2018-03

A NOVEL APPROACH FOR COVERT COMMUNICATION OVER TCP VIA INDUCED CLOCK SKEW

Knebel, Erik S.

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

A NOVEL APPROACH FOR COVERT COMMUNICATION OVER TCP VIA INDUCED CLOCK SKEW

by

Erik S. Knebel

March 2018

Co-Advisor:	Murali Tummala
Co-Advisor:	John McEachen
Second Reader:	Bryan Martin

THIS PAGE INTENTIONALLY LEFT BLANK

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

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** March 2018	**3. REPORT TYPE AND DATES COVERED** Master's thesis

4. TITLE AND SUBTITLE A NOVEL APPROACH FOR COVERT COMMUNICATION OVER TCP VIA INDUCED CLOCK SKEW	**5. FUNDING NUMBERS**
6. AUTHOR(S) Erik S. Knebel	
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) N/A	**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. IRB number ____N/A____.

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

13. ABSTRACT (maximum 200 words)

The goal of this thesis is to determine the feasibility and provide a proof of concept for a covert communications channel based on induced clock skew. Transmission Control Protocol (TCP) timestamps provide a means for measuring clock skew between two hosts. By intentionally altering timestamps, a host can induce artificial clock skew as measured by the receiver, thereby providing a means to covertly communicate. A novel scheme for transforming symbols into skew values is developed in this work, along with methods for extraction at the receiver. We tested the proposed scheme in a laboratory network consisting of Dell laptops running Ubuntu 16.04. The results demonstrated a successful implementation of the proposed covert channel with achieved bit rates as high as 33 bits per second under ideal conditions. Forward error correction was also successfully employed in the form of a Reed–Solomon code to mitigate the effects of variation in delay over the Internet.

14. SUBJECT TERMS covert channel, TCP, clock skew, cyber, TCP timestamp	**15. NUMBER OF PAGES** 119
	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 NOVEL APPROACH FOR COVERT COMMUNICATION OVER TCP VIA INDUCED CLOCK SKEW

Erik S. Knebel
Lieutenant, United States Navy
B.S., United States Naval Academy, 2009

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
March 2018

Approved by: Murali Tummala
Co-Advisor

John McEachen
Co-Advisor

Bryan Martin
Second Reader

R. Clark Robertson
Chair, Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The goal of this thesis is to determine the feasibility and provide a proof of concept for a covert communications channel based on induced clock skew. Transmission Control Protocol (TCP) timestamps provide a means for measuring clock skew between two hosts. By intentionally altering timestamps, a host can induce artificial clock skew as measured by the receiver, thereby providing a means to covertly communicate. A novel scheme for transforming symbols into skew values is developed in this work, along with methods for extraction at the receiver. We tested the proposed scheme in a laboratory network consisting of Dell laptops running Ubuntu 16.04. The results demonstrated a successful implementation of the proposed covert channel with achieved bit rates as high as 33 bits per second under ideal conditions. Forward error correction was also successfully employed in the form of a Reed–Solomon code to mitigate the effects of variation in delay over the Internet.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

A/D	analog-to-digital
ARQ	automatic repeat request
AS	autonomous system
ASCII	American Standard Code for Information Interchange
BCH	Bose-Chaudhuri-Hocquenghem
BER	bit error ratio
CPU	central processing unit
CRC	cyclic redundancy check
FCS	frame check sequence
FTP	file transfer protocol
HTTP	hypertext transfer protocol
ICMP	internet control message protocol
LSB	least significant bit
OS	operating system
PAWS	protection against wrapped sequences
PDF	probability density function
PPM	parts-per-million
RFC	request for comments
RTC	real-time clock
RTT	round-trip time
SER	symbol error rate
SNR	signal-to-noise ratio
SSH	secure shell
TCP	transmission control protocol
TSval	timestamp value
TSecr	timestamp echo reply
Tsopt	timestamp option
SYN	synchronize
UDP	user datagram protocol

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

Dedicated to my late grandfather, Alan C. Peterson I, who as a United States Navy veteran, steel mill industrial engineer, and man of flawless moral character, is an unending source of inspiration for both my personal and professional endeavors.

I would first like to thank my thesis advisors, Professor Murali Tummala and Professor John McEachen. Their guidance, insights, and mentorship were invaluable. I would also like to acknowledge and thank Lieutenant Bryan Martin for the valuable feedback provided as a second reader for this thesis. During the writing of this thesis, my parents were an unceasing source or encouragement and have my heartfelt gratitude.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

The ability to securely communicate over digital networks is constantly challenged by increasingly inventive attacks. In February 2017, a team of cryptanalysts successfully broke SHA-1, highlighting the vulnerability of what had been a gold standard of cryptography [1]. While cryptography attempts to distort a message to prevent interception, covert communications attempt to hide the very existence of the data being transmitted. Transmission control protocol (TCP) is an ideal candidate for a covert channel because of its widespread and universal use; as a result, many TCP-based schemes have been developed. Most of these schemes have subsequently been analyzed, leading to the development of countermeasures capable of detecting or destroying the covert channel [2]. There is interest in developing new schemes in order to maintain the ability to covertly communicate or to stay one step ahead of those attempting to covertly communicate. Additionally, some covert channels provide the ability to transmit useful information related to the signal, such as metadata or network diagnostics.

Although the idea of using induced clock skew as a covert channel was suggested by Kohno et al. in 2005 [3], the idea has never been developed or implemented. As of this writing, clock skew remains an obscure characteristic that is generally not monitored by network security systems. A covert channel based on clock skew has the potential to provide a data rate and channel capacity comparable to previously developed TCP-based covert channels [4] while remaining significantly less detectable.

A. THESIS GOAL AND OBJECTIVES

The goal of this thesis is to develop a practical covert communications channel that utilizes induced clock skew to embed secret messages. Unlike existing schemes, the proposed covert channel does not embed bits but instead transmits symbols through variations in clock skew in a manner analogous to modulation in which the skew is the carrier. Such a covert channel is developed in this work and tested in a laboratory environment.

The objectives of this thesis are to develop a framework for a skew–based covert communications channel, to provide a proof of concept that such a channel can operate in a laboratory environment, and to experimentally determine what parameters in that framework can be varied to maximize bit rate while reducing the likelihood of detection. In order to take full advantage of the analog nature of clock skew, a higher-order embedding scheme that allows for multiple bits per symbol is designed, built and tested. The application of error detection and error correction to the developed scheme are also explored in an attempt to improve bit rate and robustness.

B. RELATED WORK

In their 2005 work, Kohno et al. showed that TCP timestamps could be used to calculate the clock skew between two hosts [3]. This was additionally verified by Martin [5], where clock skew was used to create a hardware fingerprint for hosts on a network. Although [3] and [5] were primarily focused on using clock skew for hardware fingerprinting, [2] hypothesized that that a host that could intentionally vary its clock skew could create a covert communications channel. This idea is developed and tested in this work.

In 2002, Griffin et al. demonstrated that the TCP timestamp field could be used as a covert channel [4]. Employing naïve steganography, a scheme in which the least significant bit of the TCP timestamp field was replaced by a message bit was developed in [4]. This type of naïve embedding is highly vulnerable to the most basic steganographic analysis as described by Fridrich [6], Murdoch et al. [2], and Liu et al. [7]. The scheme presented in this thesis is fundamentally different in concept from the ideas presented in [4]. The only commonality between the two schemes is the use of the TCP timestamp field to convey covert information.

The work of [2] and [7] demonstrate additional methods for hiding data over TCP, including the use of covert timing channels and matrix embedding. Covert timing channels rely on a transmitting host sending TCP/IP packets in such a way to intentionally vary the arrival time at the receiver. Some rely on the use of timing intervals at the receiver, with packets communicating either a ‘1’ or ‘0’ depending on when they arrive. Other timing

channels convey a message in the interpacket delay at the receiver [7]. These ideas are also fundamentally different from the scheme developed in this thesis, which does not rely on specific arrival times for packets.

C. THESIS ORGANIZATION

The remainder of this thesis is organized as follows. In Chapter II, clock skew is defined and methods for its calculation are explained. System clocks, TCP, raw sockets, and error correction are briefly discussed as they pertain to the ideas developed herein. The proposed scheme for a covert communications channel is described in detail in Chapter III. The testbed and results are presented in Chapter IV. Finally, conclusions, significant results, and recommendations for future work are discussed in Chapter V. The code used for testing is contained in the appendices.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND

In this chapter, we present concepts germane to the understanding of the scheme proposed in Chapter III. System clocks, TCP timestamps, raw sockets, error correction, error detection and clock skew are discussed as they relate to the thesis objectives and proposed scheme.

A. SYSTEM CLOCKS

Computers keep track of time though the use of software clocks. This ability is important for a variety of reasons including but not limited to displaying current time, synchronization, and scheduling tasks. Each central processing unit (CPU) defines a processor tick that is an integer quantity of CPU clock cycles. When the CPU has gone through this integer number of clock cycles, it receives an interrupt and increments system time. Ticks are constant value and can be converted to seconds, allowing the computer to show time for a process, etc. [8].

The resolution of a clock is defined as the smallest change in time that clock can detect [8]. For example, a 1.0-kHz clock has a resolution of 1.0 ms. When computers shut down, they transfer their system time to a battery powered clock called a real-time clock (RTC), which continues to measure the passage of time while the computer is unpowered. When power is restored, the RTC provides information to the system clock, allowing time to be maintained in spite of shut downs or power outages [8].

B. TCP

The scheme presented in this work uses TCP as a cover in the transmission of covert messages. A widely used transport layer protocol, TCP was developed in 1981 and is described in Request for Comments (RFC) 793 [9]. Because TCP carries widely used application layer services, such as file transfer protocol (FTP), hypertext transfer protocol (HTTP), and secure shell (SSH), TCP accounts for a significant portion of all internet traffic. Its most significant difference from the other widely used transport layer protocol, user datagram protocol (UDP), is that it requires the establishment of a session, and through

the use of sequence numbers and acknowledgements, ensures that no packets have been dropped or lost. Additionally, it includes measures to relieve network congestion [10].

The TCP header is a minimum of 20 bytes and can be up to 60 bytes when all options fields are included. A TCP segment can carry up to 65,536 bytes of payload data but is usually limited to less than 1,460 bytes. This is because most network applications using TCP use Ethernet as a layer-two protocol, and the maximum size of an Ethernet frame is 1,522 bytes, including at least 22 bytes taken up in the Ethernet header and at least 20 bytes taken up in the IP header [10].

The TCP timestamp is one of the TCP options fields and was added to the protocol in RFC 1323 in 1989 [11]. The TCP timestamp helps provide protection against wrapped sequences (PAWS), a scenario involving a TCP connection that has its 32-bit sequence number exceed the maximum value of 23423 and start again at 1. When the receiver receives a segment with a sequence number with a lower sequence number than other previously received segments, it has no way of determining if the received segment is new or very old. The timestamp provides a mechanism for the receiver to determine the age of such a segment. Additionally, timestamps provide an estimation of round trip time (RTT), which provides insight into the level of network congestion. The timestamp itself is ten bytes long and consists of four fields, as shown in Figure 1. The first two fields aid in identifying the timestamp. The timestamp value (**TSval)** field is filled with the timestamp of the packet's sender. The timestamp echo reply (**TSecr**) is the value of **TSval** from the last received packet. These two values are used together to calculate RTT [12].

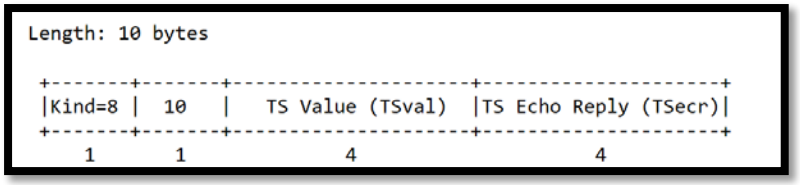


Figure 1. TCP Timestamp. Source: [11].

The clock that populates the **Tsecr** field, the timestamp option (**Tsopt)** clock, has varying resolution depending on the operating system. The resolution of **Tsopt** clock can vary from 1 Hz to 1 kHz [11].

The work of [3] demonstrated that by default a host will respond with timestamps to a TCP session initiated with timestamps. A user has the option to explicitly turn timestamps off, negating this method [3].

Because of its ubiquity, TCP provides an ideal cover for a covert channel. Its widespread use provides a vehicle for covert communications that does not raise suspicion in and of itself. Additionally, being a transport layer protocol, TCP is high enough on the protocol stack to travel from source to destination unaltered. This was verified by Handel et al. [13]. Headers from lower level protocols, such as internet protocol (IP), are often shed and reconstituted many times before reaching their destination [13].

Data can be hidden in TCP segments using either storage channels, which embed bits in header fields, or timing channels, which depend on arrival times of packets or interpacket delay [7]. Most header fields are not suitable storage channels because of either their importance in proper segment delivery or because non-standard values in certain fields might arouse suspicion [2]. A passive warden, or adversary that has the ability to view but not alter the data in transit, can detect the presence of a storage channel by using known stegoanalytic techniques on the header fields of received packets. An active warden, or adversary that has the ability to view and alter data in transit, can distort or destroy the covert message by altering data in nonessential header fields or delaying packet arrival times [7].

C. RAW SOCKETS

The TCP/IP protocol stack describes the hierarchy of tasks required to engage in network activity [10]. The human interface to a network is normally through programs that utilize application layer protocols. Sockets serve as the endpoints for network communication for the transport layer and are created by the host operating system (OS). Most application layer protocols call the OS to open either a stream socket to utilize TCP or a datagram socket to utilize UDP [14].

Raw sockets differ from stream and datagram sockets by providing access to the lower layers of the OSI model and giving the user bitwise control of protocols at all layers [14]. For example, normally a user trying to transfer a file via FTP does not have control of the destination port in the TCP header because this port number is automatically set to 21 by a stream socket created by the OS. Raw sockets allow a programmer to control all fields including but not limited to sequence number, source port, destination port, and the timestamp fields.

D. CLOCK SKEW

An understanding of clock skew calculation is fundamental to the ideas presented in this thesis. In the next two sections, we present a brief overview of clock skew and describe a method for calculating clock skew developed in [3] and [5].

1. Definition

Clock skew α is the rate at which the time measurement of two clocks, c_1 and c_2, diverge. It is given in parts per million (PPM), which are the number of μs that two clocks differ after one second has elapsed [3], [5]. Because clocks are not infinitely precise, there is always some clock skew between any two clocks. The work of Polcak et al. [15] demonstrated that the skew of a population of clocks has an approximately normal distribution centered at zero. This result can be explained by the fact that the manufacturer producing the clock intends for it to be perfectly accurate [15].

Skew typically ranges from -300 PPM to 300 PPM, as depicted in Figure 2 [3]. A survey conducted in [15] achieved similar results. The collected data in [15] was displayed in a manner that highlighted the long tails of the distribution as depicted in Figure 3.

Clock skew has some dependence on OS; [15] and Rhinehart [16] conducted surveys of clock skew values from hosts on different operating systems and found that some operating systems produced non-constant clock skew, in particular, Mac OS X.

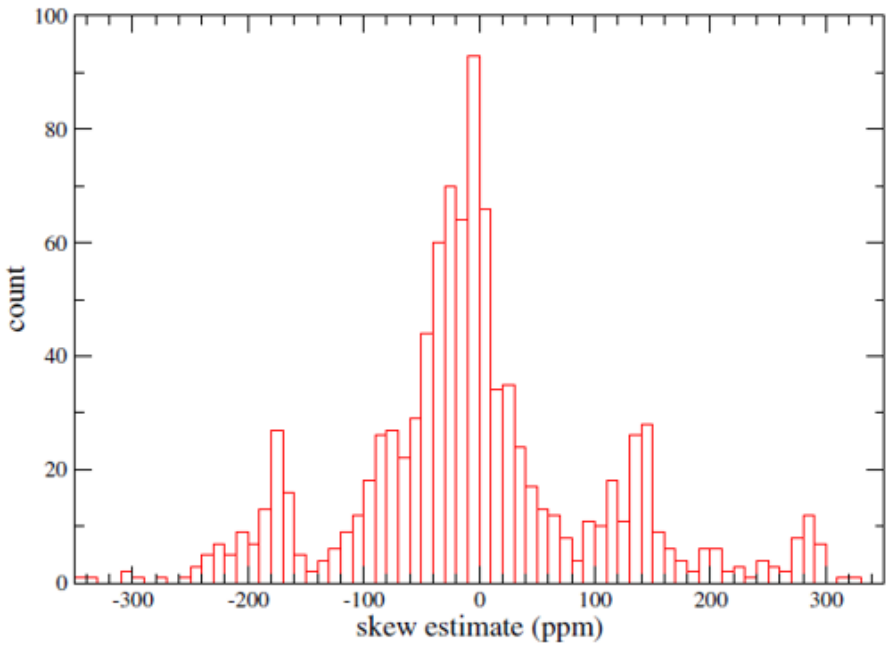


Figure 2. Distribution of Clock Skew on Real Network. Source: [3].

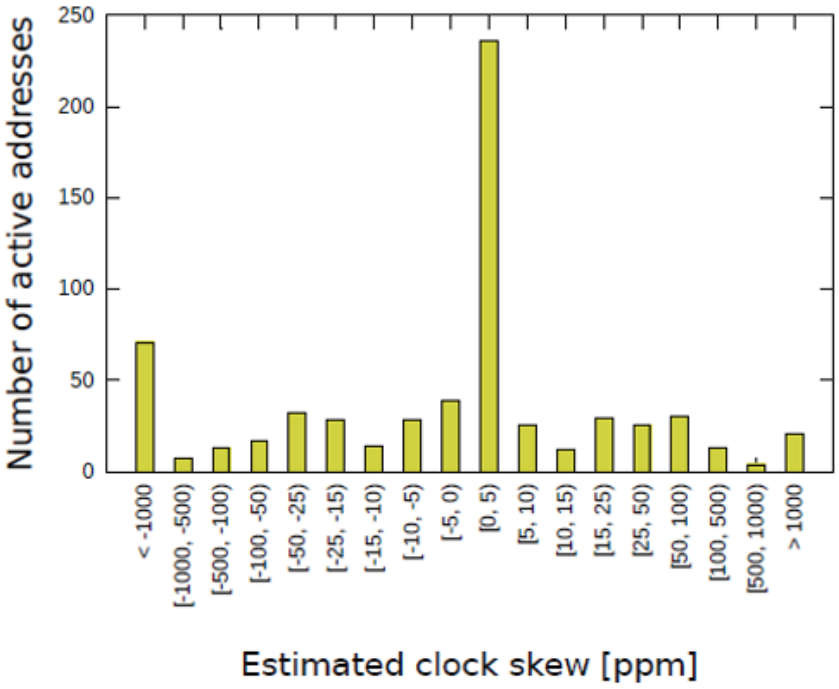


Figure 3. Distribution of Clock Skew on Real Network. Source: [15].

With the exception of the OS-specific non-constant clock skew described in [15] and [16], the results of [3] also demonstrate that clock skew remains constant over time. In experiments conducted over 38 days, constant clock skew was measured for 69 hosts running Windows XP [3]. Their result is important to the scheme developed in this work because without a transmitter and receiver having constant clock skew, the determination of induced skew values is difficult.

2. Clock Skew Calculation

Skew is always calculated relative to one of the clocks, known as the fingerprinter. Calculation of skew requires at least two synchronous time measurements from each clock [3]. In their study, Sharma et al. [17] determined experimentally that a minimum of 70 timestamp pairs were required for an accurate skew value calculation. The work of [15] refuted the work of [17], arguing instead that timestamp count was irrelevant and that it was the timespan over which the timestamps were collected that determined the accuracy of the clock skew calculation. This minimum timespan for an accurate calculation was determined experimentally to be five to ten minutes [15].

Each timestamp t_c from a clock can be modeled as a combination of base time of that clock t_{co} plus offset r_c : $t_c = r_c + t_{co.}$ In this discussion, the term timestamp refers to any time received from a clock being compared and not TCP timestamps specifically.

In the scheme described in [3] and [5], a packet sniffing application is used to produce a timestamp at the exact time of receipt of the packet. The values of timestamps received from c_1 and c_2, namely, t_{c1} and t_{c2}, are each then given by [3]

$$\begin{aligned} t_{c1} &= t_{c1o} + r_{c1} \\ t_{c2} &= t_{c2o} + r_{c2}. \end{aligned} \tag{1}$$

Rearranging (1), we obtain a method for calculating the offset of each host:

$$r_c = t_c - t_{co}. \tag{2}$$

The calculated values of r_{c1} and r_{c2} are then used to calculate drift d, which is defined as the difference in offset between two clocks at any point in time [3]:

$$d = r_{c1} - r_{c2}. \tag{3}$$

For this method of calculation to yield accurate results, t_{c1} and t_{c2} must be measured simultaneously. In the scheme proposed in [3], [5], [15], and [16], the timestamps are not taken simultaneously because the fingerprinter timestamp is not created in a packet-sniffer until the arrival of the fingerprintee's TCP timestamp. This departure from simultaneity is a result of the travel time t_t between c_1 and c_2, i.e., if the clocks have no skew between them $t_{c2} = t_{c1} + t_t$. Because the initial timestamp from the packet-sniffing application t_{c2o} includes the travel time of the first received packet t_{to}, subtracting t_{c2o} from t_{c2} to produce r_{c2} rides on the assumption that all future values of t_t will equal t_{to}. This assumption, while fundamental to the methods developed in [3], [5], [15], and [16], is not completely accurate even under ideal laboratory conditions. This means that the calculated value of r_{c2} is actually an estimate $\hat{r}_{c2}$. Consequently, the calculated value of d in (3) becomes an estimate $\hat{d} = r_{c1} - \hat{r}_{c2}$.

The skew is the rate at which drift changes and can be described by [3]

$$\alpha = \frac{dd}{dr_{c2}}. \tag{4}$$

Multiplying each side by dr_{c2} and taking the antiderivative, we have d equaling the equation of a line with slope of α and *y*-intercept of constant β_0 as shown by [3]

$$d = \alpha r_{c2} + \beta_0. \tag{5}$$

The constant β_0 is always zero when calculated this way because the subtraction of t_{co} to calculate the offset values ensures that the line formed by (5) passes through the point (d, r_{c2}) = (0,0).

While (4) and (5) describe the conceptual relationship between α, d and r, we do not have the true value of d so must instead use the received data point to estimate skew $\hat{\alpha}$ instead of the true value α. This estimate is calculated by finding the first coefficient in a least-squares regression model [16], which finds the best fit linear slope for a data set [18] where

$$\hat{\alpha} = \frac{\sum_{i=1}^{n}(r_{c2i} - \bar{r}_{c2})(d_i - \bar{d})}{\sum_{i=1}^{n}(r_{c2i} - \bar{r}_{c2})^2} \quad . \tag{6}$$

The bar above offset and drift in (6) denotes an average over all collected samples. The skew estimation can also be accomplished using a dynamic programming technique [3], [5]. The methods are equivalent for low-jitter environments [16].

We are unconcerned with calculating the second coefficient $\hat{\beta}_0$ of the least squares linear regression, which is zero because of the earlier subtraction of t_{c2o} and t_{c1o} when calculating the offsets. The index i represents a single timestamp pair, and n is the total quantity of timestamp pairs. The calculation of $\hat{\alpha}$ is illustrated graphically in Figure 4, adapted from [18].

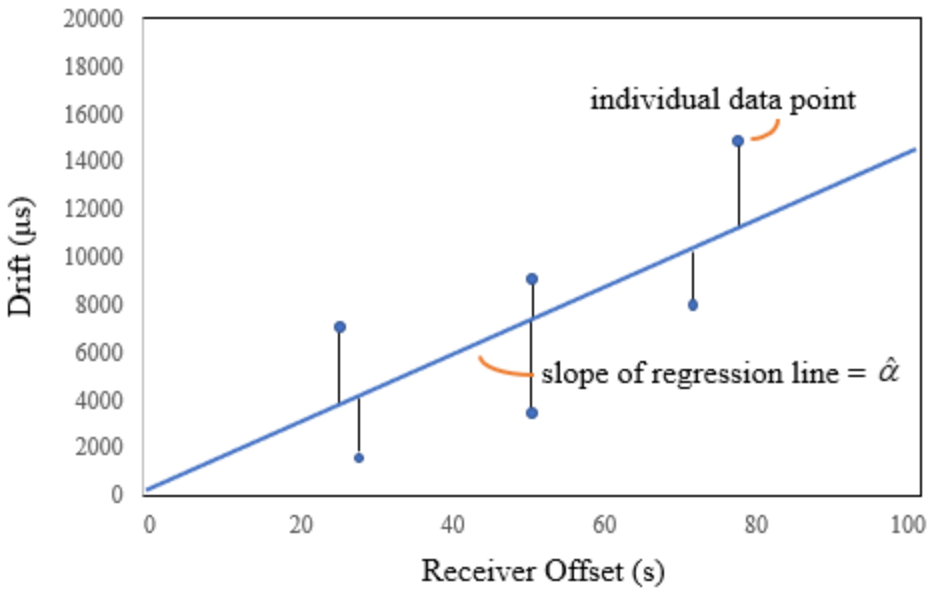


Figure 4. Least-Squares Linear Regression. Adapted from [18].

E. ERROR DETECTION AND CORRECTION

Noise may introduce errors during transmission. In digital communications, noise manifests in the form of bit errors, where a '1' is read as a '0,' or vice versa. The bit error ratio (BER) ψ_b is defined as the ratio of bit errors n_e to n packets and is given by [19]

$$\psi_b = \lim_{n\to\infty}\left(\frac{n_e}{n}\right). \tag{7}$$

Following (7), we define the symbol error ratio (SER) ψ_ρ as the ratio of symbol errors n_s to n packets and is given by [19]

$$\psi_\rho = \lim_{n\to\infty}\left(\frac{n_s}{n}\right). \tag{8}$$

Error correction and detection work by adding redundancy to a code to counteract the effects of noise. Error detection identifies that the data has been altered in transit, whereas error correction identifies the location of the change and gives the receiver the ability to fix it. The ability to detect errors can prevent corruption of data and allow for retransmission. The further ability to correct errors can reduce network congestion by eliminating the need for retransmission should an error occur.

1. Cyclic Redundancy Check

Cyclic Redundancy Check (CRC) is a widely employed error detection scheme used, for example, in Ethernet, USB, Bluetooth, and mobile network standards. Transmitters and receivers using CRC must share a generator polynomial. The message bits are divided by the generator polynomial to produce a remainder that acts as an error detection code known as a frame check sequence (FCS). The FCS is appended to the end of the transmitted data and compared to an independently calculated FCS at the receiver. If the receiver's calculation matches the received FCS value, the transmission is assumed to be error free. The naming convention for the specific implementation of CRC is based on the length of the generator polynomial, i.e., CRC-9 for a 9-bit generator polynomial. CRC is capable of detecting burst errors up to the length of its generator polynomial provided that they do not affect the CRC code itself [10].

An example of FCS calculation is provided in Figure 5 using generator polynomial '110011.' This example employs CRC-6. The first step is appending five '0's to the message bits to make room for the remainder. Then the message is divided by the generator polynomial. The remainder is the FCS and is appended to the original message bits before transmission. The quotient of the long division operation is irrelevant to FCS calculation and is not displayed in Figure 5.

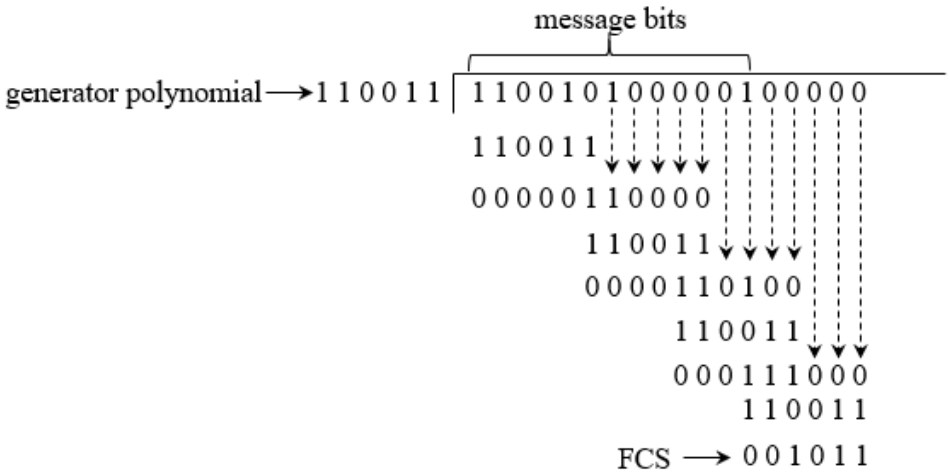


Figure 5. Example CRC-6 FCS Calculation

If a receiver detects an error through the use of an error detection scheme such as CRC, automatic repeat request (ARQ) protocols are typically employed to automate retransmission of the corrupted data [10].

2. Reed–Solomon Code

Forward error correction (FEC) codes are broadly grouped into block codes and convolutional codes. Block codes perform error correction on data in fixed size blocks, whereas convolution codes do so for potentially overlapping groups of message bits. Bose–Chaudhuri–Hocquenghem (BCH) codes are a subset of block codes, and Reed–Solomon Codes are a subset of BCH codes.

Reed–Solomon codes group m bits of a message into symbols and can have a maximum block length of $2^m - 1$ symbols, which are divided between message symbols and parity symbols. A Reed–Solomon code can correct half as many symbol errors as the number of parity symbols s_{RS} [20], e.g., with $m = 4$ bits per symbol and $s_{RS} = 4$ symbols, a Reed–Solomon code could correct two symbol errors and a maximum of eight bit errors in 11 message symbols. The block size may be split between the message and parity symbols as necessary to produce the desired level of symbol-error correction. The relationship between block size, number of parity symbols, and symbol correction is depicted in Figure 6.

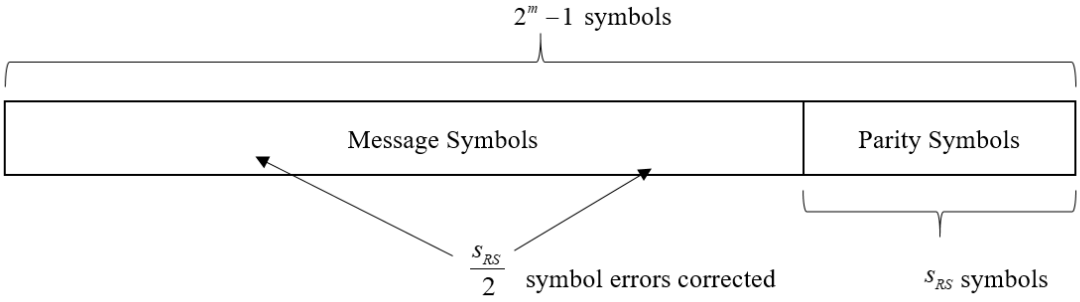


Figure 6. Reed–Solomon Code Block Size

In this chapter, we presented background concepts necessary to understand the ideas presented in this work. In Chapter III, we present a scheme dependent on the methods of clock skew calculation discussed in this chapter.

THIS PAGE INTENTIONALLY LEFT BLANK

III. INDUCED CLOCK SKEW AS A COVERT CHANNEL

In Chapter II, we explained the mathematical framework behind the clock skew measurement. As discussed in Chapter II, clock skew is the rate of change of drift between two clocks as shown in Figure 7. It can be estimated by comparing at least two simultaneously-taken timestamps from each clock. Building on these concepts, in this chapter we present a new scheme that provides a means to intentionally vary the skew calculated by the fingerprinter to communicate a covert message.

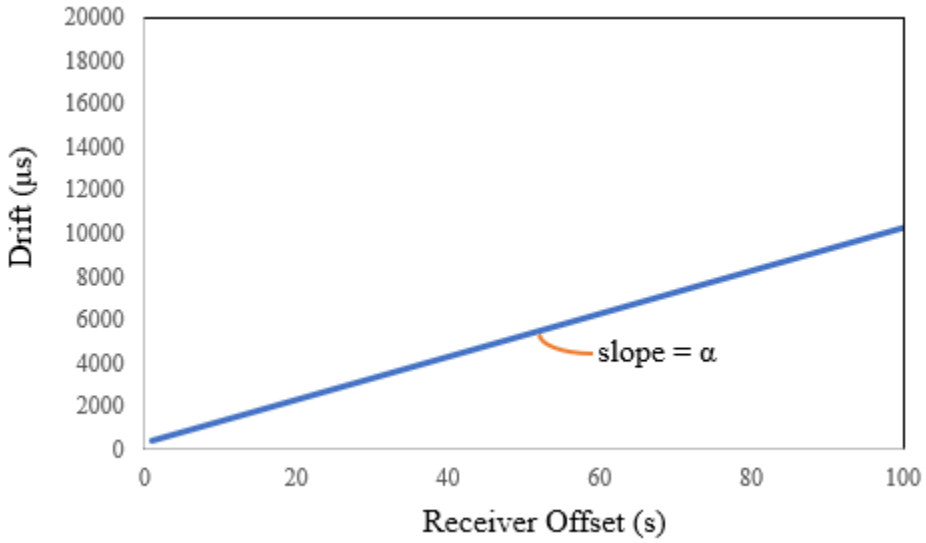


Figure 7. Clock Skew Calculation

A. PROPOSED SCHEME

The proposed scheme provides a framework for two hosts to communicate using induced clock skew over a TCP connection as shown in Figure 8. Although it was demonstrated in [3] that ICMP can also be used to measure clock skew between two hosts, TCP/IP is better suited for the creation of a practical covert channel as consistent with the goal of this thesis. This is because of TCP's ubiquity and multi-hop survival as discussed in Chapter II [18]. Additionally, the quantity of ICMP packets needed to calculate clock

skew would appear conspicuous to any adversary with the ability to monitor network traffic [3].

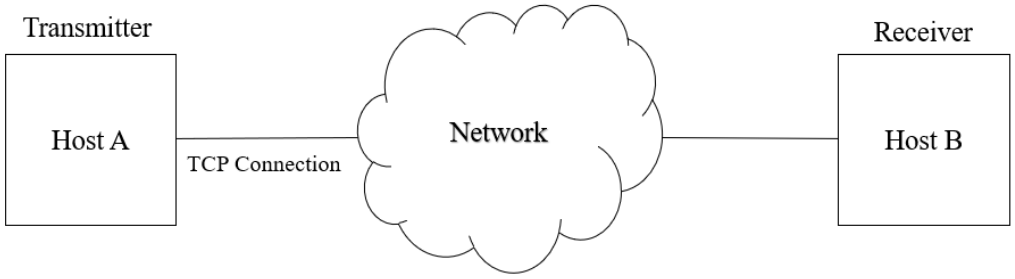


Figure 8. TCP Connection over a Network

As in [3], [5], and [15], the TCP timestamp provides the transmitter timestamp t_{c1}, and a packet-sniffing application at the receiver provides the receiver timestamp t_{c2}. With these two values, we have the necessary information to calculate clock skew at the receiver [3].

A functional diagram of the proposed scheme is displayed in Figure 9. A covert message is embedded in TCP segments, transmitted, and then extracted by the receiver. The prime on the timestamps after embedding denotes the alteration of timestamp.

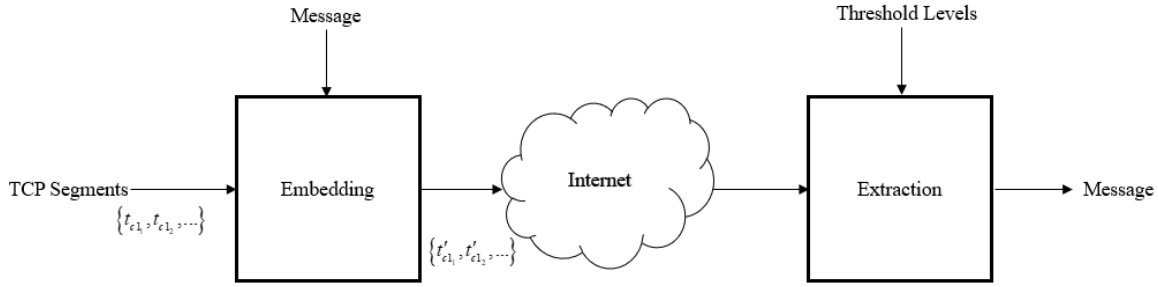


Figure 9. Scheme Functional Diagram

The steps for both embedding and extraction are expanded upon in Figure 10. The transmitter initiates a TCP connection with the timestamp option enabled. The transmitter initially transmits a quantity of unaltered packets to allow the receiver the opportunity to calculate the baseline clock skew of the transmitter. After first converting the covert message to symbols and next to induced skew values, the transmitter begins adding a small offset to each outgoing timestamp to produce the desired skew value at the receiver. The receiver reverses the process, calculating the skew for batches of received packets and then subsequently performing analog-to-digital (A/D) conversion to extract the message symbols.

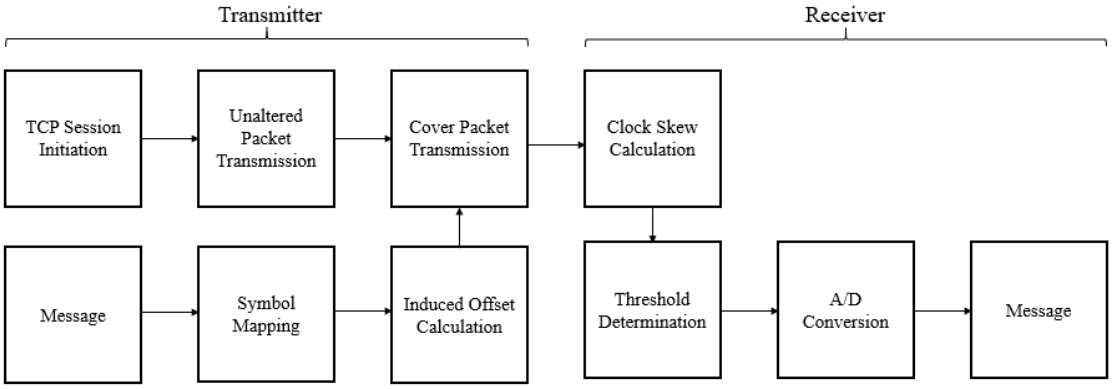


Figure 10. Scheme Overview

1. Setup

The initial packets sent by the transmitter are intentionally unaltered to allow the transmitter to estimate the baseline skew α_b of the receiver. The quantity of packets n_b used to make this calculation add overhead to the scheme because they do not contain any message information. Once the transmitter has sent n_b packets, it begins inducing skew on outgoing timestamps in intervals of n_m packets. Each subsequent batch of n_m packets has an independent induced skew γ_j that is a product of the base skew level ζ PPM and the value of the symbol ρ being transmitted as given by

$$\gamma_j = \zeta \rho_j \quad . \tag{9}$$

The transmitter and receiver must share the predetermined values of n_b, n_m, and ζ as well as the same symbol set to ensure correct extraction.

B. TRANSMISSION BIT RATE

The number of bits that can be transmitted as well as the speed at which they can be delivered are measures of effectiveness for covert channels. In the next two sections, we develop a means to estimate these values so that the effectiveness of different implementations may be compared.

a. Channel capacity

We define channel capacity for this scheme as the number of bits that can be sent on the proposed covert channel during a single TCP session. While two hosts could hypothetically keep a TCP session open indefinitely, doing so could constitute conspicuous behavior and contradict the goal of this thesis, to develop a practical covert channel. If we limit the TCP connection to the transmission of a finite payload, the channel capacity of the covert channel is limited by the size of this payload because the covert channel closes when the TCP connection closes. In the discussion of results in Chapter IV, we refer to the channel capacity by assuming that the TCP sender is transmitting a cover payload of arbitrary size. The following equation is proposed for channel capacity:

$$\kappa = \left(\frac{\nu_d}{\nu_p} - n_b \right) \frac{q}{n_m} \quad , \tag{10}$$

where the channel capacity κ of the covert channel proposed is limited by the total number of TCP segments required to transmit a cover payload of ν_d bytes with average packet payload size of ν_p. Because the number of bits per symbol q determines the number of bits each induced skew represents, this number acts as a multiplier for channel capacity. The ratio ν_d / ν_p in (10) is the number of segments that the TCP connection sends regardless of the presence of the proposed covert channel. The packets used for the baseline

skew calculation at the receiver act as overhead and are subtracted from the total packets that can be used to convey data. Because it takes many timestamps to calculate a skew value, this quantity of segments is divided by n_m. The employment of higher order schemes provides the ability to communicate more than a single bit with each skew value. This is represented by q in the numerator.

b. Bit rate

The bit rate R of the proposed scheme depends on the speed of its cover TCP connection. The delay between the transmitted individual TCP/IP packets is referred to as the interpacket delay t_{ipd} (s). The inverse of average interpacket delay of arriving TCP/IP packets provides the number of packets arriving each second. Dividing this value by the number of packets used for each skew calculation, we get an estimate of bit rate for the proposed scheme. Additionally, for the higher-order implementation, this value is multiplied by the number of bits per symbol. We, therefore, propose the following as an estimate of bit rate:

$$R = \frac{q}{t_{ipd} n_m} \quad . \tag{11}$$

All values of R given in this thesis are provided with the arbitrary but realistic value of $t_{ipd} = 10.0$ ms.

C. MESSAGE EMBEDDING

The message embedding of the proposed scheme is achieved by inducing skew into outgoing TCP segments according to (9) and Figure 11. Individual timestamps are modified to create the desired skew for each symbol at the receiver.

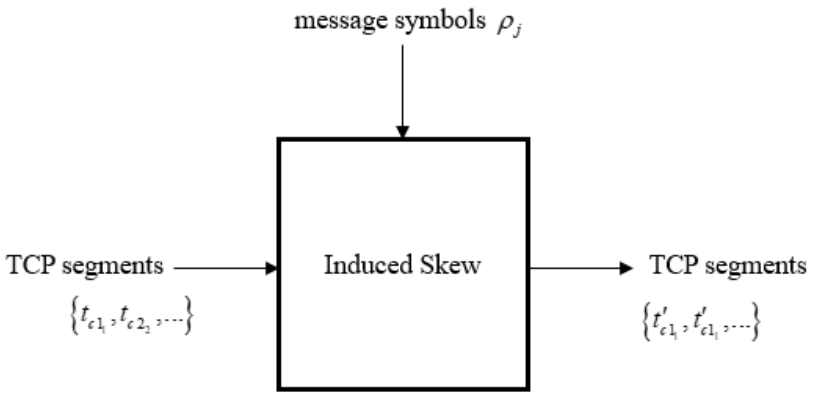


Figure 11. Message Embedding Functional Diagram

Inducing skew into the transmitted TCP segments means that the skew being transmitted is a combination of the natural skew of the transmitter and the skew induced at the transmitter. The total skew measured by the receiver is then given by

$$\alpha = \alpha_b + \gamma . \tag{12}$$

The transmitter cannot add skew directly to individual outgoing packets because skew is calculated over multiple timestamps [3]. To induce the desired skew, the transmitter must add an offset δ_i to the timestamp of each segment i that causes the methods of calculation explained in Chapter II to result in the desired induced skew.

The first step in determining δ_i is mapping each message bit group j to a corresponding symbol ρ_j. The binary implementation, the lowest order implementation of the proposed scheme, has a set of only two symbols. Clock skew is an analog property that can take on any value; a theoretically infinite number of different skews can be induced. The signal-to-noise ratio (SNR) limits the ability to differentiate between skews that are close in value and, thus, limits the size of our symbol set. Higher-order implementations, also referred to as the *q*-bit-per-symbol implementation, provide a potential means for overcoming the low data rate inherent to this concept. These implementations map q bits to $Q = 2^q$ distinct symbols. In this thesis, we develop and test message embedding and extraction for $q = 1,\ q = 2$ and $q = 4.$ The binary implementation is a special case of the *q*-bit-per-symbol implementation for $q = 1.$

The proposed mapping for $q = 1$ is given by

$$\rho_j = \begin{cases} 1, & \text{for} \quad j = 1 \\ -1, & \text{for} \quad j = 0 \end{cases}. \tag{13}$$

To map symbols to bits $q > 1$, a gray code is utilized to increase the likelihood that a single symbol error at the receiver only produces a single bit error. The symbol mapping for $q = 2$ is provided as an example of the *q*-bits-per-symbol implementation:

$$\rho_j = \begin{cases} 2, & \text{for } j = 01 \\ 1, & \text{for } j = 00 \\ -1, & \text{for } j = 10 \\ -2, & \text{for } j = 11 \end{cases}. \tag{14}$$

The symbol mapping for q = 4, the highest value of q that is tested in this thesis, is given by Table 1.

Table 1. Symbol Mapping for $q = 4$

bits j	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
symbol ρ_j	-8	-7	-5	-6	-1	-2	-4	-3	8	7	5	6	1	2	4	3

Having determined the desired γ_j based on (9), we can now determine δ_i. From (4), the relationship between the induced skew and induced offset is given by

$$\gamma = \frac{d\delta}{dr_{c1}}. \tag{15}$$

Conceptually, δ_i is the antiderivative of γ_j at the time each packet is sent, is determined in the same manner as (5), and is given by

$$\delta = \gamma r_{c1} + \beta_0, \tag{16}$$

where $\beta_\circ$ is a constant. We set $\beta_\circ$ to zero because a nonzero value produces an unhelpful and conspicuous jump in offset and drift when skew is calculated at the receiver. By replacing the offset in (16) with the bit time Δ, which we define as the offset since the

first packet in the current batch of n_m was transmitted, we ensure that each $\beta_\circ$ is zero for each skew calculation of n_m packets and there is no jump in offset at the receiver. Substituting Δ_{pj} for r_{c1} in (16) and setting $\beta_\circ$ to zero, we get

$$\delta_i = \gamma_j \Delta_{\rho_j} . \tag{17}$$

The variable Δ_{pj} differs from r_{c1} because it returns to zero after transmission of the last packet of each batch of n_m packets, as shown in Figure 12, giving the transmitter the ability to induce separate skew values on each n_m packets.

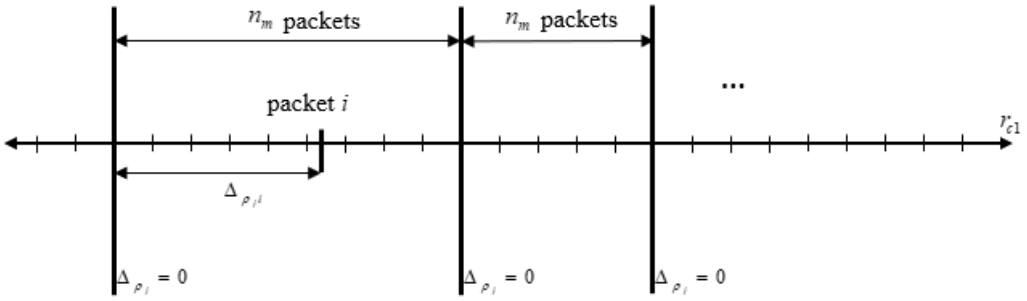


Figure 12. Relationship between Δ_{ρ_j}, n_m, and r_{c1}

The altered timestamp now has an added δ_i component. Following (1), we find the timestamps at the receiver and transmitter, respectively, are given by

$$\begin{aligned} t_{c1} &= t_{c1o} + r_{c1i} + \delta_i \\ t_{c2} &= t_{c2o} + r_{c2i} \end{aligned} . \tag{18}$$

D. MESSAGE EXTRACTION

The message extraction process recovers the embedded message by the process depicted in Figure 13. The timestamp pairs provided by the TCP timestamp and receiver timestamp are used to calculate drift with (3). Batches of n_m drift values are then used to estimate the induced skew via (6). The estimated skew then goes through A/D conversion and is mapped to a symbol. The symbols are then converted back to bits, and the message is recovered.

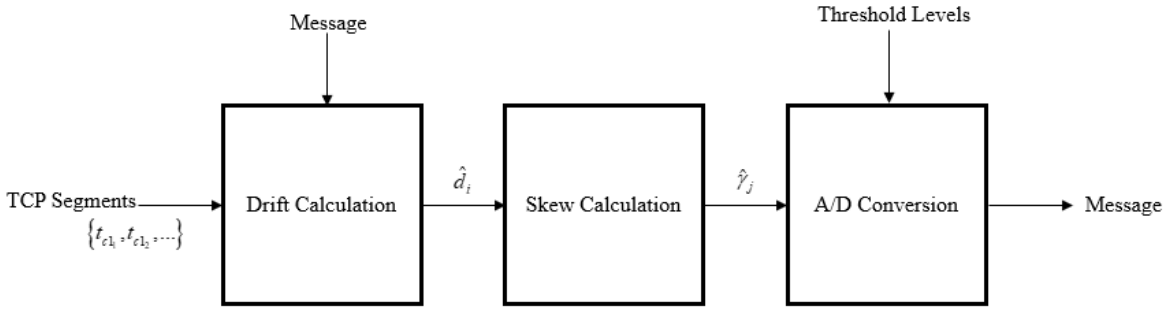


Figure 13. Message Extraction Functional Diagram

After using the first n_b packets to calculate an estimate for baseline skew $\hat{\alpha}_b$ using (6), the receiver subtracts $\hat{\alpha}_b$ from the estimated skew in each group of n_m packets to estimate induced skew $\hat{\gamma}_j$. Following (12), we see that $\hat{\gamma}_j$ is given by $\hat{\gamma}_j = \hat{\alpha}_j - \hat{\alpha}_b$. This value is then compared against a threshold value for A/D conversion. For $q = 1$, the threshold is zero, which means that positive values of $\hat{\gamma}_j$ are mapped to ρ_1, and negative values of $\hat{\gamma}_j$ are mapped to ρ_0. The receiver then maps the symbols back to bits to recover the message.

For $q>1,$ the A/D conversion process requires the creation of bins to map each of the induced skew estimates to a symbol. This mapping is given by

$$\rho_j=\begin{cases}\dfrac{Q}{2} & \text{for} \quad \hat{\gamma}_j>\left(\dfrac{Q+1}{2}\right)\zeta \\ 1 & \text{for} \quad 0\geq\hat{\gamma}_j>1.5\zeta \\ k & \text{for} \quad \left(\dfrac{2k-1}{2}\right)\zeta<\hat{\gamma}_j<\left(\dfrac{2k+1}{2}\right)\zeta, k=\pm 2,\ldots\pm\left(\dfrac{Q}{2}-1\right). \\ -1 & \text{for} \quad 0<\hat{\gamma}_j\leq -1.5\zeta \\ \dfrac{-Q}{2} & \text{for} \quad \hat{\gamma}_j<\left(-\dfrac{Q+1}{2}\right)\zeta \end{cases} \tag{19}$$

In Figure 14, the transfer characteristic is provided according to (19) for $q=3.$

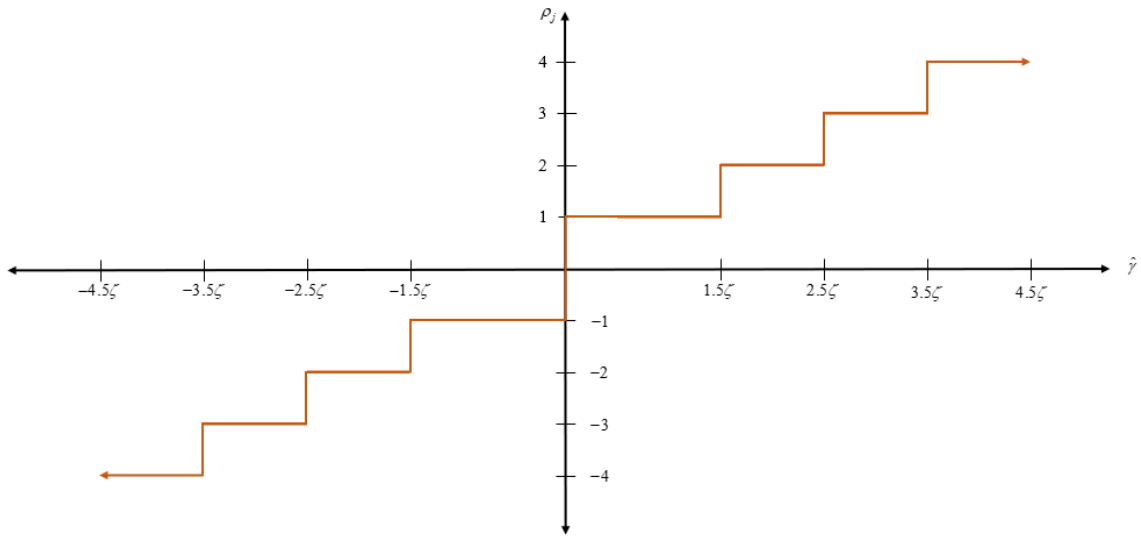


Figure 14. Transfer Characteristic of the A/D Convert Block for $q=3$

After mapping the skew estimates to symbols, the symbols are mapped to bits according to a table shared by transmitter and receiver, e.g., Table 1 is used for this mapping process by both the transmitter and receiver for $q=4.$ For mapping the symbols back to bits for $q>1,$ we used a gray code to ensure that a skew estimate that crosses a

single threshold incorrectly only produces a single bit error when the symbols are mapped back to bits. Use of gray code does not help reduce bit errors for symbol errors resulting from the erroneous crossing of more than one threshold, which can result in up to q bit errors for each symbol error.

E. NOISE

As discussed in Chapter II, the methods outlined in [3], [5], and [16] for calculating clock skew assume a constant travel time between c_1 and c_2. Because natural skew values between two clocks are typically within 300 PPM [3], [17], network jitter on the order of milliseconds can wipe out any attempt to measure clock skew by dramatically varying t_t.

Because we are communicating over a network, the travel time between the two clocks is the network delay. According to [21] and [22], we find that network delay can be modeled as a Weibull random variable. The probability density function (PDF) for the Weibull distribution is given by

$$P(t_t) = \frac{k}{\lambda}\left(\frac{t_t}{\lambda}\right)^{k-1} e^{-\left(\frac{t_t}{\lambda}\right)^k}, \tag{20}$$

where k is the shape parameter value and λ is the scale parameter value [20]. An example of the Weibull distribution is given in Figure 15 for $k = 1$ and $\lambda = 1.5$.

Noise in this scheme is manifested as a deviation in seconds from the mean value of network delay $\overline{t_t}$ and can be expressed as $w_i = t_{ti} - \overline{t_t}$. The addition of noise to transmitter offset during transit is displayed in Figure 16; however, the quantity we are concerned with is the error ε defined as $\varepsilon = \alpha - \hat{\alpha}$. Because we are using the least squares method to calculate skew, the noise in any single received timestamp does not directly affect the calculation of $\hat{\alpha}$ provided that

$$w_{avg} = \frac{1}{n}\sum_i w_i \tag{21}$$

is close to the mean of the distribution. Modeling w as Weibull random variable, we expect that the average noise will approach the mean of the distribution as the number of packets is increased. We also expect that $\hat{\alpha}$ increases in accuracy with the number of timestamp

pairs used to calculate it. This leads error to approach zero as the number of packets increases, i.e., $\lim_{n\to\infty} \varepsilon = 0$.

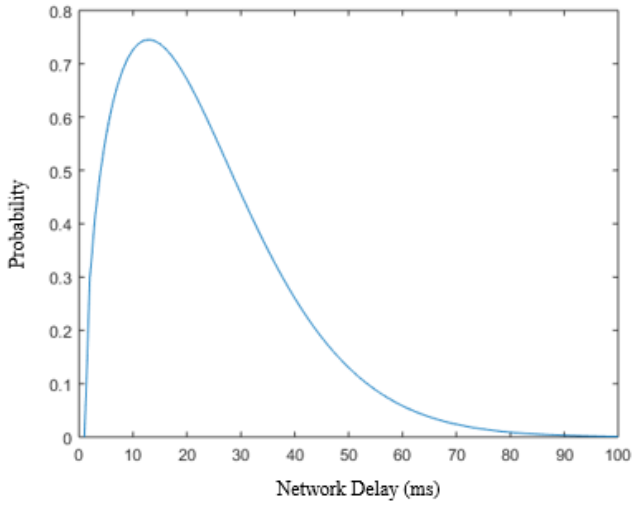


Figure 15. Weibull Distribution for $k = 1,\ \lambda = 1.5$

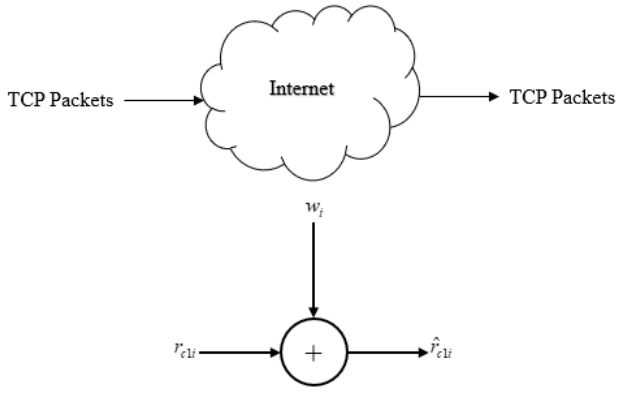


Figure 16. Noise Model: Addition of Noise to Offset

When $q = 1$, if ε is greater than ζ for any skew calculation, a symbol error and resultant bit error may occur. The scheme is more susceptible to noise for $q > 1$ because of the need for additional threshold levels to accommodate A/D conversion for larger symbol

sets according to (19). Because the threshold levels create bins of width ζ, in the *q*-bits-per-symbol implementation, a symbol error may occur for $\varepsilon \geq \zeta / 2$.

Non-constant clock skew was additionally considered as a source of noise but deemed unlikely based on the results of [3] as discussed in Chapter II. The OS-specific issues discussed in [15] and [16] are a unique case and would cause problems that would prevent the proposed scheme from operating as desired and are not considered here.

The data shown in Figure 17 provides an illustration of the noise present in skew calculations. The black line is the calculated $\hat{d}$ with slope $\hat{\alpha}$. If no noise is present, the data will fall neatly onto a line of constant slope.

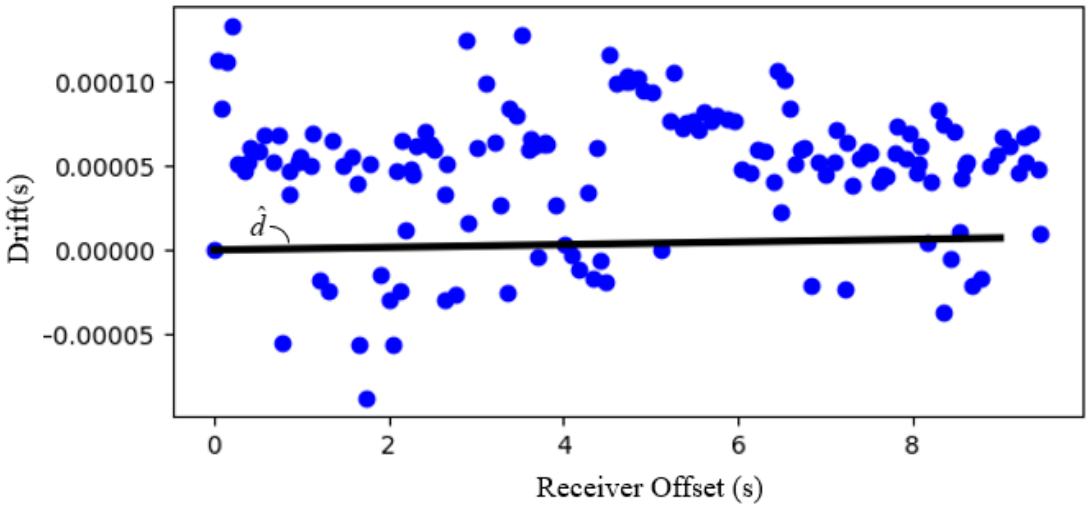


Figure 17. Noise Present in Clock Skew Calculation

1. Error Detection

Cyclic redundancy check techniques may be used to provide a simple means of error detection. Error detection alone may not be well suited for this scheme. When an error is detected, an ARQ response prompts the transmitter to resend the corrupted data. The inherently low data rate leads to a high cost for retransmission. Because of channel capacity limitations, retransmission may not be possible in many cases.

2. Forward Error Correction

FEC could be another solution. A Reed–Solomon code was chosen amongst other possible FEC codes for ease of implementation and in support of the thesis objective of determining if error correction could enhance robustness and allow for an increase in bit rate and channel capacity.

In Chapter III, we provided a framework for the scheme implemented and tested in this work. The methods for inducing skew in outgoing timestamps and estimating received skew values to extract a covert message were explained. In Chapter IV, we provide the methodology of the testing and an explanation of the results.

IV. TESTING AND RESULTS

The concepts discussed in Chapter III provided a framework for a covert channel based on induced clock skew. In this chapter, we provide detail as to the realization of such a system as well as the results of testing at various parameter values. Although there is nothing preventing such a system from operating in duplex mode, for simplicity we programmed a single host to be the transmitter and another to be the receiver.

A. TEST-BED

We used Dell XPS laptops running Ubuntu 16.04 for the transmitter and receiver. They were connected via Ethernet using CAT 5 cable across a network switch. The switch saw minimal traffic unrelated to this experiment. The hosts were connected directly via Ethernet for several trials to estimate the impact of the network switch. The results showed negligible impact with successful embedding and message extraction. The conceptual diagram of the two hosts connected across the laboratory network can be seen in Figure 18. In Figure 19, a photograph of the two host laptops is displayed.

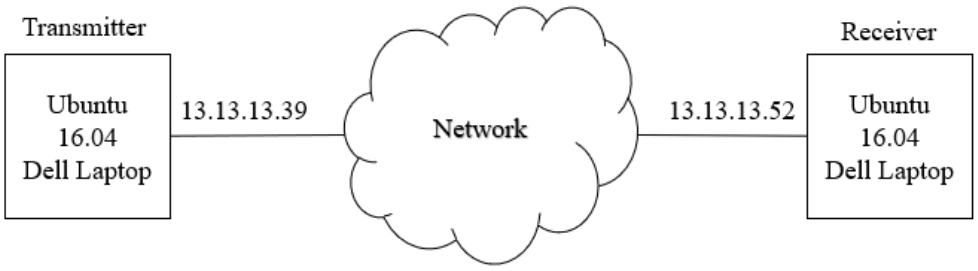


Figure 18. Testbed Conceptual Diagram

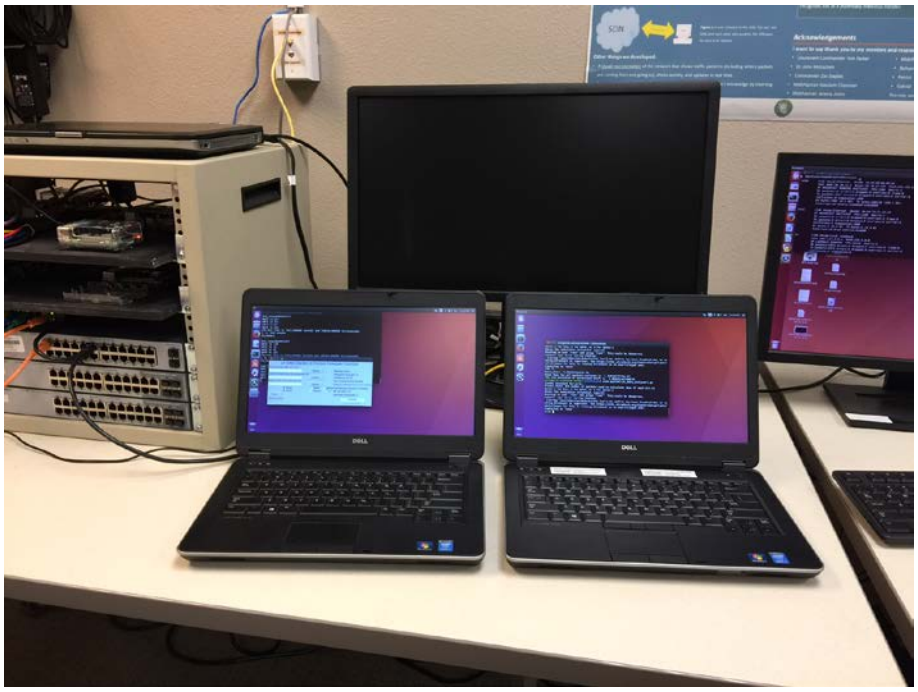


Figure 19. Photograph of Testbed Hosts

1. Transmitter

The transmission process is illustrated in the flowchart in Figure 20. For successful embedding/extraction, the transmitter and receiver must share parameter values of n_b, n_m, q, and ζ. After the establishment of a TCP session, the transmitter sends n_b unaltered packets to allow the receiver to establish the baseline skew from the transmitter. If no message is present to embed, the TCP session continues as normal without the alteration of timestamps. If a message is present, the transmitter calculates the offset to add to each outgoing timestamp by mapping the message to symbols and then applying (9) and (17). Finally, the packets with altered timestamps are transmitted across the network where noise is added according to the noise model developed in Chapter III.

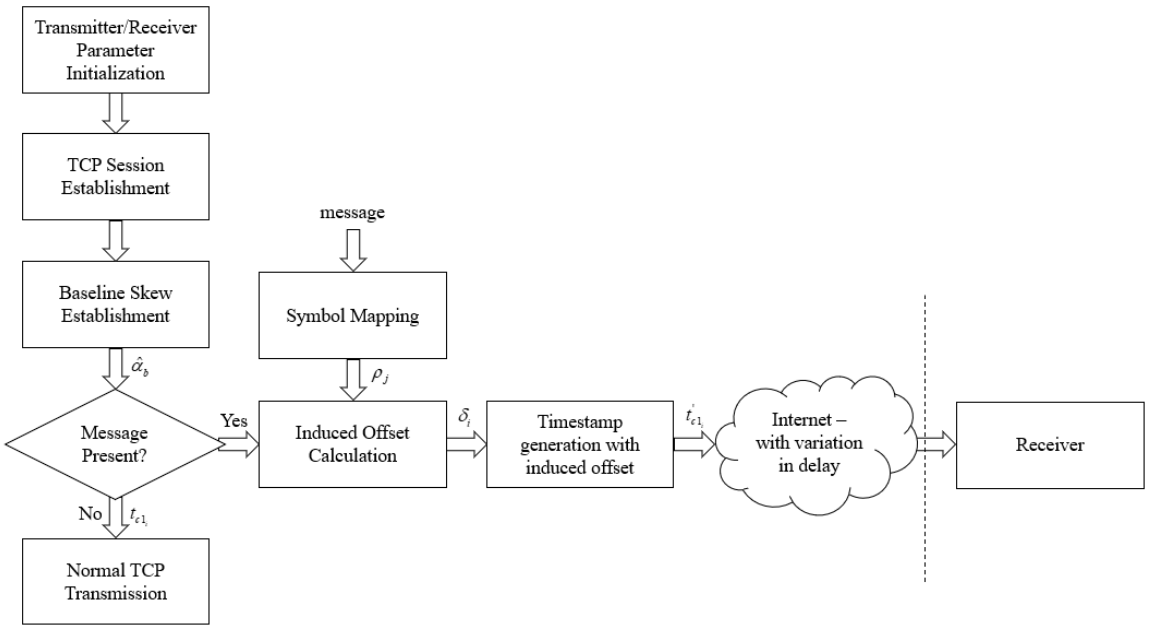
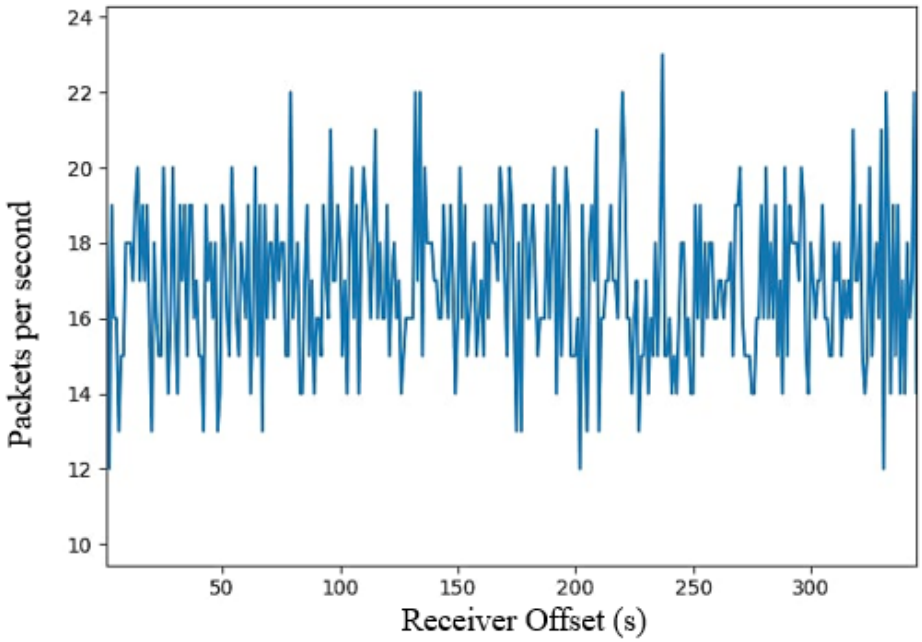


Figure 20. Transmission Flowchart

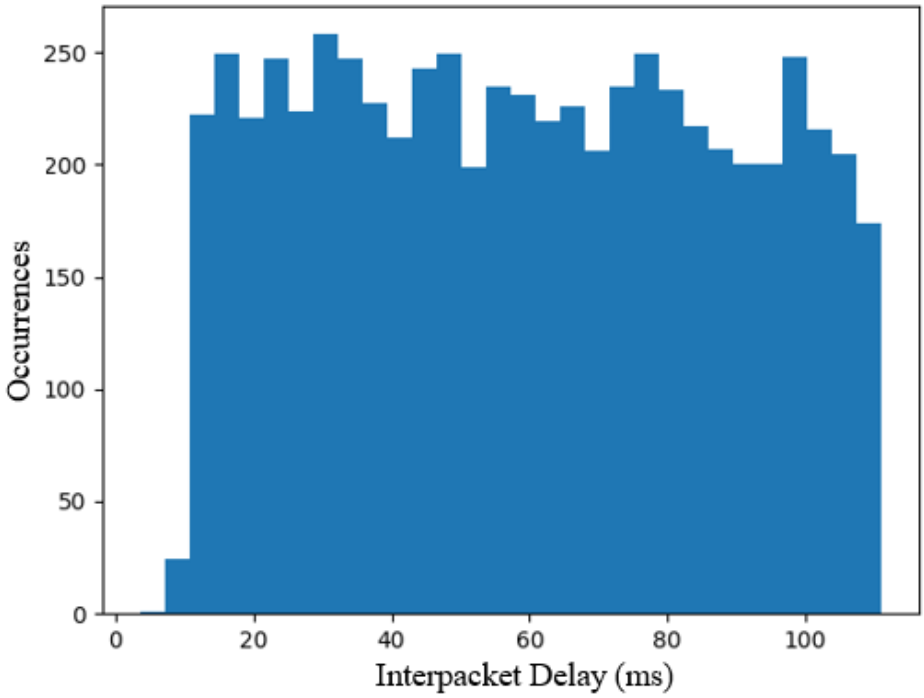
Packets were transmitted at either a constant interpacket interval of 10.0 ms or uniformly distributed random interpacket intervals of 10.0 ms to 110.0 ms. All results obtained with constant interpacket delay were also verified with uniformly distributed random interpacket delay. The random interpacket delay is illustrated by Figure 21. A plot of packets received each second and a histogram of interpacket delay for 6400 packets transmitted with uniformly distributed random interpacket delay are displayed in Figure 21(a) and (b), respectively.

Randomized interpacket intervals were chosen in order to demonstrate the scheme's resilience against non-steady arrival of packets. This parameter also serves to differentiate this scheme from the covert timing channels described in [4], [2], and [7].

It is important to note that interpacket delay and travel time are different quantities, and the scheme presented is still vulnerable to variations in travel time.



a) Quantity of Packets Received each Second.



b) Histogram of Interpacket Delay.

Figure 21. Received Packet Rate and Interpacket Delay Histogram for 6400 Packets Transmitted with Uniformly Distributed Random Interpacket Delay

Through trial and error, we chose $n_b = 150$ packets in our testing in order to balance the minimization of overhead and accurate estimation of $\hat{\alpha}_b$. Although n_b has an impact on channel capacity according to (10), it was not varied along with other parameters during testing because of its negligible impact on bit rate and no relation to likelihood of detection. As discussed in Chapter II, a minimum of 70 packets are required for a reasonably accurate calculation of clock skew [17]. Also discussed was the requirement of timestamp values spanning 5 to 10 minutes [15]. This requirement was not considered because the use of a high-frequency **Tsopt** clock allowed for quicker calculations and the fact that the receiver need not maximize accuracy for correct calculation provided the inaccuracy is not greater than the allowable error. Initially, we set $\zeta = 2500$ PPM and $n_m = 50$ packets. We then systematically lowered them in an effort to optimize bit rate and covertness. These values were sufficiently conservative to produce zero errors after hundreds of trials and, therefore, served as a starting point for fine tuning.

The transmitter was programmed in C with the exception of error correction, which utilized a MATLAB function to generate the Reed–Solomon code. Additionally, a Python GUI was developed to aid in the rapid repeat of trials. To implement the raw socket, we used C code developed by P. Buchanan [23]. A sample of the transmitter code used to add offset to the outgoing TCP timestamp is displayed in Figure 22. The variable names in the displayed code follow the convention of this thesis with the exception of 'end_long', which is the TCP timestamp. The code in Figure 22 follows (13) and (17) for the calculation of added offset. The factor of 1000000 in the code is to convert units from microseconds to seconds.

```
if(i>150){
          Delta_rho=Delta_rho+us_t;
          if(q==1){
                    if(i<nm){
                              end_long=end_long+((bit_string[j])*zeta*Delta_rho/1000000);
                              if(bit_string[j]==0){end_long=end_long-zeta* Delta_rho/1000000;}}}
                              i++;
```

Figure 22. Code Sample for Addition of Offset to Timestamp

For simplicity, the transmitter did not go through the three-way TCP handshake for establishing a TCP session but instead sent each successive packet as a new synchronization (SYN) request to a different destination port. Although the intended application for this concept is within a single TCP session, the establishment of one is not necessary for the full demonstration of this idea conceptually and the achievement of this thesis' objectives. Because each packet transmitted was a SYN packet, each packet contained no payload data. To provide a finite estimate of channel capacity at the experimentally verified values of q, ζ, and n_m, a payload size of 1460 bytes was assumed in (10) in the results section of this chapter.

2. Receiver

The extraction process at the receiver is illustrated in the flowchart in Figure 23. After receiving packets across the network with added noise, the receiver uses the received timestamps to estimate drift and then skew. If embedding is detected, the receiver subtracts the value it estimated for base skew and performs A/D conversion on the skew estimates. The symbols obtained through A/D conversion are then mapped back to bits.

The receiver collected incoming TCP/IP packets with Wireshark/Tshark. A sample Wireshark collection from one of the trials is displayed in Figure 24 for purposes of illustration. The value in the 'Time' column corresponds to the receiver timestamp for each received packet, whereas the highlighted 'Timestamp value' in the packet details pane corresponds to the transmitter timestamp for the single highlighted packet in the packet pane.

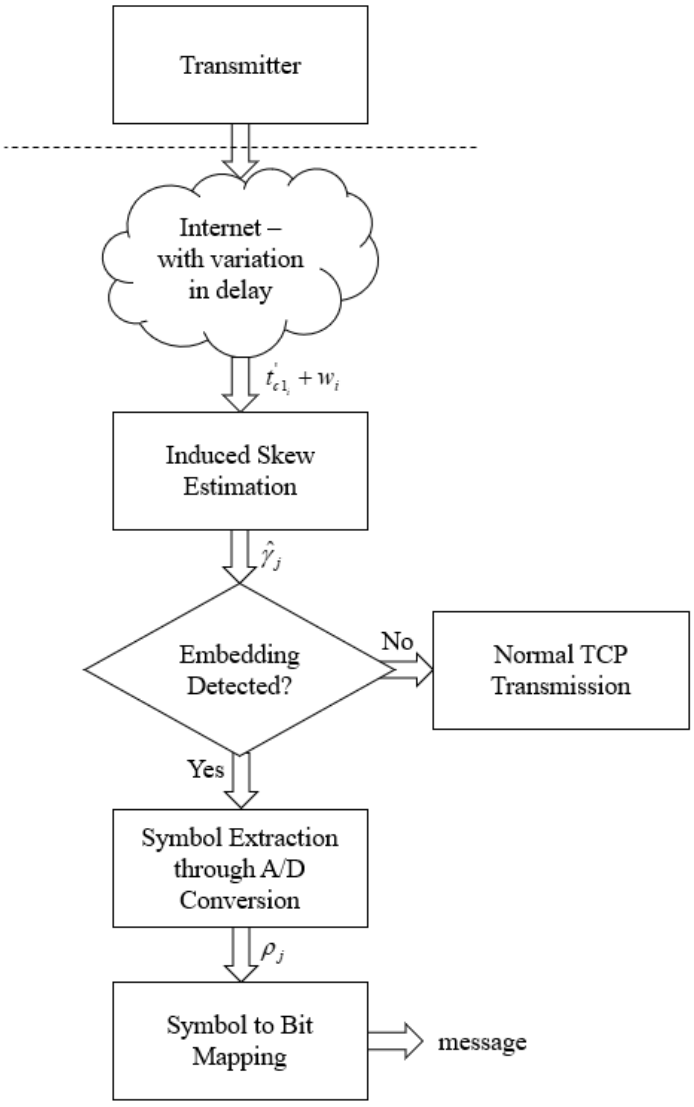


Figure 23. Receiver Flowchart

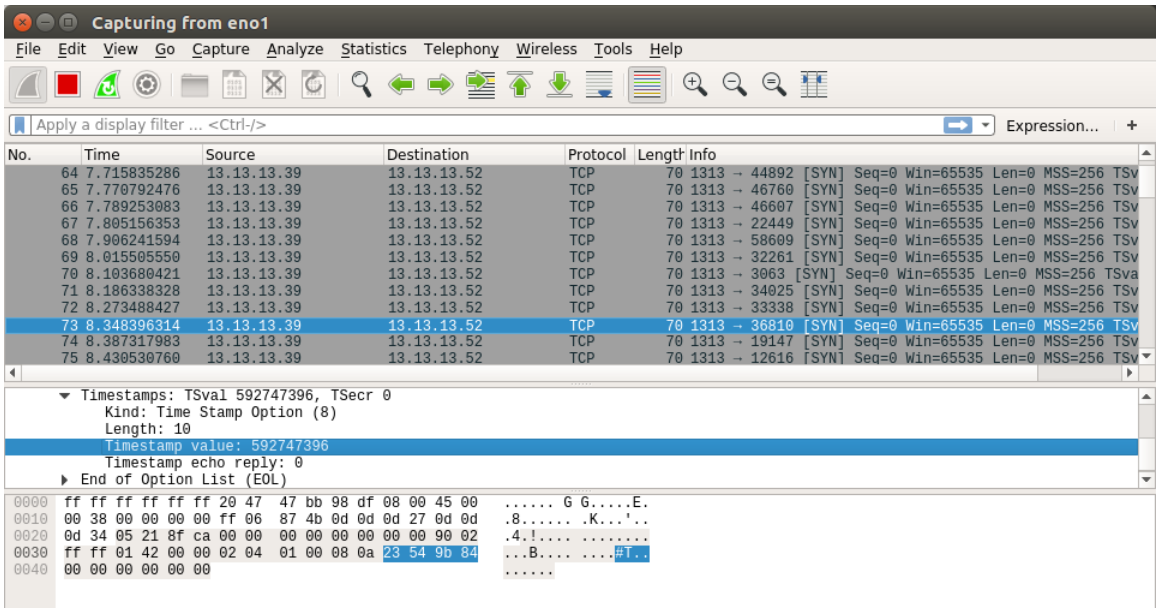


Figure 24. Wireshark Collection of TCP/IP Packets

The receiver was programmed in Python and utilized the Python Matplotlib for generating data plots. As an example, the lines of code that implement of (2) and (3) for calculating drift at the receiver are displayed in Figure 25.

```
for i in range(0,len(tc2)):
    rc2+=[tc2[i]-tc2[0]]
    rc1+=[(Tsval[i]-Tsval[0])/time_factor]
    drift+=[rc1[i]-rc2[i]]
```

Figure 25. Code Utilized for Drift Calculation

The variable ‘time_factor’ accounts for the fact that the two clocks may not be using the same units of measurement.

3. Limitations

The frequency of **Tsopt** clock was set to 1.0 MHz for the experiments conducted in this thesis, whereas real-world **Tsopt** clock frequencies range from 1.0 Hz to 1.0 kHz. The low resolution of real-world **Tsopt** clocks is the reason for the requirement of timestamps spanning at least five minutes for accurate skew calculation [15].

This departure from reality does not undermine the scheme presented in this work; increasing ζ allows for skew to be detected at the rates demonstrated even when utilizing a **Tsopt** clock with frequency less than or equal to 1.0 kHz. While the requirement that timestamps spanning five to ten minutes are collected for an accurate skew calculation holds [15], an accurate calculation is not necessary for the scheme presented in this thesis. To extract the message symbols, the calculated skew needs only to fall within threshold values that can be adjusted as necessary. For example, if timestamps spanning 30 seconds of receiver offset are collected in a scheme using a 1.0-kHz **Tsopt** clock, we may only be able to calculate skew with a resolution of 500 PPM. To create threshold values wide enough to accommodate a realistically slow **Tsopt** clock's limitations on skew calculation, we can increase ζ considerably. This increase in ζ allows for comparable bit rates to those seen in this work, although it may increase the likelihood of detection.

B. RESULTS

We first tested the proposed scheme using conservative parameters. Once we demonstrated successful operation, we lowered parameter values in an attempt to maximize bit rate and minimize likelihood of detection. Finally, we lowered parameter values until the BER became prohibitively large to test the limits of this implementation.

1. Conservative Parameters

The scheme operated as desired, achieving the main thesis objective of proof of concept. In over 100 trials at each value of q tested, messages embedded with the conservative parameters of $n_b = 150$ packets, n_m = 50 packets, and ζ = 2500 PPM were correctly extracted by the receiver. Calculated over 1000 symbols, use of these parameters achieved a SER of zero and, consequently, a BER of zero for $q = 1, 2, 4.$ In Table 2, the bit

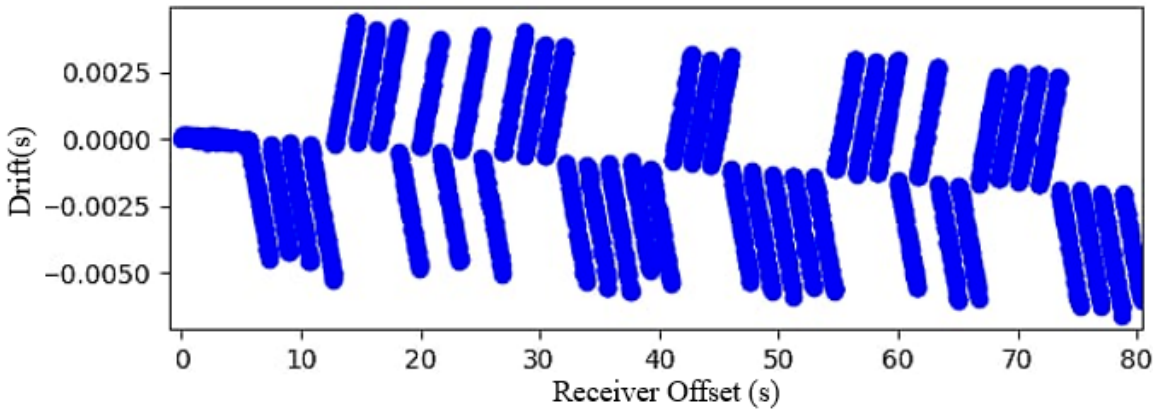
rate and channel capacity at these parameter values are provided according to (10) and (11) for each of the three values of q tested. The calculations in Table 2 assume a constant interpacket interval of 10.0 ms, a cover payload of 15.0 MB, and $\nu_p = 1460$ for the maximum payload over Ethernet.

Table 2. Bit Rate and Channel Capacity for n_m = 50 packets and ζ = 2500 PPM by q

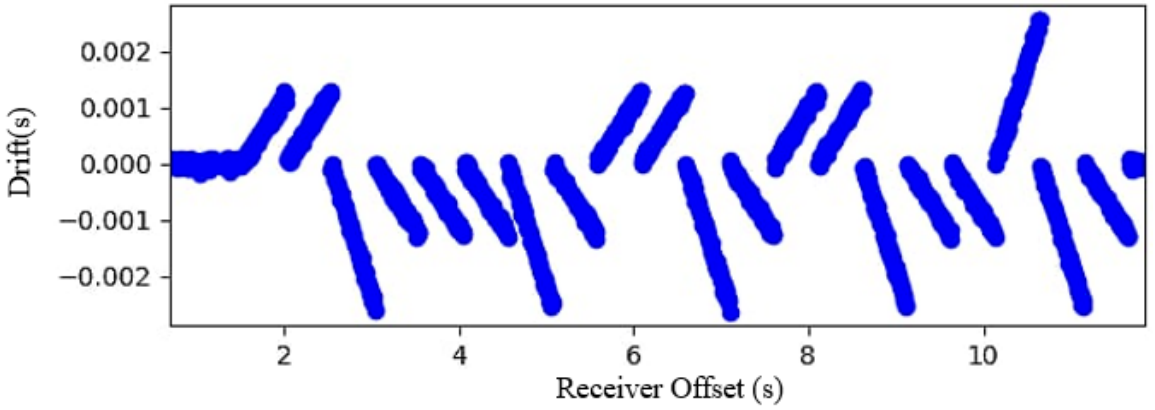
	$q=1$	$q=2$	$q=4$
bit rate (bps)	2	4	8
channel capacity (bits)	201	402	804

Plots of receiver offset versus drift are provided in Figure 25 for each of the three q values tested. In each of these figures, the waveforms corresponding to each symbol can be clearly seen due to the high level of skew induced. The differing symbol set sizes are also clearly evident from visual observation of the quantity of slopes present. Although already confirmed by the BER, these figures provide visual observation that noise was not a significant factor at these parameter values.

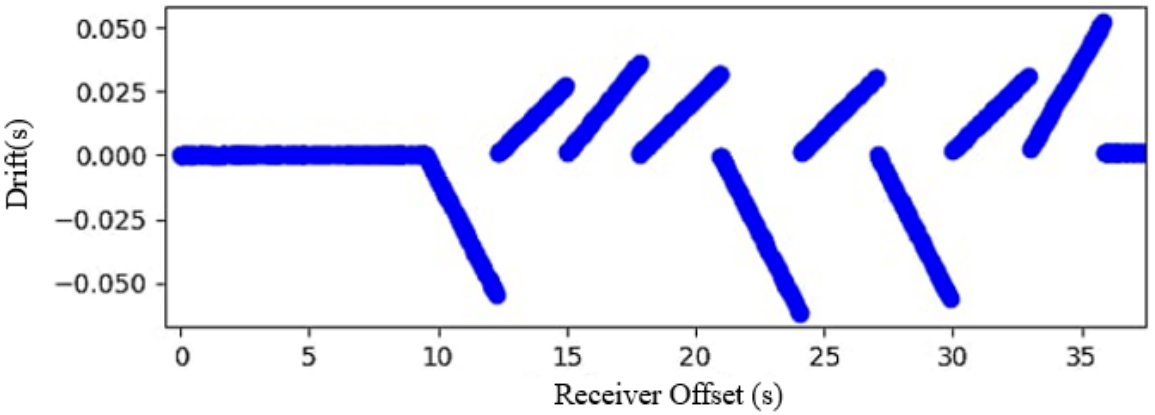
The estimated skew values for the trials shown in Figure 26 are depicted in Figure 27. Because of the large value for base induced skew, these skew estimates are cleanly centered in the bins created by (19) with the exception of the trial for $q = 1,$ which has a single threshold. In Figures 27, each blue 'x' denotes a skew estimate, while the red dashed line represents the thresholds used for A/D conversion provided by (19). Each skew estimate is centered at the receiver offset of the n_m packets used for its calculation.



a) $q = 1$

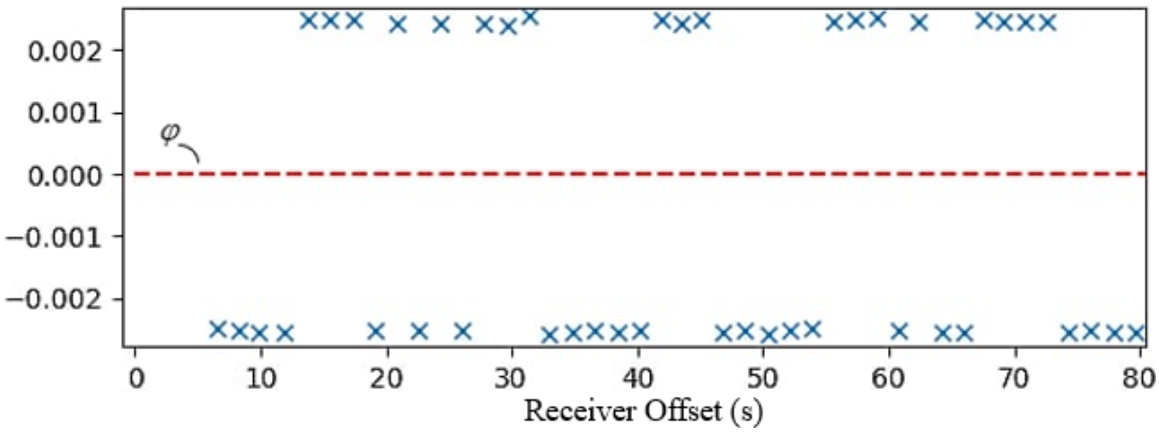


b) $q = 2$

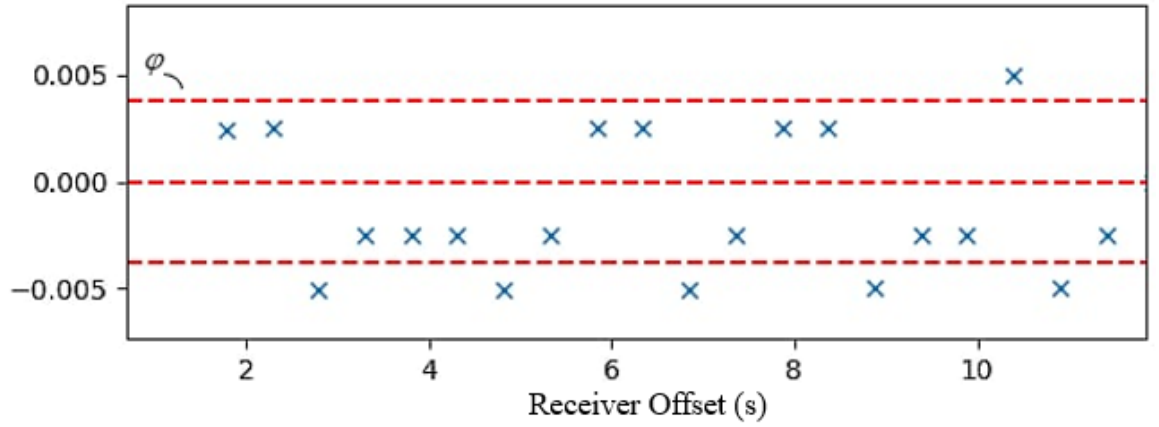


c) $q = 4$

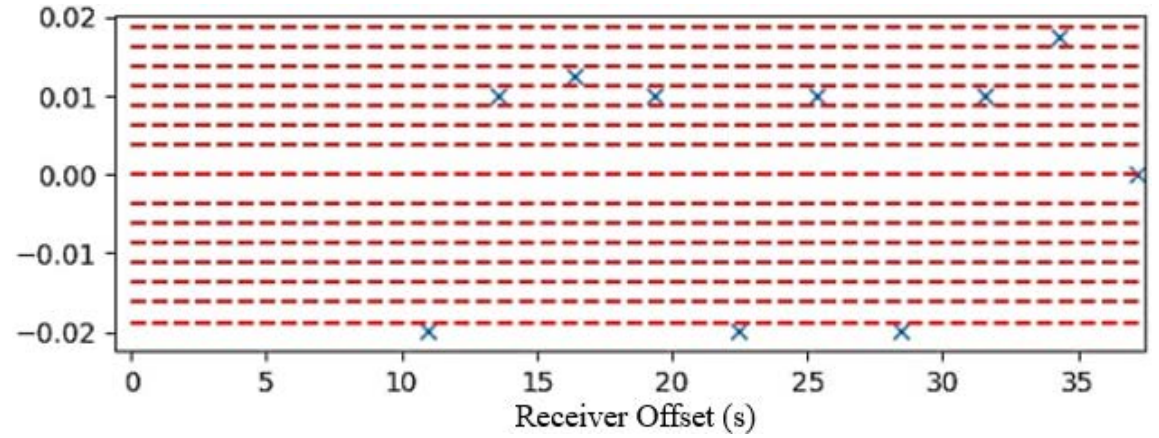
Figure 26. Plots of $\hat{d}$ for Three Trials with $\zeta = 2500$ PPM, n_m = 50 Packets



a) $q = 1$



b) $q = 2$



c) $q = 4$

Figure 27. Plots of $\hat{\gamma}$ for Three Trials with $\zeta = 2500$ PPM, n_m = 50 Packets

Although the presence of a covert message is observable from the waveforms in Figures 25, to the casual observer comparing offset, the activity is invisible. To illustrate this point, a plot of receiver offset versus transmitter offset is displayed in Figure 28 for a trial with the parameters listed in Table 2. The steady progression of transmitter offset even at a high level of induced skew shows that the channel remains hidden to the casual observer. The timestamps with embedding still appear to be constantly progressing and are not conspicuous without further investigation.

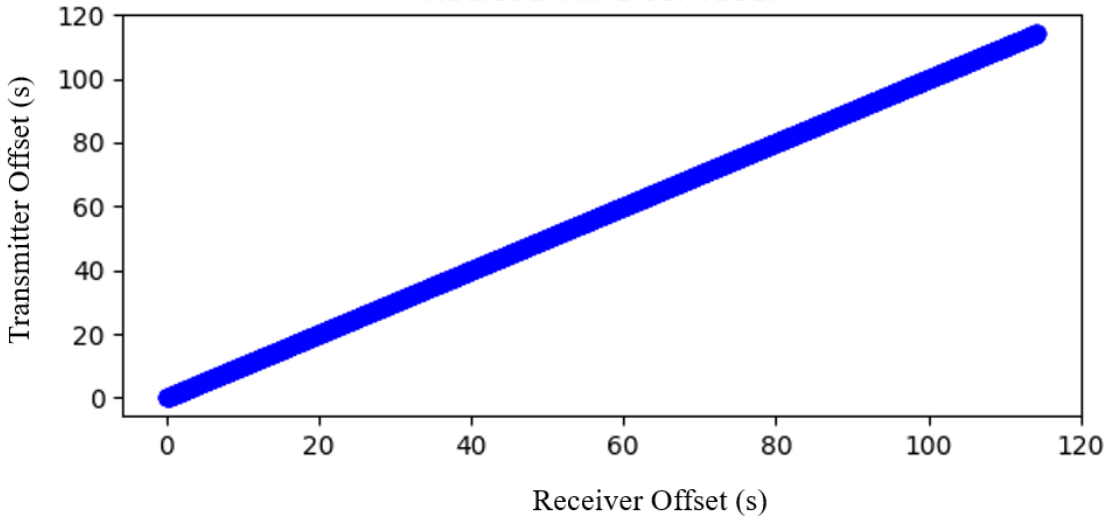


Figure 28. Plot of r_{c1} versus r_{c2} for $q = 1$, $n_m = 50$ packets, $\zeta = 2500$ PPM

2. Minimum Induced Skew Level

By lowering the value of the base induced skew and keeping n_m constant at n_m = 50, we attempted to conceal the signal in the noise. The lowest values of ζ for which we can consistently extract the message are contained in Table 3.

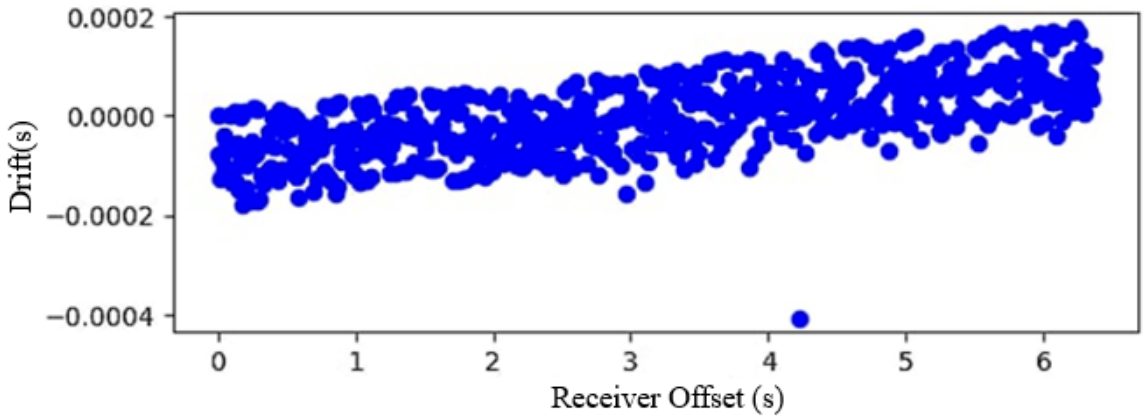
Table 3. Minimum Induced Skew by q

	$q=1$	$q=2$	$q=4$
ζ (PPM)	25	100	100

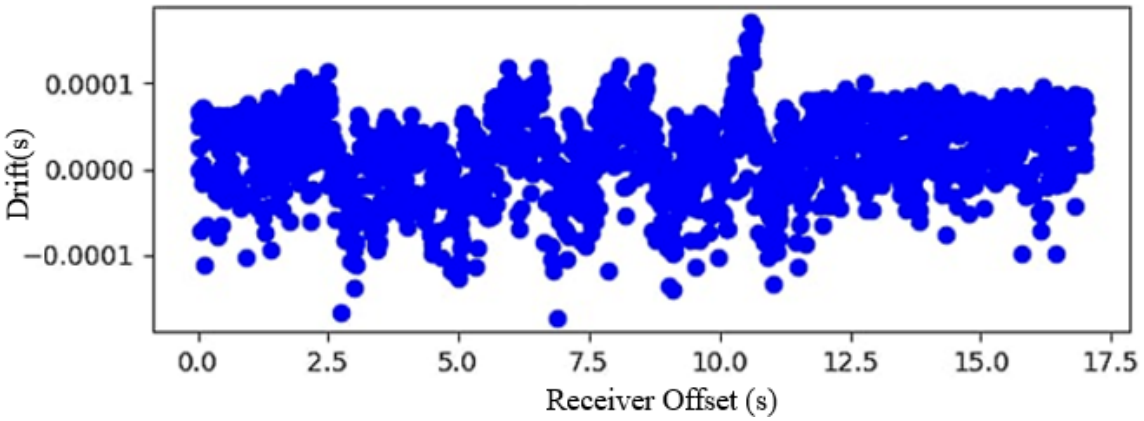
The drift and skew estimate of trials conducted with low base induced skew values according to Table 3 are depicted in Figure 29 and Figure 30, respectively. Although for $q=4$ consistent successful extraction was not demonstrated for $\zeta<100$ PPM, the trial depicted in Figures 29 and 30 had low enough noise that successful extraction was performed for $\zeta=50$ PPM.

Unlike the trials depicted in Figure 26 where $\zeta=2500,$ in Figure 29, the signal is hidden in the noise even for the observer that has gone far enough to calculate d. The waveforms in Figure 29(a) are visually indistinguishable from the drift estimates of unaltered timestamps. This signal was extracted at the receiver with zero symbol errors; however, several skew calculations were close to the threshold value as can be seen in the plot of $\hat{\gamma}$ shown in Figure 30.

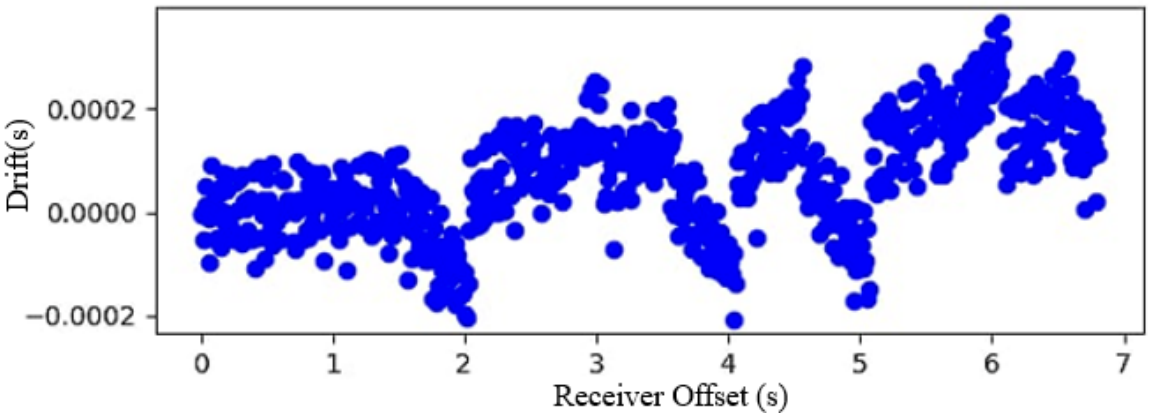
With a lower value of $\zeta,$ noise plays a much greater factor in extraction. In Figure 30, this is shown by the proximity of the skew estimates to the threshold values for each bin. Although there were no symbol errors in the three trials depicted, several estimates are very close to the threshold as compared to the trials depicted in Figure 27, in which each skew estimate is cleanly centered in its bin.



a) $q = 1,\ \zeta = 25$ PPM

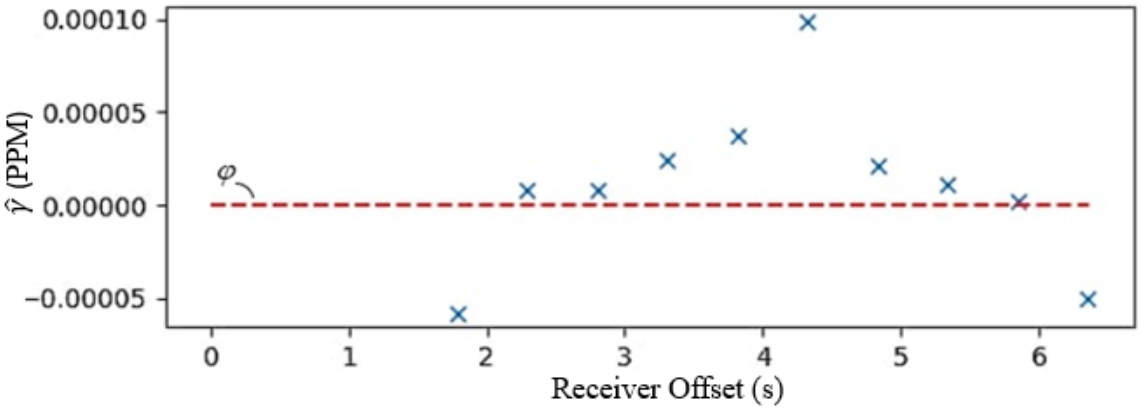


b) $q = 2,\ \zeta = 100$ PPM

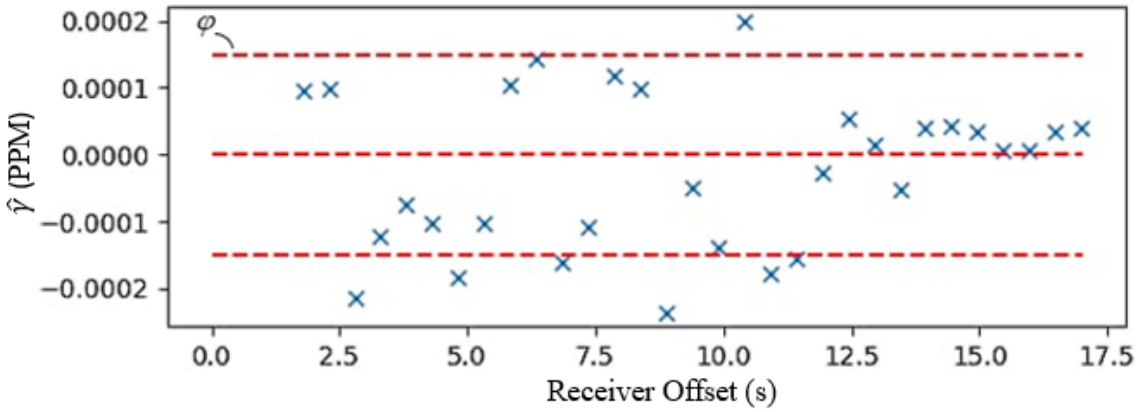


c) $q = 4,\ \zeta = 100$ PPM

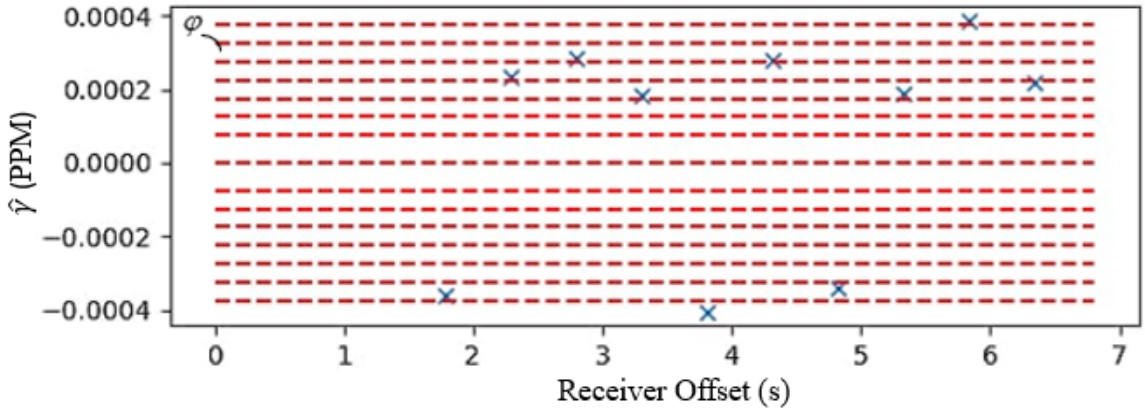
Figure 29. Plots of $\hat{d}$ for Three Trials with $n_m = 50$ Packets, and Varying ζ



a) $q = 1,\ \zeta = 25$ PPM



b) $q = 2,\ \zeta = 100$ PPM



c) $q = 4,\ \zeta = 100$ PPM

Figure 30. Plots of $\hat{\gamma}$ for Three Trials with $n_m = 50$ Packets, and Varying ζ

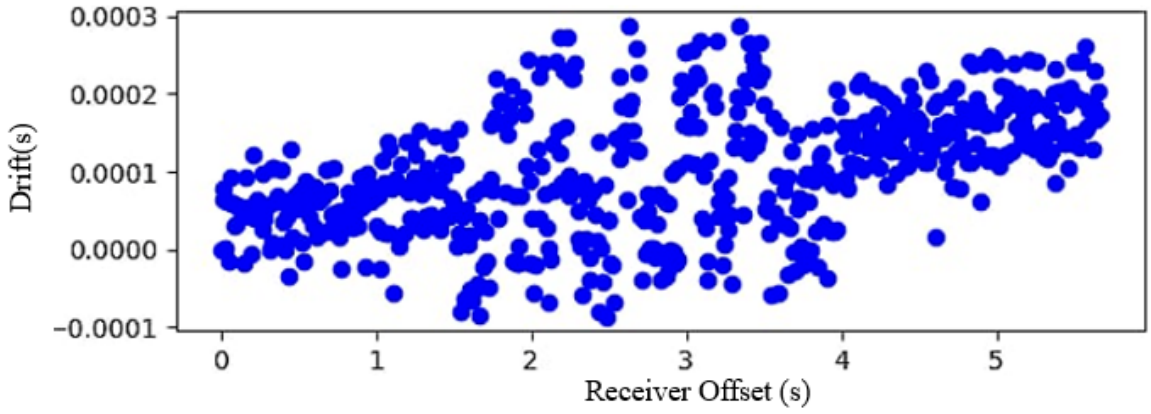
3. Maximum Bit Rate

By lowering the number of packets used to embed each skew value, we attempted to maximize the bit rate and channel capacity according to (10) and (11). The smallest values of n_m for which consistent extraction of the message was still possible are contained in Table 4. The calculation in Table 4 assume the same values for cover payload and interpacket delay as Table 2.

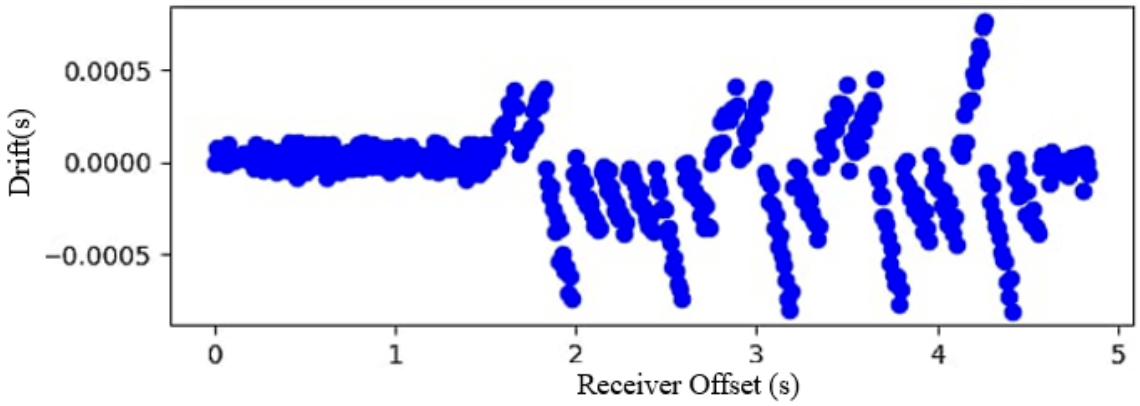
Table 4. Minimum n_m by q

	$q = 1$	$q = 2$	$q = 4$
n_m (packets)	5	15	15
bit rate (bps)	20	13.3	26.6
channel capacity (bits)	2024	1349	2699

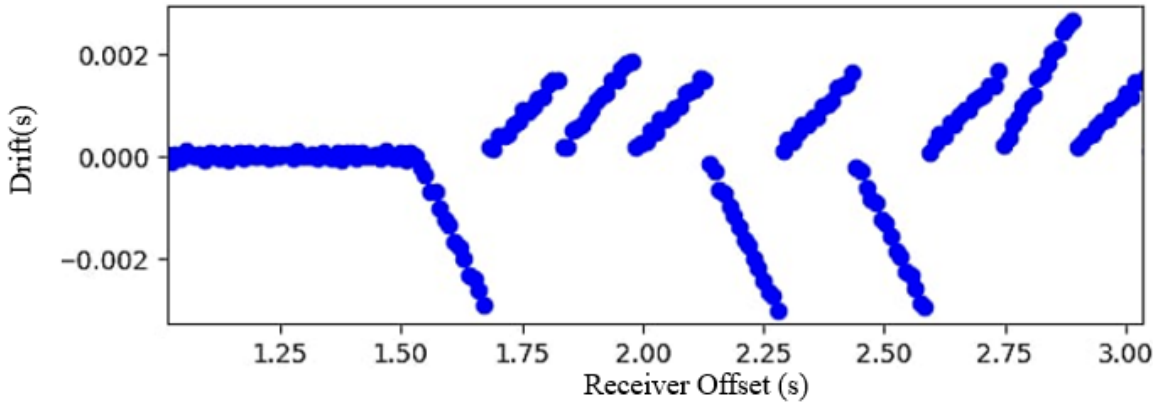
Plots of drift for trials run with the parameter values in Table 4 are shown in Figure 31. The differing values of n_m in Table 4 and Figure 31 are accounted for by the fact that that the implementation had greater noise resistance at $q = 1,$ as discussed in Chapter III and could, thus, successfully extract the message at a lower value of n_m. Although the intention was to increase bit rate, the drift waveforms for $q = 1$ are difficult to discern from the noise due to the low value of n_m, potentially making the presence of a covert message harder for an adversary to detect. The plots of estimated skew for the same trials are shown in Figure 32.



a) $q = 1,\ n_m = 5$

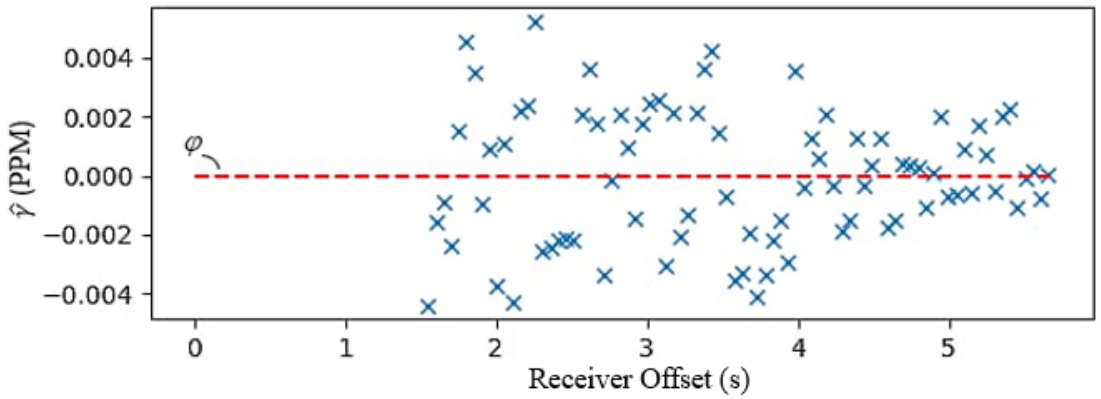


b) $q = 2,\ n_m = 15$

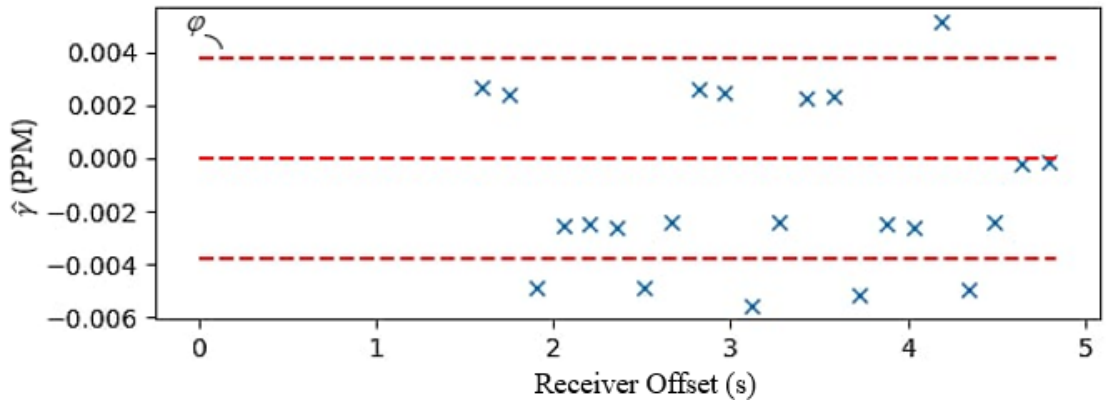


c) $q = 4,\ n_m = 15$

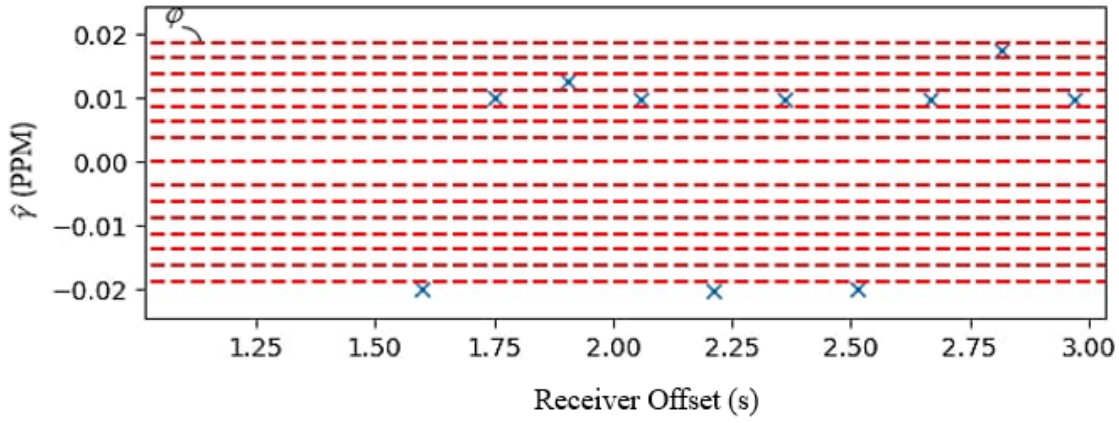
Figure 31. Plots of $\hat{d}$ for Three Trials with $\zeta = 2500$ PPM, and Varying n_m



a) $q = 1,\ n_m = 5$



b) $q = 2,\ n_m = 15$



c) $q = 4,\ n_m = 15$

Figure 32. Plot of $\hat{\gamma}$ for Three Trials with $\zeta = 2500$ PPM, Varying n_m

In the trial depicted in Figure 31 (a) and Figure 32 (a), the use of only five packets in each skew estimate is nearly hidden in the noise, indicating that a low small value of n_m may also aid in covertness. Successful extraction at this small n_m value is a significant result because it is very close to the theoretical limit of two packets and demonstrates that the requirement for 70 packets [15] is not necessary for the operation of this scheme. Although noise does not appear to be affecting the skew estimates much in Figure 32 (b) and (c), the trials displayed were relatively low-noise compared to all trials conducted.

Lowering either n_m or ζ precluded consistent successful message extraction for $n_m < 5$ packets or ζ < 25 PPM when $q = 1$ and for $n_m < 25$ packets or ζ < 100 PPM when $q > 1$. While we did observe some correlation between the impact of n_m and ζ on SER, raising the value of either of these parameters did not provide the ability to successfully extract with $n_m < 5$ or ζ < 25 PPM at any value of q. As an example, the skew estimates for a trial in which successful extraction did not occur is depicted in Figure 33.

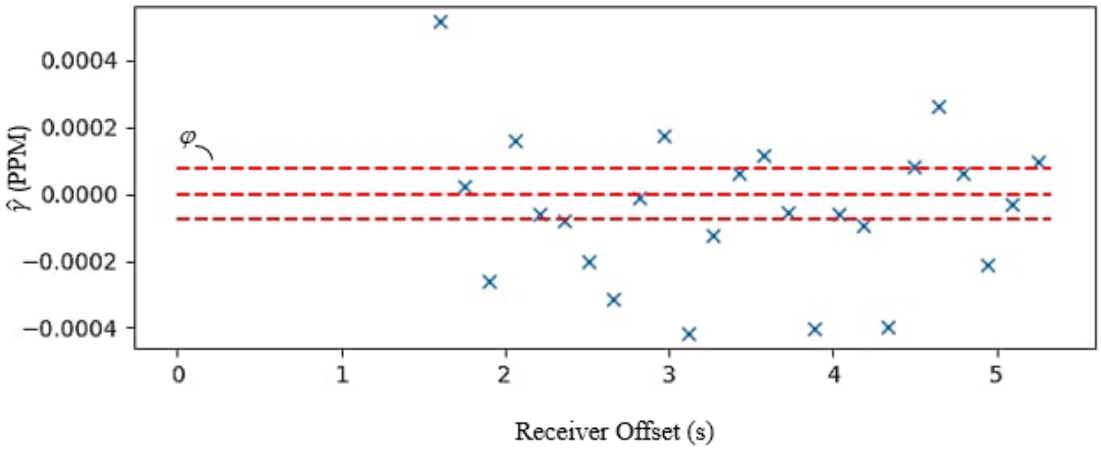


Figure 33. Plot of $\hat{\gamma}$ for $q = 2$, $n_m = 15$ Packets, $\zeta = 50$ PPM

The skew estimates in Figure 33 vary wildly; some have values many times greater than the level of skew induced. This level of noise did not allow successful extraction.

4. Error Detection

We employed CRC-9 error detection with each of the schemes presented and successfully detected errors. The FCS was created at the transmitter with C and appended to the message bits before conversion to symbols occurred. After mapping the received symbols to bits, the receiver calculated a FCS code based on the received message bits with Python and compared this value to the received FCS to determine if an error occurred. The scheme utilized the generator polynomials '111111111', '110100111', and '000101110'.

Though the use of CRC was an effective means of detecting errors, the low data rate and channel capacity of all schemes presented made the employment of ARQ an inefficient option.

5. Error Correction

We used Reed–Solomon code for FEC on the implementation with $q = 4$. The Matlab functions *rsenc* and *rsdec* provided easy generation and decoding of the Reed–Solomon code. Because the bits were already grouped into symbols through the symbol mapping process, four bits per symbol was chosen for *m.* This allowed a maximum block size of 15 symbols. In order to provide the ability to correct up to three symbol errors, we chose to use nine message symbols and six parity symbols per block. After the conversion of message bits to symbols, the transmitter calculated and appended six Reed–Solomon parity symbols to the message symbols. Before mapping back to bits, the receiver used the Matlab function *rsdec* to identify symbol errors in the nine message symbols. Using Reed–Solomon code, we successfully corrected symbol errors at parameter values $q = 4,\ \zeta = 35$ PPM, and n_m = 50 packets. The results of one such trial is illustrated in Figure 34. The yellow circled skew value is the detected symbol error.

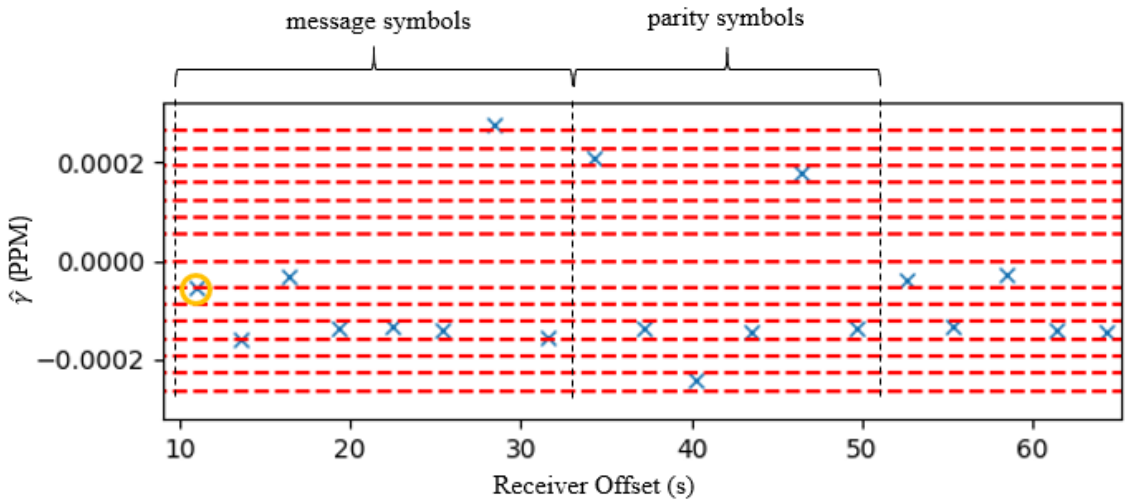


Figure 34. Error Correction for $q = 4$, $\zeta = 35$ PPM, and n_m = 50 Packets

Though the Reed–Solomon code proved effective at correcting symbol errors in the scheme presented, most trials that contained symbol errors had too many errors for any type of FEC to be employed, including errors in the FEC code itself. For example, with $q = 1$ we could extract a message at $\zeta = 2500$ PPM and n_m = 6 packets essentially error free. Changing n_m to n_m = 4 can cause as many as half of the bits to be transmitted incorrectly. In these scenarios, the FEC functioned as error detection and retransmission was necessary. FEC was employed most effectively at the parameter values listed in Table 3 and Table 4 with $q = 4$. At these values, the FEC was able to identify and correct the symbol errors that occurred.

C. DISCUSSION

Each of the schemes operated as designed when given conservative parameter values and resulted in a SER and BER of zero taken over 1000 symbols. Adjusting the value of n_m, we verified that it was possible to increase bit rate as proposed in Chapter III. Adjusting ζ values, we attempted to make the signal less detectable, even to an adversary observing drift and skew.

The maximum achievable bit rate has an upper theoretical limit because at least two TCP timestamps are required for every skew calculation. This means that at most, a single

symbol can be transmitted with every two TCP timestamps. With $q = 1$, we were able to get very close to this theoretical limit, using only five timestamps for each skew calculation, greatly undercutting the experimentally determined requirement in [15] for 70 timestamps.

Setting $q = 1$ provided greater noise resistance and, therefore, the ability to operate at a lower value of ζ and n_m. This counterintuitively led to the trials with $q = 1$ achieving a higher maximum bit rate than that achieved in trials with $q = 2$. Likewise, only for $q = 1$ was the implementation able to visually hide in the noise after drift and skew calculations. Discernable waveforms were visible for all values $q > 1$, even when ζ was minimized.

The BER of each implementation as a function of n_m is depicted in Figure 35. Once n_m was lowered below 30, bit errors began to occur for all q values.

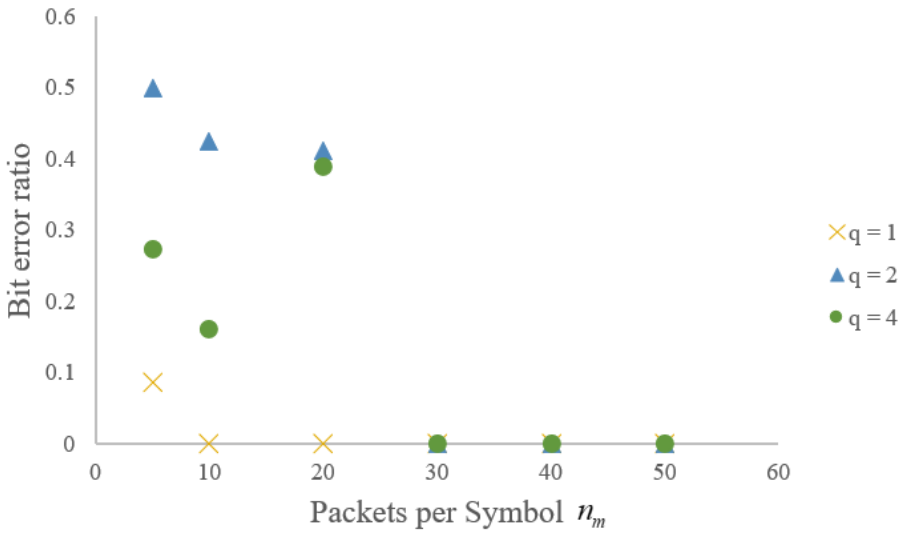


Figure 35. BER by n_m for Varying q, with ζ =2500 PPM

The BER at each value of q also increased as ζ decreased. Values $\zeta < 100$ PPM resulted in increasing bit errors at each value of q. The BER of each implementation as a function of ζ is depicted in Figure 36. Unexpectedly, at each value of ζ where bit errors were observed, more were observed for $q = 2$ than for $q = 4$.

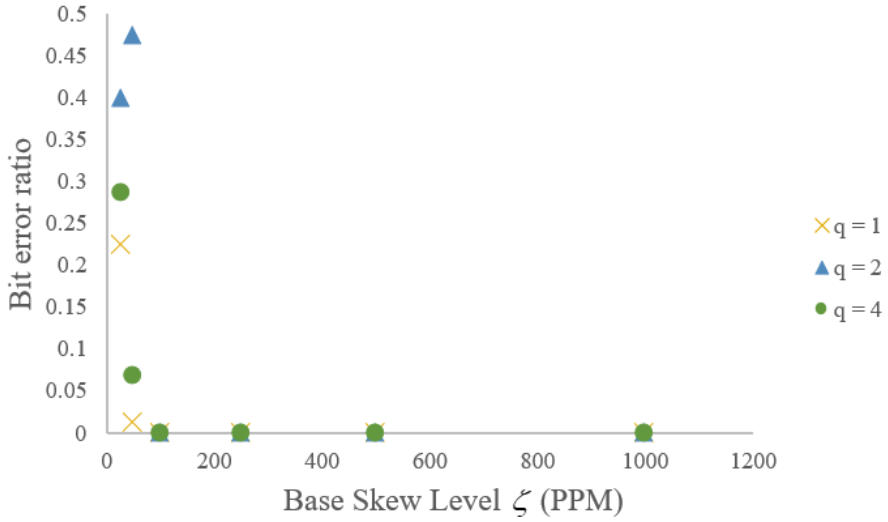


Figure 36. BER by ζ for Varying q, $n_m = 50$

Even after minimizing ζ, an adversary with knowledge of the scheme would still be able to detect its presence, especially if n_m were deduced. A histogram of received skew calculations would reveal the presence of Q distinct skews. One possible counter for detection might be varying ζ, n_m, and n_b according to a shared key within a range of acceptable values.

The effects of ζ and n_m on SER and BER appeared to be correlated at values close to their minimum; however, boosting either value did not produce an ability to further lower the other. For example, the receiver was not able to successfully extract the message for ζ = 100 PPM and n_m = 20; however, in previous trials, signals with ζ = 100 PPM given large n_m and signals at n_m = 20 packets given large ζ were successfully extracted. Values of ζ and n_m were likewise boosted to values up to 5000 PPM and 100 packets,

respectively, in an attempt to go below the minimum values presented in the results. In these trials, the receiver was still unsuccessful in extracting the message signal.

In this chapter, we described the test bed used for experimentation and presented results of the experiments. We demonstrated the successful operation of each implementation along with results of attempts to minimize various parameter values with the intent of increasing bit rate and channel capacity and decreasing the likelihood of detection. We then provided discussion of the results. In the following chapter, we conclude this thesis and provide recommendations for future work.

THIS PAGE INTENTIONALLY LEFT BLANK

V. CONCLUSION

In accordance with the thesis objectives, we developed a novel scheme that utilizes clock skew to transmit secret messages. This required the development of a method for converting symbols into skew values and then a method for transmitting those skew values. The method developed is analogous to modulation in which the induced skew is the message and the baseline skew is the carrier. It also required the development of a way to extract the skew values from received packets and perform A/D conversion to retrieve the message. Implementation of the scheme successfully demonstrated the ability to use induced clock skew as a covert channel over TCP. Our results provide a proof-of-concept for the use of induced clock skew for covert communication.

We improved bit rate and channel capacity by determining the lower limits of parameter values for successful message extraction. By implementing larger symbol sets, we utilized the analog properties of clock skew to increase the number of bits communicated by each induced skew value, which serves as a guidepost for further exploration leading to higher bit rates.

A. SIGNIFICANT RESULTS

In this thesis, we developed a novel scheme that provides the ability to covertly communicate via induced clock skew. The scheme was developed in a general way to allow for variations in parameter values such as symbol set size. This has not been previously achieved and contributes to the existing body of knowledge concerning covert communications. This scheme may serve as a starting point for study of a subset of covert channels fundamentally different in concept from existing schemes that rely on embedding bits or varying temporal parameters.

To validate the proposed scheme, we designed, implemented, and tested a first-of-its-kind program capable of sending covert messages described by the scheme. This program worked as desired, and we were able to successfully embed and extract covert messages over a TCP connection.

We have added a forward error correction code scheme based on a Reed–Solomon code to improve the performance of the proposed scheme and to mitigate the effects of travel time variations in the Internet. The forward error correction was successfully employed and provided the ability to transmit at higher bit rate and lower levels of induced skew than would have been possible otherwise.

B. FUTURE WORK

We were able to demonstrate the ability to induce clock skew in TCP timestamps, however, we did so using a **Tsopt** clock with 1.0 MHz resolution. The use of a 1.0 MHz resolution clock for **Tsopt** clock is the largest departure from real world conditions in this work. In a future effort, we recommend testing with a **Tsopt** clock having a resolution of 1.0 kHz or lower in order to present TCP timestamp values that appear routine and check the feasibility of employment under real world conditions. Reducing the **Tsopt** clock's frequency could reduce the bit rate of the proposed scheme, perhaps significantly; however, this limitation may be mitigated by the use of large ζ at the expense of increasing likelihood of detection.

Our scheme was tested in a laboratory environment with minimal traffic. This traffic-free laboratory environment provided lower noise than can be expected in real-world conditions. Travel time was relatively constant between the two hosts, and queuing delay was nonexistent because the hosts were connected through a switch. Conducting this experiment in a high-traffic environment would introduce variable travel time that could greatly impact the ability to calculate induced skew at the receiver. Additionally, conducting this experiment across multiple autonomous systems (AS) would determine its utility for real world covert communication.

The highest order implementation tested in this work consisted of a 16-symbol set. The ability to clearly distinguish these 16 unique skew values indicated that still larger values of q are feasible. A 256-symbol set would allow for each skew to convey a byte of data and roughly double the maximum data rate and channel capacity observed in this work. Larger symbol sets may even be possible. In a future effort, we recommend testing the upper limit for symbol set size.

Although the scheme presented in this work is not susceptible to the methods of stegoanalysis presented in Chapter II, it has its own weaknesses. A rigorous study of techniques for countering the proposed scheme would allow further refinement of the ideas presented in this work. Varying n_m according to a shared key combined with small ζ has the potential to make a covert message hidden in induced skew difficult to detect. This idea and other methods of decreasing the likelihood of detection warrant further exploration.

The experiments conducted in [15] and [16] highlight the non-constant clock skew produced by certain OS's when measured with the methods developed in [3]. A scheme could be developed to operate on these OS's specifically. This scheme would naturally be more resilient to detection and analysis due to the adversaries' inability to predict skew from these systems.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. TRANSMITTER CODE

The following code implements the transmitter used for the testing and experimentation presented in Chapter IV. The function *CRC* computes a CRC-9 FCS value and appends it to the message. In *main*, code developed by [23] is used to open a raw socket and fill in the values for the Ethernet, IP, and TCP headers. User input is solicited for parameter values q, n_m, ζ, and the message in American Standard Code for Information Interchange (ascii). The ascii message is converted to bits and then symbols depending on the value of q. An infinite while loop maintains transmission until user intervention. Each iteration of the while loop indexes n_m nested in the array of symbols to determine the offset needed for each outgoing timestamp.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>          // close()
#include <string.h>         // strcpy, memset(), and memcpy()
#include <sys/time.h>      // for gettime of day
#include <netdb.h>          // struct addrinfo
#include <sys/types.h>     // needed for socket(), uint8_t,
uint16_t, uint32_t
#include <sys/socket.h>       // needed for socket()
#include <netinet/in.h>      // IPPROTO_TCP, INET_ADDRSTRLEN
#include <netinet/ip.h>       // struct ip and IP_MAXPACKET
(which is 65535)
#define __FAVOR_BSD          // Use BSD format of tcp header
#include <netinet/tcp.h>      // struct tcphdr
#include <arpa/inet.h>        // inet_pton() and inet_ntop()
#include <sys/ioctl.h>        // macro ioctl is defined
#include <bits/ioctls.h>      // defines values for argument
"request" of ioctl.
#include <net/if.h>           // struct ifreq
#include  <linux/if_ether.h>       //  ETH_P_IP  =  0x0800,
ETH_P_IPV6 = 0x86DD
#include <linux/if_packet.h>  // struct sockaddr_ll (see man
7 packet)
#include <net/ethernet.h>
# include <unistd.h>                //for sleep func
#include <errno.h>            // errno, perror()
#include <math.h>             //needed for floor
```

```
#define ETH_HDRLEN 14  // Ethernet header length
#define IP4_HDRLEN 20  // IPv4 header length
#define TCP_HDRLEN 20  // TCP header length, excludes options
data
#define MAX_STRING_LEN 20

// Function prototypes
uint16_t checksum (uint16_t *, int);
uint16_t tcp4_checksum (struct ip, struct tcphdr, uint8_t *,
int);
char *allocate_strmem (int);
char **allocate_strmemp (int);
uint8_t *allocate_ustrmem (int);
uint8_t **allocate_ustrmemp (int);
int *allocate_intmem (int);

int added_code[8]={0,0,0,0,0,0,0,0};
int
bit_string[640]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};


//CRC FCS Calculation
int CRC(int msg_len)
```

```
{
int i=0;
int j=0;
int msg_bin[809];
int CRC_code[9]={1,1,1,1,1,1,1,1,1};

for(int i=0; i<(msg_len+9); i++){
if(i<msg_len){msg_bin[i]=bit_string[i];}
else{msg_bin[i]=0;}}

i=0;
//i represents a right shift of the CRC code
while (i<(msg_len))
{
printf("\ni loop top, msg_bin[%d]=%d", i, msg_bin[i]);
if (msg_bin[i]==1)
{
//j represents the xor of each digit
j=0;
while (j<9){
     if (CRC_code[j]==1)
{
          if (msg_bin[i+j]==0)
               {msg_bin[i+j]=1;}
          else if (msg_bin[i+j]==1)
               {msg_bin[i+j]=0;}}
j++;}}
i++;

added_code[0]=msg_bin[msg_len];
added_code[1]=msg_bin[msg_len+1];
added_code[2]=msg_bin[msg_len+2];
added_code[3]=msg_bin[msg_len+3];
added_code[4]=msg_bin[msg_len+4];
added_code[5]=msg_bin[msg_len+5];
added_code[6]=msg_bin[msg_len+6];
added_code[7]=msg_bin[msg_len+7];
return 0;}

//Converts input arguements from string to int variable type
int str_to_int(char *st1){
int result, dec=0,i,j,len;
len=strlen(st1);
for(i=0;i<len;i++){dec=dec*10+(st1[i]-'0');}
return dec;}
```

```
int
main (int argc, char *argv[])
{int i, j, k, l, m, mm, n, c, status, frame_length, sd, bytes,
*ip_flags,  *tcp_flags,  nopt,  *opt_len,  buf_len,  newport,
bit_count, nm, zeta, msg_pack_count, CRC_result[8], Q4[800],
Q16[400], q, wait_time, wait_rand;
  char *interface, *target, *src_ip, *dst_ip,*message_temp,
message[MAX_STRING_LEN];
  struct timeval t1, t2; //for time elapsed
  double elapsedTime, start, end, start_s, end_s, start_us,
end_us, end_us_old, us_delta, Delta_rho; // for time elapsed
  struct ip iphdr;
  struct tcphdr tcphdr;
  uint8_t       *src_mac,       *dst_mac,       *ether_frame,
*opt1,*opt2,*opt3,*opt4;
  uint8_t **options, *opt_buffer;
  struct addrinfo hints, *res;
  struct sockaddr_in *ipv4;
  struct sockaddr_ll device;
  struct ifreq ifr;
  void *tmp;
  unsigned long start_long, end_long;
  //char *msg_input = (char*) malloc( 100 );

  // Allocate memory for various arrays.
  src_mac = allocate_ustrmem (6);
  dst_mac = allocate_ustrmem (6);
  ether_frame = allocate_ustrmem (IP_MAXPACKET);
  interface = allocate_strmem (40);
  target = allocate_strmem (40);
  src_ip = allocate_strmem (INET_ADDRSTRLEN);
  dst_ip = allocate_strmem (INET_ADDRSTRLEN);
  ip_flags = allocate_intmem (4);
  tcp_flags = allocate_intmem (8);
  opt_len = allocate_intmem (10);
  options = allocate_ustrmemp (10);
  for (l=0; l<10; l++) {
    options[l] = allocate_ustrmem (40);}
  opt_buffer = allocate_ustrmem (40);

  // Interface to send packet through.
  strcpy (interface, "eno1");

  // Submit request for a socket descriptor to look up
interface.
```

```
  if ((sd = socket (PF_PACKET, SOCK_RAW, htons (ETH_P_ALL)))
< 0) {
    perror ("socket() failed to get socket descriptor for
using ioctl() ");
    exit (EXIT_FAILURE);}

  // Use ioctl() to look up interface name and get its MAC
address.
  memset (&ifr, 0, sizeof (ifr));
  snprintf  (ifr.ifr_name,  sizeof  (ifr.ifr_name),  "%s",
interface);
  if (ioctl (sd, SIOCGIFHWADDR, &ifr) < 0) {
    perror ("ioctl() failed to get source MAC address ");
    return (EXIT_FAILURE);}
  close (sd);

  // Copy source MAC address.
  memcpy  (src_mac,  ifr.ifr_hwaddr.sa_data,  6  *  sizeof
(uint8_t));

  // Report source MAC address to stdout.
  printf ("MAC address for interface %s is ", interface);
  for (l=0; l<5; l++) {
    printf ("%02x:", src_mac[l]);}
  printf ("%02x\n", src_mac[5]);

  // Find interface index from interface name and store index
in
  // struct sockaddr_ll device, which will be used as an
argument of sendto().
  memset (&device, 0, sizeof (device));
  if ((device.sll_ifindex = if_nametoindex (interface)) == 0)
{
    perror  ("if_nametoindex()  failed  to  obtain  interface
index ");
    exit (EXIT_FAILURE);
  }
  printf  ("Index  for  interface  %s  is  %i\n",  interface,
device.sll_ifindex);

  // Set destination MAC address: you need to fill these out
  dst_mac[0] = 0xff;
  dst_mac[1] = 0xff;
  dst_mac[2] = 0xff;
  dst_mac[3] = 0xff;
  dst_mac[4] = 0xff;
```

```
  dst_mac[5] = 0xff;

  // Source IPv4 address: you need to fill this out
  strcpy (src_ip, "13.13.13.39");

  // Destination URL or IPv4 address: you need to fill this
out
  strcpy (target, "13.13.13.52");

  // Fill out hints for getaddrinfo().
  memset (&hints, 0, sizeof (struct addrinfo));
  hints.ai_family = AF_INET;
  hints.ai_socktype = SOCK_STREAM;
  hints.ai_flags = hints.ai_flags | AI_CANONNAME;

  // Resolve target using getaddrinfo().
  if ((status = getaddrinfo (target, NULL, &hints, &res)) !=
0) {
    fprintf   (stderr,   "getaddrinfo()   failed:   %s\n",
gai_strerror (status));
    exit (EXIT_FAILURE);
  }
  ipv4 = (struct sockaddr_in *) res->ai_addr;
  tmp = &(ipv4->sin_addr);
  if (inet_ntop (AF_INET, tmp, dst_ip, INET_ADDRSTRLEN) ==
NULL) {
    status = errno;
    fprintf  (stderr,  "inet_ntop()  failed.\nError  message:
%s", strerror (status));
    exit (EXIT_FAILURE);
  }
  freeaddrinfo (res);

  // Fill out sockaddr_ll.
  device.sll_family = AF_PACKET;
  memcpy (device.sll_addr, src_mac, 6 * sizeof (uint8_t));
  device.sll_halen = 6;

message_temp=*argv[1];
for(l=0;l<=strlen(argv[1]);l++){message[l]=argv[1][l];}
zeta=str_to_int(argv[2]);
nm=str_to_int(argv[3]);
q=str_to_int(argv[4]);

for (l=0;l<=strlen(message);l++)
```

```
{
     j=0;
n=message[l];
while (j<8)
{
     bit_string[(i*8)+j]=(n>>j)%2;

j++;}}

//Calculating total packets to contain message information so
we can stop inducing the skew after CRC is sent

if(q==1){msg_pack_count=(((strlen(message)*8)+8)*nm);}
else if(q==2){msg_pack_count=(((strlen(message)*4)+4)*nm);}
else if(q==4){msg_pack_count=(((strlen(message)*2)+2)*nm);}

//CRC-9 calculation

CRC((strlen(message)*8));
//CRC value is contained in added_code(global) and calculaed
in CRC
bit_string[(strlen(message)*8)]=added_code[0];
bit_string[(strlen(message)*8)+1]=added_code[1];
bit_string[(strlen(message)*8)+2]=added_code[2];
bit_string[(strlen(message)*8)+3]=added_code[3];
bit_string[(strlen(message)*8)+4]=added_code[4];
bit_string[(strlen(message)*8)+5]=added_code[5];
bit_string[(strlen(message)*8)+6]=added_code[6];
bit_string[(strlen(message)*8)+7]=added_code[7];

//Created the array of message symbols for q=2
for(m=0;m<(strlen(message)*8);m=m+2)
{
if (bit_string[m]==0){
//00
if(bit_string[m+1]==0){Q4[m/2]=1;}
//01
else if (bit_string[m+1]==1){Q4[m/2]=-1;}
}

else if(bit_string[m]==1){
//11
if(bit_string[m+1]==0){Q4[m/2]=-2;}
//10
else if (bit_string[m+1]==1){Q4[m/2]=2;}}}
```

```
for(mm=0;mm<(strlen(message)*8);mm=mm+4){

//Creates the array of message symbols for q=4
if (bit_string[mm]==0){
if(bit_string[mm+1]==0){
if(bit_string[mm+2]==0){
//0000
if(bit_string[mm+3]==0){Q16[mm/4]=-8;}
//0001
else if (bit_string[mm+3]==1){Q16[mm/4]=-7;}}

else if(bit_string[mm+2]==1){
//0010
if(bit_string[mm+3]==0){Q16[mm/4]=-5;}
//0011
else if (bit_string[mm+3]==1){Q16[mm/4]=-6;}}}

else if (bit_string[mm+1]==1){

if(bit_string[mm+2]==0){
//0100
if(bit_string[mm+3]==0){Q16[mm/4]=-1;}
//0101
else if (bit_string[mm+3]==1){Q16[mm/4]=-2;}}

else if(bit_string[mm+2]==1){
//0110
if(bit_string[mm+3]==0){Q16[mm/4]=-4;}
//0111
else if (bit_string[mm+3]==1){Q16[mm/4]=-3;}}}}

else if (bit_string[mm]==1){
if(bit_string[mm+1]==0){
if(bit_string[mm+2]==0){
//1000
if(bit_string[mm+3]==0){Q16[mm/4]=8;}
//1001
else if (bit_string[mm+3]==1){Q16[mm/4]=7;}}

else if(bit_string[mm+2]==1){
//1010
if(bit_string[mm+3]==0){Q16[mm/4]=5;}
//1011
else if (bit_string[mm+3]==1){Q16[mm/4]=6;}}}

else if (bit_string[mm+1]==1){
```

```
if(bit_string[mm+2]==0){
//1100
if(bit_string[mm+3]==0){Q16[mm/4]=1;}
//1101
else if (bit_string[mm+3]==1){Q16[mm/4]=2;}}

else if(bit_string[mm+2]==1){
//1110
if(bit_string[mm+3]==0){Q16[mm/4]=4;}
//1111
else if (bit_string[mm+3]==1){Q16[mm/4]=3;}}}}}

 // Number of TCP options
  nopt = 2;

  // First TCP option - Maximum segment size
  opt_len[0] = 0;
  options[0][0] = 2u; opt_len[0]++;   // Option kind 2 =
maximum segment size
  options[0][1] = 4u; opt_len[0]++;  // This option kind is
4 bytes long
  options[0][2] = 0x1u; opt_len[0]++;  // Set maximum segment
size to 0x100 = 256
  options[0][3] = 0x0u; opt_len[0]++;

//setting first timestamp to startime
gettimeofday(&t1, NULL);
start_s=t1.tv_sec;
start_us=t1.tv_usec;

c=floor(start_s/10000);

start_s=start_s-(c)*10000;
start=start_s*1000000+start_us;
start_long=floor(start);
opt1=&start_long;
opt2=opt1+1;
opt3=opt2+1;
opt4=opt3+1;

  // Second TCP option - Timestamp option
  opt_len[1] = 0;
  options[1][0] = 8u; opt_len[1]++;   // Option kind 8 =
Timestamp option (TSOPT)
```

```
  options[1][1] = 10u; opt_len[1]++;  // This option is 10
bytes long
  options[1][2] = *opt4; opt_len[1]++;  // Set the sender's
timestamp (TSval) (4 bytes) (need SYN set to be valid)
  options[1][3] = *opt3; opt_len[1]++;
  options[1][4] = *opt2; opt_len[1]++;
  options[1][5] = *opt1; opt_len[1]++;

options[1][6] = 0x0u; opt_len[1]++;  // Set the echo timestamp
(TSecr) (4 bytes) (need ACK set to be valid)
  options[1][7] = 0x0u; opt_len[1]++;
  options[1][8] = 0x0u; opt_len[1]++;
  options[1][9] = 0x0u; opt_len[1]++;

  // Copy all options into single options buffer.
  buf_len = 0;
  c = 0;  // index to opt_buffer
  for (l=0; l<nopt; l++) {
    memcpy (opt_buffer + c, options[l], opt_len[l] * sizeof
(uint8_t));
    c += opt_len[l];
    buf_len += opt_len[l];}

  // Pad to the next 4-byte boundary.
  while ((buf_len%4) != 0) {
    opt_buffer[buf_len] = 0;
    buf_len++;}

  // IPv4 header

  // IPv4 header length (4 bits): Number of 32-bit words in
header = 5
  iphdr.ip_hl = IP4_HDRLEN / sizeof (uint32_t);

  // Internet Protocol version (4 bits): IPv4
  iphdr.ip_v = 4;

  // Type of service (8 bits)
  iphdr.ip_tos = 0;

  // Total length of datagram (16 bits): IP header + TCP
header + TCP options
  iphdr.ip_len = htons (IP4_HDRLEN + TCP_HDRLEN + buf_len);

  // ID sequence number (16 bits): unused, since single
datagram
```

```
  iphdr.ip_id = htons (0);

  // Flags, and Fragmentation offset (3, 13 bits): 0 since
single datagram

  // Zero (1 bit)
  ip_flags[0] = 0;

  // Do not fragment flag (1 bit)
  ip_flags[1] = 0;

  // More fragments following flag (1 bit)
  ip_flags[2] = 0;

  // Fragmentation offset (13 bits)
  ip_flags[3] = 0;

  iphdr.ip_off = htons ((ip_flags[0] << 15)
                      + (ip_flags[1] << 14)
                      + (ip_flags[2] << 13)
                      +  ip_flags[3]);

  // Time-to-Live (8 bits): default to maximum value
  iphdr.ip_ttl = 255;

  // Transport layer protocol (8 bits): 6 for TCP
  iphdr.ip_p = IPPROTO_TCP;

  // Source IPv4 address (32 bits)
  if ((status = inet_pton (AF_INET, src_ip, &(iphdr.ip_src)))
!= 1) {
    fprintf  (stderr,  "inet_pton()  failed.\nError  message:
%s", strerror (status));
    exit (EXIT_FAILURE);}

  // Destination IPv4 address (32 bits)
  if ((status = inet_pton (AF_INET, dst_ip, &(iphdr.ip_dst)))
!= 1) {
    fprintf  (stderr,  "inet_pton()  failed.\nError  message:
%s", strerror (status));
    exit (EXIT_FAILURE);}

  // IPv4 header checksum (16 bits): set to 0 when calculating
checksum
  iphdr.ip_sum = 0;
  iphdr.ip_sum = checksum ((uint16_t *) &iphdr, IP4_HDRLEN);
```

```
  // TCP header

  // Source port number (16 bits)
  tcphdr.th_sport = htons (1313);

  // Destination port number (16 bits)
  tcphdr.th_dport = htons (80);

  // Sequence number (32 bits)
  tcphdr.th_seq = htonl (0);

  // Acknowledgement number (32 bits): 0 in first packet of
SYN/ACK process
  tcphdr.th_ack = htonl (0);

  // Reserved (4 bits): should be 0
  tcphdr.th_x2 = 0;

  // Data offset (4 bits): size of TCP header + length of
options, in 32-bit words
  tcphdr.th_off = (TCP_HDRLEN  + buf_len) / 4;

  // Flags (8 bits)

  // FIN flag (1 bit)
  tcp_flags[0] = 0;

  // SYN flag (1 bit): set to 1
  tcp_flags[1] = 1;

  // RST flag (1 bit)
  tcp_flags[2] = 0;

  // PSH flag (1 bit)
  tcp_flags[3] = 0;

  // ACK flag (1 bit)
  tcp_flags[4] = 0;

  // URG flag (1 bit)
  tcp_flags[5] = 0;

  // ECE flag (1 bit)
  tcp_flags[6] = 0;
```

```
  // CWR flag (1 bit)
  tcp_flags[7] = 0;

  tcphdr.th_flags = 0;
  for (i=0; i<8; i++) {
    tcphdr.th_flags += (tcp_flags[i] << i);
  }

  // Window size (16 bits)
  tcphdr.th_win = htons (65535);

  // Urgent pointer (16 bits): 0 (only valid if URG flag is
set)
  tcphdr.th_urp = htons (0);

  // TCP checksum (16 bits)
  tcphdr.th_sum = tcp4_checksum (iphdr, tcphdr, opt_buffer,
buf_len);

  // Fill out ethernet frame header.

  // Ethernet frame length = ethernet header (MAC + MAC +
ethernet type) + ethernet data (IP header + TCP header + TCP
options)
  frame_length = 6 + 6 + 2 + IP4_HDRLEN + TCP_HDRLEN +
buf_len;

  // Destination and Source MAC addresses
  memcpy (ether_frame, dst_mac, 6 * sizeof (uint8_t));
  memcpy (ether_frame + 6, src_mac, 6 * sizeof (uint8_t));

  // Next is ethernet type code (ETH_P_IP for IPv4).
  // http://www.iana.org/assignments/ethernet-numbers
  ether_frame[12] = ETH_P_IP / 256;
  ether_frame[13] = ETH_P_IP % 256;

  // Next is ethernet frame data (IPv4 header + TCP header).

  // IPv4 header
  memcpy (ether_frame + ETH_HDRLEN, &iphdr, IP4_HDRLEN *
sizeof (uint8_t));

  // TCP header
  memcpy (ether_frame + ETH_HDRLEN + IP4_HDRLEN, &tcphdr,
TCP_HDRLEN * sizeof (uint8_t));
```

```
  // TCP Options
  memcpy (ether_frame + ETH_HDRLEN + IP4_HDRLEN + TCP_HDRLEN,
opt_buffer, buf_len * sizeof (uint8_t));

  // Submit request for a raw socket descriptor.
  if ((sd = socket (PF_PACKET, SOCK_RAW, htons (ETH_P_ALL)))
< 0) {
    perror ("socket() failed ");
    exit (EXIT_FAILURE);
  }

i=0;
bit_count=0;
Delta_rho=0;

//The embedding portion occurs within this loop
while(1){
newport=rand();
wait_rand=rand();
wait_rand=(wait_rand%100000)+10000;

  // Send ethernet frame to socket.
  if ((bytes = sendto (sd, ether_frame, frame_length, 0,
(struct sockaddr *) &device, sizeof (device))) <= 0) {
    perror ("sendto() failed");
    exit (EXIT_FAILURE);}

//Uniform Random interpacket delay 10-110 ms
usleep(wait_rand);

gettimeofday(&t2, NULL);

end_s=t2.tv_sec;
end_us=t2.tv_usec;
us_delta=floor(end_us)-floor(end_us_old);

c=floor(end_s/10000);
end_s=end_s-(c)*10000;
end=end_s*1000000+end_us;
us_delta=floor(end)-floor(end_us_old);
end_us_old=end;

end_long=floor(end);

if(i>150){
     Delta_rho=Delta_rho+us_t;
```

```
     if(q==1){
                    if(i<msg_pack_count){

     end_long=end_long+((bit_string[j])*zeta*Delta_rho/1000
000);
                if(bit_string[j]==0){end_long=end_long-
zeta*Delta_rho/1000000;}}}

     else if(q==2){

     if(i<msg_pack_count){end_long=end_long+zeta*(Delta_rho
/1000000)*Q4[bit_count];}}

     else if(q==4){

     if(i<msg_pack_count){end_long=end_long+zeta*(Delta_rho
/1000000)*Q16[bit_count];}}}

opt1=&end_long;
opt2=opt1+1;
opt3=opt2+1;
opt4=opt3+1;


i++;
if(i>150)
{if(i%nm==0)
{bit_count++;
Delta_rho=0;}}

//Changing destination port at random and redoing checksum
tcphdr.th_dport=htons(newport);
tcphdr.th_sum  =  tcp4_checksum  (iphdr,  tcphdr,  opt_buffer,
buf_len);

  options[1][2]  =  *opt4;    //  Set  the  sender's  timestamp
(TSval) (4 bytes)
  options[1][3] = *opt3;
  options[1][4] = *opt2;
  options[1][5] = *opt1;


  buf_len = 0;
  c = 0;  // index to opt_buffer
  for (l=0; l<nopt; l++) {
```

```
    memcpy (opt_buffer + c, options[l], opt_len[l] * sizeof
(uint8_t));
    c += opt_len[l];
    buf_len += opt_len[l];
  }

  // Pad to the next 4-byte boundary.
  while ((buf_len%4) != 0) {
    opt_buffer[buf_len] = 0;
    buf_len++;}


// TCP header
  memcpy (ether_frame + ETH_HDRLEN + IP4_HDRLEN, &tcphdr,
TCP_HDRLEN * sizeof (uint8_t));

  // TCP Options
  memcpy (ether_frame + ETH_HDRLEN + IP4_HDRLEN + TCP_HDRLEN,
opt_buffer, buf_len * sizeof (uint8_t));}

  // Close socket descriptor.
  close (sd);

  // Free allocated memory.
  free (src_mac);
  free (dst_mac);
  free (ether_frame);
  free (interface);
  free (target);
  free (src_ip);
  free (dst_ip);
  free (ip_flags);
  free (tcp_flags);
  free (opt_len);
  for (l=0; l<10; l++) {
    free (options[l]);
  }
  free (options);
  free (opt_buffer);

  return (EXIT_SUCCESS);
}

// Computing the internet checksum (RFC 1071).
//  Note  that  the  internet  checksum  does  not  preclude
collisions.
```

```
uint16_t
checksum (uint16_t *addr, int len)
{
  int count = len;
  register uint32_t sum = 0;
  uint16_t answer = 0;

  // Sum up 2-byte values until none or only one byte left.
  while (count > 1) {
    sum += *(addr++);
    count -= 2;
  }

  // Add left-over byte, if any.
  if (count > 0) {
    sum += *(uint8_t *) addr;
  }

  // Fold 32-bit sum into 16 bits; we lose information by
doing this,
  // increasing the chances of a collision.
  // sum = (lower 16 bits) + (upper 16 bits shifted right 16
bits)
  while (sum >> 16) {
    sum = (sum & 0xffff) + (sum >> 16);
  }

  // Checksum is one's compliment of sum.
  answer = ~sum;

  return (answer);
}

// Build IPv4 TCP pseudo-header and call checksum function.
uint16_t
tcp4_checksum (struct ip iphdr, struct tcphdr tcphdr, uint8_t
*options, int opt_len)
{
  uint16_t svalue;
  char buf[IP_MAXPACKET], cvalue;
  char *ptr;
  int chksumlen = 0;

  ptr = &buf[0];  // ptr points to beginning of buffer buf

  // Copy source IP address into buf (32 bits)
```

```
  memcpy      (ptr,      &iphdr.ip_src.s_addr,      sizeof
(iphdr.ip_src.s_addr));
  ptr += sizeof (iphdr.ip_src.s_addr);
  chksumlen += sizeof (iphdr.ip_src.s_addr);

  // Copy destination IP address into buf (32 bits)
  memcpy      (ptr,      &iphdr.ip_dst.s_addr,      sizeof
(iphdr.ip_dst.s_addr));
  ptr += sizeof (iphdr.ip_dst.s_addr);
  chksumlen += sizeof (iphdr.ip_dst.s_addr);

  // Copy zero field to buf (8 bits)
  *ptr = 0; ptr++;
  chksumlen += 1;

  // Copy transport layer protocol to buf (8 bits)
  memcpy (ptr, &iphdr.ip_p, sizeof (iphdr.ip_p));
  ptr += sizeof (iphdr.ip_p);
  chksumlen += sizeof (iphdr.ip_p);

  // Copy TCP length to buf (16 bits)
  svalue = htons (sizeof (tcphdr) + opt_len);
  memcpy (ptr, &svalue, sizeof (svalue));
  ptr += sizeof (svalue);
  chksumlen += sizeof (svalue);

  // Copy TCP source port to buf (16 bits)
  memcpy (ptr, &tcphdr.th_sport, sizeof (tcphdr.th_sport));
  ptr += sizeof (tcphdr.th_sport);
  chksumlen += sizeof (tcphdr.th_sport);

  // Copy TCP destination port to buf (16 bits)
  memcpy (ptr, &tcphdr.th_dport, sizeof (tcphdr.th_dport));
  ptr += sizeof (tcphdr.th_dport);
  chksumlen += sizeof (tcphdr.th_dport);

  // Copy sequence number to buf (32 bits)
  memcpy (ptr, &tcphdr.th_seq, sizeof (tcphdr.th_seq));
  ptr += sizeof (tcphdr.th_seq);
  chksumlen += sizeof (tcphdr.th_seq);

  // Copy acknowledgement number to buf (32 bits)
  memcpy (ptr, &tcphdr.th_ack, sizeof (tcphdr.th_ack));
  ptr += sizeof (tcphdr.th_ack);
  chksumlen += sizeof (tcphdr.th_ack);
```

```
  // Copy data offset to buf (4 bits) and
  // copy reserved bits to buf (4 bits)
  cvalue = (tcphdr.th_off << 4) + tcphdr.th_x2;
  memcpy (ptr, &cvalue, sizeof (cvalue));
  ptr += sizeof (cvalue);
  chksumlen += sizeof (cvalue);

  // Copy TCP flags to buf (8 bits)
  memcpy (ptr, &tcphdr.th_flags, sizeof (tcphdr.th_flags));
  ptr += sizeof (tcphdr.th_flags);
  chksumlen += sizeof (tcphdr.th_flags);

  // Copy TCP window size to buf (16 bits)
  memcpy (ptr, &tcphdr.th_win, sizeof (tcphdr.th_win));
  ptr += sizeof (tcphdr.th_win);
  chksumlen += sizeof (tcphdr.th_win);

  // Copy TCP checksum to buf (16 bits)
  // Zero, since we don't know it yet
  *ptr = 0; ptr++;
  *ptr = 0; ptr++;
  chksumlen += 2;

  // Copy urgent pointer to buf (16 bits)
  memcpy (ptr, &tcphdr.th_urp, sizeof (tcphdr.th_urp));
  ptr += sizeof (tcphdr.th_urp);
  chksumlen += sizeof (tcphdr.th_urp);

  // Copy TCP options to buf (variable length, but in 32-bit
chunks)
  memcpy (ptr, options, opt_len);
  ptr += opt_len;
  chksumlen += opt_len;

  return checksum ((uint16_t *) buf, chksumlen);
}

// Allocate memory for an array of chars.
char *
allocate_strmem (int len)
{
  void *tmp;

  if (len <= 0) {
    fprintf (stderr, "ERROR: Cannot allocate memory because
len = %i in allocate_strmem().\n", len);
```

```
    exit (EXIT_FAILURE);
  }

  tmp = (char *) malloc (len * sizeof (char));
  if (tmp != NULL) {
    memset (tmp, 0, len * sizeof (char));
    return (tmp);
  } else {
    fprintf (stderr, "ERROR: Cannot allocate memory for array
allocate_strmem().\n");
    exit (EXIT_FAILURE);
  }
}

// Allocate memory for an array of unsigned chars.
uint8_t *
allocate_ustrmem (int len)
{
  void *tmp;

  if (len <= 0) {
    fprintf (stderr, "ERROR: Cannot allocate memory because
len = %i in allocate_ustrmem().\n", len);
    exit (EXIT_FAILURE);
  }

  tmp = (uint8_t *) malloc (len * sizeof (uint8_t));
  if (tmp != NULL) {
    memset (tmp, 0, len * sizeof (uint8_t));
    return (tmp);
  } else {
    fprintf (stderr, "ERROR: Cannot allocate memory for array
allocate_ustrmem().\n");
    exit (EXIT_FAILURE);}}

// Allocate memory for an array of pointers to arrays of
unsigned chars.
uint8_t **
allocate_ustrmemp (int len)
{
  void *tmp;

  if (len <= 0) {
    fprintf (stderr, "ERROR: Cannot allocate memory because
len = %i in allocate_ustrmemp().\n", len);
    exit (EXIT_FAILURE);}
```

```
  tmp = (uint8_t **) malloc (len * sizeof (uint8_t *));
  if (tmp != NULL) {
    memset (tmp, 0, len * sizeof (uint8_t *));
    return (tmp);
  } else {
    fprintf (stderr, "ERROR: Cannot allocate memory for array
allocate_ustrmemp().\n");
    exit (EXIT_FAILURE);
  }
}

// Allocate memory for an array of ints.
int *
allocate_intmem (int len)
{
  void *tmp;

  if (len <= 0) {
    fprintf (stderr, "ERROR: Cannot allocate memory because
len = %i in allocate_intmem().\n", len);
    exit (EXIT_FAILURE);
  }

  tmp = (int *) malloc (len * sizeof (int));
  if (tmp != NULL) {
    memset (tmp, 0, len * sizeof (int));
    return (tmp);
  } else {
    fprintf (stderr, "ERROR: Cannot allocate memory for array
allocate_intmem().\n");
    exit (EXIT_FAILURE);
  }}
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. RECEIVER CODE

The following code implements the receiver used for the testing and experimentation presented in Chapter IV. The function *msg_decode* converts an ascii message into binary and then appends a computed CRC-9 FCS value. The program starts by using the python **os** and **csv** libraries to execute Tshark and strip the timestamps from packets collected via Tshark. The collected TCP timestamps and Tshark timestamps are used to calculate drift. The drift values are used with the *polyfit* function to calculate an estimate of baseline skew on the first $n_b = 150$ packets. Then *polyfit* is used in a sliding window of 'nm' packets. These estimates populate a list labeled 'mov_slope'. The index 'i' is used to find the center packet of each 'nm' packets. The element of 'mov_slope' at this value of 'i' corresponds to a transmitted skew. The A/D conversion is accomplished by a series of *if* statements that determine if a particular skew estimate meets threshold values. The baseline skew is subtracted from each threshold to ensure only the induced skew is considered. Each induced skew estimate that is compared against threshold values populates a list 'message_sym' for message symbols and a list 'message_bin'for bits. The results are then plotted using the **matplotlib** library.

```
import csv
import os
kagg')
import matplotlib.pyplot as plt
from tkinter import *
from tkinter import ttk

import time
import numpy
import matplotlib as mpl
mpl.use('t

def msg_decode(msg):

      global CRC_string, CRC_Result, q, compare_string
      decoded_msg=""
      msg_bits=len(msg)
      extras=msg_bits%8
      msg_bytes=int(numpy.floor((msg_bits/8)))
      j=0
```

```
      current_byte=""
      msg = msg[0:msg_bits-7]
      msg_copy=msg[:]
      k=0
      n=0
      CRC9_code=[1,1,1,1,1,1,1,1,1]
      compare_string=""

      msg_copy[msg_bits-8:msg_bits]=[0,0,0,0,0,0,0,0]
      for k in range(0,msg_bits-8):
            if msg_copy[k]==1:
                  n=0
                  while n<9:
                        if  CRC9_code[n]==1  and  msg_copy[k+n]==1:
msg_copy[k+n]=0
                        elif  CRC9_code[n]==1  and  msg_copy[k+n]==0:
msg_copy[k+n]=1
                        n+=1
      CRC_Result=msg_copy[msg_bits-8:msg_bits]
      if msg[msg_bits-8:msg_bits]==CRC_Result:
            print("\nCRC Check Passed!")
            CRC_string="Passed"
      else:
            print("\nCRC Check Failed :(")
            CRC_string="Failed!"

      for i in range(0,msg_bytes-1):
            current_byte=""
            j=0
            while j<8:
                  current_byte+=str(msg[(8*(i+1))-(j+1)])
                  #print("msg[",8*(i+1)-j+1,"]=",msg[(8*(i+1))-
(j+1)])
                  if j==7:
                        #print("current byte is",current_byte)
                  if current_byte=="00100000": decoded_msg+=" "
                  if current_byte=="00101101": decoded_msg+="-"
                  if current_byte=="00101110": decoded_msg+="."
                  if current_byte=="00110000": decoded_msg+="0"
                  if current_byte=="00110001": decoded_msg+="1"
                  if current_byte=="00110010": decoded_msg+="2"
                  if current_byte=="00110011": decoded_msg+="3"
                  if current_byte=="00110100": decoded_msg+="4"
                  if current_byte=="00110101": decoded_msg+="5"
                  if current_byte=="00110110": decoded_msg+="6"
                  if current_byte=="00110111": decoded_msg+="7"
                  if current_byte=="00111000": decoded_msg+="8"
                  if current_byte=="00111001": decoded_msg+="9"
                  if current_byte=="00111010": decoded_msg+=":"
                  if current_byte=="00111011": decoded_msg+=";"
```

```
if current_byte=="00111100": decoded_msg+="<"
if current_byte=="00111101": decoded_msg+="="
if current_byte=="01000001": decoded_msg+="A"
if current_byte=="01000010": decoded_msg+="B"
if current_byte=="01000011": decoded_msg+="C"
if current_byte=="01000100": decoded_msg+="D"
if current_byte=="01000101": decoded_msg+="E"
if current_byte=="01000110": decoded_msg+="F"
if current_byte=="01000111": decoded_msg+="G"
if current_byte=="01001000": decoded_msg+="H"
if current_byte=="01001001": decoded_msg+="I"
if current_byte=="01001010": decoded_msg+="J"
if current_byte=="01001011": decoded_msg+="K"
if current_byte=="01001100": decoded_msg+="L"
if current_byte=="01001101": decoded_msg+="M"
if current_byte=="01001110": decoded_msg+="N"
if current_byte=="01001111": decoded_msg+="O"
if current_byte=="01010000": decoded_msg+="P"
if current_byte=="01010001": decoded_msg+="Q"
if current_byte=="01010010": decoded_msg+="R"
if current_byte=="01010011": decoded_msg+="S"
if current_byte=="01010100": decoded_msg+="T"
if current_byte=="01010101": decoded_msg+="U"
if current_byte=="01010110": decoded_msg+="V"
if current_byte=="01010111": decoded_msg+="W"
if current_byte=="01011000": decoded_msg+="X"
if current_byte=="01011001": decoded_msg+="Y"
if current_byte=="01011010": decoded_msg+="Z"
if current_byte=="01011011": decoded_msg+="["
#if current_byte=="01011100": decoded_msg+="\"
if current_byte=="01011101": decoded_msg+="]"
if current_byte=="01011110": decoded_msg+="^"
if current_byte=="01100001": decoded_msg+="a"
if current_byte=="01100010": decoded_msg+="b"
if current_byte=="01100011": decoded_msg+="c"
if current_byte=="01100100": decoded_msg+="d"
if current_byte=="01100101": decoded_msg+="e"
if current_byte=="01100110": decoded_msg+="f"
if current_byte=="01100111": decoded_msg+="g"
if current_byte=="01101000": decoded_msg+="h"
if current_byte=="01101001": decoded_msg+="i"
if current_byte=="01101010": decoded_msg+="j"
if current_byte=="01101011": decoded_msg+="k"
if current_byte=="01101100": decoded_msg+="l"
if current_byte=="01101101": decoded_msg+="m"
if current_byte=="01101110": decoded_msg+="n"
if current_byte=="01101111": decoded_msg+="o"
if current_byte=="01110000": decoded_msg+="p"
if current_byte=="01110001": decoded_msg+="q"
if current_byte=="01110010": decoded_msg+="r"
```

```
                if current_byte=="01110011": decoded_msg+="s"
                if current_byte=="01110100": decoded_msg+="t"
                if current_byte=="01110101": decoded_msg+="u"
                if current_byte=="01110110": decoded_msg+="v"
                if current_byte=="01110111": decoded_msg+="w"
                if current_byte=="01111000": decoded_msg+="x"
                if current_byte=="01111001": decoded_msg+="y"
                if current_byte=="01111010": decoded_msg+="z"
                compare_string+=current_byte
                j+=1


    return decoded_msg


global CRC_string, CRC_Result, RS_Code, nm, mod, message_bin,
message_sym, compare_string
offset=[]
Tsval=[]
rc2=[]
rc1=[]
time_factor=1000000
drift=[]
discarded_packets=0
field_count=0
fit=[]
ys=[]
poly=[]
current_slope=[]
mov_slope=[]
b=0
c=0
nm=50
zeta=2500
minslop=0
maxslop=0
std_dev=0
Time_Travel=0
j=0
k=0
message_bin=[]
message_sym=[]
message_ascii=""
rel_max=0
rel_min=0
drift_delta=[]
last_bit_num=0
bit_index=[]
root=Tk()
```

```
row_to_write=""
q=1

os.system("sudo   tshark   -i   eno1   -Y   'tcp.srcport==1313'
>testfile.txt")
with open('testfile.txt', newline='') as f:
      data = csv.reader(f,delimiter=" ")
      for row in data:
            #print("row=",row)
            for i in row:
                  #print("Row element=",i)
                  if i != "":
                        field_count+=1
                  if field_count==2:
                        offset+=[float(i)]
                        #print("Offset is now",offset)
                        field_count+=1
                  if len(i)>3:
                        if i[0]=="T" and i[2]=="v":
                              Tsval+=[float(i.split("=")[1])]
                              #print("Tsval is now",Tsval)
            #if                              len(offset)!=len(Tsval):
offset=offset[:(len(offset)-2)]
            field_count=0

tc2=[]
rc2=[]
rc1=[]

            #print(len(row))
if  len(offset)!=len(Tsval):  print("Arrays  of  different  length.
Some garbage packet is being taken seriously")

for i in range(0,len(tc2)):
    rc2+=[tc2[i]-tc2[0]]
    rc1+=[(Tsval[i]-Tsval[0])/time_factor]
    drift+=[rc1[i]-rc2[i]]

##Calculates moving average of slope of drift values
b=0
c=0
for i in range(0,(len(drift))):
      if i<(int(numpy.floor(nm/2))): b=0
      else: b= i-(int(numpy.floor(nm/2)))
      if i>= (len(drift)-(int(numpy.floor(nm/2)))): c=(len(drift))
      else: c= i+(int(numpy.floor(nm/2)))
      current_slope=numpy.polyfit(rc2[b:c],drift[b:c],1)[0]
      mov_slope+=[current_slope]

base_skew=numpy.polyfit(rc2[0:150],drift[0:150],1)[0]
```

```
print("base_skew=",base_skew)

b=0
c=0

k=nm-1
bit_values=[]
bit_valx=[]
threshold_matrix=[]
for i in range(0,(len(drift))):
      if i>=(150+nm/2):
            k+=1
            if q==1:
                  if k==nm:
                        k=0

                        if mov_slope[i]>base_skew:
                              message_bin+=[1]
                              message_sym+=[1]
                              last_bit_num=i
                              bit_index+=[last_bit_num]
                              k=0
                              bit_values+=[mov_slope[i]]
                              bit_valx+=[rc2[i]]

                        elif mov_slope[i]<base_skew:
                              message_bin+=[0]
                              message_sym+=[-1]
                              last_bit_num=i
                              bit_index+=[last_bit_num]
                              k=0
                              bit_values+=[mov_slope[i]]
                              bit_valx+=[rc2[i]]



            if q==2:
                  if k==nm:
                        k=0
                        if        mov_slope[i]>base_skew        and
mov_slope[i]<=(1.5*mod/1000000)+base_skew:
                              message_bin+=[0]
                              message_bin+=[0]
                              message_sym+=[1]
                              last_bit_num=i
                              bit_index+=[last_bit_num]
                              k=0
                              bit_values+=[mov_slope[i]]
                              bit_valx+=[rc2[i]]
```

```
                        elif       mov_slope[i]<base_skew       and
mov_slope[i]>=(-1.5*mod/1000000)+base_skew:
                              message_bin+=[0]
                              message_bin+=[1]
                              message_sym += [-1]
                              last_bit_num=i
                              bit_index+=[last_bit_num]
                              k=0
                              bit_values+=[mov_slope[i]]
                              bit_valx+=[rc2[i]]
                        elif                          mov_slope[i]<((-
1.5*mod/1000000)+base_skew):
                              message_bin+=[1]
                              message_bin+=[1]
                              message_sym += [-2]
                              last_bit_num=i
                              bit_index+=[last_bit_num]
                              k=0
                              bit_values+=[mov_slope[i]]
                              bit_valx+=[rc2[i]]
                        elif
mov_slope[i]>((1.5*mod/1000000)+base_skew):
                              message_bin+=[1]
                              message_bin+=[0]
                              message_sym += [2]
                              last_bit_num=i
                              bit_index+=[last_bit_num]
                              bit_values+=[mov_slope[i]]
                              bit_valx+=[rc2[i]]
                              k=0

            if q==4:
                  if k>=nm:
                        if        mov_slope[i]>base_skew        and
mov_slope[i]<=(1.5*mod/1000000)+base_skew:
                              message_bin+=[0]
                              message_bin+=[0]
                              message_bin+=[1]
                              message_bin+=[1]
                              message_sym += [1]
                              last_bit_num=i
                              bit_index+=[last_bit_num]
                              k=0
                              bit_values+=[mov_slope[i]]
                              bit_valx+=[rc2[i]]
                        elif
mov_slope[i]>((1.5*mod/1000000)+base_skew)                    and
mov_slope[i]<=(2.5*mod/1000000)+base_skew: #corrected
                              message_bin+=[1]
                              message_bin+=[0]
```

```
                            message_bin+=[1]
                            message_bin+=[1]
                            message_sym += [2]
                            last_bit_num=i
                            bit_index+=[last_bit_num]
                            k=0
                            bit_values+=[mov_slope[i]]
                            bit_valx+=[rc2[i]]
                        elif
mov_slope[i]>((2.5*mod/1000000)+base_skew)                      and
mov_slope[i]<=(3.5*mod/1000000)+base_skew:
                            message_bin+=[1]
                            message_bin+=[1]
                            message_bin+=[1]
                            message_bin+=[1]
                            message_sym += [3]
                            last_bit_num=i
                            bit_index+=[last_bit_num]
                            k=0
                            bit_values+=[mov_slope[i]]
                            bit_valx+=[rc2[i]]
                        elif
mov_slope[i]>((3.5*mod/1000000)+base_skew)                      and
mov_slope[i]<=(4.5*mod/1000000)+base_skew:
                            message_bin+=[0]
                            message_bin+=[1]
                            message_bin+=[1]
                            message_bin+=[1]
                            message_sym += [4]
                            last_bit_num=i
                            bit_index+=[last_bit_num]
                            bit_values+=[mov_slope[i]]
                            bit_valx+=[rc2[i]]
                            k=0

                        elif
mov_slope[i]>((4.5*mod/1000000)+base_skew)                      and
mov_slope[i]<=(5.5*mod/1000000)+base_skew:
                            message_bin+=[0]
                            message_bin+=[1]
                            message_bin+=[0]
                            message_bin+=[1]
                            message_sym += [5]
                            last_bit_num=i
                            bit_index+=[last_bit_num]
                            k=0
                            bit_values+=[mov_slope[i]]
                            bit_valx+=[rc2[i]]
```

```
                    elif
mov_slope[i]>((5.5*mod/1000000)+base_skew)                    and
mov_slope[i]<=(6.5*mod/1000000)+base_skew:
                         message_sym += [6]

                         message_bin+=[1]
                         message_bin+=[1]
                         message_bin+=[0]
                         message_bin+=[1]

                         last_bit_num=i
                         bit_index+=[last_bit_num]
                         k=0
                         bit_values+=[mov_slope[i]]
                         bit_valx+=[rc2[i]]

                    elif
mov_slope[i]>((6.5*mod/1000000)+base_skew)                    and
mov_slope[i]<=(7.5*mod/1000000)+base_skew:
                         message_bin+=[1]
                         message_bin+=[0]
                         message_bin+=[0]
                         message_bin+=[1]
                         message_sym += [7]
                         last_bit_num=i
                         bit_index+=[last_bit_num]
                         k=0
                         bit_values+=[mov_slope[i]]
                         bit_valx+=[rc2[i]]

                    elif
mov_slope[i]>((7.5*mod/1000000)+base_skew):
                         message_bin+=[0]
                         message_bin+=[0]
                         message_bin+=[0]
                         message_bin+=[1]
                         message_sym += [8]
                         last_bit_num=i
                         bit_index+=[last_bit_num]
                         bit_values+=[mov_slope[i]]
                         bit_valx+=[rc2[i]]
                         k=0

                    elif       mov_slope[i]<base_skew       and
mov_slope[i]>=(-1.5*mod/1000000)+base_skew:
                         message_bin+=[0]
                         message_bin+=[0]
                         message_bin+=[1]
                         message_bin+=[0]
                         message_sym += [-1]
```

```
                        last_bit_num=i
                        bit_index+=[last_bit_num]
                        k=0
                        bit_values+=[mov_slope[i]]
                        bit_valx+=[rc2[i]]
                    elif                        mov_slope[i]<((-
1.5*mod/1000000)+base_skew)         and         mov_slope[i]>=(-
2.5*mod/1000000)+base_skew:
                        message_bin+=[1]
                        message_bin+=[0]
                        message_bin+=[1]
                        message_bin+=[0]
                        message_sym += [-2]
                        last_bit_num=i
                        bit_index+=[last_bit_num]
                        k=0
                        bit_values+=[mov_slope[i]]
                        bit_valx+=[rc2[i]]
                    elif                        mov_slope[i]<((-
2.5*mod/1000000)+base_skew)         and         mov_slope[i]>=(-
3.5*mod/1000000)+base_skew:
                        message_bin+=[1]
                        message_bin+=[1]
                        message_bin+=[1]
                        message_bin+=[0]
                        message_sym += [-3]
                        k=0
                        last_bit_num=i
                        bit_index+=[last_bit_num]
                        bit_values+=[mov_slope[i]]
                        bit_valx+=[rc2[i]]
                    elif                        mov_slope[i]<((-
3.5*mod/1000000)+base_skew)         and         mov_slope[i]>=(-
4.5*mod/1000000)+base_skew:
                        message_bin+=[0]
                        message_bin+=[1]
                        message_bin+=[1]
                        message_bin+=[0]
                        message_sym += [-4]
                        last_bit_num=i
                        bit_index+=[last_bit_num]
                        bit_values+=[mov_slope[i]]
                        bit_valx+=[rc2[i]]
                        k=0

                    elif                        mov_slope[i]<((-
4.5*mod/1000000)+base_skew)         and         mov_slope[i]>=(-
5.5*mod/1000000)+base_skew:
                        message_bin+=[0]
                        message_bin+=[1]
```

```
                            message_bin+=[0]
                            message_bin+=[0]
                            message_sym += [-5]
                            last_bit_num=i
                            bit_index+=[last_bit_num]
                            k=0
                            bit_values+=[mov_slope[i]]
                            bit_valx+=[rc2[i]]
                        elif                            mov_slope[i]<((-
5.5*mod/1000000)+base_skew)          and          mov_slope[i]>=(-
6.5*mod/1000000)+base_skew: #corrected
                            message_bin+=[1]
                            message_bin+=[1]
                            message_bin+=[0]
                            message_bin+=[0]
                            message_sym += [-6]
                            last_bit_num=i
                            bit_index+=[last_bit_num]
                            k=0
                            bit_values+=[mov_slope[i]]
                            bit_valx+=[rc2[i]]

                        elif                            mov_slope[i]<((-
6.5*mod/1000000)+base_skew)          and          mov_slope[i]>=(-
7.5*mod/1000000)+base_skew:
                            message_bin+=[1]
                            message_bin+=[0]
                            message_bin+=[0]
                            message_bin+=[0]
                            message_sym += [-7]
                            last_bit_num=i
                            bit_index+=[last_bit_num]
                            k=0
                            bit_values+=[mov_slope[i]]
                            bit_valx+=[rc2[i]]

                        elif                            mov_slope[i]<((-
7.5*mod/1000000)+base_skew):
                            message_bin+=[0]
                            message_bin+=[0]
                            message_bin+=[0]
                            message_bin+=[0]
                            message_sym += [-8]
                            last_bit_num=i
                            bit_index+=[last_bit_num]
                            bit_values+=[mov_slope[i]]
                            bit_valx+=[rc2[i]]
                            k=0
```

```
message_string=msg_decode(message_bin)
##  Defines  base  skew  for  Thesis  presentation  calculation.
Meaningless if message present (?)
fit=numpy.polyfit(rc2,drift,1);
poly=numpy.poly1d(fit)
ys=poly(rc2)
base_skew_unmod=numpy.polyfit(rc2,drift,1)[0]
normalized_drift=[]
for i in drift: normalized_drift+=[i-base_skew_unmod]
std_base_skew=numpy.std(normalized_drift)

print("Base Skew for all packets recieved is ", base_skew_unmod)
#("The   first   10   values   of   normalized   drift   are   ",
normalized_drift[0:9])
print("The std deviation of normalized drift is ", std_base_skew)


## To detect Time Travel ##
while j < len(rc1)-1:
     if rc1[j]>rc1[j+1]:
          Time_Travel=1
     j+=1

pos_slop=[]

for i in range(0,len(mov_slope)):
     if i in bit_index:
          if mov_slope[i]>0: pos_slop+=[mov_slope[i]]
          elif mov_slope[i]<0: pos_slop+=[mov_slope[i]*-1]
minslop=min(pos_slop)
maxslop=max(pos_slop)
std_dev=numpy.std(pos_slop)
print("\nHere are the indices of each bit and their values:")
for i in range(0,len(bit_index)):

     print("\nBit",i, "at slope index", bit_index[i], "has a value
of", pos_slop[i])

for i in range(0,len(drift)-1):
     drift_delta+=[drift[i]-drift[i+1]]


# Calculate rate of packet receipt
jjj=.1
cum_pack=0
cum_pack_seg=[]
cum_pack_seg_sec=[]
```

```
for i in range(0,len(rc2)):

      if rc2[i]>jjj:
            cum_pack_seg+=[10*(i-cum_pack)]
            cum_pack=i
            jjj+=.1

for i in range(0,len(cum_pack_seg)):
      cum_pack_seg_sec+=[(i+1)/10]



root=Tk()
result_string="The  decoded  message  is  "+message_string+"\nThe
received symbols were:"+message_bin
result_string+="\nCRC Check"+CRC_string+" with code "+CRC_Result

result_label2=ttk.Label(root,text=result_string)
result_label2.grid(column=0,row=0)
root.mainloop()

ax0=plt.subplot(232)
plt.plot(cum_pack_seg_sec,cum_pack_seg)
plt.title('Recieved Packet Rate',fontname='Times New Roman')
plt.xlabel('Reciever Offset (s)',fontname='Times New Roman')
plt.ylabel(' Packets per second',fontname='Times New Roman')
mpl.axes.Axes.set_ybound(ax0,lower=0,upper=60)

ax1=plt.subplot(233)
plt.plot(rc2, rc1,'bo')
plt.title('Recieve Time vs. Tsecr')
plt.ylabel(' Transmitter Offset (s)')
plt.xlabel('Reciever Offset (s)')

ax2=plt.subplot(234)
plt.plot(rc2,drift,'bo')
fig_size=plt.rcParams["figure.figsize"]
fig_size[1]=12
plt.rcParams["figure.figsize"]=fig_size
plt.title('Reciever Offset vs. Drift')
plt.ylabel('Drift (s)')
plt.xlabel('Reciever Offset (s)')

ax3=plt.subplot(235,sharex=ax2)
plt.plot(bit_valx,bit_values,"x")

plt.plot([0,rc2[-1]],[0,0],color='r',linestyle="--")

if q>1:
```

```
      plt.plot([0,rc2[-
1]],[1.5*mod/1000000,1.5*mod/1000000],color='r',linestyle="--")
      plt.plot([0,rc2[-1]],[-1.5*mod/1000000,-
1.5*mod/1000000],color='r',linestyle="--")
if q>2:
      plt.plot([0,rc2[-
1]],[2.5*mod/1000000,2.5*mod/1000000],color='r',linestyle="--")
#+4
      plt.plot([0,rc2[-1]],[2.5*mod/-1000000,2.5*mod/-
1000000],color='r',linestyle="--") # -4
      plt.plot([0,rc2[-
1]],[3.5*mod/1000000,3.5*mod/1000000],color='r',linestyle="--")
#+3
      plt.plot([0,rc2[-1]],[3.5*mod/-1000000,3.5*mod/-
1000000],color='r',linestyle="--")#-3
      plt.plot([0,rc2[-
1]],[4.5*mod/1000000,4.5*mod/1000000],color='r',linestyle="--
")#+5
      plt.plot([0,rc2[-1]],[4.5*mod/-1000000,4.5*mod/-
1000000],color='r',linestyle="--")#-5
      plt.plot([0,rc2[-
1]],[5.5*mod/1000000,5.5*mod/1000000],color='r',linestyle="--")
#+6
      plt.plot([0,rc2[-1]],[5.5*mod/-1000000,5.5*mod/-
1000000],color='r',linestyle="--")#-6
      plt.plot([0,rc2[-
1]],[6.5*mod/1000000,6.5*mod/1000000],color='r',linestyle="--
")#+7
      plt.plot([0,rc2[-1]],[6.5*mod/-1000000,6.5*mod/-
1000000],color='r',linestyle="--")#-7
      plt.plot([0,rc2[-
1]],[7.5*mod/1000000,7.5*mod/1000000],color='r',linestyle="--")
#+6
      plt.plot([0,rc2[-1]],[7.5*mod/-1000000,7.5*mod/-
1000000],color='r',linestyle="--")#-6


plt.tight_layout()
plt.show()


if abs(max(mov_slope))<1:
      time_str=str(time.localtime()[1])+"-
"+str(time.localtime()[2])+"-
"+str(time.localtime()[0])+";"+str(time.localtime()[3])+str(time.
localtime()[4])
      row_to_write+=time_str+"      "+mod+"      "+nm+"      "+q+"
"+len(message_sym)+" "+len(compare_string)

      with open('Trials.csv', 'w') as csvfile:
```

```
        Gwriter     =     csv.writer(csvfile,     delimiter='|',
quoting=csv.QUOTE_MINIMAL,lineterminator='\n')
          Gwriter.writerow(row_to_write)
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] S. Khandelwal, “Google achieves first-ever successful SHA-1 collision attack,” *The Hacker News*, Feb. 23, 2017. [Online]. Available: https://thehackernews.com/2017/02/sha1-collision-attack.html

[2] S. Murdoch and S. Lewis, “Embedding covert channels into TCP/IP,” *Lecture Notes in Computer Science*, vol. 3727, pp. 247–261, 2005.

[3] T. Kohno, A. Broido, and K. C. Claffy, “Remote physical device fingerprinting,” *IEEE Transactions on Dependable and Secure Computing*, vol. 2, pp. 23–108, 2005.

[4] J. Giffin, R. Greenstadt, P. Litwack and R. Tibbets, “Covert messaging through TCP timestamps,” *Proceedings of the 2nd international Conference on Privacy Enhancing Technologies*, 2002, pp. 194–208.

[5] B. J. Martin, “Detecting a multi-homed device using clock skew,” M.S. thesis, Dept. Elec. Eng., Naval Postgraduate School, Monterey, CA, USA, 2016.

[6] J. Fridrich, *Steganography in Digital Media*. Cambridge, UK: Cambridge University Press, 2010.

[7] Y. Liu, D. Ghosal, F. Armknecht, A. Sadeghi, S. Schulz and S. Katzenbeisser, “Hide and seek in time - Robust covert timing channels,” *European Symposium on Research in Computer Security*, 2009, pp. 120–135.

[8] U. Windl, D. Dalton, M. Martinec, and D. Worley, “Understanding and using the Network Time Protocol,” University of Delaware. Accessed November 21, 2017. [Online]. Available: https://www.eecis.udel.edu/~ntp/ntpfaq/NTP-s-sw-clocks.htm

[9] J. Postel, *Transmission Control Protocol*, RFC 793, Sep. 1981. [Online]. doi: 10.17487/RFC0793

[10] W. Stallings and M. Manna, *Data and Computer Communications*. Upper Saddle River, NJ, USA: Pearson, 2014.

[11] V. Jacobson, R. Braden, and D. Borman, *TCP Extensions for High Performance*, RFC 1323, May 1992. [Online]. doi:10.17487/RFC1323

[12] A. Reprakash, “TCP timestamp - Demystified,” *IT Blogtorials*, Feb. 25, 2013. [Online]. Available: http://ithitman.blogspot.com/2013/02/tcp-timestamp-demystified.html

[13] T. Handel and M. Sanford, “Hiding data in the OSI network model,” *International Workshop on Information Hiding*, 1996, pp. 23–28.

[14] J. Erickson, *Hacking*. San Francisco, CA, USA: No Starch Press, 2008.

[15] L. Polcak and B. Frankova, “On reliability of clock-skew-based remote computer identification,” in *International Conference on Security and Cryptography*, Vienna, Austria, 2014, pp. 1–8.

[16] J.A. Rhinehart, “Investigating the detection of multi-homed devices independent of operating system,” M.S. thesis, Dept. Elec. Eng., Naval Postgraduate School, Monterey, CA, USA, 2017.

[17] S. Sharma, A. Hussein and H. Saran, “Experience with heterogeneous clock-skew based device fingerprinting,” *Workshop on Learning from Authoritative Security Experiment Results*, pp. 9–18, 2012.

[18] D. Montgomery and G. Runger, *Applied Statistics and Probability for Engineers*. Hoboken, NJ: John Wiley and Sons, Inc., 2014, pp. 427–461.

[19] S. Haykin and M. Moher, *An Introduction to Analog and Digital Communications*. Hoboken, NJ, USA: Wiley Textbooks, 2012.

[20] G. Blelloch and M. Riley, “Reed–Solomon codes,” *Carnegie-Mellon University School of Computer Science*. Accessed: Oct. 7, 2017. [Online]. Available: www.cs.cmu.edu/~guyb/realworld/reedsolomon/reed_solomon_codes.html.

[21] M. Karakas, “Determination of network delay distribution over the internet,” M.S. thesis, Dept of Electronics and Elec. Eng., Middle East Technical University, Ankara, Turkey, 2003.

[22] G. Hooghiemstra and P. Van Mieghem, "Delay distributions on fixed Internet paths," *Delft University of Technology report 2001*, pp. 1020-1032, 2001.

[23] P. Buchanan, “C Language examples of IPv4 and IPv6 raw sockets for Linux,” *Dave’s Website*, Mar. 6, 2015. [Online]. Available: http://www.pdbuchan.com/rawsock/rawsock.html.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California