TOWSON UNIVERSITY

OFFICE OF GRADUATE STUDIES

Towards Wireless Bare Machines-A Study of Three Aspects of 802.11 Wireless Networks: Performance, Network Device Driver, and CCM Mode Algorithm

By

William R. Agosto-Padilla

A Dissertation

Presented to the Faculty of

Science

in partial fulfillment of the requirements for the degree

Doctor of Science

in

Information Technology

Department of Computer and Information Sciences

August 2016

TOWSON UNIVERSITY
OFFICE OF GRADUATE STUDIES

DISSERTATION APPROVAL PAGE

This is to certify that the thesis prepared by [INSERT Student's Name] William Agosto-Padilla

entitled [INSERT Title of Thesis] Towards Wireless Bare Machines-A Study of Three Aspects of 802.11 Wireless Networks: Performance, Network Device Driver, and CCM Mode Algorithm

has been approved by the thesis committee as satisfactorily completing the dissertation requirements for the degree_ [INSERT Type of Degree] Doctor of Science

(for example, Doctor of Science)

Alexander Wijesinha (Digitally signed by Alexander Wijesinha DN: cn=Alexander Wijesinha, o=Towson University, ou=Department of Computer & Information Sciences, email=awijesinha@towson.edu, c=US Date: 2016.08.17 12:25:31 -04'00')	Alexander Wijesinha	8/17/16
Chairperson, Dissertation Committee Signature	Type Name	Date
Committee Member Signature	Type Name	Date
Dr. Ramesh K. Karne (Digitally signed by Dr. Ramesh K. Karne DN: cn=Dr. Ramesh K. Karne, o=Towson University, ou=Computer Science, email=rkarne@towson.edu, c=US Date: 2016.08.17 13:14:44 -04'00')		8-17-16
Committee Member Signature	Type Name	Date
Yeong-Tae Song (Digitally signed by Yeong-Tae Song DN: cn=Yeong-Tae Song, o=Towson University, ou=CIDSC, email=ysong@towson.edu, c=US Date: 2016.08.17 14:04:21 -04'00')		
Committee Member Signature	Type Name	Date
Wei (Digitally signed by Wei DN: cn=Wei, o, ou=Yu, email=yuwilson6@gmail.com, c=US Date: 2016.08.17 13:15:39 -04'00')		
Committee Member Signature	Type Name	Date
Dean of Graduate Studies	Type Name	Date

ACKNOWLEDGEMENT

I would like deeply thank my advisor, Dr. Alexander Wijesinha and Dr. Ramesh Karne, for their continuous support and advice, and for getting me back on the right path whenever I lost focus. They motivated me the many times I hit obstacles along the way. I know that our collaboration will continue in the future, and I am looking forward to it. Thank you Dr. Ramesh Karne for giving me the opportunity to be part of the world of the bare machine computing, for being available whenever the need arises for some internal debugging in the bare code, and for the many challenges and laughs we had in the lab. To my committee members Dr. Wei Yu and Dr. Yeong-tae Song, I am extremely grateful for your willingness to serve on my committee; your feedback and opinion mean a lot to me, and for my work to be judged by you is a great honor.

My warmest and deepest thank you to my lovely and caring wife Carmen Amalia Vazquez. Thank you for your love, caring and support, and for assuming many of my responsibilities so I could get my work done--I could not have done it without you. I would also like to thank my family for the support and encouragement all this time.

To my colleagues in the bare Machine Computing Lab: Dr. Alae Loukili, Dr. Tony Tsetse, Dr. William Thompson, Mr. Hojin Chang, Ms. Rasha Almajed, and Mr. Hamdan Alabsi I say thank you for all the help and camaraderie we shared during the years of research.

To my Agency and the Laboratory for Analytic Science (LAS) mentors, Mr. Jeffrey David Harris, Mr. Michael Bender, Ms. Margaret Dillard, Ms. Lydia Santiago for their trust, confidence and unrelenting support.

Finally, I would like to also thank my dear friends Dr. John Ravekes, for his mentoring, Michelle Cayford, Cathy Wood-Rupert and their families for extending their kindness and hospitality during many of my visits from Raleigh, N.C. You have always inspired me to greater heights, and for this, I am highly appreciative.

ABSTRACT

Towards Wireless bare Machines-A Study of Three Aspects of 802.11 Wireless Networks: Performance, Network Device Driver, and CCM Mode Algorithm

By

William R. Agosto-Padilla

Bare machines applications run without the support of any operating system (OS) or other intermediary software. Since only the application code runs on the bare machine, exploits that target an underlying OS or kernel cannot compromise such applications. Bare machine applications are also more efficient than applications that require a conventional OS or kernel due to elimination of the associated overhead. Existing bare machine applications include Web servers, split servers, mail servers, SIP servers and VoIP clients. A bare machine connects to a network via an Ethernet connection. This research lays the groundwork for enabling bare machines to connect to 802.11 wireless networks in the future.

We first obtain baseline performance measurements for mobile devices that connect to an 802.11n network. Specifically, we conduct experiments to measure the throughput and delay of mobile devices in a 2.4 GHz 802.11n LAN under different levels of congestion in the wireline network, where the path from the mobile devices to a Web server traverses several routers. The results indicate that there is high variability in the performance of mobile devices even in an interference-controlled environment with no congestion in the wireline network; and that higher performance loss does not always correspond to a higher level of congestion. Packet analysis reveals that the performance variability and performance reduction are primarily due to the use of RTS-CTS and the cost of retransmitting packets on the wireless link. This suggests that mobile devices in an 802.11 network should be conservative in their use of RTS-CTS when congestion in the wireline network is likely or detected.

Next, we study a Broadcom Linux 802.11n wireless device driver. Our analysis provides a deeper insight into driver functions and their OS dependencies. We analyze the

driver module by examining its interaction with other modules, providing details of its key elements and code sizes, and comparing code sizes with its counterpart Windows driver. We also identify some design issues that will be useful for developing device drivers that are independent of any operating system, kernel, or embedded system. The study shows that transforming existing OS drivers to run on a bare machine requires knowledge of the inner workings of the Linux kernel and its interactions with the device driver.

Finally, we study CCM mode, which is the basis for the 802.11i security standard CCMP. We first study popular Linux and Windows CCM mode implementations `to understand their OS dependencies. We then implement CCM mode with no OS dependencies so that it runs on a bare machine. The bare machine CCM mode implementation derives from the original specification of CCM mode, and is not a transformation of either the Linux or the Windows versions. We compare the performance of the bare machine and Linux CCM implementations by sending UDP messages and measuring internal timings for the authenticated encryption/decryption operations. We also measure network delays. The results show that the bare machine implementation performs better than the Linux implementation in some cases, but not in others.

LIST OF FIGURES

LIST OF TABLES

TABLE OF CONTENTS

1 INTRODUCTION

Bare machine applications run without the support of any operating system (OS) or other intermediary software. Such applications have no OS-related vulnerabilities and no overhead due to an underlying OS. To date, a variety of bare machine applications have been developed including Web servers and split servers, Webmail and email servers, and SIP servers and VoIP clients. Currently, bare machine applications that run on a network require an Ethernet connection. A future goal is to enable bare machine applications to connect to 802.11 wireless networks.

This dissertation studies three aspects of 802.11 wireless networks that would be useful when implementing and evaluating the performance of future bare machine applications with 802.11 wireless connectivity. First, we study 802.11n network performance with mobile devices. Second, we study an open source 802.11n device driver that runs on Linux. Third, we study CCM mode, which is the basis for data encryption and authentication in the 802.11i security standard, and implement CCM mode on a bare machine. Our studies will provide insight into 1) establishing baselines for comparing the performance of OS-based and bare machine applications that run on 802.11; 2) developing an 802.11 network device driver for a bare machine; and 3) implementing CCM mode security for bare machine applications that run on 802.11.

The main 802.11 wireless standards used in LANs today are 802.11ac and 802.11n (the 802.11ad standard targets multimedia applications, and is for short-range wireless connectivity at gigabit rates using the 60 GHz band). Our studies on 802.11 performance and the 802.11 device driver use the 802.11n standard. Even though use of the newer

802.11ac standard is increasing, many mobile devices will continue to use currently installed 802.11n networks in homes and workplaces until 802.11ac equipment becomes commonplace.

While 802.11n can achieve theoretical maximum bandwidths up to 600 Mbps, 802.11ac targets 7 Gbps (under ideal conditions and if the full capabilities of these technologies are used by both the devices and the access points). Yet, ordinary users of mobile devices typically use preset configuration settings on their devices with any available access point, and do not have control over interference in the wireless environment.

In the first part of this research, we study mobile device performance in an 802.11n network under ordinary usage, and when the path from a mobile device to a Web server traverses congested wireline links (assuming there are no cached copies of the data close to the user). Specifically, we measure throughput and delay for mobile devices in an 802.11 network under different levels of congestion in the wireline network. Two key factors can reduce 802.11n throughput and increase delay in this case. First, the use of RTS-CTS and exponential backoff by the MAC layer algorithm in the mobile devices. Second, the TCP congestion control mechanisms used by routers and Web servers. Our experiments use a test network in the lab that consists of a small 802.11n network and a wireline component with Ethernet switches and Linux routers. An iPhone and an iPad serve as clients that make requests to an Apache Web server. The main findings are that 1) mobile device performance has a high degree of variability; and 2) higher performance losses do not always relate to the level of congestion. Our analysis of packets reveals that performance

degradation is primarily due to 802.11 collision avoidance mechanisms (i.e., RTS-CTS and backoff), and local retransmissions of packets lost on the wireless link.

In the second part of this research, we investigate approaches for developing a bare wireless device driver. When selecting an existing OS-based 802.11n device driver for study, our choices were limited because proprietary source code for 802.11 network device drivers is difficult to obtain. Our study uses a Broadcom b43 driver for Linux whose source code was available. There are two approaches to use when developing an 802.11 device driver for bare machines: either transform an existing Linux driver or develop a driver based on its specification. Our analysis shows that the Linux device driver code is complex and intertwined with the kernel, making it difficult to decouple the driver from the kernel in order to transform it to run on a bare machine.

The final part of this research focuses on AES-CCM mode with a view towards implementing it on a bare machine. The bare machine CCM mode implementation will enable protection of future bare applications that run in wireless environments. It will also eliminate the need for such applications to use hardware-based CCM or CCMP implementations. A related goal is to create an AES-CCM mode object for secure communication between any bare to bare or bare to OS-based application.

We first study two implementations of OS-based AES_128_CCM: Crypto++ on Windows and OpenSSL on Linux. We then implement CCM for a bare machine based on its original specification and test its interoperability with the Linux implementation. We also compare performance of the bare machine and Linux CCM implementations by measuring internal timings and network delays. The results show that while the bare machine implementation generally has better performance, there are several exceptions.

Future work can focus on optimizing the bare machine implementation to streamline its performance.

The rest of this dissertation is as follows. In Chapter 2, we discuss related work. Chapters 3, 4 and 5 deal with performance of 802.11n for mobile devices, analysis of the 802.11n Linux device driver, and a study of CCM mode respectively. In Chapter 6, we give the conclusion.

2 BACKGROUND AND RELATED WORK

There are many studies on TCP performance in wireless networks including 802.11 LANs. Few studies on 802.11n performance have used a real network. One such study [1] concluded that 802.11n performance in the 2.4 GHz band is affected by both powerful narrowband interference and by interference due to legacy technologies. No previous 802.11n performance study considered wireline congestion with real traffic.

In performance improvement through MAC layer retransmission [2], the transmission mechanism is stop-and-wait ARQ. A retransmission threshold is set based on error occurrence. If loss is determined to be due to congestion, the MAC sender does not retransmit. A throughput-aware rate adaptation protocol is used in [3] to reduce performance degradation in 802.11(a/g/n) networks. It is based on estimating packet transmission times and using them to identify network congestion. In [4], commercial wireless networks in which clients obtain services by connecting to the Internet via a router are studied to determine the characteristics of available Internet bandwidth. Results using a tool to access Web servers showed that the Internet bandwidth bottleneck is at the

network edge. This implies that available bandwidth to a server may be estimated by determining the available bandwidth close to the wireless device.

In [5], an analytical model is used to find the MAC layer throughput of multi-hop 802.11n mesh networks by considering frame aggregation and block acknowledgements, which are features found in the new 802.11n standard. Throughput is given in terms of several parameters including the physical layer data rate and the error rate. The protocol requires that the receiver respond to an RTS packet with a unary signal that encodes the maximum data rate based on channel quality if it is ready to receive. Otherwise, the unary signal is not sent, which causes the sender to backoff. The signal enables the sender to avoid unnecessary retransmissions.

Since significant bandwidth loss in networks with a high-speed physical layer results from collisions, a binary protocol to reduce collisions and contention time is proposed in [6]. Throughput is increased since the MAC layer overhead and collision rate are reduced. The protocol is evaluated by simulation. An earlier study [7] determined performance effects due to congestion on a real 802.11b network. One of their findings was that nodes using RTS-CTS under high congestion levels in the wireless link experienced poor performance. The performance impact of using RTS-CTS is also reflected in the results of our work. Performance studies of multipath TCP in wireless networks e.g., [8] may be useful for developing strategies that enable a mobile device to decide when to use the connection to the cellular network instead of RTS-CTS on 802.11.

Many studies have been conducted to understand the design and internals of device drivers. In [9], an in-depth study of Linux device drivers is undertaken. Static analysis tools are used to analyze driver code and determine how drivers interact with the OS and the

hardware. It does not focus on a particular driver or a wireless driver. In [10], driver development using hardware abstraction and APIs for hardware and software interfaces is discussed. While their Device Object Model is useful for separating OS and device-specific components of drivers, it does not provide any insights for transforming an OS driver so that it becomes OS independent. The focus of [11] is a system that enables Linux drivers to run in user space so that they are able to limit the impact of attacks on drivers. In [12], a technique for reverse engineering drivers is given. However, the technique cannot be easily adapted to reverse engineer an existing OS driver to run on a bare machine. The approach suggested for driver reuse in [13] is to run an existing driver and the original OS inside a virtual machine using pre-virtualization to construct the virtual machine. This requires some modifications to the OS.

In bare machine computing, all the code needed to load and execute an application is contained in the application itself, which is stored on a USB drive. The hard disk in the machine is not used, and no other software runs in a bare machine when a given application is running. Thus bare machine computing applications run on the hardware without any OS support [14], [15], and any device drivers they use are self-contained and independent of an OS. Existing bare machine drivers include a USB device driver, 3Com/Intel network interface drivers, and an audio card driver [16, 17, 18, 19, 20, 21]. Relevant standards or specification documents form the basis for writing these drivers. An alternate approach for developing bare machine drivers is to transform existing OS drivers to run on a bare machine. There are no bare machine drivers derived by transformation, although it is possible to transform a Windows SQLITE database engine to run on a bare machine [22].

The original specification of the authenticated encryption standard CCM mode is in [23]. It is also the subject of the NIST recommendation [24]. CCM mode is the basis for CCMP encryption and authentication in the 802.11i security standard, and for security in ZigBee, the low-power short-range wireless technology used in home device automation and the Internet of Things. Specifications exist for using CCM mode with many security standards including TLS [25], ECC cipher suites for TLS [26], IPsec ESP [27], IKEv2 [28], and CMS [29]. It has provably good security properties [30].

CCM mode has efficient implementations in hardware or software. There are many hardware and software implementations of CCM mode. The authors of [31] describe a reconfigurable hardware implementation of CCM mode and show that it has high throughput. The focus of [32] is the design of a compact and energy-efficient hardware implementation of CCM mode for wireless sensor networks. The experimental results for an implementation of CCM mode in C given in [33] show that it performs better than existing software implementations. The work in [34] considers a software implementation of CCM mode for multiprocessors suited for embedded systems that is faster than a single-processor implementation. The bare machine CCM mode implementation differs from such previous implementations in that it has no OS-dependencies and runs on any bare PC.

3 A STUDY OF MOBILE DEVICE PERFORMANCE IN AN 802.11n WIRELESS LAN

Performance of mobile devices in an 802.11 wireless LAN can degrade due to many reasons. Here we study the performance of mobile devices in an 802.11n wireless LAN with and without congestion in the wireline network. The sections of this chapter provide details of the test network used, the experiments conducted, the results of the study, and an analysis of the results.

3.1 Experimental Setup

To evaluate 802.11n performance with and without wireline congestion, we measure the delay and throughput associated with the response to a single browser request. For this purpose, the Wireshark analyzer [35] running on the wireless LAN as shown in Fig. 1 was used to capture the necessary packets. The delay is the total amount of time in milliseconds (ms) between the HTTP Get request and the last valid ack for the data sent by the client (browser). The throughput in Mbps is the total amount of data in all the packets (including retransmissions) transferred between the client and the server for the duration of the TCP connection divided by the connection time. Throughput includes all retransmitted packets.

3.1.1 Test Network

The experiments use the test network shown in Fig. 1. The network consists of an 802.11n wireless LAN operating in the 2.4 GHz band connected to a wireline Ethernet. (Many 802.11n networks use the more interference-prone 2.4 GHz band instead of the less crowded 5 GHz band allowed by the standard).

Details of the hardware and software used for the experiments are given in Tables I and II for the wireline network and wireless LAN respectively. We use the wireline client shown in Fig. 1 only for testing (not for the actual experiments). The wireless network includes an 802.11n access point (AP) and 802.11n wireless clients: the two mobile devices (iPhone-5 and iPad-2) and four machines (laptops or desktops) running Linux or Windows. For convenience, the client devices in Table I are referred to as iPhone-5 (Mobile Device-1), iPad-2 (Mobile Device-2), Linux-1 (L1), Linux-2 (L2), Windows-1 (W1), and Windows-2 (W2). The wireline Ethernet consists of four Linux routers R1-R4 connected to five Ethernet switches S0-S4 as shown. With the exception of switch S0 and the Ethernet NIC on edge router R1 (connected to switch S0), which are 100 Mbps, all switches and Ethernet NICs are 1 Gbps. This creates congestion on the wireline network.

3.1.2 Traffic Generation

To measure performance, one or more wireless clients use a Web browser to send a single request to the Apache/Linux Web server for a small (320 KB) page. The server connects to the wireline network via switch S4 as shown in Fig. 1. Experiments to test 802.11n performance were conducted first with no congestion traffic, and then with different levels of congestion traffic in the wireline network. The congestion traffic in the wireline network is at three different rates: 100 (low), 125 (medium), and 150 Mbps (high) respectively. The traffic consists of a 90/10% TCP/UDP mix. The MGEN traffic generator [36] with a source and sink as shown in Fig. 1 generates this traffic.

The wireless clients (mobile devices and laptops) were run alone (“single clients”), as pairs (“paired clients”), or as four clients together (“4 machines”). For “single clients”, a wireless client running alone made a single request to the Web server (i.e., without any

competition from the other wireless clients). For “paired clients”, a pair of wireless clients running simultaneously made a single request to the Web server to compete for the medium. Similarly, for “4 machines”, four wireless clients simultaneously made a single request to the Web server to compete for the medium.

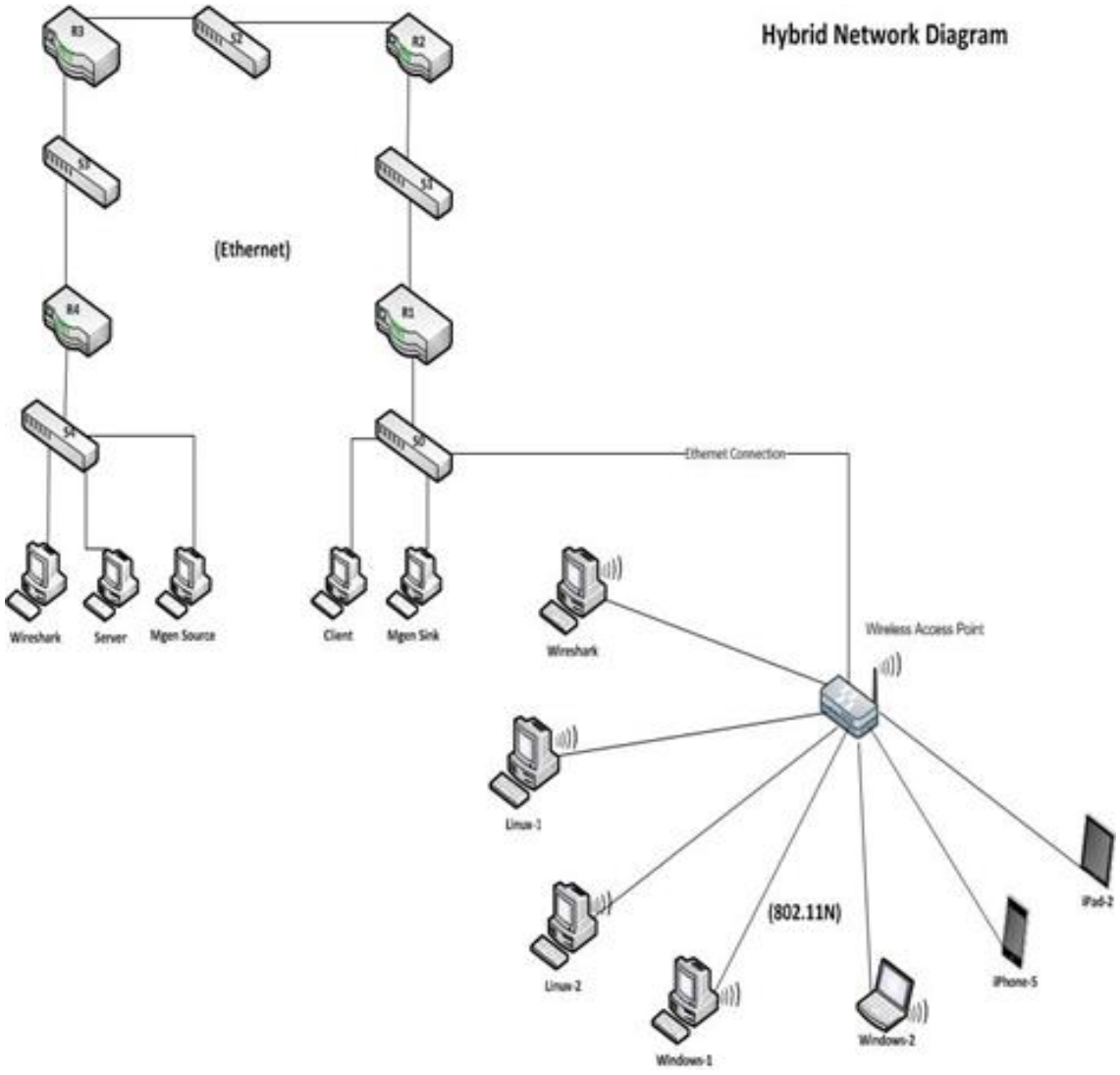


Figure 1 80211n wireless LAN and wireline Ethernet

Table 1 Ethernet Wireline Network: Hardware and Software Specification

Machine	Make	Model	Remark
Switch S0	NETGEAR	GS108T	Gigabit
Switch S1	CISCO	SG100d-08	Gigabit
Switch S2	LINKSYS: CISCO	EG005W	Gigabit
Switch S3	CISCO	SG100d-08	Gigabit
Switch S4	NETGEAR	GS108T	Gigabit
Wireshark	Dell	OPTIPLEX GX520	XP/ Wireshark Version 1.2.7 (SVN Rev 32341) -
Server	Dell	OPTIPLEX GX520	Windows server 2008 (IIS 7) **OR** Fedora 12 (Constantine) Kernel Linux 2.6.31.5-127.fc12.i686 GNOME 2.28.0 (Apache HTTP Server 2.2.16)
Mgen source	Dell	OPTIPLEX GX260	Windows XP Professional Version 2002 Service Pack 3
Mgen Sink	Dell	OPTIPLEX GX260	CentOS Version 2.16.0 Build date 02/18/2007
Client	Dell	OPTEPLEX GX520	Windows XP Professional Version 2002 Service Pack 3 **OR** Fedora 12 (Constantine) Kernel Linux 2.6.31.5-127.fc12.i686 GNOME 2.28.0
Routers Ri	Dell	OPTEPLEX GX520	Fedora 12 (Constantine) Kernel Linux 2.6.31.5-127.fc12.i686 GNOME 2.28.0

Table 2 802.11n Wireless LAN: Hardware and Software Specification

Device Function	**Details**
Access Point	Linksys 802.11n/.....
Packet Analyzer	Dell Optiplex GX520; Windows XP Professional SP2 /Wireshark version 1.6.5
Wireless Client L1	Dell OptiplexGX520; Linux Ubuntu 11.10; Kernel Linux 2.6.36....; Mozilla Firefox ..... Browser
Wireless Client L2	Dell OptiplexGX520; Linux Ubuntu 11.10; Kernel Linux 2.6.32....; Mozilla Firefox ..... Browser
Wireless Client W1	Dell Optiplex GX...; Windows 7 Professional; Mozilla Firefox .... Browser
Wireless Client W2	Dell E1705 17" Laptop; Windows 7 Professional; Mozilla Firefox .... Browser
Wireless Mobile Device-1	iPhone-5; IOS 6.0.2; Safari Browser
Wireless Mobile Device-2	iPad-2; IOS 6.0.1: Safari Browser

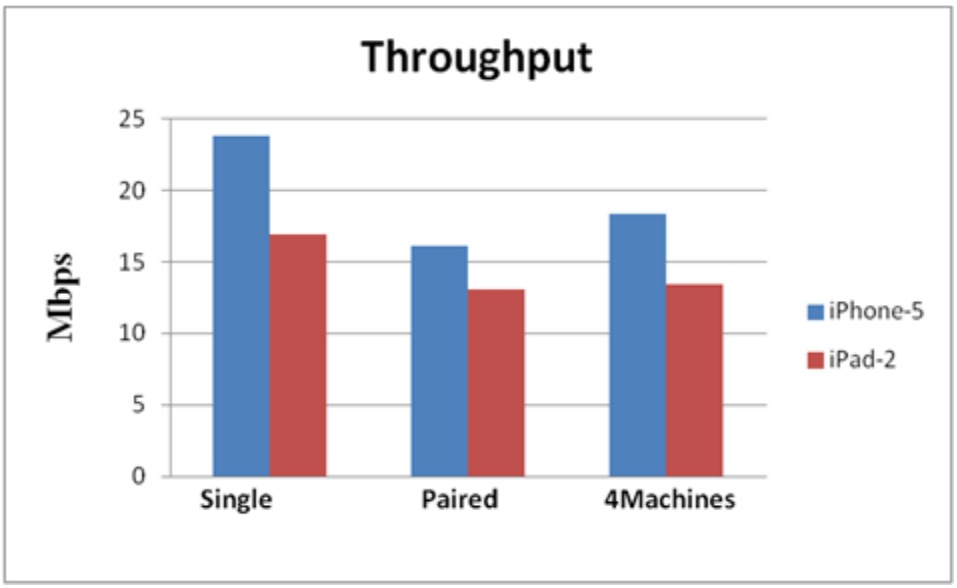
3.2 Experimental Results

All results reported are the average of three runs. Although we conducted experiments to determine performance using various combinations of the wireless clients, we only include the results using the two mobile devices, the Windows laptop W2, and the Linux desktop L1.

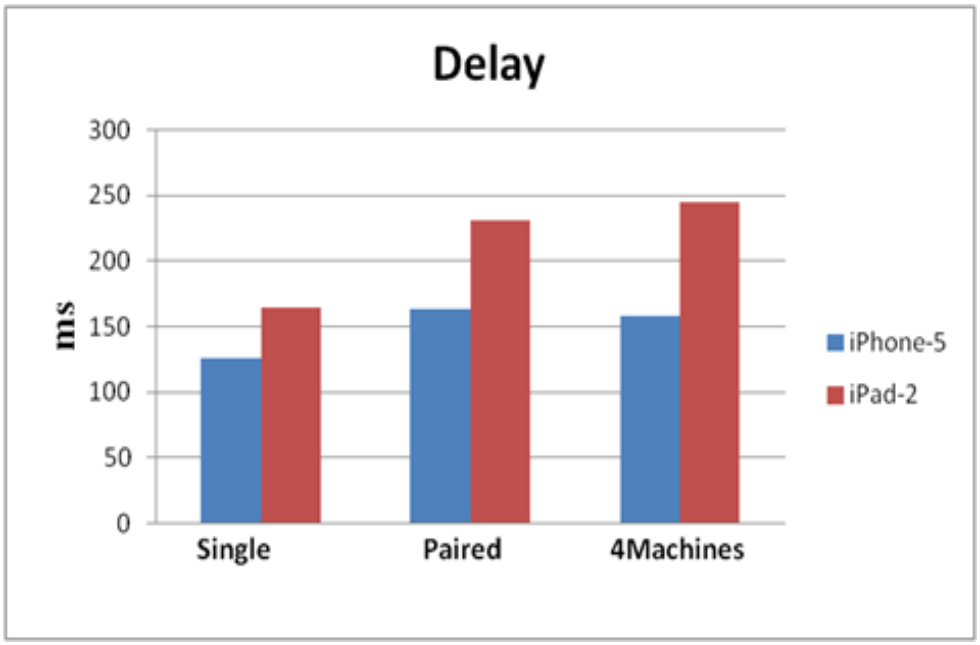
3.2.1 Performance with No Congestion

Fig.2 (a) and (b) show the throughput and delay respectively for the two mobile devices when there is no congestion traffic on the wireline network. For the 4-machine case, the mobile devices were run together with the Windows 7 laptop W2 and the Linux desktop L1. When run alone, throughput for the iPhone is 24 Mbps, which is 7 Mbps more than for the iPad. When the devices are paired, the throughput difference is only 3 Mbps and the average throughput for the devices is 14.5 Mbps. Thus, the throughput drops as expected when the devices compete for the wireless medium compared to when they run alone. However, the throughput for the iPhone is 2 Mbps less with pairing than when the four machines run together possibly due to the increase in retransmitted packets on the wireless medium when more machines compete.

For the same experiment, the highest delay (245 ms) is for the iPad with four machines, while the lowest (126 ms) is for the iPhone when it runs alone. The delays for the paired case compared to four machines is 5 ms higher for the iPhone, but 14 ms less for the iPad. The increase in delay when there is competition for the medium averages about 35 ms for the iPhone and 84 ms for the iPad, which implies that reduced processing power of a mobile device can significantly reduce its efficiency.



(a)

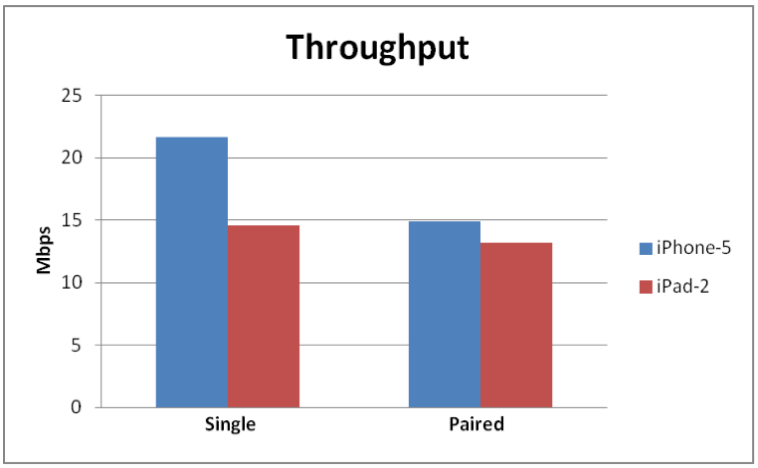


(b)

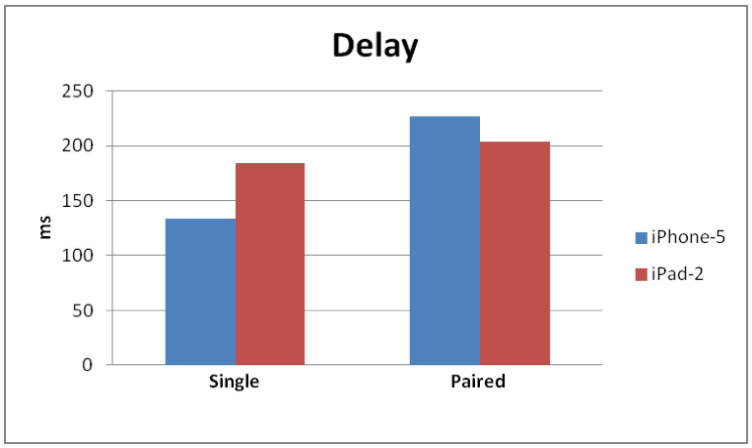
Figure 2 Performance with no Congestion

3.2.2 Performance with Congestion

Fig. 3 (a) and (b) show the throughput and delay respectively for the two mobile devices with wireline congestion traffic at 100 Mbps. For the single device case, the throughput was 22 and 15 Mbps for the iPhone and iPad respectively. When paired, throughput drops by 7 Mbps and 2 Mbps respectively for the iPhone and iPad. The corresponding delays for the single case are 133 ms for the iPhone and 184 ms for the iPad, and these delays increase by 94 and 19 ms respectively for the paired case. In general, delays increased due to congestion (by 64 ms for the iPhone in the paired case, and by 30 ms for the iPad in the single case); the only exception is that delay decreased by 28 ms for the iPad with pairing in spite of congestion.



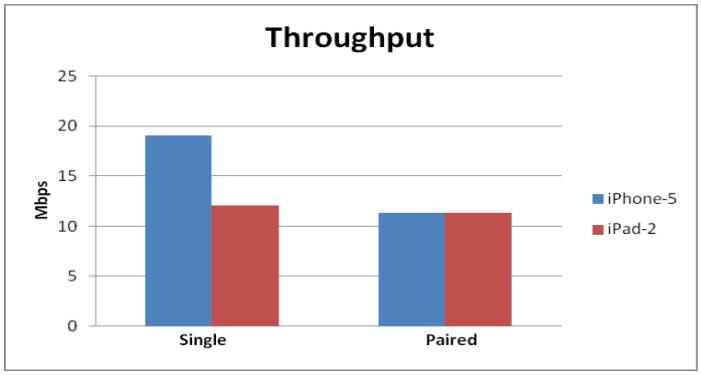
(a)



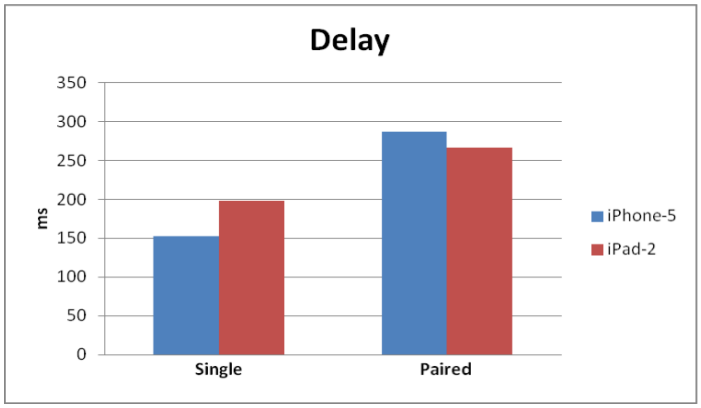
(b)

Figure 3 Single and Paired Mobile Device Performance at 100 Mbps Congestion

Fig. 4 (a) and (b) show the throughput and delay respectively for the two mobile devices with wireline congestion traffic at 125 Mbps. When the wireline congestion rate is increased from 100 to 125 Mbps, throughput drops by 3 Mbps for both devices for the single case, and by 4 Mbps and 2 Mbps for the iPhone and iPad respectively for the paired case. The respective delay increases for the iPhone and iPad are 19 and 14 ms for the single case, and 49 and 64 ms for the paired case. Although the measured throughput decrease is relatively small, the increases in delay are large and especially so with paired devices. This suggests that when there are retransmissions due to competition for the medium and/or congestion, total Wireshark throughput, which considers all packets, is not an accurate measure of 802.11 performance.



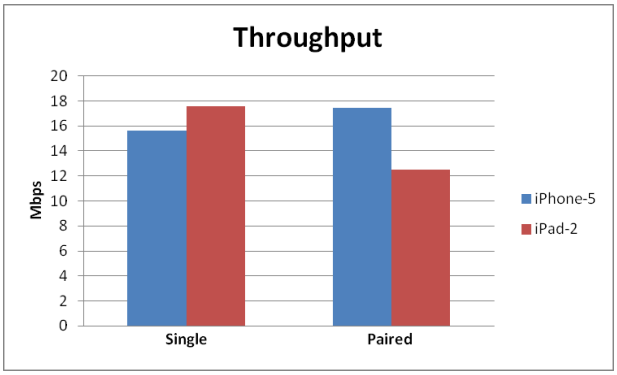
(a)



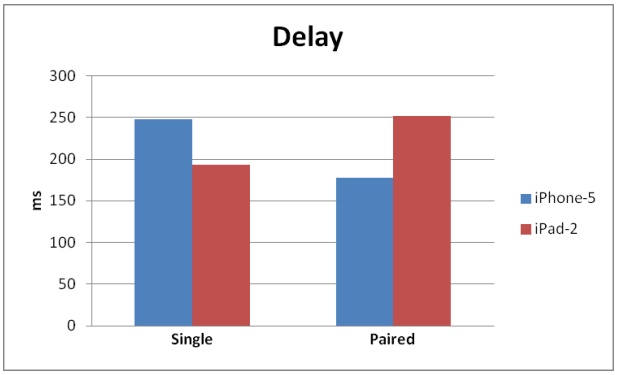
(b)

Figure 4 Mobile Device Performance at 125 Mbps Congestion

Fig. 5 (a) and (b) show the throughput and delay respectively for the two mobile devices with wireline congestion traffic at 150 Mbps. At 150 Mbps congestion, compared to 125 Mbps congestion, throughput increases by 6 Mbps with pairing for the iPhone, and by 6 and 1 Mbps respectively for the single and paired cases for the iPad. The only case in which throughput decreases compared to that at 125 Mbps congestion is for the iPhone when it runs alone (3 Mbps reduction). Again, this suggests that total throughput may not reflect the loss in performance when there are TCP or link layer retransmissions. The delays unexpectedly decrease by 108 ms for the iPhone in the paired case, and by 5 and 16 ms respectively for the iPad in the single and paired cases. It only increases for the iPhone when it runs alone (by 96 ms). The reason for this behavior could be that Wireshark timings are unreliable when there are a large number of retransmissions on the wireless medium.



(a)



(b)

Figure 5 Mobile Device Performance with 150 Mbps Congestion

3.2.3 Packet Analysis

To explain the variability in 802.11n performance for mobile devices, we analyze the actual packet exchanges and determine delays for the various parts of the exchange. Table 3 summarizes the results. Due to reasons of space, we only include the results from a typical run for the case of a single device (IPhone) accessing the medium (i.e., no competition) with no wireline congestion. This represents a best-case scenario. With the exception of the SYN packet, each delay includes the delay to access the medium prior to sending the first packet in the set. We exclude the delay prior to the sending the SYN packet since it includes the delay in manually initiating the browser request. We use the following abbreviations in the table: BA=block ack, A=802.11 ack, TA=TCP ack, and OK=200 OK.

The packets shown in the table correspond to the actual grouping of packets when transferred over the 802.11n network. For example, a typical grouping consists of between 2 to 8 data packets (1564 bytes each) followed by a block ack; an RTS/CTS exchange followed by between 1 to 12 TCP acks (116 bytes each); and a final block ack. The sizes of RTS, CTS, 802.11 acks, and block acks are 46, 40, 40 and 58 bytes respectively. These sizes include all the packet headers. Although not shown in the table, an ack (instead of a block ack) follows a single data packet. The delays shown are the total delays for each group of packets. For example, the total delay is 130 µs for a group consisting of RTS/CTS followed by 2 TCP acks, and a final block ack. This delay comprises 124 µs (access delay before RTS), 2 µs (between RTS and CTS), 1 µs (between CTS and TCP ack), 2 µs (between consecutive TCP acks), and 1 µs (preceding the final block ack). The variability

in delays is due to the initial delay in accessing the medium for any group of packets. Additional delay also resulted from retransmission of 56 packets on the wireless link.

Table 3 Actual Packet Delays

Component	Packets	Delay (μs)
TCP handshake	RTS/CTS, SYN, BA, SYN/ACK, A, RTS/CTS, TA, BA	3000
HTTP Get and 200 OK	RTS/CTS, Get, BA, TA, A, OK, A	7125
Data packets (delay for a group of k packets with 1564 bytes in each packet including all headers)	k data, BA	2755, k=8
		2501, k=7
		2315, k=6
		1949, k=5
		1501, k=4
		874, k=2
TCP acks (delay for a group of k packets with 116 bytes in each packet including all headers)	RTS/CTS, k acks, BA	352, k=12
		867, k=8
		352, k=7
		379, k=6
		81, k=5
		123, k=4
		249, k=3
		130, k=2
		628, k=1
TCP close	data/FIN, BA, RTS/CTS, 8 TA, BA, RTS/CTS, 2 TA, BA, RTS/CTS, TA, BA, RTS/CTS, FIN/ACK, BA, TA, A	4995

4 A STUDY OF A LINUX 802.11n WIRELESS DEVICE DRIVER

Transforming an existing network device driver to run on a bare machine is difficult due to its complexity and OS-dependent characteristics. To gain insight into the driver transformation problem, we studied a popular wireless Linux driver: Broadcom b43 [21]. This particular wireless driver is convenient since its source code is available. Source code for wireless drivers is often proprietary and some Linux drivers have been reverse-engineered [22, 23].

The main goal in transformation is to avoid understanding low-level functionality and details of the driver code as much as possible. However, it is necessary to determine how the b43.ko module communicates with external modules and the Linux kernel. In this paper, we examine those aspects of the Linux b43xx wireless driver that are essential as a first step towards transforming the Linux driver into its bare machine counterpart. The rest of this chapter is as follows. Section 4.1 gives the Linux wireless driver module system view. Section 4.2 includes the key code details and structures. Sections 4.3 and 4.4 describe the module interactions. Section 4.4 discusses the OS/device driver interfaces and system calls, and Section 4.5 presents driver design issues.

4.1 System View

Linux OS modules use object code interfaces instead of a binary executable. That is, linking a module does not occur until its run time, or inserting a module into the kernel is dynamic. While the module approach is a very convenient and efficient for kernel management, it is not suited for bare machine transformation. The dependencies of a Linux driver module are somewhat complex, and it is difficult to decouple the module from the Linux kernel. In general, the kernel provides a variety of services including process

management, memory management, module management, dynamic linking and error checking. Transforming the existing driver source code requires understanding details of the driver and its interactions with the rest of the system.

Fig. 6 shows a system view of the Broadcom b43 wireless driver for Ubuntu 3.9.3. In this figure, we have labeled the key interfaces as #KSC and #EMC and the data symbols as #EDO for convenience. We explain later the significance of the numbers shown alongside the edges. The wireless module b43.ko, which incorporates essential driver functionality, has many dependencies on other modules and the kernel. In particular, b43.ko interacts with BCMA (Broadcom Microcontroller Architecture), SSB (Sonics Silicon Backplane), skbuf (socket buffer structure), LED (Light Emitting Diode), DMA (direct memory access), PRINT (print facilities) and IEEE802.11 (wireless standard). In addition, b43.ko also uses some system calls provided by the OS.

The b43.ko module exports symbol names (functions and variables) in order to provide a global scope so that other modules can use these interfaces. The module also has certain parameters defined externally and passed to the module when it is instantiated. These variables can change the internal behavior of the module using #ifdef statements.

In order to test the b43.ko module, we need the whole system running as a single monolithic executable. We use a variety of techniques to test the module including PRINTK statements, log traces, Objdump, KGDB, and intercepting interrupts using sys_call_table.

4.2 Code View

The “linux-source-3.9.3” version for Ubuntu 12.10 version has 29 “C” source code files and “30” header files. Each source file has a corresponding header file and the extra header file b43.h consists of data definitions, constants and inline functions for b43 module. In addition, there are 41 header files for libraries and system calls. Table 4 shows the grouping of b43 source files based on their functionality. The total source code size is 45,483 lines and the header code size is 7,446 lines (52,929 lines in all). Fig. 7 shows the code percentages for the b43 code groups identified in Table 4. It is evident that this device driver covers many features and options not required for a typical usage of the driver. For example, when we tested the module with a desktop PC running Ubuntu, LED Control, Debug, and PCMCIA are not used. When we eliminate code sizes that are not relevant to a typical application, the total code size becomes 35,943 lines (that is 68% of the original code).

We also analyzed the b43.ko module using the Source Insight tool [24]. Table 5 shows the module summary it gave for b43.ko. It indicates the number of functions (899), function prototypes (220) and external variables (43). The largest contributors to b43 are constants, functions and structure members. The b43.h header file contains many data structures, definitions and system header files. Fig. 8 shows the components of this header file and their relationships analyzed by using the Doxygen tool [25]. There are 18 system header files in this header including 802.11mac.h, ssb.h, and kernel.h. The right side of the figure shows the relation between the 802.11 physical layer header files phy_a.h, phy_g.h, phy_common.h, and nl802.11.h.

We also classify the 899 functions found in Table 5 based on their functional unit as defined in Table 4. Fig. 9 shows the number of functions in each unit. In this classification, the largest unit is phy (43%). If we ignore functions that are not used in our desktop PC and b43.ko wireless module connection with security turned off, we only need 583 functions (which are 65% of total functions implemented in the module). We also generated a log file with PRINTK statements in each function, which shows that the only functions activated are those in the files dma.c, leds.c, main.c, phy_common.c, phy_n.c, xmit.c, and b43.h (inline). This amounts to 440 out of 899 functions, which is 49% of the total functions implemented in the module. This implies that the b43.ko module has about 50% of functions that may be unused at a given time for a given setup.

We have also compared the Linux wireless driver with a Windows version of its counterpart (WMP300Nv1.sys). The source code for this Windows driver is not available. Table 6 shows a comparison of assembly source lines of code and number of "call" statements in the Windows and Linux drivers. The Linux driver assembly code is almost three times smaller as it is a module and requires other interfaces at execution. The number of CALL statements in the code includes local calls and external interfaces.

4.3 External Interactions

While the code views shown in the previous section show the structure of the b43.ko module, it does not reflect module dependencies and kernel interactions. There are many tools available to identify the external interactions and OS related calls for this module. For example, "objdump" (available in Linux and Windows) has a variety of options that give useful information about the object module. Similarly, the GNU's "nm" utility provides undefined symbols (UND) that indicate system calls and other library

interfaces. In addition, the Linux command "/proc/kallsyms" in Linux lists all symbols in the kernel.

The parameters passed to the module can be obtained by using "ls –l /sys/module/b43/parameters" command. In this module there are nine parameters that can be passed to the module (bad_frames_preempt, btcoex, fwpostfix, hwpctl, hwtkip, nohwcrypt, pio, qos, and verbose). We removed the module from the kernel using "sudo rmmod b43", and checked the exported symbols using "cat/proc/kallsyms | grep b43_". Surprisingly, we found that three symbols remain in the "/proc/kallsyms" file even after removal of the module (b43_pci_bridge_driver, b43_pci_bridge_tbl, and b43_pci_ssb_bridge_exit). One possible is that other modules using the b43_ label may have defined these three symbols since there are no such symbols referred to in the b43.ko module. We inserted the module into the kernel using "sudo modprobe b43" and checked the exported symbols again. This time, there were 771 symbols in the exported table. These symbols consist of 471 functions and the rest are data symbols. The b43.ko module exports these 471 functions for other modules to use. Because we did not find any EXPORT statements in the b43 source code, it appears that the command "modprobe" generated these EXPORT symbols during the insert or dynamic link time.

To understand the 471 functions above and to get more details on b43.ko, we used the "objdump" and "nm". The command "objdump –t b43.ko > objdump.txt" gives all symbols in the b43.ko module. There were 998 lines captured in the objdump.txt file. Likewise, the command "grep g objdump.txt > globals.txt" gives all global symbols in the b43.ko module, which were found to 195 symbols in this module. The command "grep .text globals.txt > globalfun.txt" showed that there are 147 functions in this module. The

rest of the global symbols (195-147 = 48) are global data parameters that consist of the .data and .rodata sections of the module (#EDO in Fig. 6). The command "grep UND objdump.txt > syscalls.txt" identified 127 system calls that were needed in the module. The "nm" command also gave similar results. The command "nm –g b43.ko > nmext.txt" resulted in 322 global symbols. The command "grep U nmext.txt > nmsyscalls.txt" also showed that there are 127 system calls in the module, which serves to confirm this number. The 127 external system calls include all the interfaces shown in Fig. 6 except the #EDO. To transform the Linux b43 wireless driver into a wireless driver that runs on a bare machine, we need the equivalents of these 127 kernel/system calls (#KSC and #EMC interfaces in Fig. 6).

We also collected more data for the BCM43xx Windows wireless driver (WMP300Nv1.sys, which is not open source). Fig. 10 shows its system view and relation to HAL (hardware abstraction layer), NDIS (Network Driver Interface Specification) and Kernel. All these components are DLL elements that work with the wireless driver. The HAL component has three interfaces, which consist of acquire spin lock, release spin lock and stall processor. The NDIS has 41 interfaces, which consist of wrapper interfaces to enable the driver to run in a Linux environment. There are 29 kernel interfaces, which are similar to the #KSC interfaces in Fig. 6. The total number of external interfaces or system calls for the Windows driver is 73, compared to 127 for the Linux driver. Linux needs more interfaces, although it is a module and not an executable. We determined that eight interfaces here are similar to #KSC in Fig. 6, while the rest are unique to Windows. There are less system calls in the Windows driver compared to its Linux counterpart because the Windows encapsulates more functions into its driver. Windows bases its DLL approach on

executable modules instead of object modules enabling the driver to be more self-contained.

4.4 Kernel/Device Driver Interfaces

The #KSC, #EMC interfaces and #EDO data symbols as identified in the previous section enable the b43.ko module to interact with the Linux kernel and other modules. The 127 system interfaces identified previously for b43 consist of 63 #KSC and 64 other interfaces (15 SSB, 23 IEEE802.11, 5 SKBUFF, 6 DMA, 2 LED, 11 BCMA, and 2 LED) as shown in Fig. 6. Similarly, the 48 EDO interfaces consist of .data (4), .rodata (43, read only data), and .gnu.linkonce.this_module (1) objects. These objects represent local and remote data objects.

4.5 Driver Design Issues

The b43.ko module operates with the Linux OS. While the Windows driver provides NDIS support to port the driver to the Linux environment, it still requires some Linux system calls after porting. An alternate approach is to provide appropriate and equivalent interfaces to #KSC, #EMC, and #EDO enabling the driver to run with other operating systems and on a bare machine. Ideally, the wireless device driver specification can be an ADO (Abstract Data Model) that is analogous to the USB (Universal Serial Bus) standard, SCSI (Small Computer System Interface) standard, or other standards available in computing. In our view, it should be a USB standard as many devices already use USB interfaces. This will allow us to wrap all wireless commands inside a USB command payload and the driver to execute these commands while hiding the complexity.

Then we can design a wireless device driver as a self-contained object that provides a high-level API to applications including the kernel. This API does not micro-manage the

wireless driver. That is, an application programmer directly invokes its API, and there is no kernel or embedded system running in the machine. Also, it should behave similar to other wired network interface card (NIC) drivers, which have a standard set of operations such as Initialize(), Reset(), Read(), Write(), Configure() and so on. It should provide an API that is standard and analogous to SCSI or USB interfaces.

Another issue with Linux and b43.ko interdependencies is the complexity of the module architecture. The b43.ko module has too many external interfaces making it difficult to de-couple the driver module from its operating environment. By modeling a wireless device driver as a self-contained object it becomes possible to eliminate all other module interactions and provide a standard interface directly to the application programmer. This is the approach used to develop device drivers for a variety of bare machine applications. These drivers require no OS, kernel, or embedded system to support their operation.

The b43.ko module also incorporates general functionality used rarely in a typical operating environment. Such functionality makes the module itself complex and large. This in turn makes it harder to understand, port, and test its operation. Bare machine drivers can solve these problems; however, more research is required to understand how to transform existing OS drivers to remove their OS dependencies.

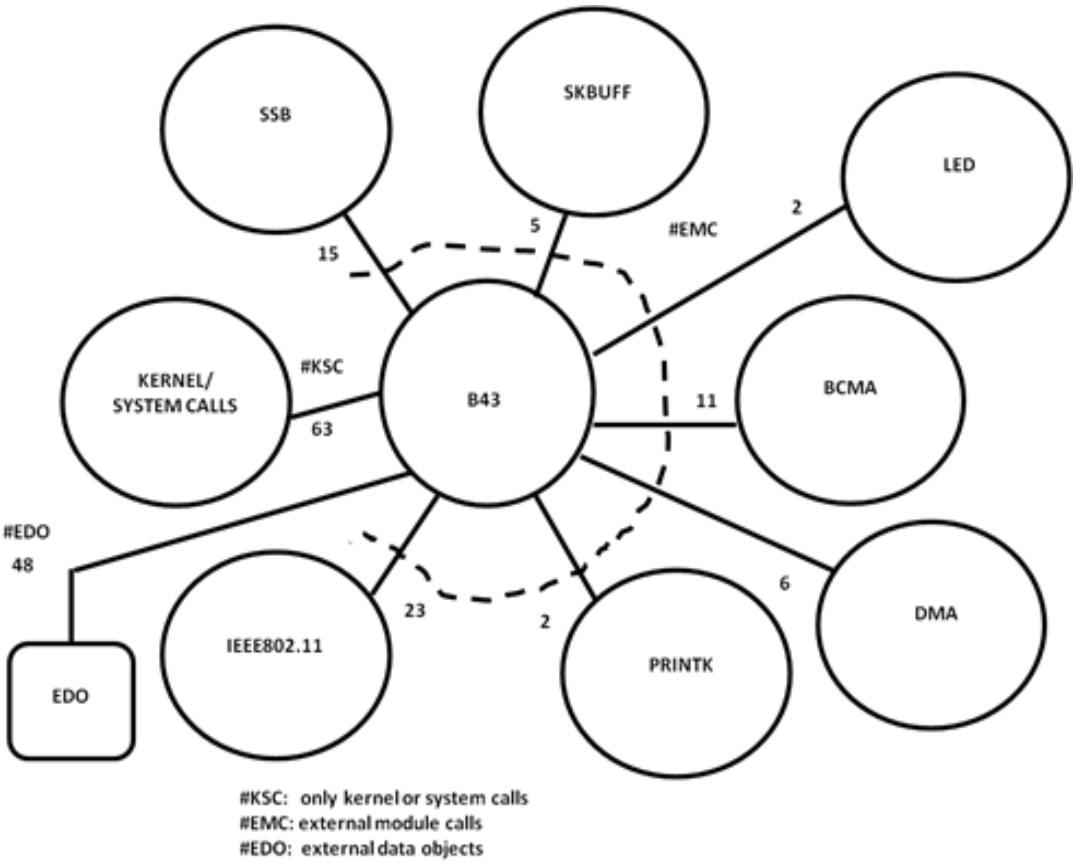


Figure 6 System View (Linux Module)

Table 4 B43 Source Files

bus.c	Bus Abstraction
debugfs.c	Debug
dma.c	DMA
leds.c	LED Control
lo.c	Local Oscillator
main.c	Main
pcmcia.c	PCMCIA Card Interface
phy_a.c	PHY -a
phy_common.c	PHY common
phy_g.c	PHY -g
phy_ht.c	PHY -h
phy_lcn.c	PHY -l
phy_lp.c	PHY -lp
phy_n.c	PHY -n
pio.c	Parallel I/O
radio_2055.c	Radio Dev Data Tables
radio_2056.c	Radio Dev Data Tables
radio_2057.c	Radio Dev Data Tables
radio_2059.c	Radio Dev Data Tables
rfkill.c	Radio Enable
sdio.c	Secure Data Card Int
sysfs.c	Virtual File System
tables.c	Radio Device Data Tables
tables_lpphy.c	Radio Device Data Tables
tables_nphy.c	Radio Device Data Tables
tables_phy_ht.c	Radio Device Data Tables
tables_phy_lcn.c	Radio Device Data Tables
wa.c	Work arounds
xmit.c	TX/RX Functions

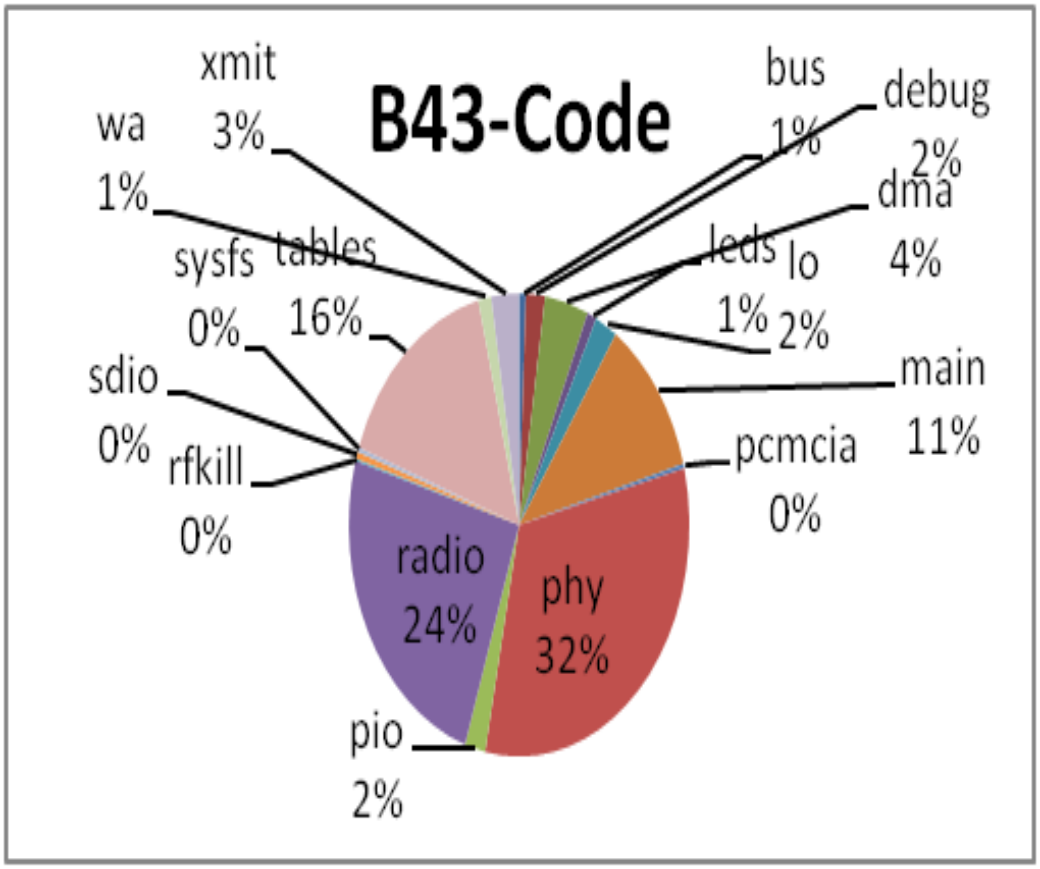


Figure 7 Percentages of Code Sizes per Group

Table 5 B43 Module Dissection

Summary of Items on b43 driver
Structures (109) Macros (68)
Unions (10) Function Prototypes (220)
Enumerations (24) Method Prototypes (23)
Constants (4275) Structure Members (891)
Enum Constants (118)
Variables (257)
External Variables (43)
Functions (899)

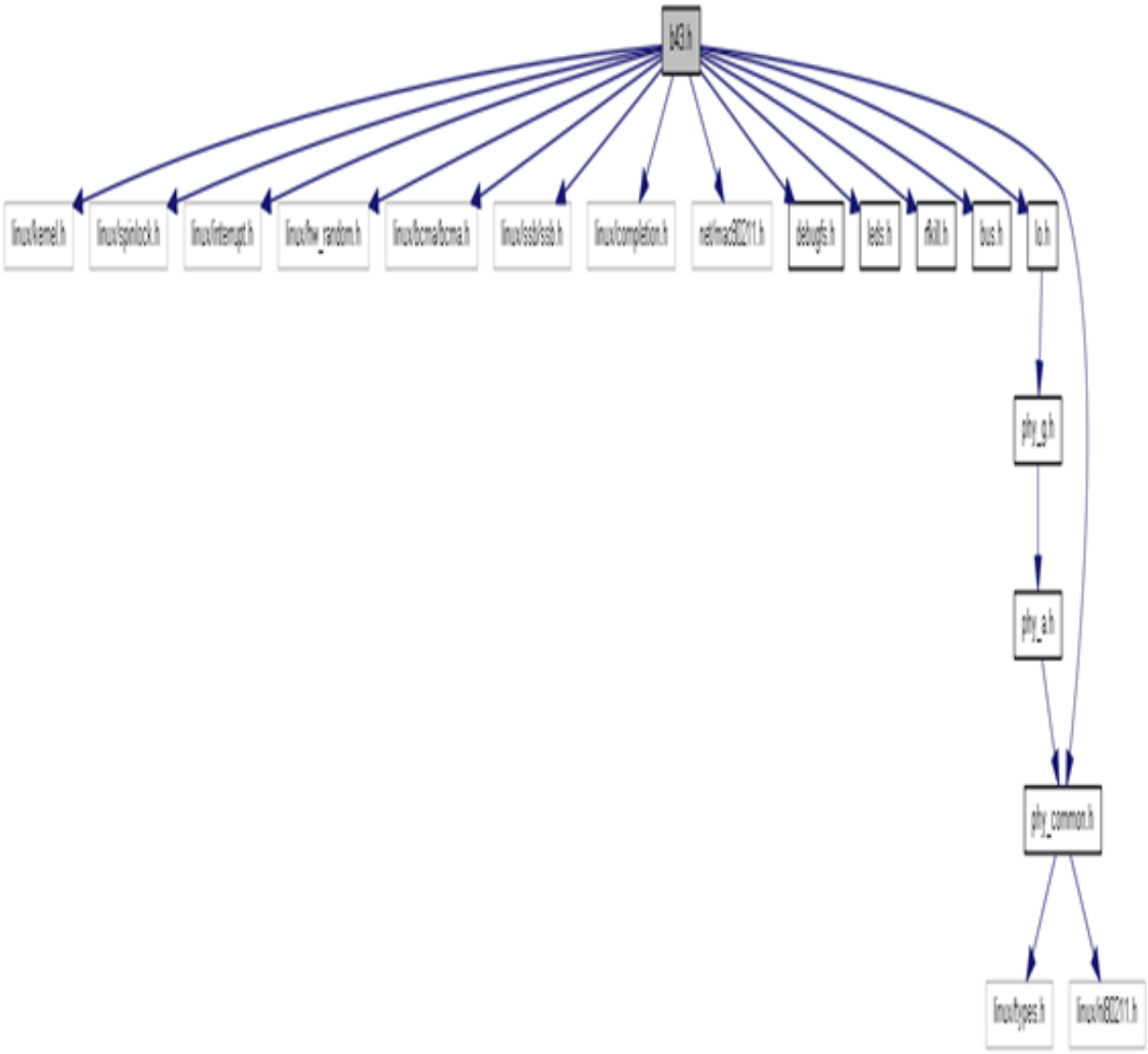


Figure 8 b43.h Header Structure

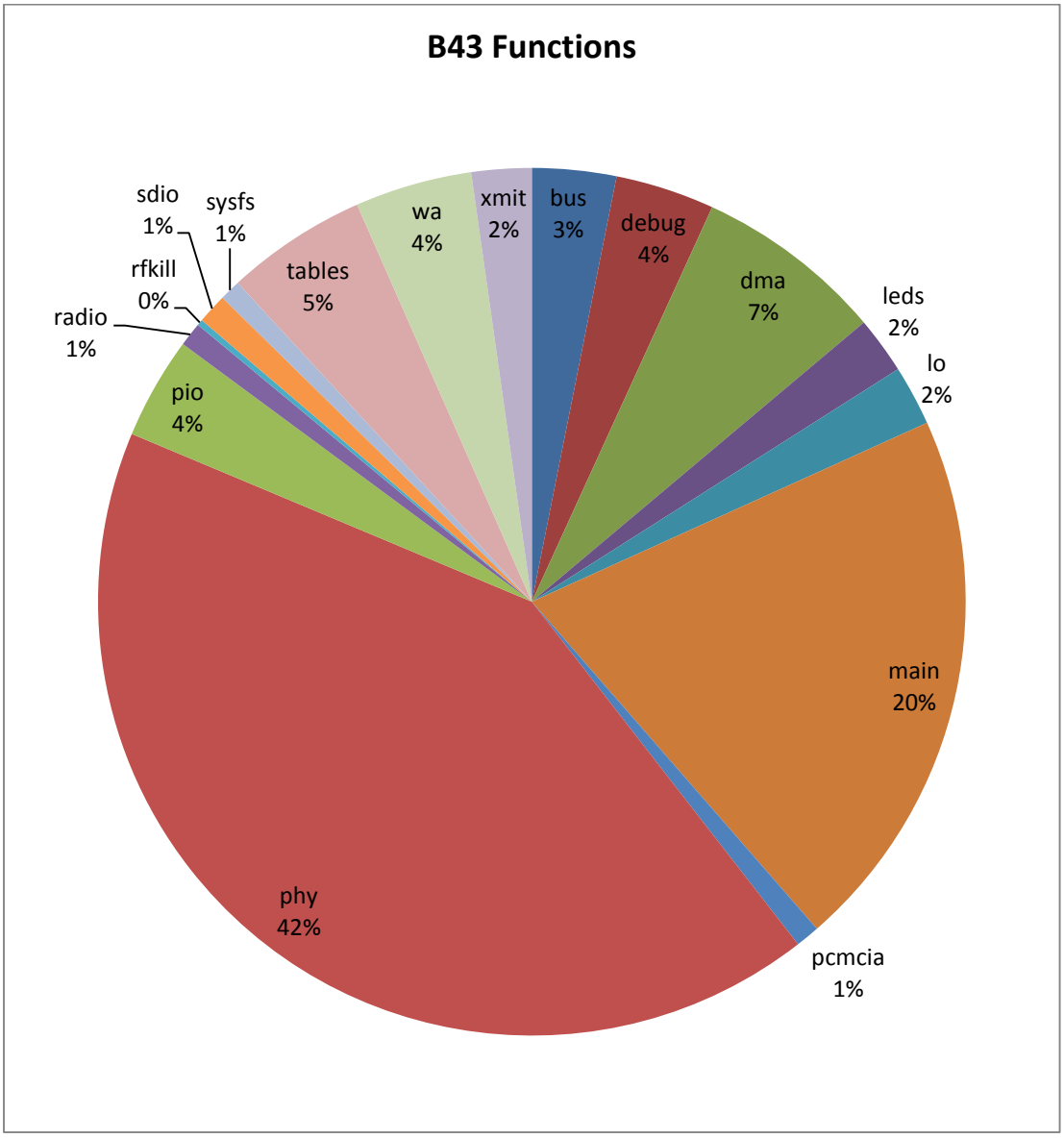


Figure 9 Percentages of Functions per Group

Table 6 (LoC, # of Calls) for Linux and Windows

Driver	**Assembly Lines of Code**	**Number of CALLs in the code**
Linux	53,863	7340
Windows	160,490	9798

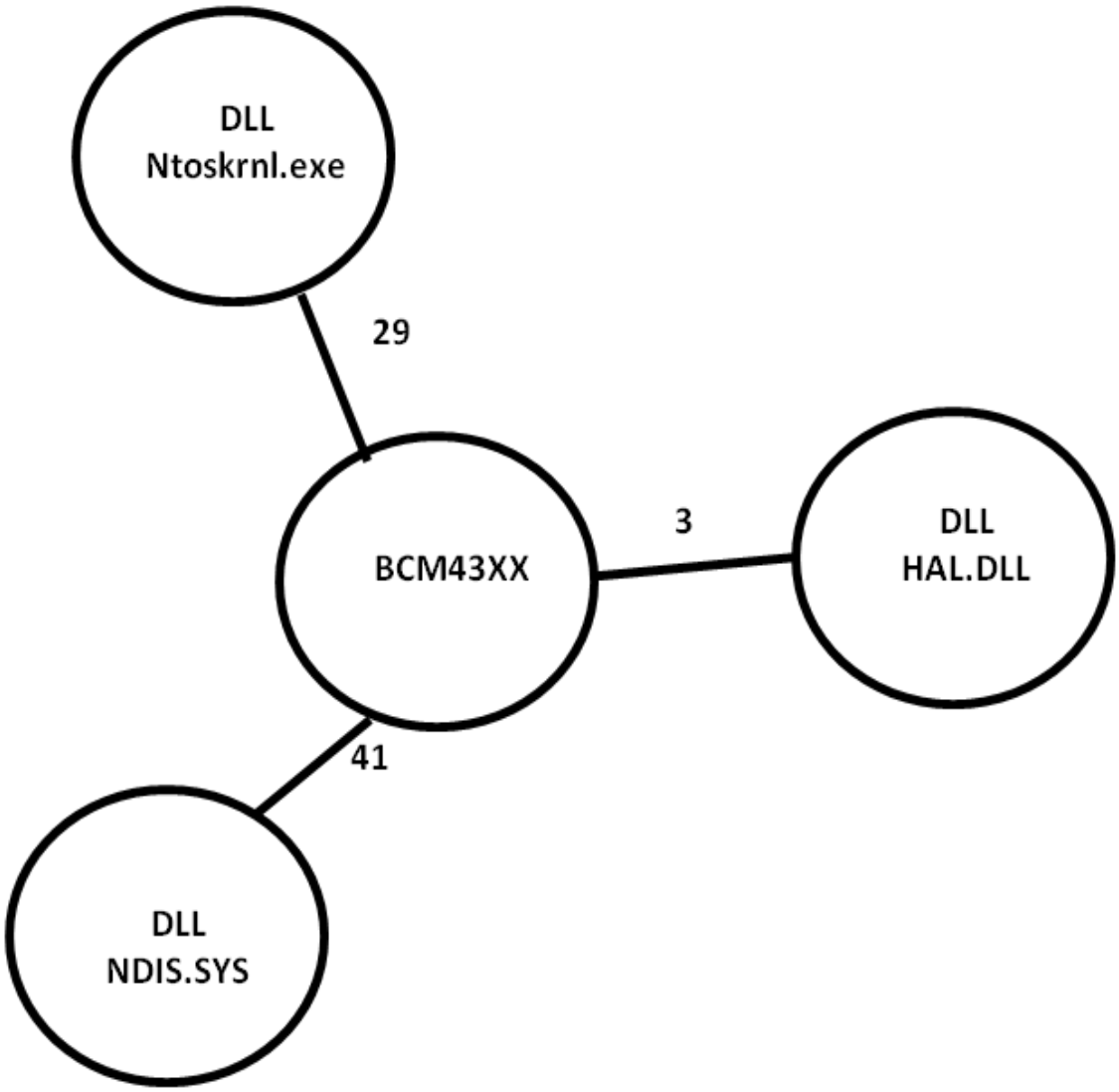


Figure 10 System View (Windows Driver Interfaces)

5 A STUDY OF CCM MODE

The main goal of our study of CCM mode is to develop an implementation that runs on a bare machine. We provide below some basic background on CCM mode, discuss both OS-based and bare machine implementations, and conduct studies to verify interoperability and evaluate the performance of the bare machine implementation.

5.1 802.11i Security Aspects

AES-CCM mode is the basis for the 802.11i security standard CCMP (Counter Mode Cipher Block Chaining Message Authentication Code Protocol). As discussed below, AES-CCM mode is an authenticated encryption standard that combines counter mode encryption with CBC MAC authentication. CCMP uses an 8-byte authentication tag and a 2-byte message length field. The key size is 128 bits. CCMP protects the data transferred over an 802.11 wireless link. Other aspects of 802.11i security deal with authenticating users and access points, and establishing and managing keys.

5.2 AES-CCM Mode

AES-CCM Mode uses AES counter mode for encryption and AES CBC MAC for authentication. CCM mode allows variable length encoding with tradeoffs involving the sizes of the fields specifying the lengths of the message, the authentication tag, the nonce, and the counter. For example, within a 16-byte IV that encodes a 1-byte flags field, the size of the message length field is L bytes and the size of the nonce length field is 15-L bytes. Similarly, the size M of the authentication tag field is even and can vary between 4 and 16 bytes.

5.3 Implementation

As discussed in the related work section, there are many implementations of AES CCM mode, but none is directly usable in a bare machine. One approach to developing a bare machine implementation is to remove system dependencies from an OS-based implementation. The other approach is to implement CCM mode for a bare machine using its specification.

5.3.1 AES-CCM Linux and Windows Implementations

AES-CCM implementations exist for both Linux and Windows. We studied the OpenSSL implementation on Linux Ubuntu 16.04, 15.10 [37], and the Crypto++ 5.6.3 implementation on Windows 7 [38]. The Linux OpenSSL implementation in C uses two functions aes_ccm_encrypt and aes_ccm_decrypt. Each function calls other functions that implement the internals of AES-CCM. For example, the aes_ccm_encrypt function calls EVP_CIPHER_CTX_new, EVP_EncryptInit_ex, EVP_CIPHER_CTX_ctrl, EVP_EncryptUpdate, EVP_EncryptFinal_ex, and EVP_CIPHER_CTX_free. The Windows Crypto++ implementation in C++ requires the classes AuthenticatedEncryptionFilter and AuthenticatedDecryptionFilter, and calls its members ChannelPut, ChannelMessageEnd, GetLastResult, SetRetrievalChannel, and MaxRetrievable. Since these dependencies are OS-specific, it is difficult to use these Linux or Windows CCM mode implementations as a basis for developing an OS-independent CCM mode implementation that runs on a bare machine.

5.3.2 Bare Machine Implementation

The methods aesccmencrypt and aesccmdecrypt respectively implement the CCM mode encrypt/authenticate and verify/decrypt operations in the bare machine implementation. The aesccmencrypt method has two parameters, message and message length. When it completes, the message includes the IV and additional authenticated data (in the clear), and the message plus the authentication tag (both encrypted). The method returns the total length of the result i.e., the length of authenticated encrypted message and authenticated data plus the length of the IV and the authentication tag. The aesccmdecrypt method also has two parameters. The message parameter points to the all the bytes resulting from encryption and the total length as described above. When verification is successful (which also means decryption is successful), the message parameter points to the decrypted original message and the method returns the size of the original message. The bare machine CCM mode implementation requires an existing implementation of the AES CBC mode algorithm used previously in many bare machine applications.

We integrated the bare machine CCM mode implementation with a simple bare machine application that can send and receive UDP messages. As in other bare machine applications, the UDP application includes a main task and receive task together with code for booting and executing the application. We modified the UDP handler method in the bare machine UDP implementation enabling it to call the aesccmencrypt and aesccmdecrypt methods to perform authenticated encryption of UDP messages using CCM mode.

5.4 AES-CCM Performance Study

This study presents delay measurements for AES-CCM performance using a small test network that connects the server and client via an Ethernet switch. The server and client run on identical Dell GX 960 machines. We connected a third Dell GX 960 running Windows 7 to the switch to capture packets exchanged between the server and the client. We test Bare-Bare (BB), Linux-Linux (LL), and Bare-Linux (BL) combinations of the client and the server. For the BL combination, we consider both cases of a bare client connecting to a Linux server and a bare server connecting to a Linux server. Fig. 11 shows the test network used for the experiments.

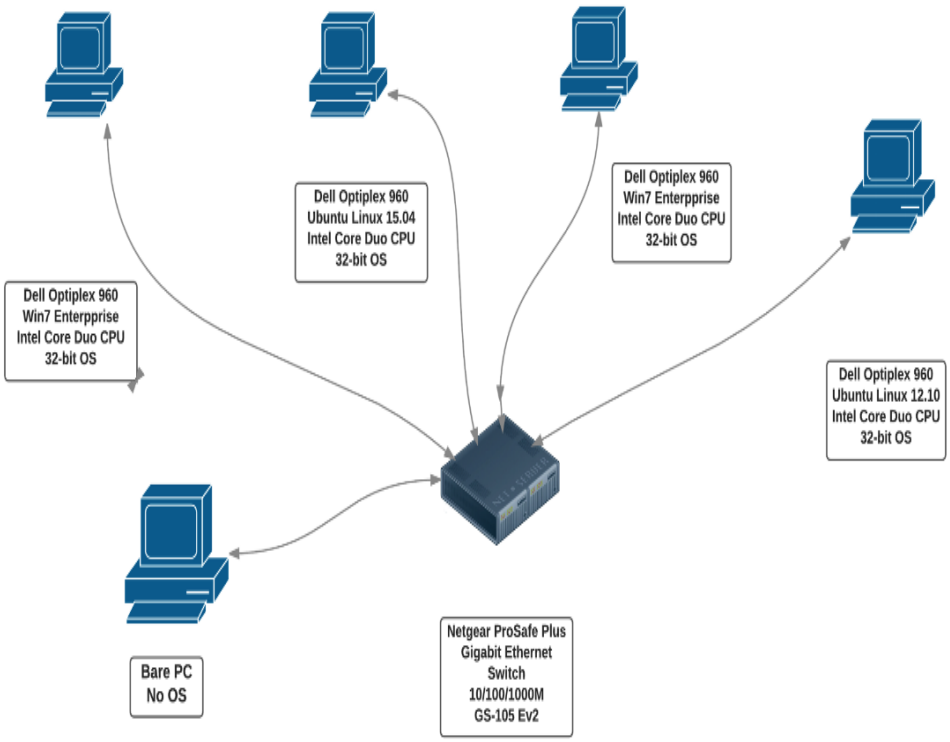


Figure 11 AES-CCM test network

5.4.1 Experiments

Fig. 12 shows the UDP client/server communication scheme with the messages exchanged, and the timing points used for encrypted and unencrypted messages. First, the client sends an unencrypted message to the server. When the server receives the message, it responds by sending m unencrypted messages to the client. After it receives all these messages, the client sends one CCM authenticated encrypted message to the server, which the server verifies and decrypts. It then sends m CCM authenticated encrypted messages to the client, which the client verifies and decrypts. All unencrypted and encrypted UDP messages carry a packet of the same size n as their UDP payload. In encrypted UDP messages, we only encrypt this UDP payload (i.e. we encrypt the packet of size n, but not the UDP header).

For the AES-CCM tests, we use m=1000 packets and packet sizes n=50, 250, 500, 750, and 1000 bytes. The server and client are restarted for each run with a given packet size. No other traffic is on the network during the test runs. The source code of the server and client applications include timing points to find internal timings for 1000 unencrypted and encrypted packets respectively. The internal timings for encrypted packets include total time to encrypt and send 1000 packets at the server, and total time to receive and decrypt 1000 packets at the client. We also find the contribution of network delay by using Wireshark timestamps of packets captured during the transfer.

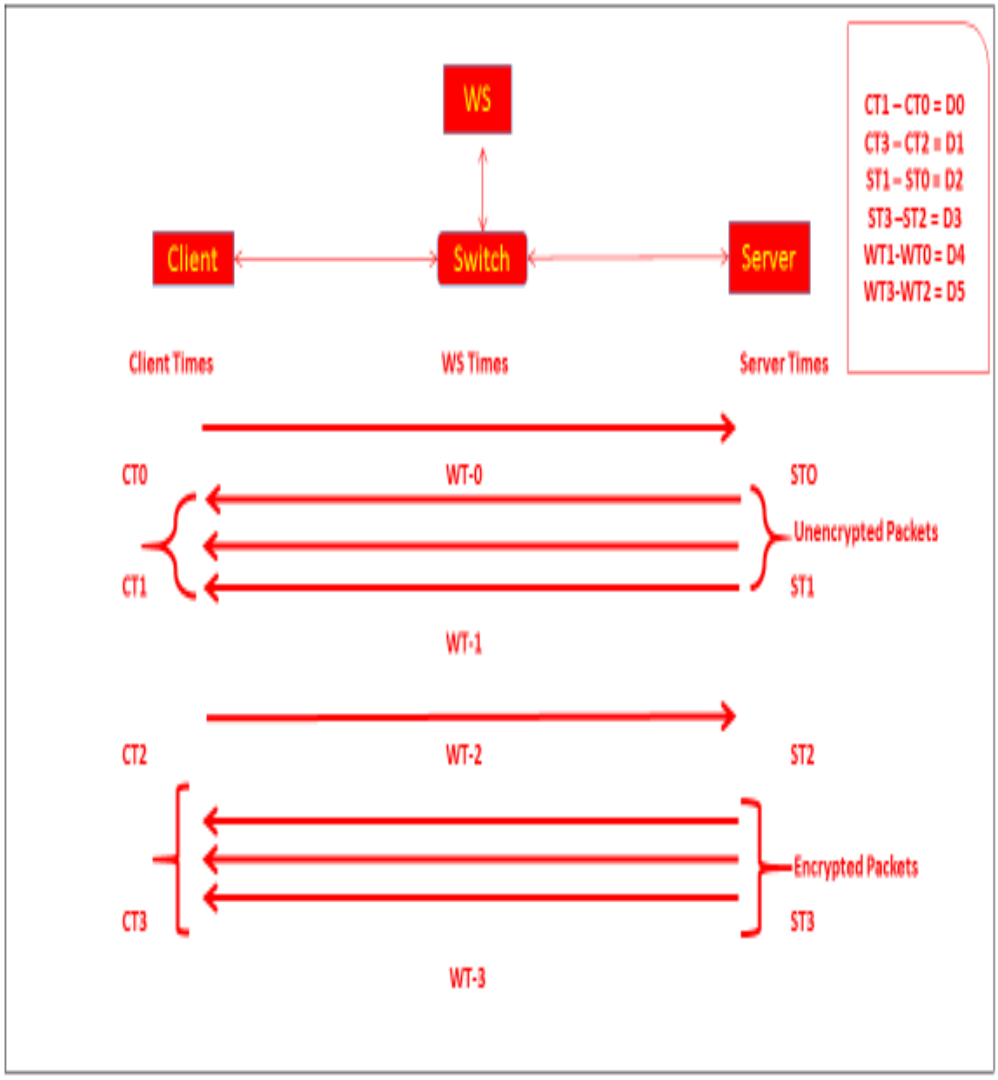


Figure 12 AES-CCM communication scheme

5.4.2 Encryption Time

Referring to Fig. 12, the difference of internal timings D3=ST3-ST2 measured at the server is the total time to encrypt, process, and send 1000 packets of a given size to the client. Similarly, D2=ST1-ST0 is the total time for the server to process and send 1000 unencrypted packets of a given size. Fig. 13 shows these times measured using the bare server when sending to a bare client or a Linux client, and Fig.14 shows the corresponding times using the Linux server.

For encrypted and unencrypted packets, as the size varies from 50 bytes to 1000 bytes, bare server times increase from 11.5 to 66.5 ms and 4.25 to 9.25 ms respectively with the bare client, and from 24 to 79.75 ms and 16.25 to 21.5 ms respectively with the Linux client. The corresponding increases for Linux server times for encrypted packets are 10 to 84 ms with both the bare and Linux clients; and for unencrypted packets are 6 to 79 ms and 6 to 80 ms respectively with the bare and Linux clients. When sending to the bare client, the bare server takes less time than the Linux server does for encrypted packets of all sizes except 50 bytes, and unencrypted packets of all sizes. When sending to the Linux client, the bare server takes more time than the Linux server does for encrypted packets of all sizes less than or equal to 500 bytes, and less time than the Linux server does for unencrypted packets of all sizes except 50 bytes (the difference for 50 bytes with unencrypted packets is only 250 microseconds).

5.4.3 Decryption Time

Referring to Fig. 12 again, the difference of internal timings D1=CT3-CT2 measured at the client is the total time to receive, process, and decrypt 1000 packets of a given size sent by the server. Similarly, D0=CT1-CT0 is the total time for the server to receive and process 1000 unencrypted packets of a given size. Fig. 15 shows these times measured using the bare client when receiving from a bare server or a Linux server, and Fig.16 shows the corresponding times using the Linux client.

For decrypted packets, as the size varies from 50 bytes to 1000 bytes, bare client times increase from 31.5 to 89.25 ms and 32 to 90.25 ms respectively with the bare and Linux servers; and for unencrypted packets from 13.75 to 85.5 ms with both the bare and Linux servers. The corresponding increases for Linux client times for decrypted packets are 24 to 91 ms and 15 to 91 ms respectively with the bare and Linux servers; and for unencrypted packets are 17 to 86 ms, and 10 to 86 ms respectively with the bare and Linux servers. For decrypted packets, the bare client takes more time than the Linux client does with both the bare and Linux servers for packets of all sizes except 1000 bytes (the difference for 1000 bytes with the bare server is 1.75 ms, and with the Linux server is 750 microseconds). For unencrypted packets, the bare client takes less time than the Linux client does with both the bare and Linux servers for packets of all sizes except 50 bytes.

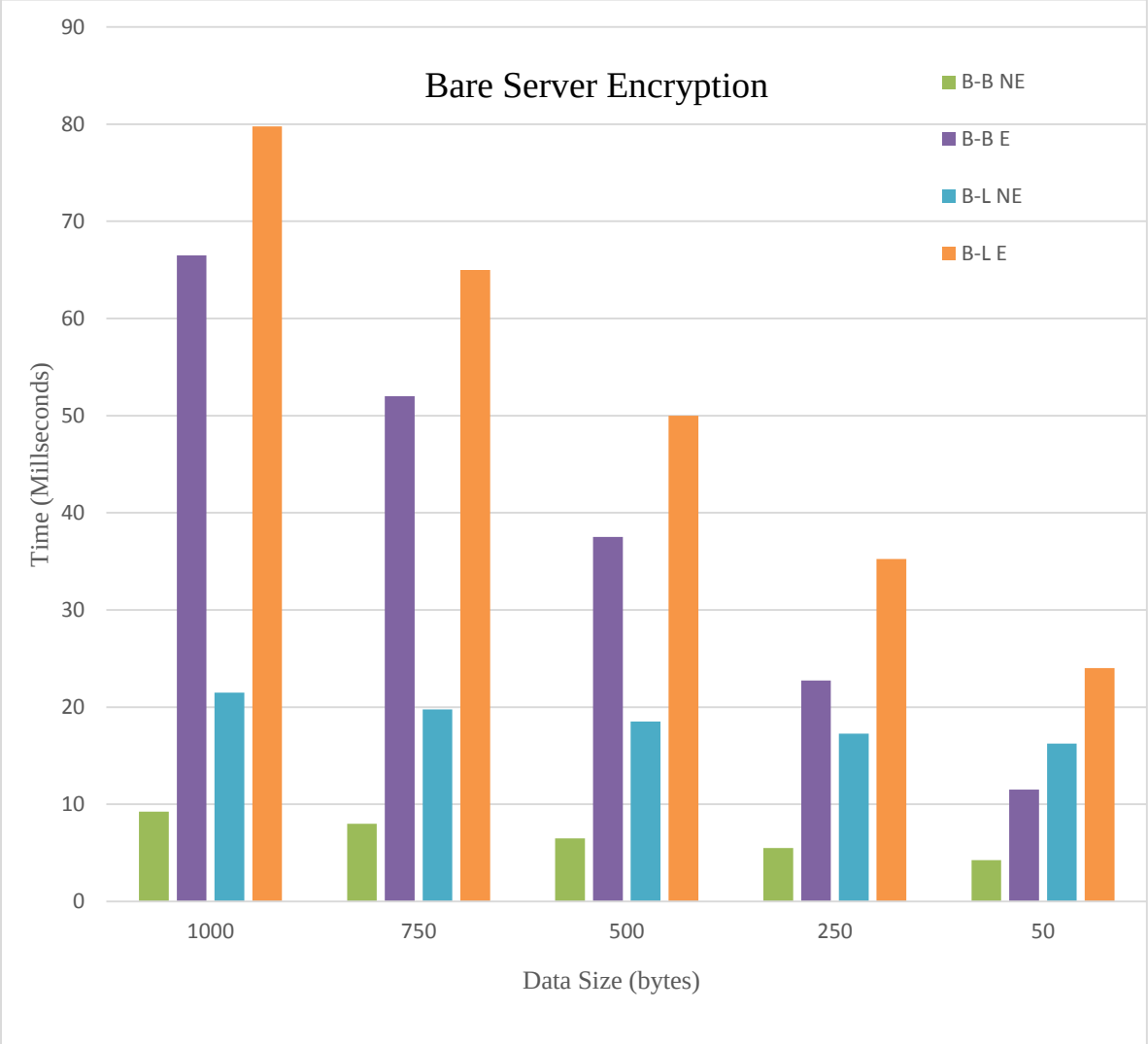


Figure 13 Bare Server Encryption: Internal Timings

Figure 14 Linux Server Encryption: Internal Timings

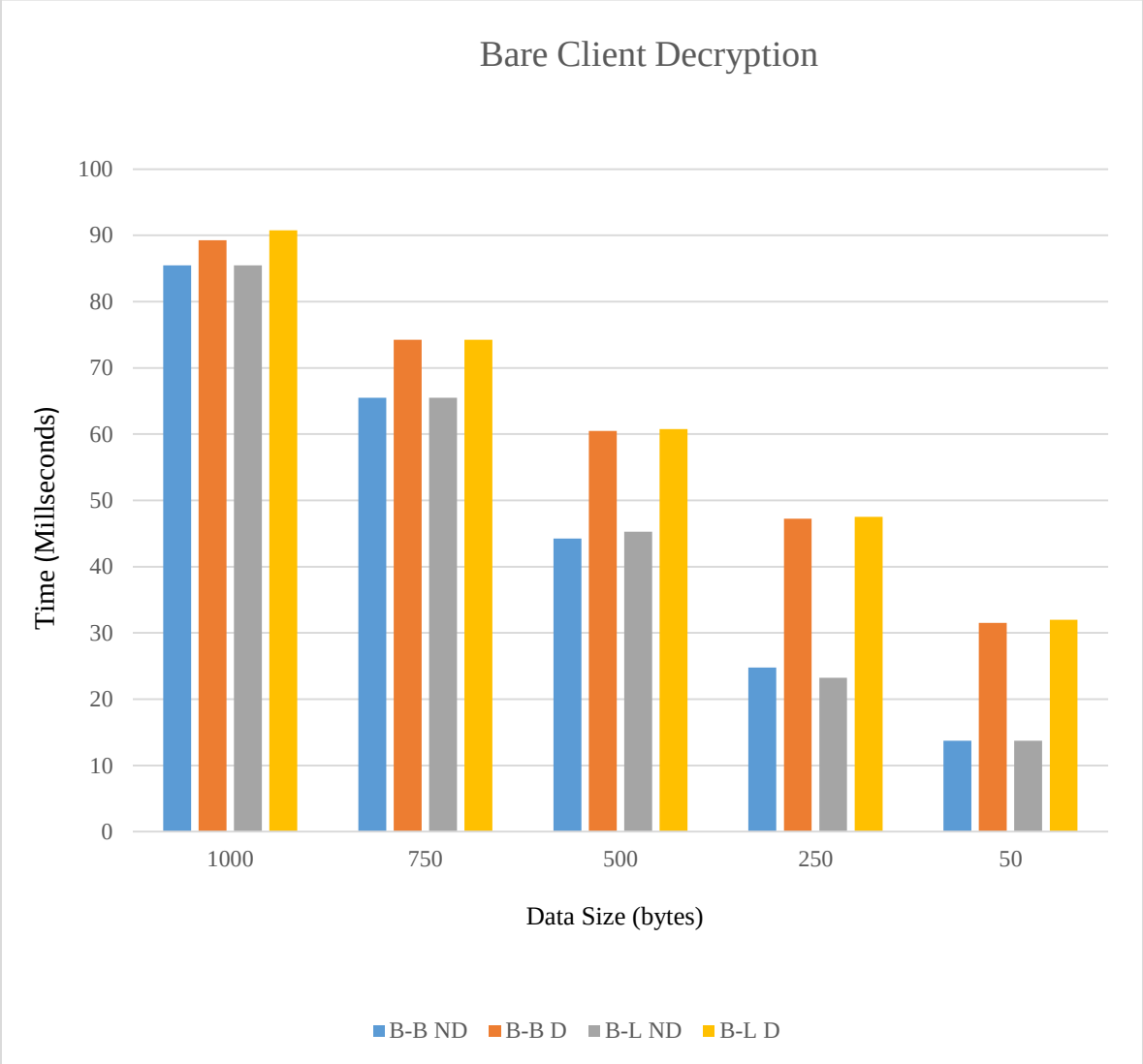


Figure 15 Bare Client Decryption: Internal Timings

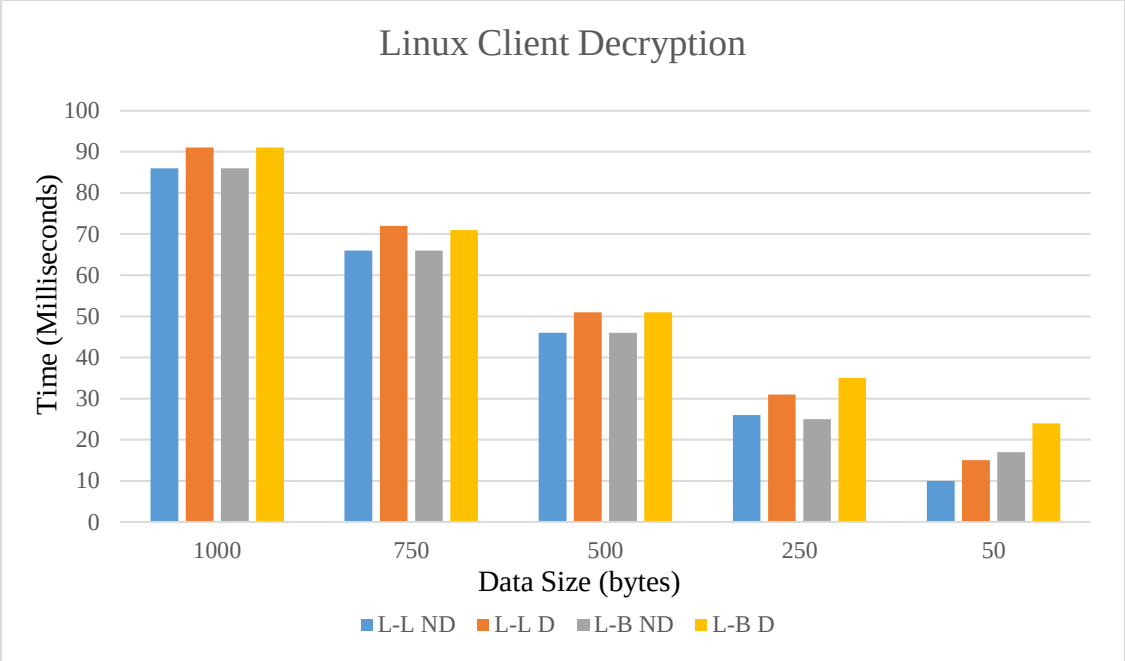


Figure 16 Linux Client Decryption: Internal Timings

5.4.4 Network Delay

In Fig. 12, the difference of Wireshark timestamps D4=WT1-WT0 reflects the round trip time (RTT) and the processing time for unencrypted packets. Likewise, the difference of Wireshark timestamps D5=WT3-WT2 reflects the RTT and the encryption time. Since we already measured the processing time for unencrypted packets using internal timings, our measurements based on Wireshark timestamps only consider D5. D5 also reflects the response time for encrypted packets. We compare the value of D5 with a difference of Wireshark timestamps that approximately represents the encryption time.

Figs. 17-21 show the response times and approximate encryption times for the bare server-bare client (BS-BC), Linux server-Linux client (LS-LC), bare server-Linux client (BS-LC), and Linux server-bare client (LS-BC) combinations respectively. As expected, response times and encryption times increase with increasing packet size, and encryption times are lower than response times for all packet sizes. The lowest response times for all packet sizes (12.8 ms to 89.0 ms) are for the bare server when it is sending to the bare client. The response times for the same bare server are higher for all packet sizes when it is sending to the Linux server. Response times for the Linux server are equal or higher with the Linux client than with the bare client for all packet sizes except 50 bytes. The lowest encryption times for all packet sizes (12.7 ms to 88.7 ms) are also for the bare server when it is sending to the bare client, and the encryption times for the bare server are higher for all packet sizes when it is sending to the Linux server. Encryption times for the Linux server have identical values for all packet sizes with both the bare and Linux clients.

5.4.5 Issues and Future Work

We now discuss some performance issues that are relevant to the bare CCM mode implementation. Future versions of the bare CCM mode implementation could address these issues by examining our results, doing more performance studies, and analyzing the code to eliminate inefficiencies in the current implementation.

First, consider the difference between the server time for encrypted packets and the time for unencrypted packets, which represents the encryption overhead. This encryption overhead is larger for all packet sizes with the bare server compared to the Linux server regardless of whether the Linux client or the bare client is used. For example, as packet size increases from 50 to 1000 bytes, the bare server encryption overhead varies between 7.25 to 57.25 ms and from 7.75 to 58.25 ms respectively with the bare and Linux clients. The corresponding values for the Linux server are 4 to 9 ms and 4 to 10 ms. Also, the decryption overhead is larger for the bare client compared to the Linux client with both the bare and Linux servers for all packet sizes with one exception (bare client with the bare server and 1000-byte packets). For example, as packet size increases from 50 to 1000 bytes, the bare client decryption overhead varies between 3.75 to 22.5 ms and 5.25 ms to 24.25 ms with the bare server and Linux server respectively. The corresponding values for the Linux client are 5 to 10 ms and 5 to 6 ms. To help analyze bare server encryption and bare client decryption performance, it may be useful to determine the various components of response time and their contribution.

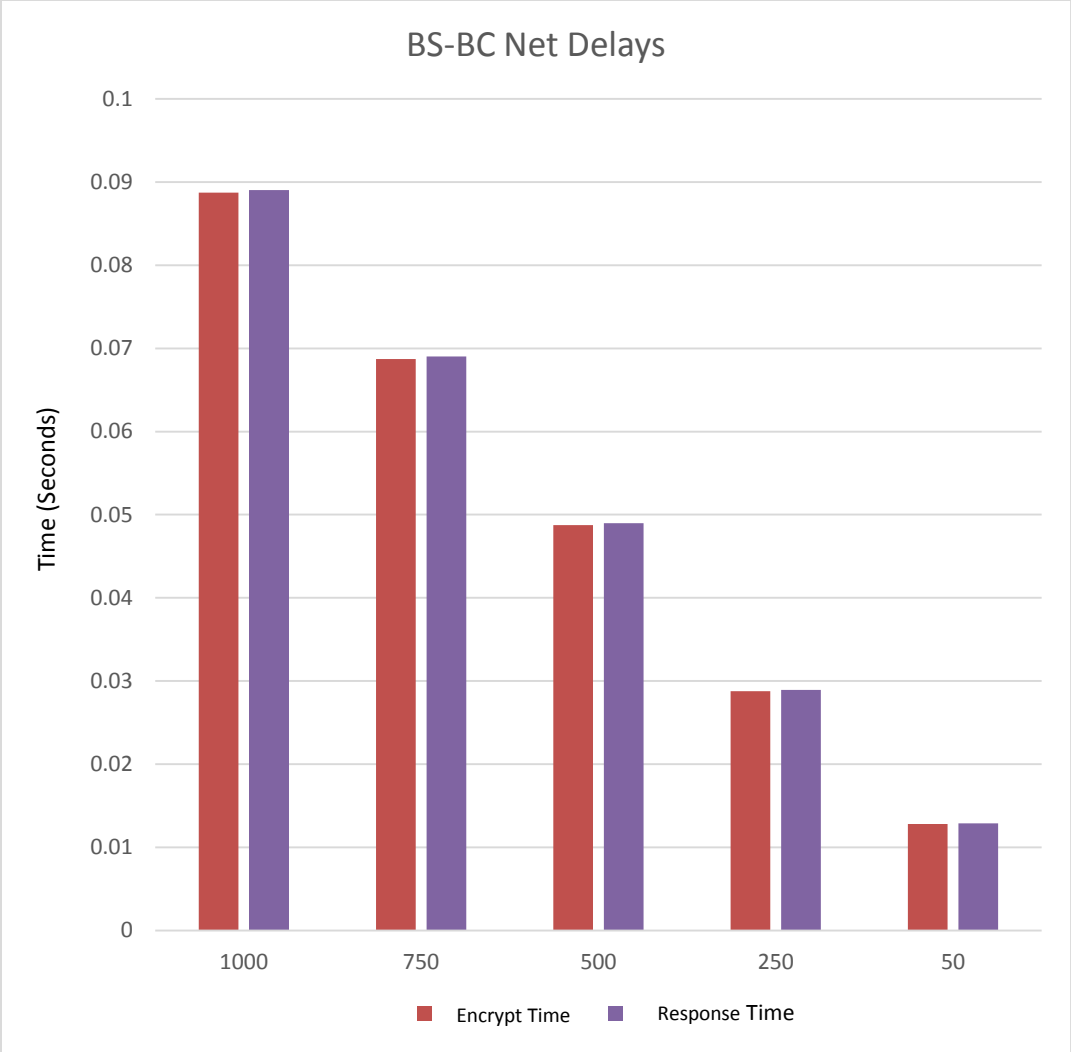


Figure 17 Bare Server-Bare Client Network Delay

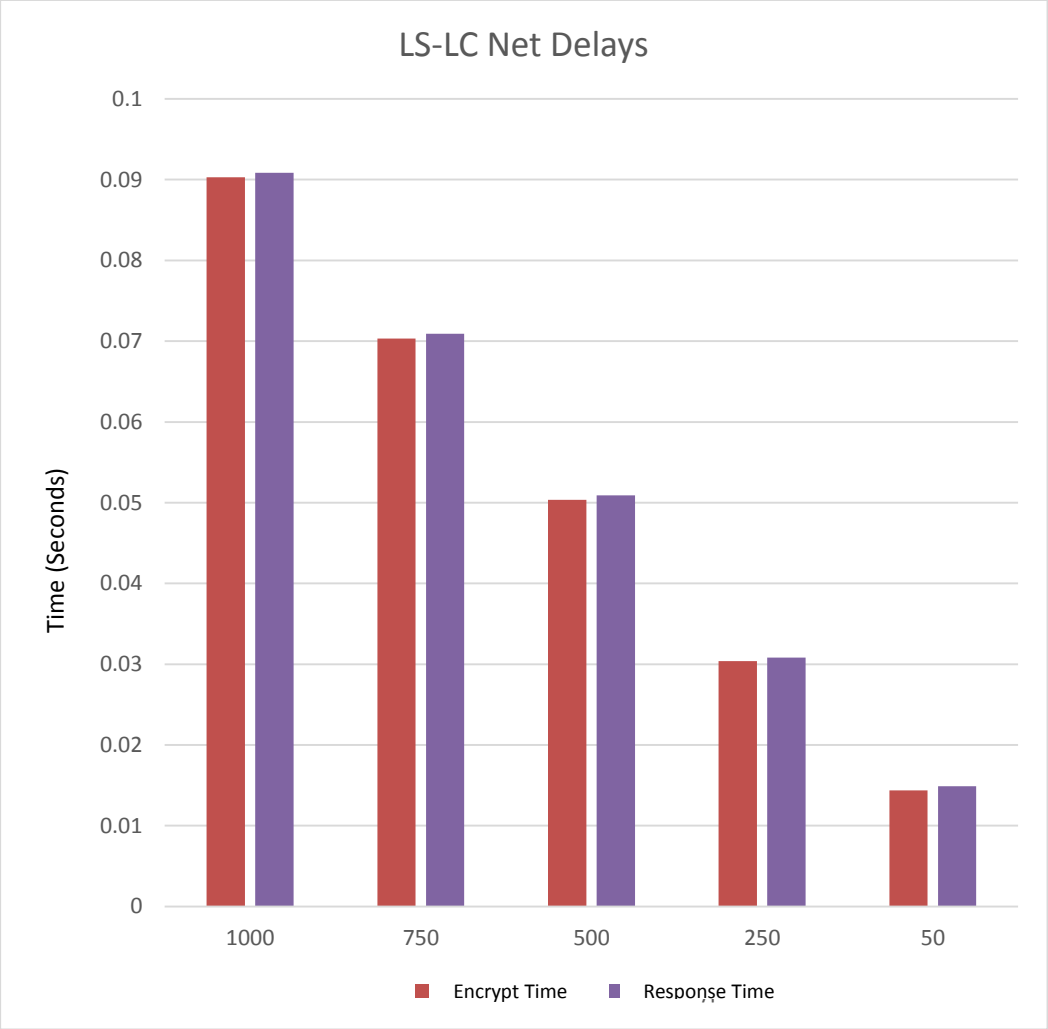


Figure 18 Linux Server-Linux Client Network Delay

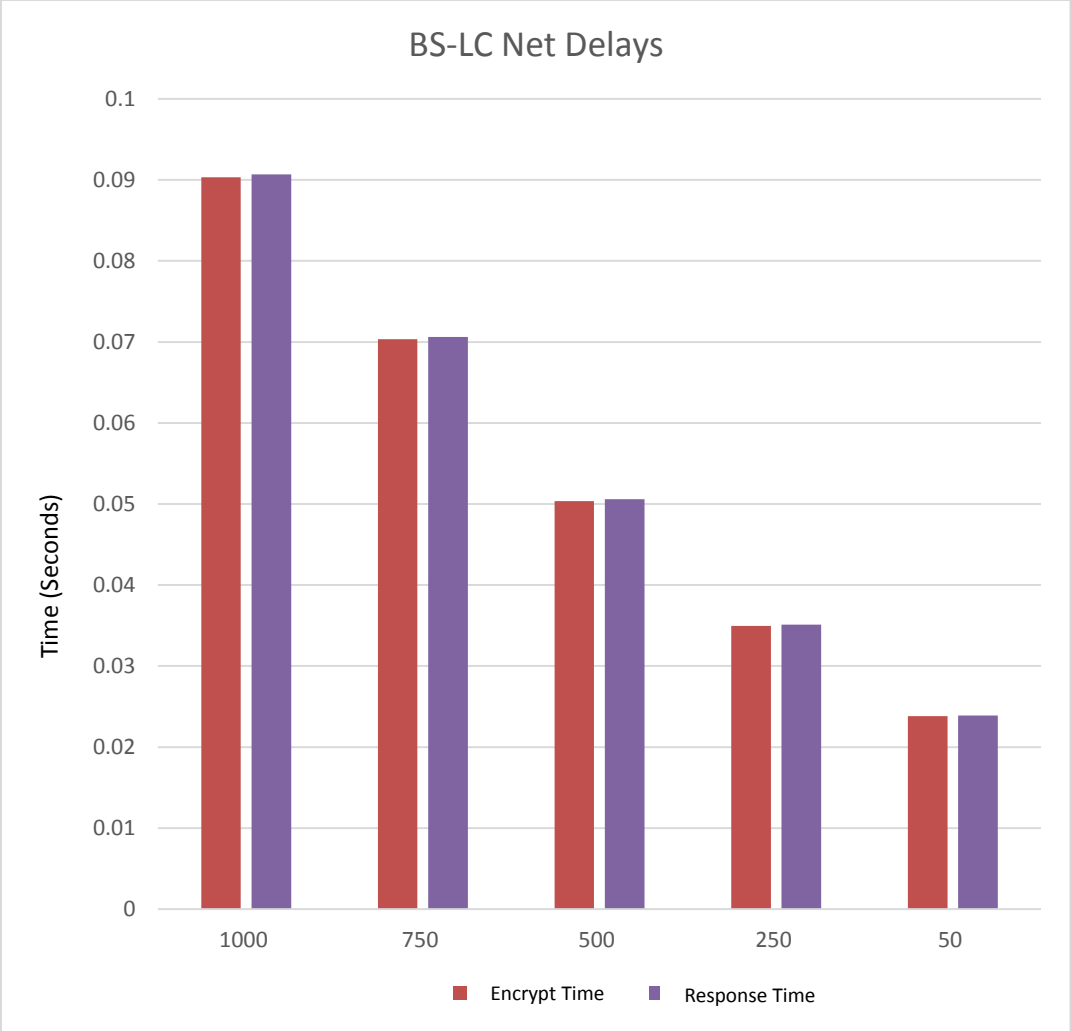


Figure 19 Bare Server-Linux Client Network Delay

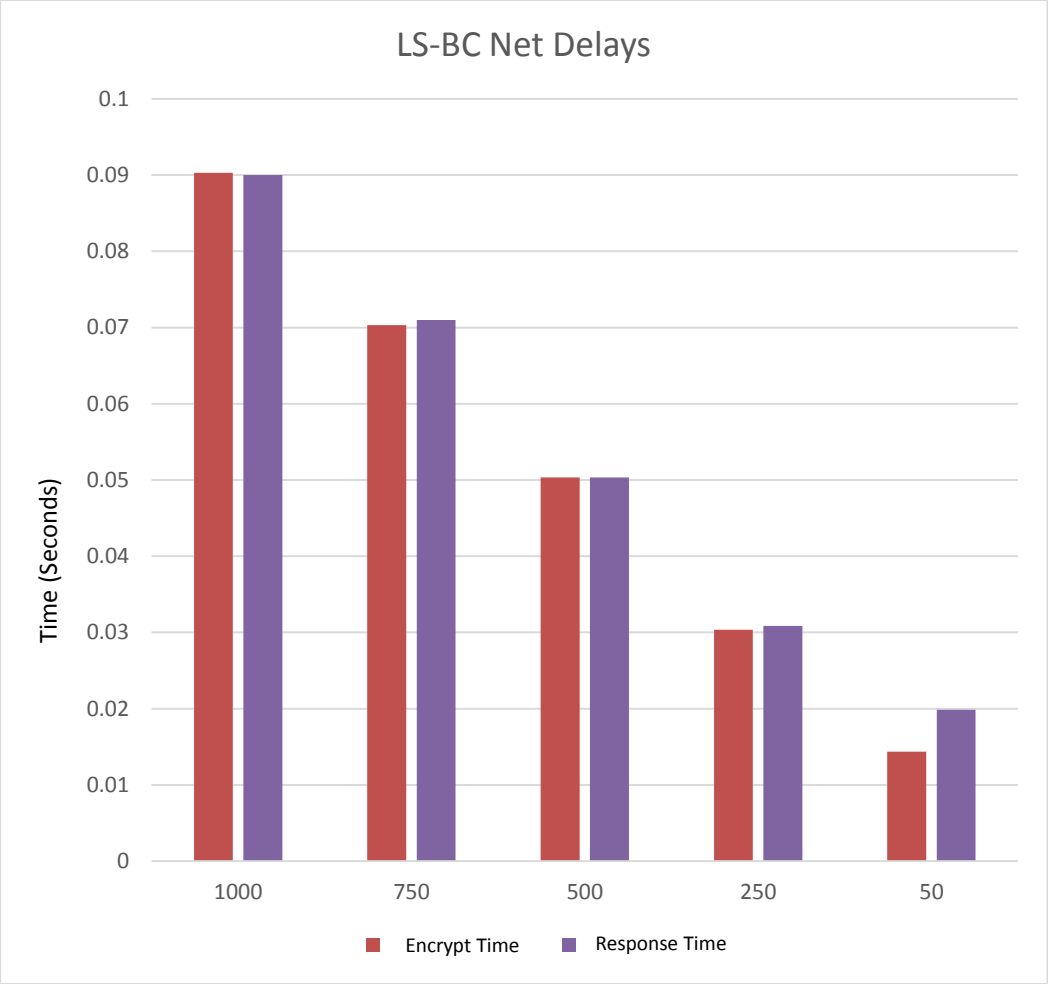


Figure 20 Linux Server-Bare Client Network Delay

6 CONCLUSION

This dissertation focused on studying three aspects of 802.11wireless networks: performance with mobile devices, analysis of an 802.11 device driver for Linux, and implementation of CCM mode on a bare machine. Our research serves as a first step towards developing secure bare applications that can connect to 802.11 wireless networks.

In the first part of our work, we measured 802.11n performance due to congestion on the wired backbone or access link by sending a single request from a wireless browser on a mobile device to a Web server along a path with several routers. Our results using two mobile devices indicate that file transfer can be inefficient when there is congestion on the wired side. In general, delay increases significantly when there is congestion, and a large increase in delay can occur even when there is a relatively low level of congestion. Experimental results for delay using passive packet captures at high levels of congestion do not appear to be reliable, and using throughput reported by passive packet capturing tools may result in throughput increases due to considering all packets exchanged. Analysis of actual packets exchanged shows that the main reason mobile devices in an 802.11n network have poor performance under congestion in the wireline network is due to RTS/CTS exchanges and the retransmission of packets.

In the second part of our work, we studied the b43.ko Linux wireless driver module and discussed its system view, code view, and external interactions with other modules. We classify the code and functions used in the module based on their type of operations and functionality. Our analysis shows that the Linux 802.11n device driver is complex and that it would be difficult to transform its source code to run on a bare machine. However, design issues identified with the design of b43.ko provide a starting point for future

development of device drivers that are independent of any OS. The driver details provided will be useful for developing techniques that can identify and transform parts of the Linux driver source code to run on a bare machine.

The last part of our work focused on implementing CCM mode for a bare machine. Transforming the Linux OpenSSL or Windows Crypto++ AES-CCM code to run on a bare machine is not easy due to their dependence on OS-specific libraries. Our implementation of AES-CCM for a bare machine directly uses the relevant specification documents. We tested the interoperability of the bare implementation with the Linux OpenSSL implementation. Our studies using internal timings and network delay show that more work is necessary to improve the bare CCM mode implementation so that it performs better than the Linux implementation in all cases.

This research could serve as the basis for several future projects. The study on mobile device performance in an 802.11n network indicates that more research on approaches to increasing throughput and reducing delays would be useful. The study using the 802.11n Linux network device driver shows the need for new tools that could automate the analysis of driver source code. Finally, the performance study on the bare machine implementation of CCM mode provides insight into developing a more efficient implementation for both bare machine and OS-based applications.

REFERENCES

[1] S. Fiehe, J. Riihijarvi and P. Mahonen, "Experimental study on performance if IEEE 802.11n and impact of interferers in the 2.4 Ghz ISM band," in *6th International Wireless Communications and Mobile Computing Conference (IWCMC)*, 2010.

[2] S. A. Mondal and F. B. Luqman, "Improving TCP performance over wired-wireless networks," *ScienceDirect Networks*, vol. 51, pp. 3799-3811, 2007.

[3] E. Ancilloti, B. Raffaele and M. Conti, "Design and performance of throughput-aware rate adaptation protocols for IEEE 802.11 wireless networks," *ScienceDirect Performance Evaluation*, vol. 66, pp. 811-825, 2009.

[4] X. Xing, Y. Jiang and S. Mishra, "Understanding characteristics of available bandwidth in wireless environment," in *2nd International Conference on Ambient Systems, Networks, and Technologies (ANT)*, 2011.

[5] S. Frohn, S. Gubnaer and C. Lindemann, "Analyzing the effective throughout in multi-hop IEEE 80211n networks," *Comput. Commun.*, vol. 34, no. 16, pp. 1912-1921, 2011.

[6] I. Kedzo, J. Ozegovic and A. Kristic, "BPC-A binary priority countdown protocol," in *20th International Conference in Siftware, Telecommunications and Computer Networks (SoftCOM)*, 2012.

[7] A. P. Jardosh, K. N. Ramachandran, K. C. Almeroth and E. M. Belding-Royer, "Understanding congestion in IEEE 802.11b wireless networks," in *5th ACM Conference on Internet Measurement (IMC)*, 2005.

[8] Y. C. Chen and e. al, "A measurement-based study of multipath TCP performance over wireless networks," in *ACM Conference on Internet Measurement (IMC)*, 2013.

[9] A. Kadav and M. M. Swift, "Understanding modern device drivers," in *Asplos XVII International Conference on 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[10] A. Amar, S. Joshi and D. Wallwork, "Generic Driver Model," May 2014. [Online]. Available: http://www.design-reuse.com/articles/a8584/generic-driver-model.html. [Accessed May 2015].

[11] S. Boyd-Wickizer and N. Zeldovich, "Tolerating malicious device drivers in Linux," in *USENIX Annual Technical Conference*, Boston, 2010.

[12] V. Chipounov and G. Gandea, "Reverse engineering of binary device drivers with RevNIC," in *1st EuroSys Confeernce on Systems*, Belgium, 2006.

[13] J. LeVasseur, V. Uhlig, J. Stoess and S. Gotz, "Unmodified device driver reuse and improved system dependability via virtual machines," in *6th Symposium on Operating Systems Design and Implementation*, San francisco, 2004.

[14] R. K. Karne, K. V. Jaganathan, N. Rosa and T. Ahmed, "Dispersed Operating System Computing (DOSC)," in *Object-Oriented Programming, Systems, Languages and Applications*, San Diego, 2005.

[15] R. K. Karne, A. L. Wijesinha, U. Okafor and P. Appiah-Kubi, "Eliminating the operating system via the bare machine computing paradigm," in *The 5th International Conference on Future Computational Technologies and Applications*, Valencia, 2013.

[16] R. K. Karne, S. Liang, A. L. Wijesinha and P. Appiah-Kubi, "A bare PC mass storage USB driver," *International Journal of Computers and Their Applications (IJCA),* 2014.

[17] L. He, R. K. Karne and A. L. Wijesinha, "Design and performance of a bare PC web server," *International Journal of Computers and Their Applications*, 2008.

[18] G. H. Ford, R. K. Karne, A. L. Wijesinha and P. Appiah-Kubi, "The design and implementation of bare PC email server," in *33rd Annual IEEE International Computer Software and Applications Conference*, Seattle, 2009.

[19] G. H. Khaksari, A. L. Wijesinha, R. K. Karne, L. He and S. Girumula, "A peer-to-peer bare PC VoIP application," in *4th Annual IEEE Consumer Communications and Networking Conference*, Las Vegas, 2007.

[20] P. Appiah-Kubi, R. K. Karne and A. L. Wijesinha, "A bare PC TLS webmail server," in *International Conference on Computing, Networking and Communications (ICNC)*, Maui, Hawaii, 2012.

[21] B. Rawal, R. K. Karne and A. L. Wijesinha, "Mini webserver clusters based on HTTP request splilting," in *13th IEEE International Conference on High Performance Computing and Communications (HPCC)*, Banff, Canada, 2011.

[22] U. Okafor, R. K. Karne, A. L. Wijesinha and P. Appiah-Kubi, "A methodology to transform OS based applications to a bare machiine application," in *The 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (IEEE TrustCom-13)*, Melbourne, Australia, 2013.

[23] H. N. Whiting, R. Housley and N. Ferguson, "Counter with CBC-MAC, RFC 3610, September 2003.," September 2003.

[24] M. Dworkin, "Recommendation for block cipher modes of operation: the CCM mode for authentication and confidentiality," in *NIST Special Publication 800-38C,* May 2004.

[25] D. McGrew and D. Bailey, "AES-CCM cipher suites for Transport Layer Security," in *RFC 6655,* July 2012.

[26] D. McGrew, D. Bailey, M. Campagna and R. Dugal, "AES-CCM Elliptic Curve Cryptography (ECC) cipher suites for TLS," RFC 7251, June 2014.

[27] R. Housley, "Using Advanced Encryption Standard (AES) CCM mode with IPsec Encapsulating Security Payload (ESP)," RFC 4309, December 2005.

[28] D. Black and D. McGrew, "Using authenticated encryption algorithms with the encrypted payload of the Internet Key Exchange version 2 (IKEv2) protocol," RFC 5282, August 2008.

[29] R. Housley, "Using AES-CCM and AES-GCM authenticated encryption in the Cryptographic Message Syntax (CMS)," RFC 5084, November 2007.

[30] J. Johnson, "On the security of CTR+CBC-MAC," *Selected Areas in Cryptography, Lecture Notes in Computer Science,* vol. 2595, pp. 76-93, February 2003.

[31] E. Lopez-Trejo, F. Rodriguez and A. Diaz-Perez, "An FPGA implementation of CCM mode using AES," in *International Conference on Information Security and Cryptology, pp. 208-215*, 2005.

[32] L. Huai, X. Zou and Y. Han, "An energy-efficient AES-CCM implementation for IEEE802.15.4 wireless sensor networks," in *Networks Security, Wireless Communications and Trusted Computing (NSWCTC)*, 2009.

[33] A. Samiah, A. Aziz and N. Ikram, "An efficient software implementation of AES-CCM for IEEE 802.11iWireless St.," in *Computer Software and Applications Conference (COMPSAC)*, 2007.

[34] J. H. Yoo, "Fast software implementation of AES-CCM on multiprocessors," *Algorithms and Architectures for Parallel Processing, Lecture Notes in Computer Science*, vol. 7017, pp. 300-311, 2011.

[35] Wireshark, "Wireshark," 15 April 2016. [Online]. Available: http://www.wireshark.org. [Accessed 15 May 2016].

[36] N. R. Laboratory, "Mgen," 15 April 2016. [Online]. Available: http://cs.itd.nrl.navy.mil/work/mgen. [Accessed 5 May 2016].

[37] OpenSSL.org, "TLS/SSL and Crypto Library," 15 April 2016. [Online]. Available: https://www.openssl.org/. [Accessed 15 April 2016].

[38] O. S. Community, "CCM Mode, Crypto++ 5.6.3 Free C++ class library of cryptographic schemes," 15 April 2016. [Online]. Available: https://www.cryptopp.com/wiki/CCM_MODE. [Accessed 15 April 2016].

CURRICULUM VITAE

NAME: William R. Agosto Padilla

[redacted]

PROGRAM OF STUDY: Information Technology

DEGREE AND DATE TO BE CONFERRED: Doctor of Science, August 2016

EDUCATION:

Doctor of Science, Information Technology
Towson University, Towson, Maryland – May 2016

Master of Science, Information and Telecommunication Systems (concentration in Computer Communications and Internets)
Johns Hopkins University, Baltimore, Maryland – May 2008

Computer Systems Internship, National Security Agency, Department of Defense, Fort Meade, Maryland -- 2000

Master of Arts, Clinical and Cognitive Psychology (Clinical Psychology, Psychometrics, and Children Diagnostics)
Loyola College, Baltimore, Maryland – May 1996 (degree pending)

Bachelor of Arts, Behavioral and Experimental Psychology (concentration in Biopsychology and Neuro-perception Systems)
University of Maryland, College Park, Maryland – September 1993

CERTIFICATES:

Education and Training Officer, National Cryptologic School
Department of Defense, National Security Agency, Fort Meade, Maryland

Graduate Certificate, Information and Telecommunications Systems
Johns Hopkins University, Baltimore, Maryland

Voice Language Analyst, National Cryptologic School, Department of Defense, National Security Agency, Fort Meade, Maryland

Graphics Language Analyst, National Cryptologic School, Department of Defense, National Security Agency, Fort Meade, Maryland

Adjunct Faculty, National Cryptologic School, Department of Defense, National Security Agency, Fort Meade, Maryland

SKILLS AND ABILITIES:

Knowledgeable and fluent in several programming languages, including C, C++, Java, Python, SAS, Perl, JavaScript, SQL, and HTML.

AWARDS AND HONORS:

Exceptional Analyst Fellowship, Office of the Director of National Intelligence (DNI), National Intelligence University, Defense Intelligence Agency, Department of Defense

Top Cryptologic Linguist (9 times), United States Marine Corps, Department of the Navy, Washington DC

Defense Meritorious Service Medal (DMSM), Exceptional Achievement, National Security Agency, Department of Defense, Fort Meade Maryland

ARTICLES / PULICATIONS:

Psychosocial Aspects of Computing Hackers; 2002

Evaluating Performance Reduction in 802.11n Wireless LANs due to Network Congestion; 2015

A Study of a Linux Wireless Device Driver Module; 2016

PROFESSIONAL POSITIONS HELD & EXPERIENCE:

Adjunct Professor of Computer Information Sciences
Shaw University,
Raleigh, North Carolina; 01/2015 – Present

Global Networking and Exploitation Analyst
National Security Agency, Department of Defense
Fort Meade, Maryland; 05/2000 – 05/2013

Scientific Linguist, Voice Language Analyst
National Security Agency, Department of Defense
Fort Meade, Maryland; 01/1997 – 05/2001

Psychology Associate—Psychometrician, Office of Psychological Services,
National Security Agency, Department of Defense
Fort Meade, Maryland; 01/1996 – 01/1997

Education and Training Officer, National Cryptologic School
National Security Agency, Department of Defense,
Fort Meade, Maryland; 01/1988 – 05/1990

Adjunct Faculty, Defense Language Institute,
Language Training Command, Department of Defense,
Fort Meade, Maryland

Signal Intelligence Analyst
National Security Agency, Department of Defense
Fort Meade, Maryland; 01/1988 – 01/1997