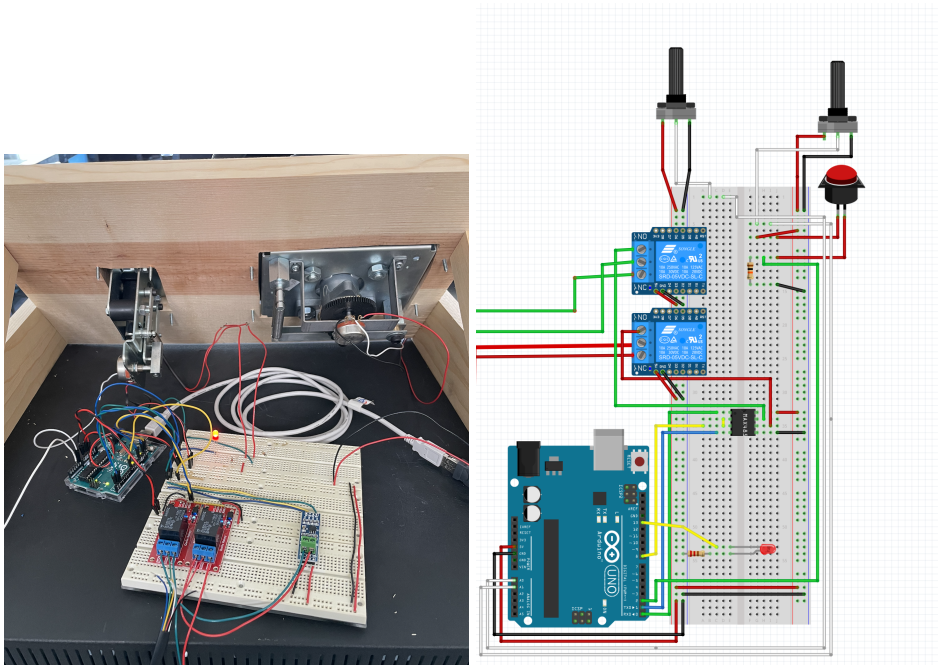

Figure 3.7. Photograph and Fritzing schematic of the helm/lee helm console with the emergency manual push button circuitry

Arduino 2 powers the VFD emulator, which was added in order to demonstrate the PTP mechanism, absent a full-path solution from HMI to RTU to PLC to VFD. Arduino 2 has a very simple configuration, with only a MAX485 module to simulate the VFD receiving legitimate commands from the RTU and an LED to simulate the on/off actions of the VFD. The MAX485 module is powered by 5V DC, grounded to Arduino GND, with terminals A and B connected to the corresponding terminals on the RTU. Additionally, the MAX485's DE and RE pins are coupled and connected to Arduino pin 2. MAX485 DO is connected to Arduino pin 0 (Rx) and MAX485 DI is connected to Arduino pin 1 (Tx). The LED, grounded by a 220Ω resistor, is connected to Arduino pin 13. The image on the left in Figure 3.8 shows the setup for Arduino 1 and the diagram on the right in Figure 3.8 provides a full schematic for the circuitry.

3.3.2 Raspberry Pi

The common assembly across all Raspberry Pi 4s was that each was connected via category 6 ethernet cable to the Netgear switch, which was also connected via category 6 ethernet

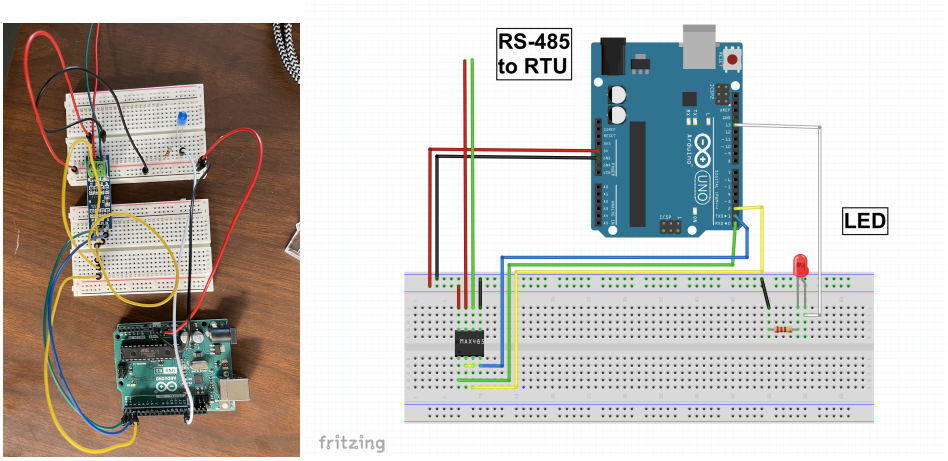

Figure 3.8. Photograph and Fritzing schematic of the second Arduino unit, powering the VFD emulator

cable to the Linksys router. The individual assembly steps for the HMI and RTU are described below.

HMI Physical buttons, LEDs, and potentiometers were installed through the HMI's available GPIO pins in order to reduce processing overhead incurred by driving an LCD touch panel display (per the original design).

The three push buttons, corresponding to the three LED status lights (green, red, and blue), represent states on, off, and forward/reverse respectively. The push buttons are each hard wired with pull-up resistors via the 3.3V DC power supplied by the Raspberry PI board. Although pull-up conditions can be applied in software through the Raspberry Pi GPIO configuration, hard wiring reduced code length and increased the reliability and accuracy of button presses. For the first button, a 10kΩ resistor from 3.3V DC was connected to the rung shared by Raspberry Pi pin 36 and a 1kΩ resistor into the left terminal of the push button. The right terminal of the push button is wired directly to GND. This same configuration is duplicated for the other two push buttons, except button 2 is wired to Raspberry Pi pin 38 and button 3 is wired to Raspberry Pi pin 40. Pins 36, 38, and 40 were configured via GPIO setup as digital input pins.

The three LED status lights are wired in a similar fashion as both Arduinos, but with the output from Raspberry Pi pin 11 wired into a 220Ω resistor and then into the anode side

(long leg) of the LED. The cathode side (short leg) of the LED is wired to GND. As the VFD emulator is only configured to receive ON or OFF commands from the RTU, the potentiometer is not currently configured to send data to the HMI; the transmission of potentiometer data would require either an analog to digital converter (ADC) or another Arduino unit.

RTU The RTU is configured more simply than the HMI, and not unlike Arduino 2. The RTU communicates with Arudino 2, the VFD emulator, via a corresponding MAX485 module that is connected via the Raspberry Pi GPIO pins. This MAX485 draws 5V DC from the Raspberry Pi pin 2 and returns to GND on pin 6. As with Arduino 2 (the VFD emulator), terminals A and B on the MAX 485 connect via hookup wire to the corresponding A and B terminals on Arduino 2's MAX485 module. This established the RS-485 communication link between the two. As with the other MAX485 modules, DE and RE are coupled, but the coupled wire is connected to Raspberry Pi pin 7. MAX485 DO is connected to Raspberry Pi pin 10 and DI is connected to Raspberry Pi pin 8.

Figure 3.9 shows the setup for both HMI and RTU Raspberry Pi units and Figure 3.10 is the full schematic diagram for the circuitry, also via Fritzing.

3.3.3 Click PLC

The CLICK PLC system is wired for power in accordance with reference [32], using 24V DC power into the terminal blocks as shown in Figure 3.11. For communications with the PC running the CLICK! PLC Software, Comm Port 1 (TCP/IP) has a category 6 ethernet cable attached to the router providing the LAN for the project. It is worth acknowledging that this configuration does technically expose the PLC to the attacking Raspberry Pi unit that is also on the same LAN and subnet as the HMI, RTU, and now PLC. This configuration was chosen in order to simplify the experimental setup and that attack vector is neither considered nor explored any further in this work. A future implementation that would avoid this risk could, instead of TCP/IP, utilize the Comm Port 2 (RS-232) and a module for the RS-232 serial communication protocol. The VFD section that follows discusses the communication connections required to control the VFD via the PLC.

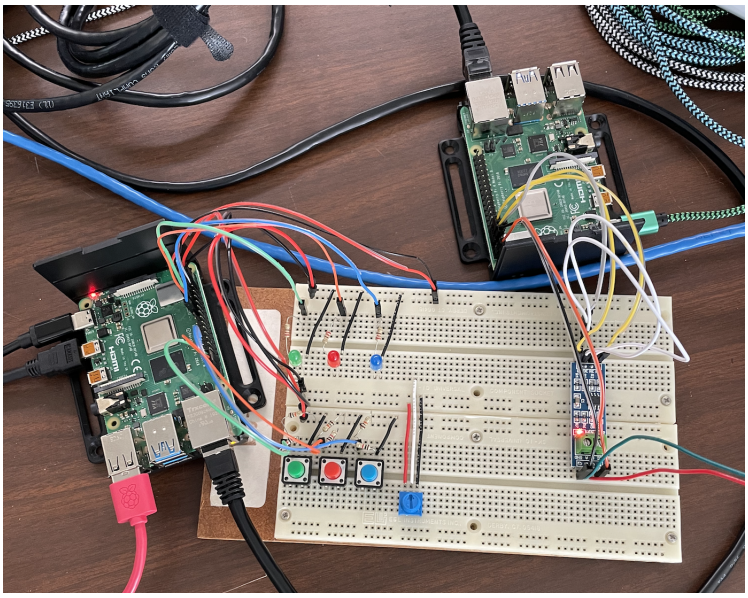

Figure 3.9. Photograph of the configuration of the HMI (left Raspberry Pi) and the RTU (right Raspberry Pi)

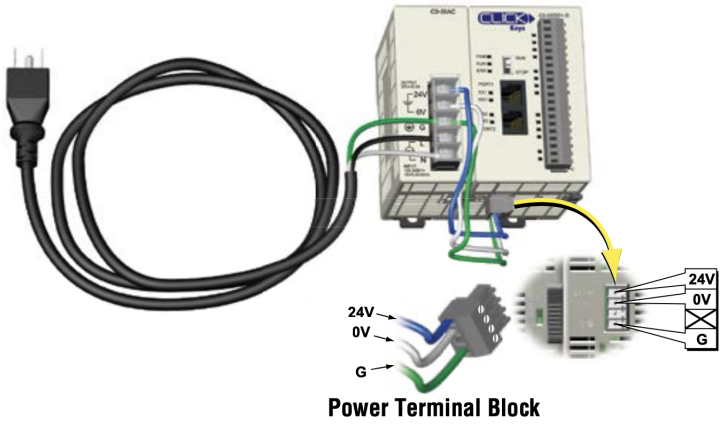

Figure 3.11. Wiring of CLICK PLC for power, from [32].

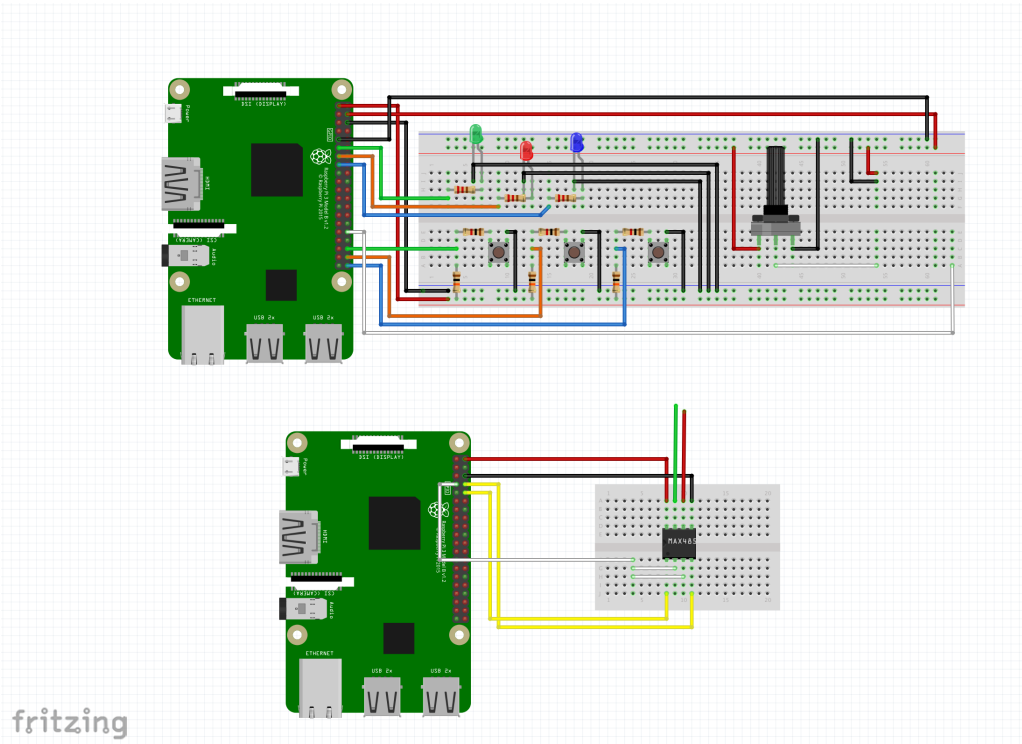

Figure 3.10. Full Schematic of HMI and RTU Configuration, using Fritzing

3.3.4 VFD

The AutomationDirect GS1 series VFD was wired in accordance with [24], using the 1-phase input power connections and 3-phase output power connections. Figure 3.12 shows the necessary connections for each VFD from the AC motor or power source. The communications are provided via the AutomationDirect pre-terminated RS-485 cable (GS-485HD15-CBL-2), ultimately connected from Comm Port 3 on the PLC to the RJ-45 port on the bottom of the VFD. Note that the wiring in this specific implementation is re-routed through the relays as part of the emergency manual override functionality.

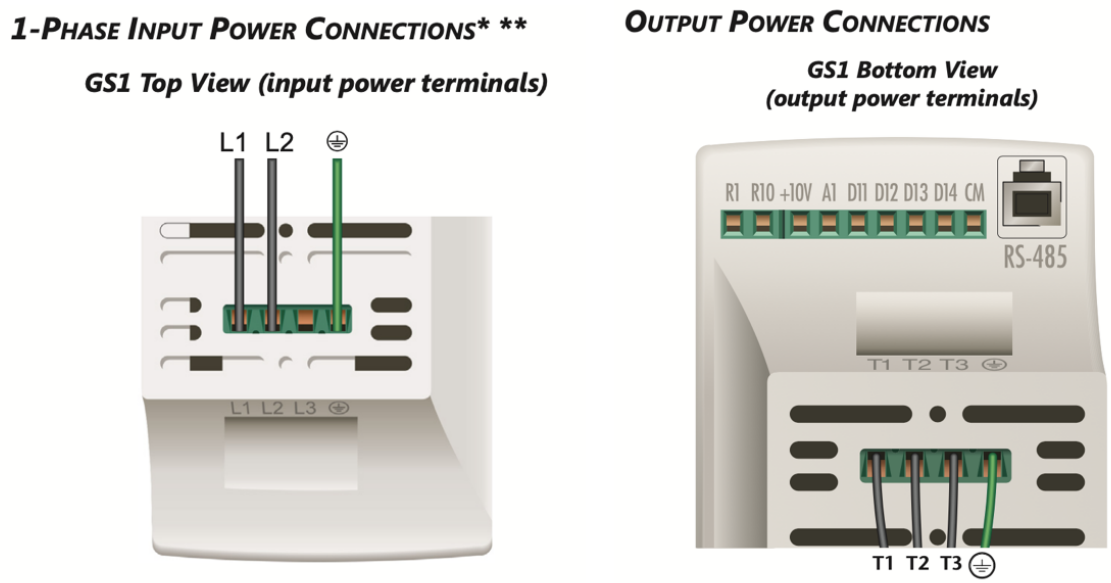

Figure 3.12. Wiring of 1-phase input and 3-phase output, from [24].

3.3.5 Full System Schematic

Figure 3.13 shows the full configuration layout of our test bed. It is worth noting that the configuration shown does incorporate the PLC connected via the same network hardware as both the HMI and RTU.

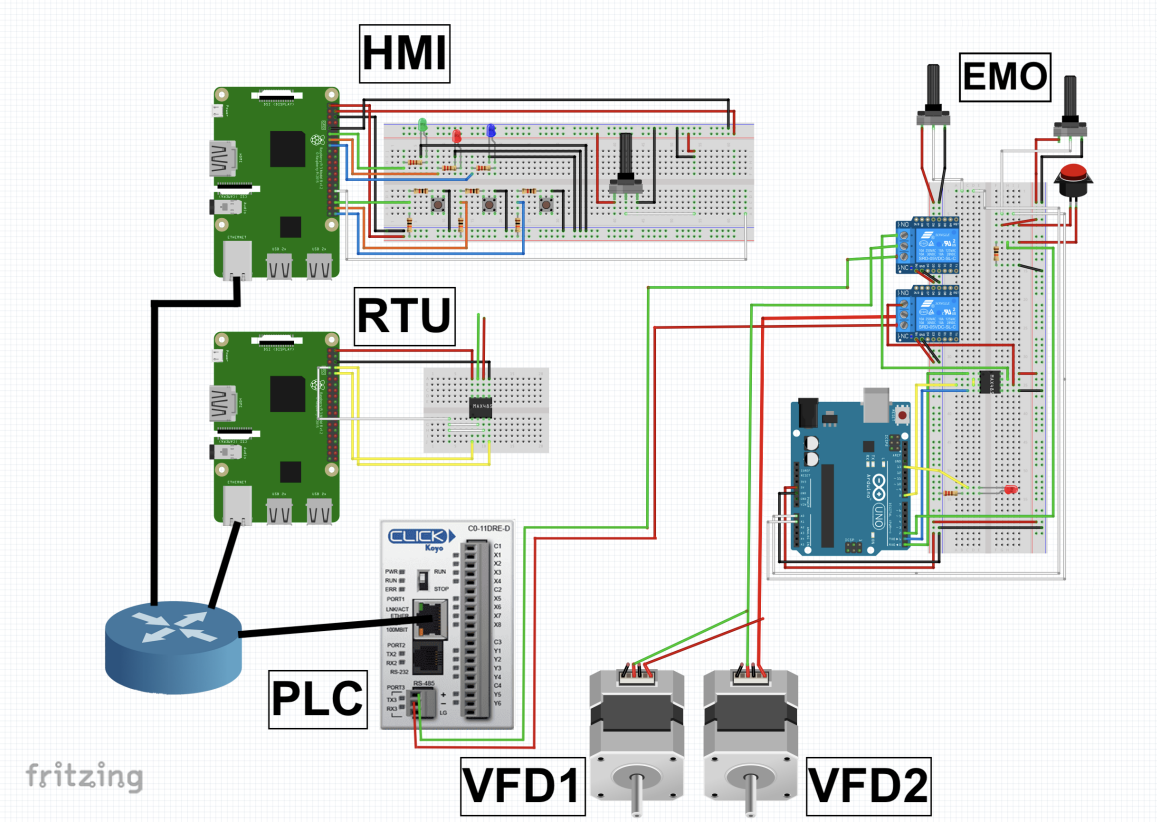

Figure 3.13. Full system schematic, using Fritzing

In the next section, we discuss the configuration of the components of this test bed.

3.4 Configuration

3.4.1 Networking

The nodes on the network were configured with static IP addressing corresponding to Table 3.2. A MacBook Pro with a dynamically assigned IP address was utilized for configuration and testing on the nodes via SSH. WiFi was disabled on the router.

Hostname	IP Address	MAC Address
alice	192.168.1.38	DC:A6:32:30:FF:F3
bob	192.168.1.39	DC:A6:32:30:E6:F6
eve	192.1681.36	DC:A6:32:1B:02:06
clickplc	192.168.1.20	00:D0:7C:12:67:6E

Table 3.2. Static IP address assignments for experimentation LAN

3.4.2 Raspberry Pi

All three of the Raspberry Pi units received the following initial configuration settings (via `sudo raspi-config`), as seen in Figure 3.5:

1. 1 > S1 (Wireless LAN) > [Set to join hotspot for updates and package installs]
2. 1 > S4 (Hostname) > [Set to alice/bob/eve]
3. 1 > S5 (Boot / Auto login) > B2 (Console Autologin)
4. 1 > S6 (Network at Boot) > Enabled
5. 1 > S7 (Splash Screen) > Disabled
6. 3 > P2 (SSH) > Enabled
7. 3 > P3 (VNC) > Disabled
8. 3 > P4 (SPI) > Enabled (not req'd for Eve)
9. 3 > P5 (I2C) > Enabled (not req'd for Eve)
10. 3 > P8 (Remote GPIO) > Enabled (not req'd for Eve)

Additionally, all three received the following common installation packages, installed via standard Linux installation methods:

```
sudo apt install python3
sudo apt install linuxptp
sudo apt install mosquitto mosquitto-clients
sudo apt-get install paho-mqtt
```

The *linuxptp* installation package created a new directory in `/etc/linuxptp`, where two configuration files are now located, *ptp4l.conf* and *timemaster.conf*. *timemaster.conf* is the configuration file for the *timemaster* service that governs fallbacks to NTP timesources or in instances when there are multiple PTP domains that require greater specificity. *ptp4l.conf*, the configuration file for the *ptp4l* service, is the focus here. The full configuration file for each node can be found in Appendix A.3, A.4, or A.5.

All three nodes had settings that needed to be uniform across all nodes in order to properly communicate via PTP. For example, all three nodes needed to have software timestamping, the appropriate interface, and the correct clock servo mechanism in agreement. Most default settings were maintained, as they provided a sufficient synchronization. For example, the use of two-step, the delay mechanism (end to end, vice peer to peer), and clock settings were maintained. There were some key changes that needed to be implemented according to the circumstances of each node. For example, Bob (the RTU) needed to have *slaveOnly* set to 1, as there was no condition in which Bob would become the PTP grandmaster. Additionally, in order to ensure that Bob would re-synchronize following a change in masters, *step_threshold* was set to the same value as *first_step_threshold* (0.00002). Also, adding [eth0] to the end of the configuration file for each eliminates the need to add the option to specify an interface when running *ptp4l* either from command line or starting it as the background service.

3.4.3 Arduino

After the hardware assembly was completed and the board was wired correctly, the Arduino configuration was very simple. Each board was loaded with the corresponding Arduino Sketch file, and then the onboard microcontroller does the rest of the work. Arduino 1 was loaded with the sketch file in Appendix A.1.2 and Arduino 2 was loaded with the sketch file in Appendix A.1.1.

Next, we discuss the overarching strategy for using precise timing in this system to mitigate a MotS attack.

3.5 Implementation Approach

With the test bed assembled and configured, we address the second phase of our project: examining a method of defeating a MotS (Eve) through effective use of precise timing. As

the MotS, Eve is presumed to have physical access to the network, can see data flowing across it and between the nodes, capture data, and even send data to the various nodes. But, Eve is unable to delete valid data from the system; she is only able to inject her own malicious and false commands into the system. Eve has knowledge of how the two legitimate nodes, Alice and Bob, work together via MQTT and PTP. Additionally, Eve is ultimately capable of even hijacking the PTP communication cycle, for example, by advertising a better priority bit in the hopes of distorting the best master clock algorithm in her favor.

Successful defense against these attacks rests on Bob, the RTU and PTP slave also connected via RS-485 to the VFD emulator, using the timestamp information required in the MQTT message to determine if the associated command also inside the MQTT message is valid or not. The commands currently accepted between the two are either "on" or "off," controlling the LED on the VFD emulator.

Alice, the HMI and the PTP master, selects an arbitrary time, changes its system clock accordingly, and then begins to synchronize with Bob using *linuxptp* suite, specifically *ptp4l*. Again, the *phc2sys* service is not needed here as the NIC in the Raspberry Pi 4 models do not support hardware timestamping. Using configuration guidance from the Red Hat Customer Portal, *ethtool* is a useful utility in order to verify key capabilities of the NIC and kernel at large [33]. Figure 3.14 shows the use of *ethtool* to validate the NIC and kernel capability to handle at least software timestamping. The key fields to be present in the Capabilities section of the output of *ethtool* are "software-transmit", "software-receive", and "software-system-clock." Hardware timestamp-capable units will have additional entries that correspond to: "hardware-transmit," "hardware-receive," and "hardware-raw-clock" [33].

```
pi@eve:~ $ sudo ethtool -T eth0
Time stamping parameters for eth0:
Capabilities:
        software-transmit     (SOF_TIMESTAMPING_TX_SOFTWARE)
        software-receive      (SOF_TIMESTAMPING_RX_SOFTWARE)
        software-system-clock (SOF_TIMESTAMPING_SOFTWARE)
PTP Hardware Clock: none
Hardware Transmit Timestamp Modes: none
Hardware Receive Filter Modes: none
```

Figure 3.14. Screenshot of *ethtool* used to check timestamping capability, in this case of Eve.

Bob, now synchronized to Alice's arbitrary time, has a known reference point for checking the validity of the incoming messages. Separately, an average MQTT message processing and propagation delay was computed and that value is also used by Bob to conduct the validity checks. If the command packets received are within the mean time differential then they are considered valid.

This is particularly useful for the defender because a MotS that is either trying to guess the timestamps or, based on intercepted traffic is trying to preempt them by sending ones slightly ahead, has a very narrow window for its messages to be accepted as valid. Additionally, the defender, Bob, can safely drop all packets timestamped ahead of the synchronized arbitrary time, as receiving data from the future is an impossible legitimate scenario for Bob. What remains is the very narrow window of time in which Eve must correctly guess (or compute) the precise time and also transmit it within the known offset.

In the next section, we describe the methodology that governs normal command and control within our experimental test bed.

3.6 Python Scripts

Python 3 was the preferred choice to implement the code design because of its simplicity, extensibility (i.e. in importing the *paho-mqtt* library, among others), and its ability to handle date time objects and strings. MQTT delivers the messages as strings, and it is important to be able to convert the timestamps in the received messages back into a datetime object for accurate computation of time differentials. The basic MQTT publish and subscribe framework that we utilized drew from the CS3250 Cyber Physical Systems Laboratory #3 by Singh and Prince [34]. The pseudocode of the program logic follows here, and the Python 3 code can be found in Appendix A.6 and A.7.

3.6.1 Alice Psuedocode

```
Timing:
    Alice sets system time as arbitrary
    ptp4l synchronizes Bob to Alice's arbitrary time
Command Messaging:
    while True:
```

```
        Check for button press
        if Green button was pressed:
            Create "on" payload
            Get current time & add to "on" payload
            Send payload via MQTT
        if Red button was pressed:
            Create "off" payload
            Get current time & add to "off" payload
            Send payload via MQTT
        Receive status update payload from Bob
```

3.6.2 Bob Psuedocode

```
Timing:
    Bob uses ptp4l to synchronizes to Alice's arbitrary time
Command Messaging:
    while True:
        Check for new MQTT message (via on_message)
            if new message received:
                Get current time
                Decode message (parse command and timestamp)
                Compute difference of current time & timestamp
                if difference < average MQTT time difference:
                    Command is valid
                    Forward Command to VFD emulator
                if difference >= average MQTT time difference:
                    Command is invalid
                    Disregard/Drop
```

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4:
Testing & Results

This section describes the testing methodology and discusses the results captured from our test bed. Subsection 4.1.1 describes the testing process of the *linuxptp* software suite, detailing a series of tests that compare different configurations in search of optimal settings. Subsection 4.1.2 details the configuration and testing of the MQTT control messaging. Finally, section 4.2 describes the key findings regarding our use of arbitrary time and the effects of the rogue master takeover attack.

4.1 Testing

In this section we discuss the testing of the key constituent protocols in our testbed PTP and MQTT. For data visualization, we utilized Google Colab, which is a cloud based coding environment similar to Jupyter Notebook. There were a number of advantages conferred by using Google Colab such as ready access to a wide array of powerful libraries like Pandas, NumPy, SciPy, Matplotlib, and more. Additionally, the cloud backup and version history components of Google Colab proved to be useful advantages in the course of the data analysis. The Python code that we used to generate the data and figures in this section can be found in Appendix A.8. In addition to Google Colab, we used the popular packet capture program Wireshark to examine the multicast PTP traffic flowing through the network.

First, we will discuss PTP synchronization using *ptp4l*.

4.1.1 PTP Synchronization

With the *linuxptp* package installed on the relevant nodes according to Section 3.4 and utilizing the configuration found in Appendices A.3 and A.4, we ran the PTP synchronization under a number of different scenarios. For example, we first gathered the initial synchronization data by running *ptp4l* directly from the command line, with the output of the process going to standard out. Subsequent data collections compared various aspects of the *ptp4l* application such as clock servo methods linear regression (*linreg*) versus pi constant (pi). For another example, we compared altered network configurations such as the

set up described by the full network diagram in Figure 3.2 versus a "bare bones" switch-only configuration. These comparisons and others are described in greater detail in this section.

Initial Synchronization

Our first data collection with *linxuptp* on Alice and Bob was an approximately 23 minute run, with output from the various services being printed to standard out (stdout) over the secure shell (SSH) connection of the MacBook Pro. This initial synchronization run will also be used to describe some of the features, behaviors, and output of the *ptp4l* program in practice. Figure 4.1 shows the result of this synchronization run with the calculated offset from the master's time along the y-axis in nanoseconds and the total run time along x-axis in seconds. The *ptp4l* application prints new computed master offset values to standard out approximately every second.

Figure 4.2 shows an example of typical PTP logging messages via *ptp4l* on Bob. The command used to execute this run on Bob can be seen in the first line of Figure 4.2. Option `-f` with `/etc/linuxptp/ptp4l.conf` specifies the configuration file for *linuxptp* to consult. Additionally, options `-s -m` in the same figure explicitly declare the run to use *SlaveOnly* mode and route output to standard out, respectively. In order for synchronization to occur, Alice needed to also be running *ptp4l*, using the configuration found in A.3, also with the `-m` option on the command line to also route output to standard out over SSH. Alice can use a lower *priority1* and *priority2* value in order to ensure she takes on the role of PTP grandmaster, however specifying Bob as *SlaveOnly* as previously described will achieve the same effect: Alice assumes the role of PTP grandmaster and Bob assumes PTP slave.

We take this opportunity to examine some of the output of *ptp4l* using Figure 4.2 from Bob. After the first line in Figure 4.2 that was described above, each line of the *ptp4l* standard output begins with `ptp4l[X]:`, where `X` is a float that represents the current uptime, or the number of seconds since the last reboot. In the Linux kernel, this value is stored in `/proc/uptime`. Our data visualization script remaps this number to begin at 0 s.

In the first few lines of Figure 4.2, the *ptp4l* output, there are important port state changes to examine. According to [33] and the *ptp4l* manual page, port 0 refers to a Unix domain socket used to manage PTP, while port 1 refers to the specific interface in use by *ptp4l*–in our case this is Raspberry Pi *eth0* interface, as specified in the configuration file.

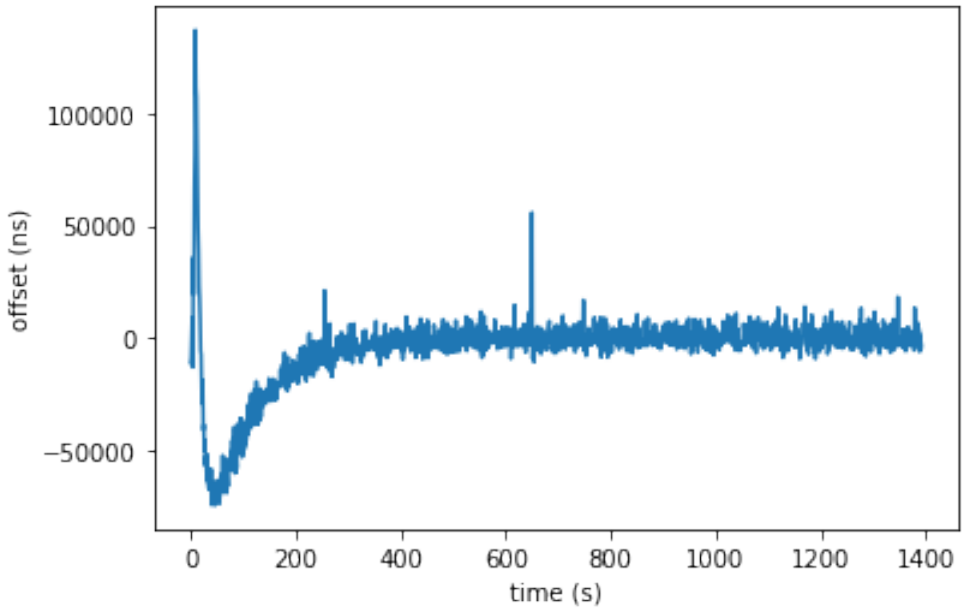

Figure 4.1. Initial PTP synchronization using *ptp4l* from the *linuxptp* package

We can see in the first line of *ptp4l* output that Bob's correct interface is now in the LISTENING state, looking for a grandmaster from which to receive timing data. As the corresponding service had already been running on Alice, we can see in the third line of Figure 4.2 that Bob detected a new foreign master. This was based on the *announce* messages sent by the *ptp4l* program on Alice.

The next four lines reflect Bob working through the best master clock algorithm (BMCA) as specified in [11] and enacted by [12]. All of the lines of Bob's standard output in Figure 4.2 that contain `master offset` are the data lines that we include and parse in order to analyze further. These data lines contain five key data elements: kernel uptime in seconds, master offset, clock servo state (`s0`/`s1`/`s2`), frequency, and path delay. Uptime and master offset have been described previously, but clock servo state is an important value to describe as well. A clock servo is a mechanism used to correct the offset drift between a master and subordinate slave nodes [35]. In the context of *linuxptp* and *ptp4l*, the clock servo state refers to the degree to which the *ptp4l* servo can change the system time to match what it has computed as the correct time, based on the PTP messages it has received from the

Statistic	Value
Number of Values	1393
Mean offset	-5484.5 ns
Mean abs(offset)	9194.2 ns
Median	3916ns
Standard Deviation	1548.1
Mean (log normalized)	8.24 ns
Median (log normalized)	8.2 ns
Standard Deviation (log normalized)	1.38

Table 4.1. Key statistics from the initial synchronization with *ptp4l*.

master. State `s0` is unlocked, meaning the slave's clock is not yet ready to track the master time [12]. State `s1` is clock step, meaning the slave's clock servo is ready to correct its time based on the master's calculated offset [12]. Finally, state `s2` is locked, meaning the slave's clock servo has locked onto the master's timing offset and is gradually altering its time, consistent with the new incoming offset values [12], [33]. The master offset values in states `s0` and `s1` can be extreme prior to synchronization, and this is especially true with our use of an arbitrary time value at the master. Because of the extreme values in those first two clock servo states, the tendency of those values to artificially skew results, and the *ptp4l* application's swift shifts into state `s2` after approximately 16 seconds in state `s0`, the data we use excludes those first two clock servo states. The remainder of Bob's output from *ptp4l* in this run contains only `s2` data and would only include additional information such as port state changes if there was a change in master, of which there was not in this run.

It required just under seven minutes during this initial run and baseline performance reference for the offset to stabilize near what we will later describe as our optimal achieved offset range for software timestamping. In Figure 4.1, this stabilization occurs near the 400 s mark. Using the data visualization methods described at the beginning of this chapter that can be found in Appendix A.8, we will examine key metrics of this initial run. Table 4.1 displays the key statistics we extracted from this initial synchronization. Of note, the standard deviation was extremely high (1548.1) until logarithmically normalized (1.38).

```
pi@bob:~ $ sudo ptp4l -f /etc/linuxptp/ptp4l.conf -s -m
ptp4l[426078.884]: port 1: INITIALIZING to LISTENING on INIT_COMPLETE
ptp4l[426078.884]: port 0: INITIALIZING to LISTENING on INIT_COMPLETE
ptp4l[426084.148]: port 1: new foreign master dca632.fffe.30fff3-1
ptp4l[426085.199]: selected local clock dca632.fffe.30e6f6 as best master
ptp4l[426088.145]: selected best master clock dca632.fffe.30fff3
ptp4l[426088.145]: foreign master not using PTP timescale
ptp4l[426088.145]: port 1: LISTENING to UNCALIBRATED on RS_SLAVE
ptp4l[426089.144]: master offset -73688524551 s0 freq +500000 path delay    392702
ptp4l[426090.144]: master offset -73689525981 s0 freq +500000 path delay    392702
ptp4l[426091.143]: master offset -73690431350 s0 freq +500000 path delay    309053
ptp4l[426092.142]: master offset -73691431669 s0 freq +500000 path delay    309053
ptp4l[426093.142]: master offset -73692316429 s0 freq +500000 path delay    212190
ptp4l[426094.141]: master offset -73693374143 s0 freq +500000 path delay    260621
ptp4l[426095.140]: master offset -73694357829 s0 freq +500000 path delay    260621
ptp4l[426096.139]: master offset -73695413228 s0 freq +500000 path delay    309053
ptp4l[426097.139]: master offset -73696391260 s0 freq +500000 path delay    309053
ptp4l[426098.138]: master offset -73697402513 s0 freq +500000 path delay    309053
ptp4l[426099.137]: master offset -73698399158 s0 freq +500000 path delay    315435
ptp4l[426100.136]: master offset -73699391656 s0 freq +500000 path delay    315435
ptp4l[426101.135]: master offset -73700330661 s0 freq +500000 path delay    260621
ptp4l[426102.134]: master offset -73701291419 s0 freq +500000 path delay    255003
ptp4l[426103.134]: master offset -73702317701 s0 freq +500000 path delay    256433
ptp4l[426104.133]: master offset -73703350190 s0 freq +500000 path delay    287019
ptp4l[426105.132]: master offset -73704324294 s1 freq -487762 path delay    272053
ptp4l[426106.132]: master offset     -11007 s2 freq -488873 path delay    272053
ptp4l[426106.132]: port 1: UNCALIBRATED to SLAVE on MASTER_CLOCK_SELECTED
ptp4l[426107.132]: master offset      -8643 s2 freq -488646 path delay    272053
ptp4l[426108.132]: master offset       2552 s2 freq -487524 path delay    241467
ptp4l[426109.132]: master offset       9257 s2 freq -486844 path delay    241467
ptp4l[426110.132]: master offset     -12898 s2 freq -489072 path delay    241467
```

Figure 4.2. Example prints to standard out using *ptp4l* from the *linuxptp* package

Subsequent runs did not produce the same six to seven minute "cold start" as this initial run. In fact, they began basic stabilization inside our tolerable offset threshold from the beginning. These additional runs will be described in greater detail later. But before we proceed, we will further examine this initial run from the point of synchronization after the cold start, near the 400 s mark of 4.1. Figure 4.3 shows the rest of the run from this point forward and gives a glimpse of the normal offset range for *ptp4l* with software timestamping in our test bed. Table 4.2 shows the key statistics for this stabilized segment of the initial synchronization. Here, we see the initial standard deviation (3019.5) is substantially better than the full initial synchronization as shown in Table 4.1, and when normalized, the standard deviation is very close to 1 (1.09).

Next, we will examine a few different *ptp4l* testing scenarios, beginning with a "long run" of the service.

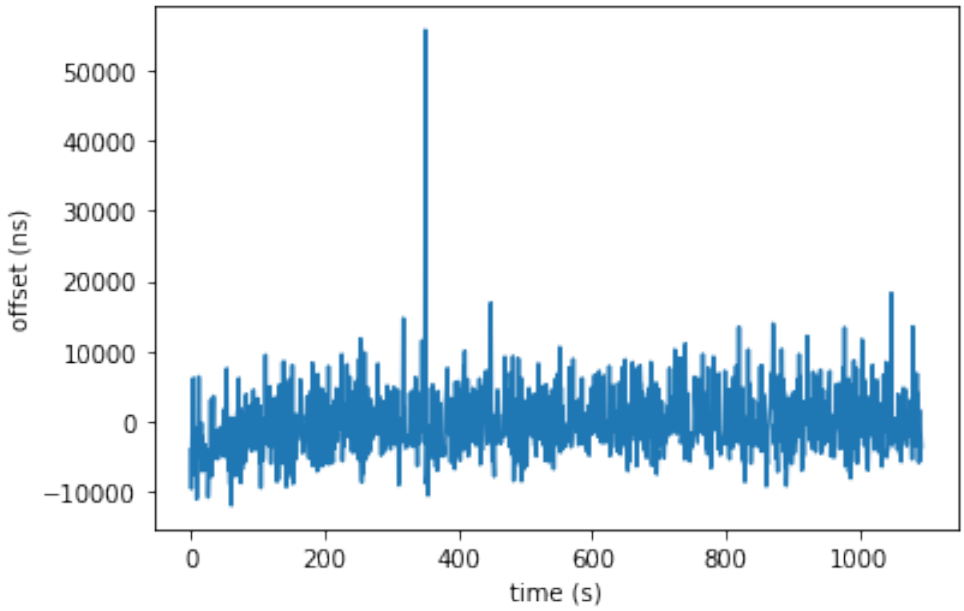

Figure 4.3. Initial PTP synchronization, following offset stabilization using *ptp4l* from the *linuxptp* package

Long Run Data

In order to assess the stability of the *ptp4l* service over a longer period of time, we captured data for over two hours (7600 s). The long run proved to be very stable over the two hour period, with some identifiable outliers where the offsets exceeded the optimal offset window. Figure 4.4 shows the 7600 second data capture with identifiable outlier clusters near the 300 s, 4000 s, 5000 s, 6000 s, 6500 s, and 7500 s marks. The most clear outliers occurred near the 4000 s and 5000 s marks. For this data set, we found an average offset of -10 ns, a median offset of -442 ns, an average absolute offset of 4187 ns, and a standard deviation of 5410. After logarithmic normalization, the standard deviation is 1.18. Of note, just the four largest outlier offset values were correctly identified using a Z-score of 2.75. By trial and error, a Z-score of 2.35 resulted in the successful identification of all six offset outlier clusters, at the aforementioned intervals in Figure 4.4.

Statistic	Value
Number of Values	1094
Mean offset	-481 ns
Mean abs(offset)	3568.4 ns
Median	3066 ns
Standard Deviation	3019.5
Mean (log normalized)	7.8 ns
Median (log normalized)	8 ns
Standard Deviation (log normalized)	1.09

Table 4.2. Key statistics from the initial synchronization, post cold start with *ptp4l*.

Switch-Only Network Configuration

As previously discussed in Section 3.1.6, we examined a more simplistic network configuration like the one in Figure 3.2, where the connection between the switch and the router had been severed, leaving only the Raspberry Pi nodes in communication by way of the Netgear Switch and via statically assigned IP addresses. Using this basic networking configuration for an approximately 15 minute run, we found a mean offset of 83 ns, a median of -502 ns, and an a mean absolute offset of 3021 ns. The standard deviation after logarithmic normalization was 1.07. With this run, a Z-score of 2.25 was needed in order to correctly identify the two primary outlier offset values.

Standard Out versus Background Service

In comparing these two options, we are by extension evaluating to what degree increased processor or increased network stress may impact the master offset values on Bob. The standard out option, printing to a console connected via SSH, is the mechanism creating this increase in network traffic, as each print of standard out is transmitted to the connected console over that remote SSH connection. Meanwhile, with the nodes running *ptp4l* as a service, they do not transmit more traffic than necessary and record the behavior of the service in `/var/log/messages`. The entries into this file by the *ptp4l* service are almost identical to what is printed via standard out as seen in 4.2, but with the current system date

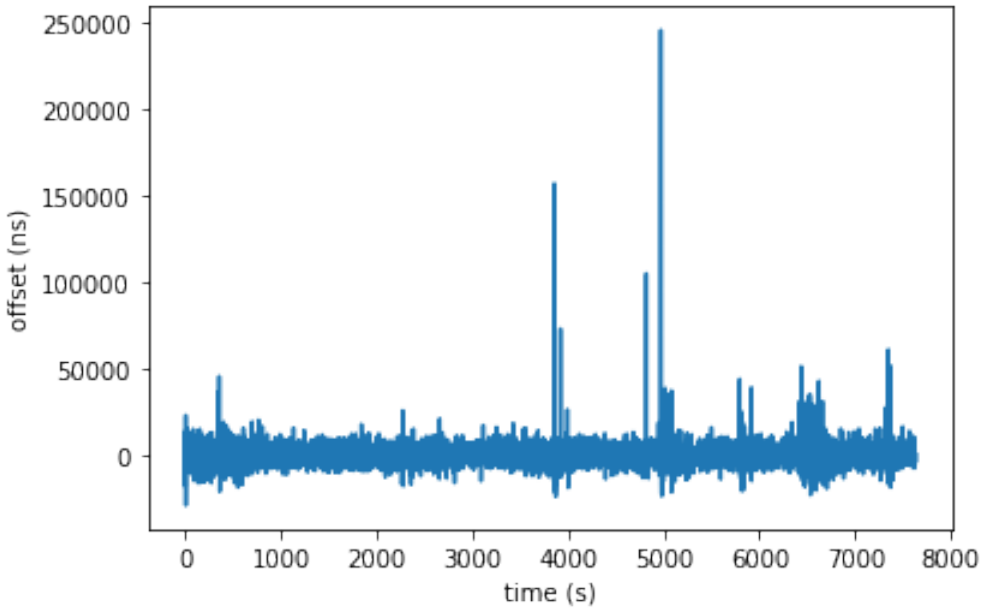

Figure 4.4. Long run of *ptp4l* service, showing offset versus time.

and time appended to the beginning of each line.

The distinction between processor versus network stress was examined in greater and far more specific detail by Huwyler, who showed that central processing unit (CPU) load did not have a substantial impact on the calculated offsets, but high volumes of induced noise in the network did considerably impact performance of the *ptp4l* service [36]. This work, and a deeper look into the impact of network noise that we observed, are discussed in greater detail in Section 4.1.2.

Our comparison of these two approaches to using *ptp4l* kept all other configuration details such as clock servo mechanism and delay measurement constant. For standard out, we found a mean offset of 384 ns, a median of -234 ns, and a mean absolute offset of 3586 ns. For the service, we found a mean offset of -640 ns, a median of -122 ns, and a mean absolute offset of 3442 ns.

We logarithmically normalized both data sets in order to use a consistent benchmark to compare performance between the approaches described in this subsection. This normalization

tells us more about how far the average results are from that calculated mean. As such, the normalized standard deviation of the *ptp4l* service (1.04) illustrates the offsets in that data set were slightly closer to their average than that of the standard out methodology, with a standard deviation of 1.06. We chose the service approach because the standard deviation for the standard out approach was greater than the *ptp4l* service.

Comparing *linreg* vs *pi* Clock Servo Mechanisms

As described by [33], *ptp4l* accepts either of two primary clock servo mechanisms, *linreg* (linear regression) or *pi* (pi constant). The *linreg* mechanism is an adaptive one with reduced configuration options, but higher processing demands. Our results from running a synchronization using *linreg* showed a median offset of -95 ns, a mean offset of -57 ns, and a mean absolute offset of 3583 ns. Our log normalized standard deviation using *linreg* was 1.2. With *pi*, we ascertained a median offset of -1094 ns, a mean offset of -741 ns and a mean absolute offset of 4244 ns. Our log normalized standard deviation with *pi* was 1.12. The 750 ns advantage of *linreg* over *pi* in mean absolute offset values, along with comparable standard deviations, lead us into the discussion of why we chose *linreg* over *pi*.

We selected *linreg* as the operative clock servo for a few reasons. First, based on the aforementioned data that we collected and analyzed, *linreg* did show better performance versus the *pi* constant servo. Second, owing to the lower degree of configuration required and its adaptive nature, *linreg* has additional advantages over the alternative. Finally, as shown by [36], increased processor load was not the prime driver of instability in offset values—network load was. Therefore, the additional processing burden of *linreg* over *pi* was not considered to be a major detriment in our test bed, and the results from our test bed confirmed this conclusion.

Wireshark Data

To gather and examine network data, we utilized the popular packet analyzer Wireshark (v.3.4.9). Figure 4.5 depicts a normal PTP two-step message exchange, as captured by Wireshark. In that figure, the protocol is shown to operate in the manner specified by [11]. The multicast address in the center of the figure, 224.0.1.129, could represent anyone on that network listening to multicast traffic on the given ports (UDP 319 & UDP 320). It is worth highlighting here that of the timestamps needed by the slave to correctly calculate the

offset, only t_1 (time of *sync* transmission) and t_4 (time of *delay_req* receipt) are recovered by Wireshark's packet sniffing. Timestamps t_2 (time of *sync* receipt) and t_3 (time of *delay_req* transmission) are recorded only by the slave. Figure 2.1 refers.

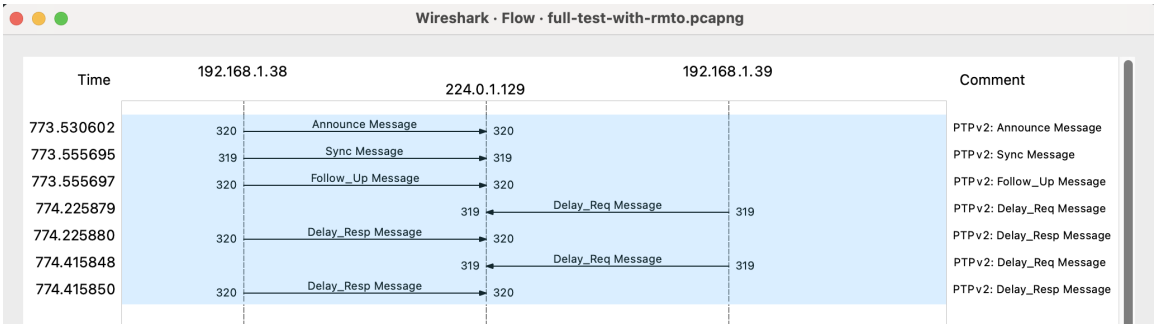

Figure 4.5. Wireshark packet flow diagram depicting a standard PTP message exchange in practice.

Next, we will discuss the mechanics of the command and control in our test bed, using MQTT.

4.1.2 MQTT Control Messaging

Control Overview

The publish-and-subscribe MQTT framework that we used was derived from [34]. On top of the basic publish and subscribe framework of [34], we built core modules in order to enable necessary functionality such as registering button presses on the HMI, or checking the log for PTP offset stability on the RTU. For reference, the Python 3 code for each node can be found in appendices A.6 and A.7.

HMI The HMI uses the MQTT methods of `on_connect` and `on_message` to handle the connections between MQTT publisher and subscriber, and subsequently the messages passing between them. There needed to be a publisher and a subscriber on each node in order to facilitate the bidirectional communication loop. In addition to these two functions, a `buttonHandler` function was created using the *multiprocessing* library in order to allow the HMI to continuously watch for button presses while the main function continues handling

MQTT message traffic. ON and OFF commands were created in hardware by the green and red buttons and lights. Using a Boolean `sendState` variable, new MQTT commands were only sent to the RTU if the state had changed, meaning the opposing button had been pressed. This ON/OFF logic was handled in software, but could have also been implemented in hardware using a toggle switch.

The HMI is the publisher of the "mqtt/commands" topic, which is subscribed to by the RTU. Vice versa, the RTU is the publisher of the "mqtt/updates" topic, which is subscribed to by the HMI. MQTT does not natively support timestamping within the protocol, so it was necessary for us to append the timestamp to the data payload. When a new and valid button press is logged by `buttonHandler`, HMI takes a timestamp and if the button pressed was green, it appends that timestamp to the "on/" payload and publishes it to "mqtt/commands", thereby sending it to the RTU.

RTU Using the same basic publish and subscribe framework described above, the RTU had different demands that required new functions. There is an important distinction in the way the system handles getting and sending a timestamp: all payload in MQTT messages are published as a Python string, however when the timestamp was originally made, it was as a Python `datetime` object. For this reason, we had to add a function that the RTU could use to convert the received timestamp string into a Python `datetime` object, which could then be used to perform calculations such as the transmitted & received timing difference. Using the `split` method, RTU divides the received payload into a command segment and a datetime segment. Then, using methodology described by [37], RTU converts the received timestamp segment back into a Python `datetime` object.

Because our PTP system with software timestamping could not approach nanosecond-level precision of hardware timestamps, we deemed it appropriate to use a conversion method that kept `datetime` objects in terms of microsecond precision. After the conversion, the RTU conducts a simple subtraction of the timestamp found in the master's payload from the timestamp the RTU took when it received the payload. It was important to perform this computation inside the RTU's `on_message` function, because the time difference computation needed to be as instantaneous as possible. Receiving the payload, storing it in a global variable, exiting `on_message` and returning to `main()` all before completing that time difference calculation would incur delay due to movement between functions on the stack.

The validity decision takes advantage of the ongoing synchronization between HMI and RTU according to the arbitrary time decided by the HMI & PTP master. The precise alignment of legitimate nodes along this arbitrary time value is what makes it difficult for Eve, the attacker, to perform FCI in the system under normal conditions. The first validity decision made by the RTU is whether or not the received timestamp is ahead of its own. If so, RTU can automatically discard this message as it is not possible for a valid message to come from its valid master at a future time. The part of the validity decision is whether or not the computed timestamp difference is either greater than or less than or equal to the `validityTolerance`. We will describe the manner in which we determined the appropriate `validityTolerance` in the next section.

After the validity decision is made inside `on_message`, the Boolean global variable `validCmd` is changed accordingly and is passed to `main()`, where the RTU makes its next conditional decision prior to sending any commands to the VFD emulator. In order to ensure the RTU was only receiving valid commands while it was synchronized by the *ptp4l* service running in the background, we created an additional `logWatcher` function that con-tinuously scans through the most recent *ptp4l* service entries in the `/var/log/messages` directory, computes the current average offset, and then determines if the offset values are within the valid range of stability. The three values (in nanoseconds) used by `logWatcher` were determined based on the offset ranges determined by the preceding sections. Nomi-nally, any computed average offset greater than `offsetError = 10000` puts the system into an error state, prohibiting acceptance of new commands and explicitly printing an error to standard out. In between `offsetError = 10000` and `offsetWarning = 5000`, new commands are accepted, but a warning is printed to standard out. If the computed average is between `offsetWarning = 5000` and `offsetOptimal = 3000`, commands are accepted with no warnings printed unless in debug mode. This is the normal operating range for the system, and it is worth noting here that a DoS attack is certainly possible, but was not specifically evaluated in our test bed as there is a large pre-existing body of work reviewing the susceptibility of PTP to DoS attacks. Any computed running averages of the offset that are below `offsetOptimal = 3000` are considered optimal with all valid commands also accepted.

Determining Average Time

In order to determine the average expected time delay between authentic HMI to RTU messages, we modified the control programs to print both payloads, delimited by a `/`. This phase required alternate pressing of each button on the HMI in order to transmit a new payload each time. This process was repeated approximately every second, under normal otherwise operating constraints for the test bed. Using *pandas* dataframes in Google Colab, we parsed the output of the test to include just the two timestamp values for each button press, calculated the difference between slave-received and master-transmitted timestamps, and applied the `mean()` function to compute the average of all differences. The mean difference we calculated was 0.00124s. We then used this value as the basis for the `validityTolerance` variable previously described.

Full Test

We conducted full tests for approximately 25 minutes. The 25 minute time frame intentionally allows for approximately 15 minutes of PTP synchronization time via the *ptp4l* service before the MQTT Python scripts are started on both nodes. The last 10 minutes of the test examines the behavior of the MQTT communications and introduces the MotS attacker, Eve, who uses the attack script in Appendix A.2. The results of those tests are described in Section 4.2.

4.2 Test Analysis

Arbitrary Time

The arbitrary time we chose varied, but was typically within the most recent 24 hour period. The difference between the arbitrary time and the slave's current time is not particularly relevant because, as described in section 4.1.1, clock state `s1` is when the biggest timing adjustments are made by the slave's clock servo. After it enters state `s2`, the timing adjustments are more gradual. Therefore, an arbitrary time that is days or weeks off will be accounted and adjusted for before the slave's clock begins to stabilize according to the offsets it is calculating once locked in `s2`.

Eve has a very narrow window of time that she needs to accurately guess in order to land successful FCIs with Bob. Even if the correct date is known, given the mean difference of

0.00124 s, as calculated in Subsection 4.1.2, Eve has an approximately 0.0000014% chance of guessing correctly. This, based on: 0.00124 s divided by 86400 seconds per day. Even though Eve is able to to watch the multicast UDP network traffic flowing between Alice and Bob, she is missing the two critical values of t_2 and t_3, in order to calculate her own accurate offsets from Alice. This shortcoming in Eve's attack strategy led her to attempt the rogue master takeover, described in Subsection 4.2.1.

Before proceeding, we will show the effect of the use of arbitrary time in the two invalid circumstances of ahead of arbitrary time as well as outside the `validityTolerance` value. Figure 4.6 shows Alice, Bob, and Eve in that order, from left to right. As Bob has been time synchronized to Alice's arbitrary time using *ptp4l*, the commands Alice sends to Bob are validated and are accepted. Of the four valid messages shown on Bob's console, the longest time difference between timestamps was 0.00122 s, which is still inside our `validityTolerance` of 0.00124 s. Next, as shown in the far right console and highlighted in red, Eve attempts to send false commands to Bob. These commands are rejected because they are grossly outside the `validityTolerance` threshold. The calculated time difference of these invalidated commands are around 368532 s. Negative time difference values reflect received timestamps that are ahead of the synchronized arbitrary time.

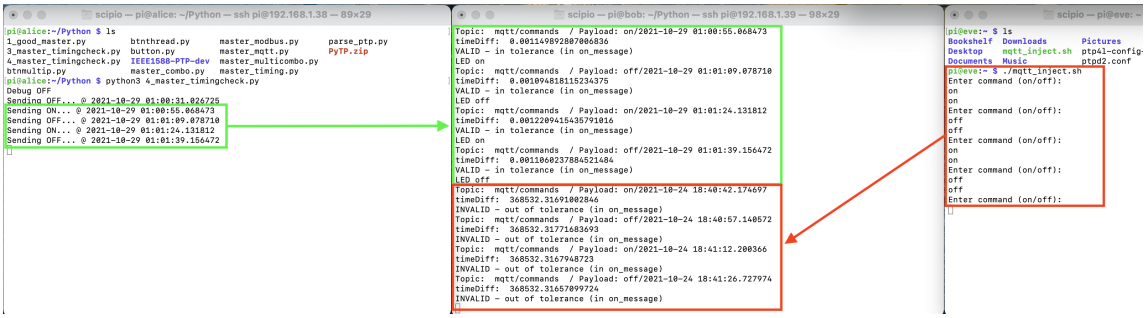

Figure 4.6. Full test output depicting valid messages accepted and invalid messages rejected.

In the next Subsection, we review the results and impact of Eve's rogue master takeover attack.

4.2.1 Rogue Master Takeover

After Eve's FCI attempts were rejected based on timing, Eve initiated a rogue master takeover attack, similar to that described in [14]. Eve's most important considerations for this to be successful were to ensure that delay mechanism and clock servo mechanism were in agreement with what Alice and Bob were already using. Figure 4.7 shows the next phase of Eve's attack. In Figure 4.7 in green we see Alice's valid commands, accepted by Bob. In the same Figure in red we see Eve's successful rogue master takeover, where the top right console window says of Eve "`assuming the grand master role`". Now that she has control, Eve again attempts to transmit her commands to Bob. In Figure 4.7, these commands are also rejected as they are still outside the threshold, though they are much closer to `validityTolerance`. These new commands have a smallest calculated time difference of 0.0261 s.

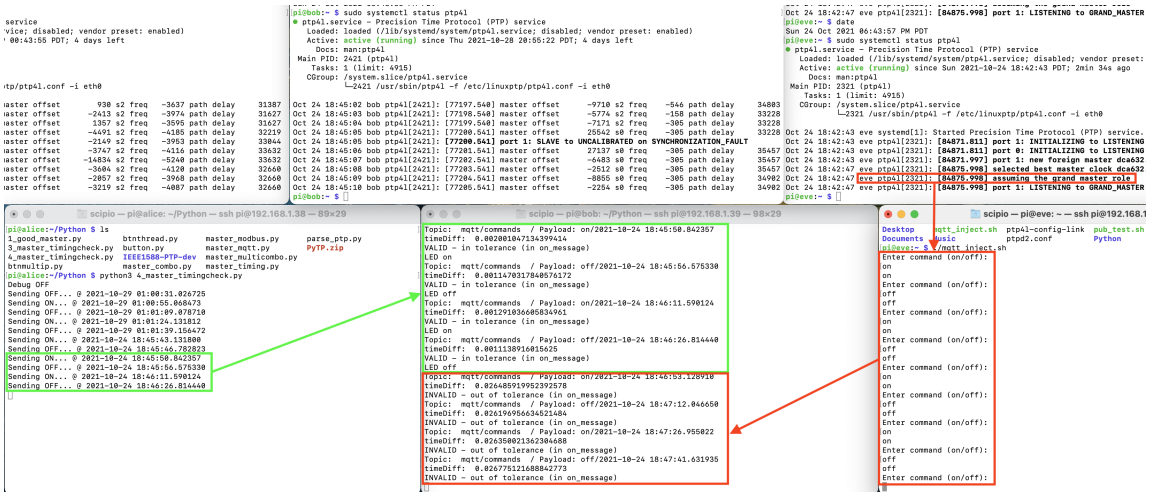

Figure 4.7. Full test output after Eve's rogue master takeover.

This iteration of the test was run without the `logWatcher` capability enabled in order to evaluate the effect of Eve's rogue master takeover as well as the FCI attempts. With `logWatcher`, the rogue master takeover attack grossly disrupts stability of the system and strongly resembles a DoS (though it was not our intention to evaluate DoS here). The effects of that rogue master takeover attack on timing offsets is depicted in Figure 4.8, drawn from Bob's *ptp4l* logfile. In Figure 4.8, time on the x-axis was remapped to the kernel time values represented in the *ptp4l* log, so that specific moments on the line graph could be more easily

referenced in the log. This allowed us to confirm that at kernel time 76808.495 s, Bob's offsets began wildly fluctuating as it entered a re-synchronization period with the new PTP master, Eve.

Additionally, Figure 4.8 has two reference lines drawn across the graph. The overlaid red line represents the `offsetWarning = 5000` and the overlaid yellow line represents `offsetError = 10000`. These lines are used to highlight the potential disruption to the system that the rogue master takeover attack would have caused under normal operating conditions. The normal operating conditions shown are consistent with the testing described in Section 4.1.1, and resulted in running averages well below the `offsetError` conditions and normally inside the `offsetWarning` conditions, until the moment of the takeover.

In conclusion, despite Eve's eventual success in exploiting security shortcomings of PTP (i.e., lack of authentication) by wresting control of the timing system away from Alice, Eve is still not able to inject false commands. The net effect of creating that timing instability as shown in 4.8 at the point of RM Takeover is an action that more closely resembles a DoS, and is counterproductive to Eve's goal of surreptitiously injecting false commands.

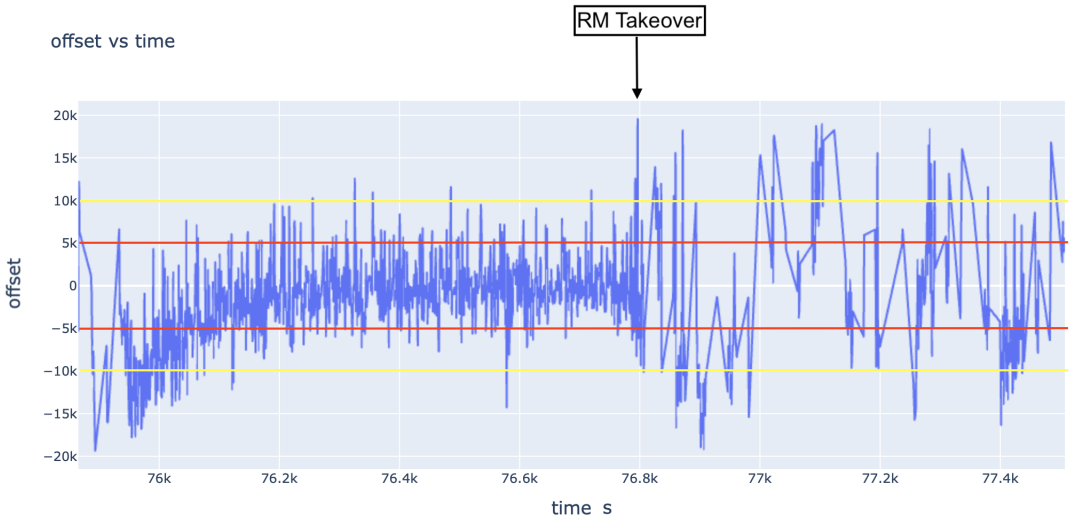

Figure 4.8. Line graph depicting offset versus time for the full test, including the rogue master takeover attack.

Eve successfully took control of timing, but destabilized the system in the process and still had her false commands rejected by Bob.

In the next chapter we discuss these findings further and present conclusions about the results using our PTP test bed.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5:
Conclusion

This chapter discusses the conclusions we reached through our research, the aspects of our work that were successful, the aspects of our work that were unsuccessful, key advantages of our approach, and finally the limitations to our approach.

5.1 Discussion

In our OT test bed, we have demonstrated that timing can be a useful aspect of security. By mitigating certain kinds of attacks through the use of an arbitrary time value, we were able to reject certain FCI attempts by the MotS attacker. This is an intentionally narrow focus on a small subset of the broader category of kinds of MotS attacks, and there are still many ways for a MotS to circumvent or disrupt the system entirely (e.g. disruption via rogue master takeover or an explicit DoS). Limitations in our work that are discussed here highlight the necessity for a breadth of security approaches to safeguard an OT system. In physical security, a door handle lock is probably not sufficient to prevent a burglar or home invader. Instead, a broad array of approaches such as deadbolts or security cameras in addition to the handle lock offer more comprehensive and reliable mitigation of malicious activity. For OT, timing can assist in mitigating certain attacks, but the reliance on that accurate timing for core OT reliability functions may make its application to security unappealing.

In the following subsections, we take a deeper look into what did and did not work in our test bed.

5.1.1 What Worked

PTP software timestamp offset stabilization was predictable and consistent across most of our applications in the test bed. Additionally, our master offset values were notably smaller than similar COTS demonstrations by [38] and were on par with that of [36]. Figure 5.1 shows an overlay of two horizontal green lines, on top of Figure 5.9 from [38], depicting the typical master offset ranges we received in our testbed versus a similar COTS implementation in [38].

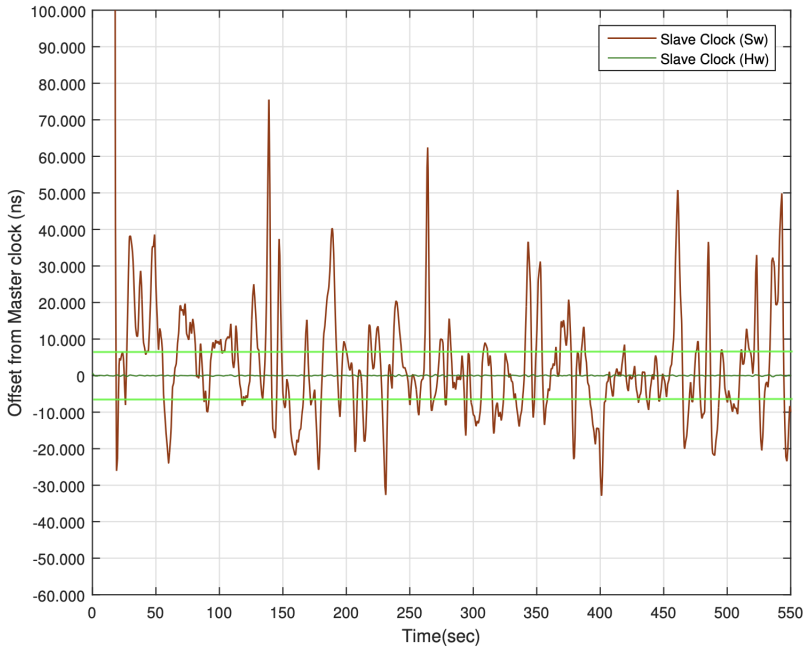

Figure 5.1. Hardware versus software timestamps from Ahmed (2018), overlaid with our typical offset range given by the bright green horizontal lines. Adapted from: [38, figure 5.9].

We were able to synchronize the operation of our testbed system and tie its functionality to tolerable ranges of offset values from the master. The MQTT command and control proved to be an effective and simple way of handling message publication to constituent nodes in our OT test bed. MQTT was highly modular in implementation, allowing for the addition of missing features, such as native timestamping, directly into the message payload.

Using arbitrary time, we were able to explicitly reject certain FCI attempts. There were clear windows in which Eve was not able to inject false commands, at least until execution of the rogue master takeover.

We demonstrated the effectiveness of the rogue master attack, accentuating the importance of bringing the *linuxptp* package into alignment with [13] and [14].

5.1.2 What Didn't Work

The additional network noise generated by the consistent updates from the slave node caused noticeable and detrimental effects in our test bed. This is admittedly a difficult challenge to address, as updates from the slave are necessary. For example, being able to regularly retrieve the true current operating frequency of a VFD is critical to safe and reliable operation. The system should strike a balance between frequency of update messaging and stress on the network. In order to prevent disruption to our system, we ceased the MQTT message updating from the slave to the master. Creating a separate `multiprocessing` instance to handle message updates at shorter intervals might strike a better balance. Slowing the update rate inside `main()` via a longer `time.sleep()` value would disrupt the processing of inbound messages and the transmission of outbound messages. Further, it would introduce greater probability that the master might miss messages from the slave during those `time.sleep()` periods.

5.2 Advantages

The system we utilized has a few key advantages. First, the COTS approach enabled quick acquisition, setup, and deployment of a very reliable PTP-based synchronization and MQTT-based command and control network.

Additionally, both the PTP implementation we used via *linuxptp* as well as the MQTT implementation via *mosquitto/paho-mqtt* utilize UDP multicast. This affords an additional advantage of being highly scalable. Although we did not implement additional slave nodes, doing so would be very straightforward. This multicast approach worked well in our network, primarily because there were only two primary types of network traffic: PTP and MQTT. In an implementation on a consolidated network with additional types of traffic (i.e. mail, web, etc.), this multicast might not be a tenable solution.

Finally, the nature of the PTP communication algorithm, whereby only two of the four necessary timestamps are actually transmitted, was an advantage to our ends. In order to get the same level of precision as the legitimate slave node, the MotS needs to ascertain those two other timestamps. In the next chapter, we will discuss a potential application of emerging technology to improve the attack.

5.3 Limitations

In testing with this configuration, using only the software timestamping on the Raspberry Pi's, PTP offset stabilization could take up to 15 minutes. The work of [39] has indicated this cold start can be in excess of 20 minutes. This latency could have serious implications in time-sensitive systems, especially ones like a shipboard ICS that requires fast and reliable on/off system control. In initial testing, the 10-15 minute synchronization was on cold start and subsequent synchronizations did not take the same length of time to stabilize. Subsequent trials were synchronized to a similar average offset as the initial cold start from the moment the clock servo locked in state `s2` .

As discussed in Section 2.3.2, use of this arbitrary time premise could prohibit the use of certain security mechanisms that rely on an accurate and true time of day. For example, some systems gather their PTP timing from a global navigation satellite system (GNSS) valid time source. In this kind of system, it isn't practical to use an arbitrary time value in PTP without first disconnecting the GNSS time source, as GNSS is a higher clock priority than the local oscillator used by a node like a Raspberry Pi [13]. Additionally, some other OT systems may require the true time of day for reliable operations. For example, an assembly line at a car manufacturer might initialize certain systems an hour before workers arrive for the first shift.

Next, management of the system via SSH incurs additional network load which could potentially be detrimental, depending on what other operations are occurring over that remote management connection.

Finally, despite initial intentions to not specifically examine DoS, we confirmed that the system is very susceptible to a DoS-style attack. The rogue master takeover's induction of bad timing offsets in a full test resulted in locking up the system in the error state. Preventing this level of system seizure due to distortion of the running offsets calculated by the RTU is challenging with *linuxptp*, because entries are only written to the log approximately every second. Computing a more accurate running offset and simultaneously reducing the impact of average-distorting false offsets would require more frequent logging.

In the final chapter, we discuss potential areas for future work, informed by the considerations we have reviewed in this chapter.

CHAPTER 6:
Future Work

Finally, in this chapter we discuss future work in the OT and precision timing space.

6.1 Test Bed Improvements

The first potential for future work and improvement we will discuss relates to our own OT test bed.

Hardware Timestamping The first and clearest recommendation for future work in this domain is to leverage hardware timestamps. Reducing the offsets by an order of magnitude through hardware timestamping is very useful and [38] provided a clear visual representation of the difference in precision afforded by this design choice. The *linuxptp* software suite is very powerful and regularly updated, routinely bringing necessary improvements and continuing to accommodate the use of hardware timestamps.

The Real-Time Hat, by InnoRoute, is a "hat" that interfaces with Raspberry Pi units and offers the NIC-level interfacing with the PTP protocol powered by *linuxptp*. Hats, or other COTS equipment with NICs that support IEEE1588 could be utilized instead of a Raspberry Pi. Additionally, the use of network equipment that specifically supports IEEE1588, such as some produced by Cisco, can further improve timing accuracy of the system. Switches that specifically support IEEE1588 can increase accuracy by acting as a PTP boundary clock. The IEEE1588-compliant switches can then timestamp the PTP messages at the hardware layer, at the moment they are transmitted, further reducing inaccuracies from processing and queuing [40].

Straying from the pure COTS approach, there are still other methods of precise timing, including some used by renowned international research organizations. For example White Rabbit, a field-programmable gate array-based timing solution used by the researchers at CERN, "achieves sub-nanosecond accuracy in Ethernet based networks" and can be integrated into existing PTP implementations [41]. Although described as cost-effective, this level of precision may not be necessary in most OT systems and could even be overkill for

security-centric applications. White Rabbit is worth highlighting, though, for its commercial availability and its capability to achieve picosecond-level accuracy in existing PTP systems.

These hardware-based approaches could be examined individually, or could be a piece of another possible approach which is the use of a separate channel for the timing network. This would involve another parallel network connected to a secondary ethernet interface on the nodes. For example, in this implementation *eth0* could be connected to the PTP timing network, while *eth1* is connected to the MQTT control network. There is potential to incur processing overhead, but based on the findings of [36] in regards to processing load versus network load, that overhead might be negligible compared to security and stability advantages afforded by a separate physical channel.

Security Security improvements to the PTP side of the test bed could implement recommendations from [14]. Broadly, [14] refers to the use of IPSec or MACsec as a starting point for a security requirement. The first and most significant requirement outlined in the RFC was the "must" of clock identity authentication and authorization [14]. The proper security mechanisms that support both authentication and authorization can have positive material impacts in mitigating an attack like the rogue master we demonstrated.

As described in [14], authentication of slaves can clarify in the network which nodes are authorized to receive the timing information sent by the master. Additionally, proper authentication of slaves can mitigate DoS attacks against the master [14]. Spoofing prevention is a "must" described by [14] as spoofing can be very detrimental to the system. An unmitigated spoofed master (distinct from the rogue master we evaluated here) usually presents itself as a PTP transparent clock and can transmit false timing data to the slaves in the network.

For the MQTT side of the test bed, *mosquitto* and *paho-mqtt* support TLS for security within the newest version of MQTT protocol. Using encryption in the command and control messaging could mitigate both MotS and potentially MitM attacks against the slave (and/or master) node. It is worth noting that encryption and decryption of payloads could add additional latency to the system. The level of impact encrypting payloads might have on timing latency could also be specifically evaluated in future work.

Arbitrary Time Although we have previously discussed the limitations of using arbitrary time within systems that can tolerate its application, there is potential to leverage arbitrary

time in more meaningful ways. For example, by introducing a cryptographic aspect to the selection of the arbitrary time by the master, we could improve resilience against a MotS attack. This would subject the PTP system to the same challenges discussed in Chapter 1. Specifically, challenges surrounding key storage and distribution, highlighted by [7], reemerge when considering a cryptographic approach to the arbitrary time premise.

6.2 MitM

Part of the original proposal of this research was to evaluate the capacity of precise timing to determine whether or not a command had been altered en route. This premise focused on a true MitM, whereby the additional processing delay caused by the changes made by the MitM were measurable due to such a high degree of precision in timing.

This preliminary objective was not practically addressable here given our use of software timestamps. However, addressing this problem with an extremely high degree of precision using hardware timestamping at the lowest possible layer of the protocol stack might prove effective. There is additional potential inside this MitM exploration to leverage artificial intelligence (AI) and machine learning (ML) in concert with precise hardware-level timestamping to measure and evaluate processing delays added by a MitM. As mentioned in Chapter 5, there is potential to apply emerging technologies such as AI and ML to aid an attacker's traffic analysis, possibly providing a better way to accurately guess or infer the missing timestamps t_2 and t_3.

6.3 Summary

We have presented the assembly of an OT system with a modular and scalable PTP-synchronized command and control protocol that is able to leverage precise timing in order to mitigate a specific kind of MotS attack. Though there are still many vulnerabilities in the system, the fusion of different security strategies in concert with timing may make approaches like this more viable in other OT implementations.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX: Configuration Files, Code, and Scripts

A.1 Arduino Code

A.1.1 RS-485 VFD Emulator

```
/*Derived from Arduino Sketch example code found in Arduino IDE
RS-485 ASCII (vice modbus) via Arduino:
https://forum.arduino.cc/t/modbus-rs485-to-gs1-vfd/400756
*/

int enablePin = 2;
const int ledPin =  13;      // the number of the LED pin
int ledState = LOW;         // the current state of the output pin
long int on = 1;            //establish value 1 as on
long int off = 0;           //establish value 0 as off

void setup(){
  Serial.begin(19200);            // initialize serial at baudrate 9600:
  pinMode(enablePin, OUTPUT);
  delay(10);
  digitalWrite(enablePin, LOW);    //(Pin2 LOW to receive value from Master)
  pinMode(ledPin, OUTPUT);         //set pinMode for the LED as output
  digitalWrite(ledPin, ledState);  //set initial state (LOW)
}

void loop(){
  while (Serial.available()){   //While Serial data has arrived
  long int command = Serial.parseInt(); //Receive INTEGER value from Master
                                        // through RS-485
    Serial.println(command);   //Print received & parsed cmd to monitor
    if (command == on){
      Serial.println("On");
      //ledState != ledState;       //change the led state
```

```
      digitalWrite(ledPin, HIGH);   //
      //delay(100);
      //digitalWrite(ledPin, LOW);
    } if (command == off){
      Serial.println("Off");
      digitalWrite(ledPin, LOW);
    }
    //digitalWrite(ledPin, ledState);     //write the new value to the led
  }
}
```

A.1.2 Emergency Manual to Override Push Button

```
/*Derived from Arduino Sketch example code found in Arduino IDE and
SparkFun Qwiic example:
https://learn.sparkfun.com/tutorials/qwiic-single-relay-hookup-guide/all

RS-485 ASCII (vice modbus) via Arduino:
https://forum.arduino.cc/t/modbus-rs485-to-gs1-vfd/400756

This implementation is set for a single relay (RELAY_A).
RELAY_B is commented out
*/

//RELAY:
#include <Wire.h>
#include "SparkFun_Qwiic_Relay.h"
#define RELAY_A_ADDR 0x18
//#define RELAY_B_ADDR 0x19
Qwiic_Relay relayA(RELAY_A_ADDR);
//Qwiic_Relay relayB(RELAY_B_ADDR);

//SERVO:
#include <Servo.h>
Servo myservo;  // create servo object to control a servo
```

```
// twelve servo objects can be created on most boards
int start = 0;    // variable to store the servo position
int finish = 90;

//BUTTONS:
// constants won't change. They're used here to set pin numbers:
const int buttonPin = 2;     // the number of the pushbutton pin
const int ledPin =  13;      // the number of the LED pin
// Variables will change:
int ledState = LOW;         // the current state of the output pin
int buttonState;             // the current reading from the input pin
int lastButtonState = LOW;   // the previous reading from the input pin
// the following variables are unsigned longs because the time, measured in
// milliseconds, will quickly become a bigger number than can be stored in an int.
unsigned long lastDebounceTime = 0;  // the last time the output pin was toggled
unsigned long debounceDelay = 50;  // debounce time; increase if output flickers

//RS485:
#define MASTER_EN 8   //connected to RS485 enable pin (currently both DE & RE)

// the setup routine runs once when you press reset:
void setup() {
  relayA.turnRelayOff();

  myservo.attach(9);  // attaches the servo on pin 9 to the servo object

  // initialize the LED pin as an output:
  pinMode(ledPin, OUTPUT);
  // initialize the pushbutton pin as an input:
  pinMode(buttonPin, INPUT);
  // set initial LED state
  digitalWrite(ledPin, ledState);

  // initialize serial communication at 19200 (formerly 9600) bits per second:
  Serial.begin(19200);
```

```
  //RS485 Setup:
  pinMode(MASTER_EN, OUTPUT);     //Declare Enable pin as output (to Slave)
  digitalWrite(MASTER_EN, LOW);   //make RE/DE enable pin LOW
}

// the loop routine runs over and over again forever:
void loop() {


  // read the state of the switch into a local variable:
  int reading = digitalRead(buttonPin);

  // check to see if you just pressed the button
  // (i.e. the input went from LOW to HIGH), and you've waited long enough
  // since the last press to ignore any noise:

  // If the switch changed, due to noise or pressing:
  if (reading != lastButtonState) {
    // reset the debouncing timer
    lastDebounceTime = millis();
  }

  if ((millis() - lastDebounceTime) > debounceDelay) {
    // whatever the reading is at, it's been there for longer than the debounce
    // delay, so take it as the actual current state:

    // if the button state has changed:
    if (reading != buttonState) {
      buttonState = reading;

      // only toggle the LED if the new button state is HIGH
      if (buttonState == HIGH) {
        ledState = !ledState;
      }
```

```
  }
}

// set the LED:
digitalWrite(ledPin, ledState);

if (ledState == HIGH) {
  //relayA.turnRelayOn();
  //set the servo
  //myservo.write(finish);    // tell servo to go to position in variable 'pos'
  relayA.turnRelayOn();
  delay(15);                  // waits 15ms for the servo to reach the position
} else {
    relayA.turnRelayOff();
    //myservo.write(start);   // tell servo to go to start position
    delay(15);                // waits 15ms for the servo to reach the position
    //relayA.toggleRelay();
}

// save the reading. Next time through the loop, it'll be the lastButtonState:
lastButtonState = reading;

//SERIAL PRINT TO SLAVE VIA RS-485:
//digitalWrite(MASTER_EN, HIGH); //Make enable pin high to send data to slave
//delay(5);                     //required minimum delay of 5ms?

//SERIAL READ:
// read the input on analog pin 0:
int sensorValue = analogRead(A0);
// print out the value you read:
Serial.println(sensorValue);
Serial.flush();
//delay(500);         // delay in between reads for stability
//delay(1000);    //From 485MASTER sketch
```

```
  //RS485 Receive mode enable
  //digitalWrite(MASTER_EN, LOW);         //Receive mode back on

}
```

A.2 Eve Attack Bash Script

```
#!/bin/bash
# mqtt_inject.sh
# AUTHOR: LT CHARLES ALLEN, USN
# mqtt_inject.sh accepts user input for command (on or off), packages
# it into the message format, and transmits to the target IP
# using mosquitto_pub.

TARGET_IP="192.168.1.39"
TARGET_TOPIC="mqtt/commands"

#Perpetual Loop:
while true; do

echo "Enter command (on/off): "
read COMMAND
echo "${COMMAND}"

# TIME IN NANOSECONDS:
#NOW=‘date +"%Y-%m-%d %T.%N"‘
# TIME IN MILLISECONDS:
#NOW=‘date +"%Y-%m-%d %T.%3N"‘
# TIME IN MICROSECONDS:
NOW=‘date +"%Y-%m-%d %T.%6N"‘

#echo "${COMMAND}/${NOW}"

mosquitto_pub -h ${TARGET_IP} -t ${TARGET_TOPIC} -m "${COMMAND}/${NOW}"
```

```
done
```

A.3 Alice ptp4l.conf

```
[global]
#
# Default Data Set
#
twoStepFlag 1
slaveOnly 0
priority1 128
priority2 128
domainNumber 0
#utc_offset 37
clockClass 248
clockAccuracy 0xFE
offsetScaledLogVariance 0xFFFF
free_running 0
freq_est_interval 1
dscp_event 0
dscp_general 0
#
# Port Data Set
#
logAnnounceInterval 1
logSyncInterval 0
logMinDelayReqInterval 0
logMinPdelayReqInterval 0
announceReceiptTimeout 3
syncReceiptTimeout 0
delayAsymmetry 0
fault_reset_interval 4
neighborPropDelayThresh 20000000
#
```

```
# Run time options
# - logging_level 7 for all available messages
assume_two_step 1
logging_level 6
path_trace_enabled 0
follow_up_info 0
hybrid_e2e 0
net_sync_monitor 0
tx_timestamp_timeout 1
use_syslog 1
verbose 0
summary_interval 0
kernel_leap 1
check_fup_sync 0
#
# Servo Options
#
pi_proportional_const 0.0
pi_integral_const 0.0
pi_proportional_scale 0.0
pi_proportional_exponent -0.3
pi_proportional_norm_max 0.7
pi_integral_scale 0.0
pi_integral_exponent 0.4
pi_integral_norm_max 0.3
step_threshold 0.0
first_step_threshold 0.00002
max_frequency 900000000
#clock_servo pi
clock_servo linreg
sanity_freq_limit 200000000
ntpshm_segment 0
#
# Transport options
#
```

```
transportSpecific 0x0
ptp_dst_mac 01:1B:19:00:00:00
p2p_dst_mac 01:80:C2:00:00:0E
udp_ttl 1
udp6_scope 0x0E
uds_address /var/run/ptp4l
#
# Default interface options
#
network_transport UDPv4
delay_mechanism E2E
time_stamping software
tsproc_mode filter
delay_filter moving_median
delay_filter_length 10
egressLatency 0
ingressLatency 0
boundary_clock_jbod 0
#
# Clock description
#
productDescription ;;
revisionData ;;
manufacturerIdentity 00:00:00
userDescription ;
timeSource 0xA0

[eth0]
```

A.4 Bob ptp4l.conf

```
[global]
#
# Default Data Set
#
```

```
twoStepFlag 1
slaveOnly 1
priority1 128
priority2 128
domainNumber 0
#utc_offset 37
clockClass 248
clockAccuracy 0xFE
offsetScaledLogVariance 0xFFFF
free_running 0
freq_est_interval 1
dscp_event 0
dscp_general 0
#
# Port Data Set
#
logAnnounceInterval 1
logSyncInterval 0
logMinDelayReqInterval 0
logMinPdelayReqInterval 0
announceReceiptTimeout 3
syncReceiptTimeout 0
delayAsymmetry 0
fault_reset_interval 4
neighborPropDelayThresh 20000000
#
# Run time options
#
assume_two_step 1
#assume_two_step 0
logging_level 6
path_trace_enabled 0
follow_up_info 0
hybrid_e2e 0
net_sync_monitor 0
```

```
tx_timestamp_timeout 1
use_syslog 1
verbose 0
summary_interval 0
kernel_leap 1
check_fup_sync 0
#
# Servo Options
#
pi_proportional_const 0.0
pi_integral_const 0.0
pi_proportional_scale 0.0
pi_proportional_exponent -0.3
pi_proportional_norm_max 0.7
pi_integral_scale 0.0
pi_integral_exponent 0.4
pi_integral_norm_max 0.3
step_threshold 0.00002
#step_threshold 0.0
first_step_threshold 0.00002
max_frequency 900000000
#clock_servo pi
clock_servo linreg
sanity_freq_limit 200000000
ntpshm_segment 0
#
# Transport options
#
transportSpecific 0x0
ptp_dst_mac 01:1B:19:00:00:00
p2p_dst_mac 01:80:C2:00:00:0E
udp_ttl 1
udp6_scope 0x0E
uds_address /var/run/ptp4l
#
```

```
# Default interface options
#
network_transport UDPv4
delay_mechanism E2E
time_stamping software
tsproc_mode filter
delay_filter moving_median
delay_filter_length 10
egressLatency 0
ingressLatency 0
boundary_clock_jbod 0
#
# Clock description
#
productDescription ;;
revisionData ;;
manufacturerIdentity 00:00:00
userDescription ;
timeSource 0xA0


[eth0]
```

A.5 Eve ptp4l.conf

```
[global]
#
# Default Data Set
#
twoStepFlag 1
slaveOnly 0
#priority1              125
priority1 128
#priority2 125
```

```
priority2 128
domainNumber 0
#utc_offset 37
clockClass 248
clockAccuracy 0xFE
offsetScaledLogVariance 0xFFFF
free_running 0
freq_est_interval 1
dscp_event 0
dscp_general 0
#
# Port Data Set
#
logAnnounceInterval 1
logSyncInterval 0
logMinDelayReqInterval 0
logMinPdelayReqInterval 0
announceReceiptTimeout 3
syncReceiptTimeout 0
delayAsymmetry 0
fault_reset_interval 4
neighborPropDelayThresh 20000000
#
# Run time options
#
assume_two_step 0
logging_level 6
path_trace_enabled 0
follow_up_info 0
hybrid_e2e 0
net_sync_monitor 0
tx_timestamp_timeout 1
use_syslog 1
verbose 0
summary_interval 0
```

```
kernel_leap 1
check_fup_sync 0
#
# Servo Options
#
pi_proportional_const 0.0
pi_integral_const 0.0
pi_proportional_scale 0.0
pi_proportional_exponent -0.3
pi_proportional_norm_max 0.7
pi_integral_scale 0.0
pi_integral_exponent 0.4
pi_integral_norm_max 0.3
step_threshold 0.0
first_step_threshold 0.00002
max_frequency 900000000
clock_servo linreg
#clock_servo pi
sanity_freq_limit 200000000
ntpshm_segment 0
#
# Transport options
#
transportSpecific 0x0
ptp_dst_mac 01:1B:19:00:00:00
p2p_dst_mac 01:80:C2:00:00:0E
udp_ttl 1
udp6_scope 0x0E
uds_address /var/run/ptp4l
#
# Default interface options
#
network_transport UDPv4
delay_mechanism E2E
time_stamping software
```

```
tsproc_mode filter
delay_filter moving_median
delay_filter_length 10
egressLatency 0
ingressLatency 0
boundary_clock_jbod 0
#
# Clock description
#
productDescription ;;
revisionData ;;
manufacturerIdentity 00:00:00
userDescription ;
timeSource 0xA0

[eth0]
```

A.6 Alice/HMI Python Script

https://gitlab.nps.edu/charles.allen/allen-thesis-ptp-security.git

A.7 Bob/RTU Python Script

https://gitlab.nps.edu/charles.allen/allen-thesis-ptp-security.git

A.8 Data Visualization Scripts

```
import pandas as pd
import numpy as np
from scipy import stats
import matplotlib.pyplot as plt
import plotly.express as px
import csv
import statistics
```

```
#Create lists to hold key data from logfile
time = []             #create list of time values
offset = []           #create list of offset values
freq = []             #create list of freq values
path_delay = []       #create list of path_delay values
state = []            #create list of state values

#Read in and parse the logfiles for key data
with open('linreg-slave.txt', 'r') as f:        #open the ptp4l output .txt file
  lines = f.readlines()                         #read in each line of the file
  for line in lines:                            #begin to parse through each line
    line = " ".join(line.split())
    cont = line.split(' ')                      #split each line on space delimiter
    if 'offset' in line and (cont[9] == 's2'):  #parse lines for just those that
                                                #   contain "offset" and "s2"
      if float(cont[8]) != 0:                   #checking for offset values that
                                                #    are not = 0
        time.append(float(cont[5][1:-1]))       #add time value to time list
        offset.append(float(cont[8]))           #add offsest value to offset list
        #state.append(cont[9])                  #add state value to state list
        freq.append(int(cont[11]))              #add freq value to freq list
        path_delay.append(float(cont[14]))      #add path_delay to path_delay list

#Housekeeping data manipulations
time = [i - time[0] for i in time]    #Map time values from kernel time to 0 base
offset = np.asarray(offset)           #make offset list a numpy array
offset = np.absolute(offset)          #use np to take absolute values of offset
offset = np.log(offset)               #use np to take the log of each value in list
print(len(offset))                    #print the length of the offset list

#Printing basic statistics:
print("Average absOffset: ", np.mean(offset))
print("Standard Deviation: ", np.std(offset))
print("Median: ",np.median(offset))
```

```
#Z-scores & ourliers
z = np.asarray(stats.zscore(offset))   #set up z to handle z-score computations
threshold = 3                          #set z-score threshold = 3
outliers = np.where(z > 2.35)          #create outliers list, adding those
                                       #    outside the specified value of z
print(len(outliers[0]))                #print number of outliers
for o in outliers[0]:                  #print the outlier value and the time
                                       #    of the outlying data point
  print(time[o],offset[o])             # -> this might involve commenting out
                                       #    the mapping of time value to 0 in
                                       #    order to check against the log file

#averageOffset = Average(offset)
#print("Average offset: ", averageOffset)
#offsetMed = statistics.median(offset)
#print("Median Offset", offsetMed)

print("")

#Display Plot/Histogram
#plt.plot(time,offset)
#plt.show()
plt.hist(offset, bins='auto')
plt.show()
```

THIS PAGE INTENTIONALLY LEFT BLANK

List of References

[1] J. Guan, J. H. Graham, and J. L. Hieb, "A digraph model for risk identification and mangement in SCADA systems," in *Proceedings of 2011 IEEE International Conference on Intelligence and Security Informatics*. IEEE, 2011, pp. 150–155.

[2] B. Galloway and G. P. Hancke, "Introduction to industrial control networks," *IEEE Communications surveys and tutorials*, vol. 15, no. 2, pp. 860–880, 2013.

[3] R. Lee, M. Assante, and T. Conway, "Analysis of the cyber attack on the ukrainian power grid: Defense use case," SANS Institute, Bethesda, MD, USA, 2016. [Online]. Available: https://nsarchive.gwu.edu/sites/default/files/documents/3891751/SANS-and-Electricity-Information-Sharing-and.pdf

[4] Y. Cherdantseva, P. Burnap, A. Blyth, P. Eden, K. Jones, H. Soulsby, and K. Stoddart, "A review of cyber security risk assessment methods for SCADA systems," *Computers & security*, vol. 56, pp. 1–27, 2016.

[5] D. Fauri, B. de Wijs, J. den Hartog, E. Costante, E. Zambon, and S. Etalle, "Encryption in ICS networks: A blessing or a curse?" in *2017 IEEE International Conference on Smart Grid Communications (SmartGridComm)*. IEEE, 2017, pp. 289–294.

[6] W. Turton, M. Riley, and J. Jacobs, "Colonial pipeline paid hackers nearly $5 million in ransom," May 13, 2021. [Online]. Available: https://www.bloomberg.com/news/articles/2021-05-13/colonial-pipeline-paid-hackers-nearly-5-million-in-ransom/

[7] N. Rao, S. Srivastava, and S. K.S, "PKI deployment challenges and recommendations for ICS networks," *International journal of information security and privacy*, vol. 11, no. 2, pp. 38–48, 2017.

[8] W. Broad and D. Sanger, "Worm was perfect for sabotaging centrifuges," Nov. 18, 2010. [Online]. Available: https://www.nytimes.com/2010/11/19/world/middleeast/19stuxnet.html/

[9] D. Mills, J. Martin, J. Burbank, and W. Kasch, "Network Time Protocol Version 4: Protocol and Algorithms Specification," Internet Requests for Comments, IETF, RFC 5905, June 2010. Available: https://datatracker.ietf.org/doc/html/rfc5905

[10] J. Eidson, "IEEE-1588 standard for a precision clock synchronization protocol for networked measurement and control systems: A tutorial," Agilent Technologies, Oct. 10, 2005. [Online]. Available: https://www.nist.gov/system/files/documents/el/isd/ieee/tutorial-basic.pdf

[11] "TC-9", “IEEE standard for a precision clock synchronization protocol for networked measurement and control systems,” *"IEEE" Std 1588-2008 (Revision of IEEE Std 1588-2002)*, pp. 1–269, 2008. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4579760

[12] R. Cochran, *linuxptp*, ver. 2, 2021. [Online]. Available: https://github.com/richardcochran/linuxptp

[13] *"1588-2019 - IEEE standard for a precision clock synchronization protocol for networked measurement and control systems"*. IEEE, 2020.

[14] T. Mizrahi, “Security requirements of time protocols in packet switched networks,” Internet Requests for Comments, IETF, RFC 7384, October 2014. Available: https://datatracker.ietf.org/doc/html/rfc7384

[15] T. Rid and B. Buchanan, “Attributing cyber attacks,” *Journal of Strategic Studies*, vol. 38, no. 1-2, pp. 4–37, 2015.

[16] N. Saxena, L. Xiong, V. Chukwuka, and S. Grijalva, “Impact evaluation of malicious control commands in cyber-physical smart grids,” *IEEE Transactions on Sustainable Computing*, vol. 6, no. 2, pp. 208–220, 2021.

[17] I. Allahi, B. Khan, A. S. Nagra, R. Idrees, and S. Masud, “Performance evaluation of IEEE 1588 protocol using raspberry pi over WLAN,” in *2018 IEEE International Conference on Communication Systems (ICCS)*. IEEE, 2018, pp. 315–320.

[18] W. Alghamdi and M. Schukat, “Precision time protocol attack strategies and their resistance to existing security extensions,” *Cybersecurity*, vol. 4, no. 1, pp. 1–17, 2021.

[19] C. DeCusatis, R. M. Lynch, W. Kluge, J. Houston, P. A. Wojciak, and S. Guendert, “Impact of cyberattacks on precision time protocol,” *IEEE Transactions on Instrumentation and Measurement*, vol. 69, no. 5, pp. 2172–2181, 2020.

[20] B. Moussa, M. Debbabi, and C. Assi, “A detection and mitigation model for PTP delay attack in an IEC 61850 substation,” *IEEE transactions on smart grid*, vol. 9, no. 5, pp. 3954–3965, 2018.

[21] B. Moussa, C. Robillard, A. Zugenmaier, M. Kassouf, M. Debbabi, and C. Assi, “Securing the precision time protocol (PTP) against fake timestamps,” *IEEE communications letters*, vol. 23, no. 2, pp. 278–281, 2019.

[22] E. Itkin and A. Wool, “A security analysis and revised security extension for the precision time protocol,” *IEEE transactions on dependable and secure computing*, vol. 17, no. 1, pp. 22–34, 2020.

[23] R. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 26, no. 1, pp. 96–99, 1983.

[24] *GS1 Series Drives User Manual*, 2nd ed., Automation Direct, Cumming, GA, 2011, pp. 3–6–3–8.

[25] AutomationDirect.com. "CLICK PLC - How To Control a 3-Phase AC Motor Using a GS1 Drive and a CLICK PLC" Jul. 26, 2013. [YouTube video]. Available: https://www.youtube.com/watch?v=3ZpFqL1200I

[26] ECM Web. "The basics of ladder logic" Dec. 1, 2003. [Online]. Available: https://www.ecmweb.com/archive/article/20891380/the-basics-of-ladder-logic

[27] R. Coppen, *MQTT Version 5.0: OASIS Standard*, ver. 5, 2019. [Online]. Available: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html

[28] S. Kreuzer, G. Neville-Neil, and W. Owczarek, *ptpd*, ver. 2, 2019. [Online]. Available: https://github.com/ptpd/ptpd

[29] V. Aggarwal, *IEEE1588-PTP*, ver. 2, 2021. [Online]. Available: https://github.com/bestvibes/IEEE1588-PTP

[30] R. Cochran and C. Marinescu, "Design and implementation of a PTP clock infrastructure for the Linux kernel," in *2010 IEEE International Symposium on Precision Clock Synchronization for Measurement, Control and Communication*. IEEE, 2010, pp. 116–121.

[31] R. Cochran, C. Marinescu, and C. Riesch, "Synchronizing the Linux system time to a PTP hardware clock," in *2011 IEEE International Symposium on Precision Clock Synchronization for Measurement, Control and Communication*. IEEE, 2011, pp. 87–92.

[32] *CLICK PLC User Manual*, 6th ed., Automation Direct, Cumming, GA, 2021.

[33] "Using PTP," Red Hat Customer Portal, 2021. [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/deployment_guide/ch-configuring_ptp_using_ptp4l

[34] G. Singh and C. Prince, "Multi-node controller with gateway," Laboratory description for Cyber Physical Systems, Graduate School of Operational and Information Sciences., Naval Postgraduate School, Monterey, CA, January 2021.

[35] D. Macii, D. Fontanelli, and D. Petri, “A master-slave synchronization model for enhanced servo clock design,” in *2009 International Symposium on Precision Clock Synchronization for Measurement, Control and Communication*. IEEE, 2009, pp. 1–6.

[36] J. Huwyler, “PTP time synchronization for flocklab 2,” Master’s thesis, Dept. of Info. Tech. and Elec. Eng., ETH Zurich, 2020, semester thesis. Available: https://pub.tik.ee.ethz.ch/students/2020-FS/SA-2020-01.pdf

[37] Kite. “How to get the time with microsecond precision in Python.” Accessed 2021. [Online]. Available: https://www.kite.com/python/answers/how-to-get-the-time-with-microsecond-precision-in-python

[38] M. Ahmed, “Implementation and performance analysis of precision time protocol on Linux based system-on-chip platform,” 2018, master Project.

[39] V. Jordan. ("2020. [Online.]"). “Sync up your clocks! Better PTP settings on Raspberry Pi.”. [Online]. Available: https://medium.com/inatech/sync-your-clocks-better-ptp-settings-on-raspberry-pi-37a9a54e4802

[40] “PTP - precision time protocol: IEEE 1588 v1 and v2 PTP boundary clock capabilities in industrial managed switches.” Available: https://www.perle.com/supportfiles/precision-time-protocol.shtml

[41] F. Girela, “Software defined timing: the synchronization solution for data centers,” Seven Solutions, 2019. [Online]. Available: https://wsts.atis.org/wp-content/uploads/sites/9/2019/03/4_03_7Solutions_Lopez_Software-Defined-Timing.pdf

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California